

Spring 4-19-2024

## A Further Performance Comparison of Operations in the File System and in Embedded Key-Value Databases

Nicholas Cunningham

*Southern Adventist University*, [nicholascunningham@southern.edu](mailto:nicholascunningham@southern.edu)

Follow this and additional works at: [https://knowledge.e.southern.edu/mscs\\_reports](https://knowledge.e.southern.edu/mscs_reports)

---

### Recommended Citation

Cunningham, Nicholas, "A Further Performance Comparison of Operations in the File System and in Embedded Key-Value Databases" (2024). *MS in Computer Science Project Reports*. 12.  
[https://knowledge.e.southern.edu/mscs\\_reports/12](https://knowledge.e.southern.edu/mscs_reports/12)

This Thesis is brought to you for free and open access by the School of Computing at Knowledge Exchange. It has been accepted for inclusion in MS in Computer Science Project Reports by an authorized administrator of Knowledge Exchange. For more information, please contact [jspears@southern.edu](mailto:jspears@southern.edu).

A FURTHER PERFORMANCE COMPARISON OF OPERATIONS IN THE FILE  
SYSTEM AND IN EMBEDDED KEY-VALUE DATABASES

by

Nicholas A. Cunningham

A PROJECT

Presented to the Faculty of

The School of Computing at the Southern Adventist University

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Germán H. Alférez, Ph.D.

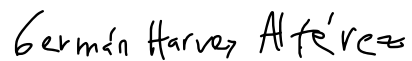
Collegedale, Tennessee

April, 2024

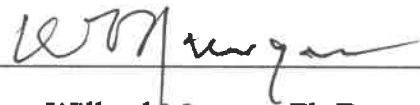


# A FURTHER PERFORMANCE COMPARISON OF OPERATIONS IN THE FILE SYSTEM AND IN EMBEDDED KEY-VALUE DATABASES

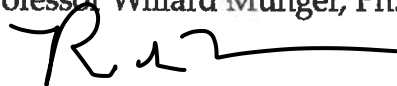
Approved by:



Professor Germán H. Alférez, Ph.D., Adviser



Professor Willard Munger, Ph.D.



Professor Richard Halterman, Ph.D

Date Approved 4/18/2024

A FURTHER PERFORMANCE COMPARISON OF OPERATIONS IN THE FILE  
SYSTEM AND IN EMBEDDED KEY-VALUE DATABASES

Nicholas A. Cunningham, M.S.

Southern Adventist University, 2024

Adviser: Germán H. Alférez, Ph.D.

A routine scenario when developing PC applications is storing data in small files or records and then retrieving and manipulating that data with a distinctive identifier (ID). In these scenarios, the developer can save the records using the ID as the filename or use an embedded on-disk key-value database. However, many file systems can have performance issues when handling many small files. As a result, developers would rather avoid depending on an embedded database if it offers little benefit or has a detrimental effect on performance for their use case. Our contribution is to compare several key-value databases—SQLite3, LevelDB, RocksDB, and Berkeley DB—based on many parameters, including the file system—NTFS, as opposed to ext4, the file system utilized on a previous project [1]—and explain the outcomes. Moreover, the metrics and technologies to be evaluated extend the metrics evaluated in our previous research work. We compare these key-value databases on two machines: a solid-state drive and a hard disk drive. Our research used the Windows Subsystem for Linux 2 (WSL 2) to work with.

# Contents

|                 |          |
|-----------------|----------|
| <b>Contents</b> | <b>v</b> |
|-----------------|----------|

|                        |            |
|------------------------|------------|
| <b>List of Figures</b> | <b>vii</b> |
|------------------------|------------|

|                       |          |
|-----------------------|----------|
| <b>1 Introduction</b> | <b>1</b> |
|-----------------------|----------|

|                                 |   |
|---------------------------------|---|
| 1.1 Problem Statement . . . . . | 1 |
|---------------------------------|---|

|                          |   |
|--------------------------|---|
| 1.2 Objectives . . . . . | 2 |
|--------------------------|---|

|                          |   |
|--------------------------|---|
| 1.3 Motivation . . . . . | 3 |
|--------------------------|---|

|                            |   |
|----------------------------|---|
| 1.4 Organization . . . . . | 3 |
|----------------------------|---|

|                     |          |
|---------------------|----------|
| <b>2 Background</b> | <b>5</b> |
|---------------------|----------|

|                                     |   |
|-------------------------------------|---|
| 2.1 Theoretical Framework . . . . . | 5 |
|-------------------------------------|---|

|                                 |   |
|---------------------------------|---|
| 2.1.1 The File System . . . . . | 5 |
|---------------------------------|---|

|                                    |   |
|------------------------------------|---|
| 2.1.2 Embedded Databases . . . . . | 8 |
|------------------------------------|---|

|   |   |
|---|---|
| 2.1.3 File System Performance . . . . . | 9 |
|---|---|

|                                 |    |
|---------------------------------|----|
| 2.1.4 Storage Devices . . . . . | 10 |
|---------------------------------|----|

|   |    |
|---|----|
| 2.1.5 Windows Subsystem for Linux . . . . . | 11 |
|---|----|

|                                |    |
|--------------------------------|----|
| 2.2 State of the Art . . . . . | 11 |
|--------------------------------|----|

|  |    |
|--|----|
| 2.2.1 Hines, Cunningham, and Alférez . . . . . | 12 |
|--|----|

|                                    |    |
|------------------------------------|----|
| 2.2.2 Educational Theses . . . . . | 12 |
|------------------------------------|----|

|          |   |           |
|----------|---|-----------|
| 2.2.3    | Comparing Key-Value Databases . . . . .       | 14        |
| 2.2.4    | Comparing File Systems . . . . .              | 14        |
| <b>3</b> | <b>Methodology</b>                            | <b>17</b> |
| 3.1      | Benchmark Implementation . . . . .            | 18        |
| 3.2      | Implementation of Utility Functions . . . . . | 25        |
| 3.3      | Taking Measurements . . . . .                 | 33        |
| 3.4      | Monitoring Performance . . . . .              | 33        |
| <b>4</b> | <b>Results</b>                                | <b>35</b> |
| 4.1      | SSD Undisturbed Results . . . . .             | 35        |
| 4.2      | HDD Undisturbed Results . . . . .             | 37        |
| 4.3      | SSD Disturbed Results . . . . .               | 38        |
| 4.4      | HDD Disturbed Results . . . . .               | 39        |
| 4.5      | Discussion . . . . .                          | 40        |
| 4.5.1    | Insert Operation . . . . .                    | 40        |
| 4.5.2    | Update Operation . . . . .                    | 40        |
| 4.5.3    | Get Operation . . . . .                       | 41        |
| 4.5.4    | Remove Operation . . . . .                    | 41        |
| 4.5.5    | Space Efficiency . . . . .                    | 42        |
| 4.5.6    | File Systems vs Embedded Databases . . . . .  | 42        |
| 4.5.7    | Effects of Compression . . . . .              | 42        |
| <b>5</b> | <b>Conclusions and Future Work</b>            | <b>43</b> |
|          | <b>Bibliography</b>                           | <b>45</b> |

# List of Figures

|     |   |    |
|-----|---|----|
| 2.1 | A concept map of the underpinnings of our project . . . . .           | 6  |
| 2.2 | Summary of the best storage options on the previous project . . . . . | 13 |
| 4.1 | The results of the undisturbed SSD . . . . .                          | 36 |
| 4.2 | The results of the undisturbed HDD . . . . .                          | 37 |
| 4.3 | The results of the disturbed SSD . . . . .                            | 38 |
| 4.4 | The results of the disturbed HDD . . . . .                            | 39 |





# Chapter 1

## Introduction

This chapter presents the following sections: Section 1.1 presents the problem statement, Section 1.2 presents the objectives, Section 1.3 presents the motivation, and Section 1.4 presents the organization.

### 1.1 Problem Statement

Our previous research work allowed us to compare the speeds of CRUD operations and the space efficiency of embedded databases [1]. However, that work only ran on the ext4 file system. Testing on one type of file system also led us to ask whether an outlier was due to the embedded database or the file system that we were using. There is a need for extending that previous work to analyze key-value storage performance on solid-state drives (SSDs) and hard disk drives (HDDs). It is also necessary to analyze the causes of outlier and performance degradations in the experiments.

Knowledge about the speed and efficacy of data is vital to understanding the efficacy of the database systems, the file system, and the hardware, as well as which are the most effective in their spatial efficiency and speed. We expect that an extension of our previous

research work will allow developers to delineate between the different options and make an informed decision on which tools are best for their scenario or even which scenarios are best for the tools they can use.

## 1.2 Objectives

The main objective of this project is to delineate the differences in operation speed and space efficiency between four embedded databases and two different directory structures. This process was evaluated on two Windows machines and performed on the NTFS file system. Our previous research work benchmarked the speed and efficiency of an ext4 file system, the primary file system of Linux machines. By mounting an NTFS volume and accessing it through the Windows Subsystem for Linux 2 (WSL 2) [2], we can benchmark the effectiveness of using an embedded database on a Windows device. We also compared the efficacy of two storage devices– a SSD and a HDD.

These are the objectives of this project:

- Compare the speed in nanoseconds for CRUD operations of SQLite3, Berkeley DB, RocksDB, LevelDB, and the file system. The speed will determine which of these storage methods is most effective at storing the information quickly.
- Compare the space efficiency of the RAM at peak memory. The amount of memory needed to execute the algorithm is important because it can determine the speed at which applications can run.
- Test the bounds of the aforementioned key-value databases by creating a benchmark to run on an NTFS file system and different storage devices (SSD and HDD).
- Test the impact of the aforementioned configurations while a YouTube video is running on Google Chrome and compare it when the video is not playing.

- Compare the effects of compressible and incompressible data.

### **1.3 Motivation**

Although there have been approaches to compare some of the leading embedded databases to the file system, none have been very vast in their assessments of the ideal embedded database or how they compare to the file system. None of these comparisons include testing while other applications are running in the background or testing on different storage devices. Still, minimal has been tested with multiple hard drives and programs running in the background. This benchmark would allow people who work with databases to make informed decisions on which embedded databases work well for which product.

### **1.4 Organization**

This document is organized as follows:

- Chapter 2 presents a theoretical framework of the concepts for this project.
- Chapter 3 outlines the methodology for planning and constructing the benchmark.
- Chapter 4 details the project evaluation.
- Chapter 5 presents the conclusions and future work.



# Chapter 2

## Background

The theoretical framework and state-of-the-art introduce terms and include relevant works related to the topics of embedded databases and file system performance.

### 2.1 Theoretical Framework

Figure 2.1 presents a concept map explaining the concepts in the research work.

#### 2.1.1 The File System

A way to collect data is by using the file system. The file system stores and arranges data on any digital storage device. The file system also takes control of the vacant space. Essentially, when a file is stored on a storage medium (HDD, SSD, USB drives, etc.) that has a specific capacity to be used both to read and write information, each byte of information on it has a particular offset from the storage start known as an address and can be referenced by its address. In this regard, storage can be treated as a grid with a set of numbered cells (each cell is a single byte). And any item saved to the storage gets its own cells [3].

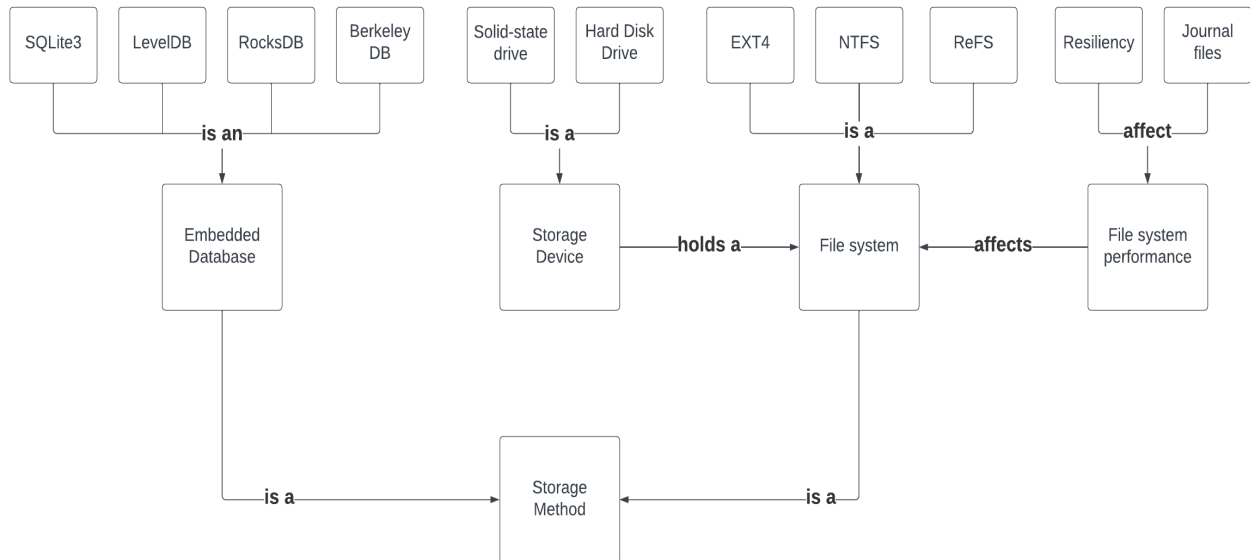


Figure 2.1: A concept map of the underpinnings of our project

There are different types of file systems. Some of the most common file systems for Windows are NTFS, FAT, exFAT, ReFS, and HPFS [4].

NTFS (New Technology File System) was introduced in July 1993 with Windows NT 3.1 and is the most widely used file system for Windows computers. NTFS has journaling capabilities, which means it can track changes not yet committed to the file system’s central part by recording the changes in a “journal,” which is usually a circular log. Some of the advantages of NTFS are the improvements over FAT in application speed and space efficiency by compressing the applications. Some disadvantages are fragmentation issues, flash memory (such as SSD drives) not having head movement delays, and the high access time of mechanical hard disk drives. Hence, fragmentation has only a minor penalty. There also might be boot issues if some system files are needed at boot time [5].

FAT (File Allocation Table) was developed in 1977 for floppy disks and later adapted for hard disks and other devices. FAT file systems are effectively tables that act as an index for their content. The structure is arranged into the boot sector, the File Allocation Table,

and the data storage area. Variants came with an increase in storage capacity. The original 8-bit FAT had 8-bit table elements and was used for Microsoft MS-DOS, a system released by Microsoft in 1981 [6]. FAT12 and FAT16 were applied to old floppy disks and are not extensively used today. FAT32, however, is still widely used because of its broad capability. It can also be used on Linux and macOS devices, which makes it essential for portable devices. FAT32 does not have native support for more than 32 GB storage capacities. For this reason, it can be used on Windows-compatible external storage, disk partitions under 32 GB when formatted with this OS's built-in tool, or up to 2 TB when other means are employed to format the storage. The file system also doesn't allow creating files that exceed 4 GB [4].

The exFAT file system is the successor to FAT32 in the FAT family of file systems. It was introduced with Windows Embedded CE 6.0 in November 2006. Its goals were to retain the simplicity of FAT-based file systems, enable very large files and storage devices, and incorporate extensibility for future innovation. exFAT is, like FAT, widely compatible with other operating systems. It also enables users to store files larger than 4 GB and has no file or partition size limits. Although it is compatible with other operating systems, it is less compatible than FAT. It also does not have any journaling capabilities nor the advanced features of NTFS, which has been around since the 1990s [7].

ReFS (Resilient File System) is the latest development of Microsoft released with Windows Server 2012 and later added to Windows 8.1. Now, it is also available for Windows 11. ReFS has been designed to address some of NTFS's shortcomings, especially concerning data corruption. ReFS enhances failure tolerance through its Copy-on-Write mechanism, storing older metadata copies in different locations for easy file system integrity restoration and data loss prevention. It also employs checksums to detect potential data corruption. ReFS is a unique Windows format with B+-trees as an on-disk structure for representing metadata and file data. This makes it ideal for ample storage and high availability systems.



Still, it lacks the stability of NTFS and compatibility with other Windows-based devices, making it less suitable for other systems [8].

### **2.1.2 Embedded Databases**

Embedded databases are database management systems included in an application instead of a separate server. Thus, they do not need a separate server or even an Internet connection to be used. Embedded databases are best used when information needs to be stored on a local machine.

Oracle's embedded database, Berkeley DB, is a fast, open-source database developed at the University of California. It is used in popular open-source products like Linux and BSD Unix, Apache Web server, and OpenOffice productivity suite [9].

Google's LevelDB is a lightweight Bigtable storage implementation with a native C++ API and third-party API wrappers for Python, PHP, Go, Node.js, and Objective C. It is distributed under the New BSD License and is a lightweight implementation of Bigtable storage design [10].

Meta created RocksDB, a LevelDB fork that prides itself on its high performance, fast storage optimization, adaptability, and basic and advanced database operations. It is also a storage engine in other databases like ArangoDB, Ceph, CockroachDB, MongoRocks, MyRocks, Rocksandra, TiKV, and YugabyteDB [11].

SQLite [12] is a widely deployed SQL database engine that implements a self-contained, server-less, zero-configuration system. Its source code, primarily C, is in the public domain and includes a native C library and command line client. It is included in various operating systems, including Android, Dropbox, iOS, OS X, and Windows 10 [13].

### 2.1.3 File System Performance

Modern file systems like FAT, ext3, and ext4 allocate space for files in a unit of a cluster or block, regardless of the file size. This can lead to space waste if many files are smaller than the cluster size. Additionally, directory lookup can be affected if many files are directly under a single folder. NTFS and ReFS are substantially better at maximum volume and file size, allowing up to 8 PB (petabytes, 1,000x 1 terabyte and 1,000,000x 1 gigabyte) and 35 PB, respectively [14].

ReFS offers built-in resiliency, checking files as they are read or written to prevent data corruption. It also periodically checks all files on the drive to identify and repair corrupted data. ReFS supports large amounts of data and has a mirror-accelerated parity feature for efficient storage and high performance [8]. Windows cannot boot from ReFS [14] because its increased system resource consumption can create file integrity issues and an inability to convert NTFS data to ReFS. It also lacks features like compression and encryption [8].

NTFS uses journal files to record metadata changes and restore file system consistency after system failures. It uses access control lists and user-level encryption for security and allows users to restrict access. NTFS also features compression, hard link, and disk quota to save system resources and space [14]. NTFS's primary weakness is its limited compatibility with non-Windows operating systems and other removable device support [5].

ext4 is widely seen as the faster of the two most popular file systems, having faster file system checks, better performance, improved file handling and fewer fragmentation issues, lower overhead, faster operations, and improved data reliability [14]. However, it lacks compatibility, being less popular than Windows's NTFS [5]. It also lacks advanced file permissions and access control, built-in encryption, and has issues with larger file sizes and protection against data loss [15].

Memory cards typically use FAT32 file systems, which cannot store files larger than

4GB. NTFS, on the other hand, has higher disk utilization and faster read and write speeds. While FAT32 is suitable for smaller-capacity memory devices, NTFS is more suited for larger disks [16].

#### **2.1.4 Storage Devices**

A storage device is any computing hardware used for temporarily or permanently storing data files and objects, either internal or external, to a computer, server, or computing device. There are two types of storage devices: primary and secondary storage devices [17]. Primary storage devices are designed to hold critical data for the operation of the device temporarily and are internal to the device. Secondary storage is slower than primary storage due to its lack of direct access to the CPU. However, it compensates for this by offering more excellent data retention, being twice as cheap as primary storage, and storing significantly more information [17].

The two secondary storage types important to this project are solid-state drives and hard disk drives. Both are used to store and retain the device's files and to work with the system's memory and processor to access data.

Hard disk drives (HDDs) consist of a spinning platter, a magnetic disk with tracks and sectors for data storage, and an actuator arm that reads and writes data across the platter. The platter spins on a spindle to speed up the process. Solid-state drives (SSDs) are built from silicon memory chips; therefore, there is no rotational delay and barely any seek time. Due to this, SSDs are more sought after for users who require high performance. A Dell white paper [18] compared hard disk drives to solid state drives regarding price and performance (using input/output operations per second (IOPS), which determines how many single operations per second can be handled). They determined that hard disk drives have comparable speeds and a better price for sequential workloads, but Solid-state drives are

better performers for random workloads. Solid-state drives have recently been considered preferable because they are better than hard disk drives at loading large amounts of data but have less capacity per drive and are more expensive.

### **2.1.5 Windows Subsystem for Linux**

Windows Subsystem for Linux (WSL) is a feature of Windows that allows developers to use Linux without a separate environment. It was first released on August 2, 2016. It was used as a compatibility layer for running Linux WSL 1 and was released in August 2016 as a compatibility layer for running Linux binary executables on the Windows kernel[19]. It is available for Windows 10, Windows 10 LTSC/LTSC, Windows 11, Windows Server 2016, Windows Server 2019 and Windows Server 2022. WSL 2 was announced in May 2019 [20], introducing a true Linux kernel with Hyper-V capabilities. WSL 2 runs in a managed virtual machine and implements the entire Linux kernel, making it compatible with more Linux binaries than WSL 1. In June 2019, WSL 2 began to be available to Windows 10 customers through the Windows Insider program.

## **2.2 State of the Art**

Due to the advances in computer architecture, comparing different computer systems' performances becomes more complicated just by looking at their specifications as computer architecture develops. As a result, tests that enabled the comparison of various designs have been proposed [21, 22]. These works have been essential in the birth of this research. They have laid the groundwork for the project.

### **2.2.1 Hines, Cunningham, and Alférez**

In our previous work [1], we explored the ext4 file system's performance and compared it to some of the more popular open-source embedded key-value databases. In the present research work, we delve into the performance of the operations on the primary Windows file system, NTFS, on HDDs and SSDs and determine what impact using another program in the background will have on the benchmark results.

In [1], the benchmark results were exported to a CSV file, then moved to a chart (Figure 2.2) to show which storage option was best, either by speed (in microseconds) or in space efficiency of the RAM. The chart was color-coded to highlight the most successful embedded database through the different record sizes and counts. Cells are color-coded by store: yellow is for Berkeley DB, red is for LevelDB, gray is for RocksDB, light blue is for Flat File System, dark blue is for nested File System, and green is for SQLite. It also compared compressible data and incompressible data. Berkeley DB was the quickest with the insert and update features, especially among record sizes below 10 KiB. The flat file system was faster and had larger record sizes than any key-value database. The previous project did not compare the effects of different hard drives, nor did it compare these effects with other programs in the background. This paper also tested on ext4, the journaling file system for Linux. However, running our benchmark on Windows Subsystem for Linux 2 (WSL 2) enables us to use the more popular New Technology File System (NTFS), a proprietary journaling file system developed by Microsoft that is used on Windows devices.

### **2.2.2 Educational Theses**

Some of the works comparable to ours are more educatory than exploratory. Najafzade and Mariezcurrena's work [23] is an exploratory work to ours, teaching about embedded databases and an introduction to Berkeley DB— how to use the operations and how to create

| Operation        | Record Size     | Incompressible     |                    |                   |                   |                    | Compressible      |                   |                    |                   |                  |
|------------------|-----------------|--------------------|--------------------|-------------------|-------------------|--------------------|-------------------|-------------------|--------------------|-------------------|------------------|
|                  |                 | 100                | 1,000              | 10,000            | 100,000           | 1,000,000          | 100               | 1,000             | 10,000             | 100,000           | 1,000,000        |
| insert           | <1 KiB          | Berkeley (2 μs)    | Berkeley (4 μs)    | Berkeley (5 μs)   | Berkeley (7 μs)   | Berkeley (8 μs)    | Berkeley (9 μs)   | Berkeley (6 μs)   | Berkeley (6 μs)    | Berkeley (7 μs)   | Berkeley (8 μs)  |
|                  | 1 - 10 KiB      | Berkeley (7 μs)    | Berkeley (8 μs)    | Berkeley (10 μs)  | Flat FS (22 μs)   | Berkeley (12 μs)   | Berkeley (7 μs)   | Berkeley (8 μs)   | Berkeley (9 μs)    | Flat FS (22 μs)   | Berkeley (12 μs) |
|                  | 10 - 100 KiB    | Flat FS (45 μs)    | Flat FS (47 μs)    | Berkeley (45 μs)  | Flat FS (45 μs)   |                    | Flat FS (47 μs)   | Berkeley (45 μs)  | Berkeley (47 μs)   | Berkeley (95 μs)  |                  |
|                  | 100 KiB - 1 MiB | Flat FS (296 μs)   | Flat FS (300 μs)   | Flat FS (296 μs)  |                   |                    | Flat FS (307 μs)  | Flat FS (296 μs)  | Nested FS (952 μs) |                   |                  |
| update           | <1 KiB          | Berkeley (1 μs)    | Berkeley (3 μs)    | Berkeley (4 μs)   | Berkeley (5 μs)   | Berkeley (6 μs)    | Berkeley (3 μs)   | Berkeley (4 μs)   | Berkeley (4 μs)    | Berkeley (6 μs)   | Berkeley (7 μs)  |
|                  | 1 - 10 KiB      | Berkeley (6 μs)    | Berkeley (8 μs)    | Berkeley (10 μs)  | LevelDB (27 μs)   | RocksDB (35 μs)    | Berkeley (6 μs)   | Berkeley (8 μs)   | Berkeley (9 μs)    | LevelDB (19 μs)   | LevelDB (18 μs)  |
|                  | 10 - 100 KiB    | Berkeley (45 μs)   | Berkeley (49 μs)   | Berkeley (47 μs)  | Berkeley (55 μs)  |                    | Berkeley (44 μs)  | Berkeley (48 μs)  | Berkeley (49 μs)   | Berkeley (69 μs)  |                  |
|                  | 100 KiB - 1 MiB | Nested FS (428 μs) | Nested FS (416 μs) | Berkeley (575 μs) |                   |                    | Berkeley (520 μs) | Berkeley (601 μs) | Berkeley (749 μs)  |                   |                  |
| get              | <1 KiB          | Berkeley (1 μs)    | LevelDB (2 μs)     | Berkeley (2 μs)   | Berkeley (3 μs)   | Berkeley (4 μs)    | Berkeley (1 μs)   | LevelDB (2 μs)    | Berkeley (2 μs)    | Berkeley (3 μs)   | Berkeley (4 μs)  |
|                  | 1 - 10 KiB      | LevelDB (2 μs)     | Berkeley (4 μs)    | Berkeley (4 μs)   | Flat FS (7 μs)    | Berkeley (1266 μs) | LevelDB (2 μs)    | Berkeley (4 μs)   | Berkeley (4 μs)    | Flat FS (7 μs)    | SQLite (23 μs)   |
|                  | 10 - 100 KiB    | RocksDB (7 μs)     | Flat FS (13 μs)    | Flat FS (13 μs)   | Flat FS (12 μs)   |                    | RocksDB (7 μs)    | RocksDB (11 μs)   | Nested FS (14 μs)  | Flat FS (12 μs)   |                  |
|                  | 100 KiB - 1 MiB | LevelDB (65 μs)    | LevelDB (75 μs)    | Flat FS (68 μs)   |                   |                    | Flat FS (82 μs)   | Nested FS (78 μs) | Flat FS (68 μs)    |                   |                  |
| remove           | <1 KiB          | Berkeley (1 μs)    | Berkeley (3 μs)    | Berkeley (4 μs)   | LevelDB (5 μs)    | LevelDB (5 μs)     | Berkeley (2 μs)   | Berkeley (4 μs)   | Berkeley (4 μs)    | LevelDB (5 μs)    | Berkeley (7 μs)  |
|                  | 1 - 10 KiB      | Berkeley (6 μs)    | LevelDB (6 μs)     | LevelDB (7 μs)    | LevelDB (5 μs)    | LevelDB (4 μs)     | Berkeley (6 μs)   | Berkeley (7 μs)   | Berkeley (9 μs)    | Flat FS (20 μs)   | Flat FS (28 μs)  |
|                  | 10 - 100 KiB    | LevelDB (7 μs)     | RocksDB (17 μs)    | RocksDB (17 μs)   | Nested FS (32 μs) |                    | Flat FS (23 μs)   | Nested FS (27 μs) | Flat FS (26 μs)    | Nested FS (32 μs) |                  |
|                  | 100 KiB - 1 MiB | RocksDB (26 μs)    | RocksDB (34 μs)    | Flat FS (87 μs)   |                   |                    | Flat FS (87 μs)   | Flat FS (85 μs)   | Flat FS (121 μs)   |                   |                  |
| space efficiency | <1 KiB          | SQLite (50%)       | SQLite (56%)       | LevelDB (77%)     | RocksDB (91%)     | RocksDB (92%)      | SQLite (52%)      | SQLite (57%)      | LevelDB (85%)      | RocksDB (91%)     | RocksDB (112%)   |
|                  | 1 - 10 KiB      | SQLite (76%)       | SQLite (83%)       | SQLite (86%)      | LevelDB (97%)     | RocksDB (99%)      | SQLite (75%)      | SQLite (83%)      | LevelDB (113%)     | RocksDB (133%)    | RocksDB (138%)   |
|                  | 10 - 100 KiB    | Flat FS (96%)      | Flat FS (97%)      | SQLite (98%)      | LevelDB (99%)     |                    | Flat FS (96%)     | LevelDB (104%)    | RocksDB (147%)     | RocksDB (156%)    |                  |
|                  | 100 KiB - 1 MiB | Flat FS (100%)     | Flat FS (100%)     | Flat FS (100%)    |                   |                    | LevelDB (115%)    | LevelDB (125%)    | RocksDB (149%)     |                   |                  |

Figure 2.2: Summary of the best storage options on the previous project

the database. Our research touches on this and uses many other embedded databases and the file system. V. N. R. Patchigolla, J. Springer, and K. Lute's [24] briefly approach the optimal solution to storing and retrieving data from mobile devices. We expand this work by focusing more on embedded database management systems and the impact of CRUD operations on multiple open-source SQL embedded database management systems.

### 2.2.3 Comparing Key-Value Databases

Slabinoha et al., in their thesis *entitled Comparative Analysis of Embedded Databases Performance* [25], cover a comparative analysis of embedded databases and their performance on SQLite, LMDB, and UnQLite. Specifically, they compared the interpretation of these systems through their own Python program. This project deals with a single storage type and uses Python software. We extend this research by using many different file systems, types of hard drives, and the file system.

Obradovic, Dujlovic, and Kelec [26] also did a performance analysis on SQLite and went more in-depth on its impact on the Android platform. Our research extends to the Windows subsystem for Linux 2 (WSL 2) on a Windows device and many different hardware platforms, and it also works on compressed and uncompressed data.

The work of Hassan and Sarhan [27] is closest to our work. Their research also looked into four popular open-source relational embedded databases and tested them on their merits in creating, retrieving, updating, and deleting information. The four relational embedded databases they used are H2, HSQLDB, Apache Derby, and SQLite. We further this research using more than one computer and more than one file system in our data.

The research from Konrad Fraczek and Malgorzata Plechawska-Wojcik [28] from the Lublin University of Technology's Institute of Computer Science in Lublin, Poland, compares the performance of relational and non-relational databases in web applications. They measured the speeds of read and write operations. Our research also compares update and delete operations as well.

### 2.2.4 Comparing File Systems

In Sterniczuk's work [29], he compared ext4 and NTFS in Ubuntu with SSD disks. The analysis copies files between two partitions and measures the time of the operation in the

bash language. Our research furthers this by comparing using the Windows Subsystem for Linux 2 (WSL 2), a widely used feature in the Windows operating system. *A Forensic Comparison of NTFS and FAT32 File Systems* by Rubarsky, Lane, and City [30] deals with a performance and a bare-boned research comparison and contrast of the two most popular Windows file systems, New Technology File System (NTFS) and the File Allocation Table (FAT32). We extend this research by comparing different storage devices– HDDs and SSDs.





## Chapter 3

### Methodology

This research work aims to create and use a tool to benchmark the performance of multiple operations and the space efficiency of embedded databases in different scenarios. To this end, we created a script to build an image that downloads the needed packages to run the benchmark, prepares the device, and shows where the script is to run the benchmark. The benchmark script then runs the benchmark and saves the results in a CSV file. Our previous research did this on ext4 [1], while this one approaches this on NTFS. This benchmark was done using WSL 2 and the Ubuntu distribution, as it is the most widely deployed Linux distribution overall [31].

The benchmark algorithm generates random incompressible data with the Mersenne Twister pseudo-random number generator (in C++ standard library). It generates random compressible data from 250 public domain books from *the Gutenberg Project* [32]. It uses pre-downloaded data for the CRUD operations (create, read, update, delete) due to an issue with the former results: the benchmark was evaluated with a different amount of data on every test. The benchmark algorithm also analyzes usage trends (amongst the embedded databases and how they react to the data). Moreover, it measures the performance of the embedded databases (measured through microseconds for the speed and percentage of the

RAM used in the space efficiency).

In the experiments, we used one machine with an Intel Core i5 processor, 8 GB of Crucial RAM DDR3 1600 MHz CL11 Desktop Memory CT102464BD160B with 250 GB of a Western Digital WD2500AAKX SATA HDD. We ran Microsoft Windows Subsystem for Linux 2 (WSL 2) on Windows 11. We used another machine with an Intel Core i5 processor, 8 GB of Crucial RAM DDR3 1600 MHz CL11 Desktop Memory CT102464BD160B, and 250 GB Kingston NV2 M.2 2280 NVMe Internal SSD. We ran Microsoft Windows Subsystem for Linux 2 (WSL 2) on Windows 11. Both machines used the Peripheral Component Interconnect or Conventional PCI bus architecture and the NTFS file system instead of ext4, used in the previous project [1].

### 3.1 Benchmark Implementation

The code in Listing 3.1 performs basic tests to ensure that different store types work correctly. The source code is available online <sup>1</sup>. It creates various kinds of stores (SQLite3Store, LevelDBStore, RocksDBStore, BerkeleyDBStore, FlatFolderStore, NestedFolderStore) and tests basic operations like insert, update, remove, and get on each store. It also tests the handling of null characters in keys and values, deletes if records exist, multiple record insertion, and bulk data insertion into the stores.

---

<sup>1</sup><https://github.com/thaamazingone/MastersThesis>

```
1 #include <filesystem>
2 #include <memory>
3 #include <map>
4 #include <string>
5 #include <functional>
6
7 #define DOCTEST_CONFIG_IMPLEMENT
8 #include "doctest/doctest.h"
9
10 #include "stores.h"
11 #include "utils.h"
12
13 namespace tests {
14     namespace fs = std::filesystem;
15     using fs::path;
16     using namespace std::string_literals;
17     using std::string, std::vector, std::map, std::pair, std::
18         function, std::unique_ptr, std::make_unique;
19     using stores::Store;
20
21     const string filepath = path("out") / "tests" / "store";
22
23     vector<function<unique_ptr<Store>()>> storeFactories{
24         [](){ return make_unique<stores::SQLite3Store>(filepath); },
25
26         [](){ return make_unique<stores::LevelDBStore>(filepath); },
27
28         [](){ return make_unique<stores::RocksDBStore>(filepath); },
29
30         [](){ return make_unique<stores::BerkeleyDBStore>(filepath);
31     },
```

```
27     [](){ return make_unique<stores::FlatFolderStore>(filepath);
    },
28     [](){ return make_unique<stores::NestedFolderStore>(filepath
    , 2, 3, 32); },
29 };
30
31
32 TEST_CASE("Test Stores") {
33     fs::remove_all("out/tests");
34     fs::create_directories("out/tests/");
35
36     SUBCASE("Basic") {
37         for (auto storeFactory : storeFactories) {
38             auto store = storeFactory();
39             string key = utils::randHash(32);
40
41             store->insert(key, "value");
42             REQUIRE(store->get(key) == "value");
43             REQUIRE(store->count() == 1);
44
45             store->update(key, "updated");
46             REQUIRE(store->get(key) == "updated");
47
48             store->remove(key);
49             REQUIRE(store->count() == 0);
50
51             REQUIRE_THROWS(store->get(key));
52         }
53     }
54
55     SUBCASE("Nulls") {
```

```
56         for (auto storeFactory : storeFactories) {
57             auto store = storeFactory();
58
59             string key = utils::randHash(32);
60
61             store->insert(key, "hello\0world"s);
62             REQUIRE(store->get(key) == "hello\0world"s);
63
64             store->update(key, "\0goodbye\0"s);
65             REQUIRE(store->get(key) == "\0goodbye\0"s);
66         }
67     }
68 }
69
70 TEST_CASE("Test deletes if exists") {
71     fs::remove_all("out/tests");
72     fs::create_directories("out/tests/");
73
74     for (auto storeFactory : storeFactories) {
75         string key = utils::randHash(32);
76
77         fs::remove_all(filepath);
78         {
79             auto store = storeFactory();
80             REQUIRE(store->filepath == filepath);
81
82             store->insert(key, "value");
83         }
84         REQUIRE(fs::exists(filepath));
85
86     }
```

```
87         auto store = storeFactory();
88         REQUIRE_THROWS(store->get(key));
89     }
90 }
91 }
92
93 TEST_CASE("Test multiple records") {
94     fs::remove_all("out/tests");
95     fs::create_directories("out/tests/");
96
97     for (auto& storeFactory : storeFactories) {
98         auto store = storeFactory();
99         for (int i = 0; i < 25; i++) {
100             string key = utils::randHash(32);
101             string value = utils::randBlob(64);
102             store->insert(key, value);
103             REQUIRE(store->get(key) == value);
104         }
105     }
106 }
107
108 TEST_CASE("Test bulk insert") {
109     fs::remove_all("out/tests");
110     fs::create_directories("out/tests/");
111
112     for (auto& storeFactory : storeFactories) {
113         auto store = storeFactory();
114         string a = utils::randHash(32), b = utils::randHash(32),
115         c = utils::randHash(32);
116         vector<pair<string, string>> data{ {a, "1"}, {b, "2"}, {
117         c, "3"} };
118     }
```

```

116         store->bulkInsert(data);
117         REQUIRE(store->get(b) == "2");
118         REQUIRE(store->count() == 3);
119     }
120 }
121 }

```

Listing 3.1: Code from “tests.cpp”

Listing 3.1 shows the code used to perform some basic tests. Lines 1-21 show the necessary imports. Lines 22-30 create a factory for different storage systems. Each factory is a lambda function that creates a unique pointer to a specific kind of Store, such as SQLite3Store (line 23), LevelDBStore (line 24), RocksDBStore (line 25), BerkeleyDBStore (line 26), FlatFolderStore (line 27), and NestedFolderStore (line 28). The numbers “2, 3, 32” on line 28 are the parameters to make it a nested file system. The number “2” denotes the amount of hexadecimal characters per level. The number “3” denotes the level of nesting or subordination of a component to another. The number “32” is a hexadecimal representation of the ASCII character “2”. The factories are stored in a vector of functions, and each factory is constructed with the appropriate file path for storage. The code uses the C++ filesystem library for path manipulation and the Doctest testing framework for testing.

Line 32 initializes the test case, and line 33 removes all existing “out/tests” directory. Line 34 creates a new “out/tests” directory. Inside the test suite, there are two subcases. One is “Basic” (line 36), which shows for each store factory in the storeFactories collection, it creates a store instance (line 38), inserts a key-value pair (lines 41-42), checks if the value can be retrieved correctly (line 42), checks the count of items in the store (line 43), updates the value of the key (lines 45-46), removes the key (lines 48-49), and verifies that the key does not exist anymore (line 51). The other subcase is “Nulls” (line 55), which



shows for each store factory in the `storeFactories` collection, it creates a store instance (line 57), inserts a key-value pair (line 61), checks if the value can be retrieved correctly (line 62), and updates the value of the key (lines 64-65).

The next test case is named “Test deletes if it exists” (line 70). First, it removes all files (line 71) and directories inside the `"out/tests"` directory. Then, it creates the `"out/tests"` directory if it does not exist (line 72). It iterates over a list of store factories and performs the following actions for each factory. First, it generates a random key, then it removes any existing file at a specific `filepath` (line 77), creates a store using the current factory (lines 79-80), inserts a key-value pair into the store (line 82), and then closes the store. It then checks if the file exists at the specified `filepath` (line 84), and finally tries to retrieve the value associated with the key from the store (line 87) and expects an exception to be thrown (line 88).

The next test involves inserting multiple records into different data stores. It removes any existing test data (line 94), creates a directory for test output (line 95), and then iterates over a list of store factories (line 97). For each store factory, it creates a store instance and inserts 25 random records into it (lines 98-102). It then tests that each inserted record can be retrieved correctly by verifying that the retrieved value matches the inserted value using the `REQUIRE` macro (line 103). The `REQUIRE` macro ensures the pointer value is correct for the benchmark to work.

The second is a unit test for a data store implementation’s “bulk insert” functionality. It removes any existing test data directory (line 109) and creates a new one (line 110). It then iterates over different store factories, creates a new store instance using each factory (line 113), generates random strings for keys (line 114), creates a vector of key-value pairs (line 115), calls the `bulkInsert` method of the store with this data (line 116), and finally asserts that the store retrieves the correct value for a specific key and that the total count of items in the store is 3 (lines 117-118). The `bulkInsert` method is the way that this code inserts

large amounts of code into a database.

## 3.2 Implementation of Utility Functions

The code also provides various utility functions, as shown in Listing 3.2, for calculating disk usage, measuring time, obtaining peak memory usage, and formatting file sizes in a human-readable format. Specifically, it includes functions for generating random blobs of data, generating random hashes, generating keys based on SHA1 hash, measuring the execution time of a function, calculating disk usage of a file or directory, obtaining peak memory usage of the process, and formatting file sizes in a human-readable format.

```
1 #include <string>
2 #include <random>
3 #include <functional>
4 #include <chrono>
5 #include <iomanip>
6 #include <filesystem>
7 #include <vector>
8 #include <cmath>
9 #include <algorithm>
10 #include <fstream>
11 #include <sys/resource.h>
12
13 #include <boost/process.hpp>
14 #include <boost/uuid/detail/sha1.hpp>
15
16 #include "utils.h"
17
18 namespace utils {
19     namespace fs = std::filesystem;
```

```

20     using fs::path;
21     using std::string, std::vector, std::ofstream, std::ifstream;
22     namespace chrono = std::chrono;
23     namespace process = boost::process;
24     using boost::uuids::detail::sha1;
25
26     std::random_device randomDevice;
27     std::mt19937 randGen(randomDevice());
28
29     string randBlob(size_t size) {
30         std::uniform_int_distribution<unsigned char> randChar(0, 0
xFF);
31         string blob;
32         blob.resize(size);
33         std::generate(blob.begin(), blob.end(), [&]() { return
randChar(randGen); });
34         return blob;
35     }
36
37     string randBlob(Range<size_t> size) {
38         return randBlob(randInt(size.min, size.max));
39     }
40
41     ClobGenerator::ClobGenerator(const path& textFolder) :
textFolder(textFolder), filesTotalSize(0) {
42
43         for (auto file : fs::directory_iterator(this->textFolder)) {
44             if (file.path().extension() == ".txt") {
45                 this->fileSizes.push_back({file.path(), file.
file_size()});

```

```

46         filesTotalSize += file.file_size();
47     }
48 }
49 }
50
51 string ClobGenerator::operator()(size_t size) {
52     size_t start = randInt<size_t>(0, this->filesTotalSize -
size);
53     size_t end = start + size;
54
55     size_t pos = 0;
56     auto fileInfo = this->fileSizes.begin();
57     for (; pos + fileInfo->size < start; fileInfo++) {
58         pos += fileInfo->size;
59     }
60
61     string clob(size, '\0');
62     while (pos < end) {
63         ifstream file(fileInfo->file, ifstream::in ifstream::
binary);
64         if (pos < start) {
65             file.seekg(start - pos);
66             pos = start;
67         }
68
69         auto bufferPos = pos - start;
70         auto remainingFile = fileInfo->size - file.tellg();
71         file.read(&clob[bufferPos], std::min(remainingFile, size
- bufferPos)); // read up to EOF or end of buffer
72
73         pos += remainingFile;

```

```
74         fileInfo++;
75     }
76
77     return clob;
78 }
79
80 string ClobGenerator::operator()(Range<size_t> size){
81     return (*this)(randInt(size.min, size.max));
82 }
83
84 string intToHex(long long i, int width) {
85     std::stringstream stream;
86     stream << std::setfill('0') << std::setw(width) << std::hex
<< i;
87     return stream.str();
88 }
89
90 string randHash(int size) {
91     std::uniform_int_distribution<unsigned char> randNibble(0x0,
0xF);
92     string hash;
93     hash.resize(size);
94     for (int i = 0; i < size; i += 1)
95         hash[i] = intToHex(randNibble(randGen), 1)[0];
96     return hash;
97 }
98
99 string genKey(size_t i) {
100     sha1 hash;
101     hash.process_bytes(reinterpret_cast<void*>(&i), sizeof(i));
```

```

102     hash.process_byte(136);
103     sha1::digest_type digest;
104     hash.get_digest(digest);
105
106     std::stringstream ss;
107     for (auto part : digest)
108         ss << std::setfill('0') << std::setw(sizeof(part) * 2)
<< std::hex << part;
109     return ss.str().substr(0, 32);
110 }
111
112
113     chrono::nanoseconds timeIt(std::function<void()> func) {
114         auto start = chrono::steady_clock::now();
115         func();
116         auto stop = chrono::steady_clock::now();
117         return chrono::duration_cast<chrono::nanoseconds>(stop -
start);
118     }
119
120
121     long long diskUsage(const path& filepath) {
122         process::ipstream out;
123         process::child du(process::search_path("du"), "-s", "--block
-size=1", filepath.native(), process::std_out > out);
124
125         string outputStr;
126         string line;
127         while (out && std::getline(out, line) && !line.empty())
128             outputStr += line + "\n";

```

```
129     du.wait();
130
131     return std::stol(outputStr);
132 }
133
134 size_t getPeakMemUsage() {
135     struct rusage info;
136     getrusage(RUSAGE_SELF, &info);
137     return info.ru_maxrss;
138 }
139
140 void resetPeakMemUsage() {
141     ofstream clearRefs("/proc/self/clear_refs");
142     clearRefs << "5";
143 }
144
145
146 string prettySize(size_t size) {
147     vector<string> units{"B", "KiB", "MiB", "GiB"};
148     int unitI = std::min<size_t>(std::log(size) / std::log(1024)
, units.size());
149     int unitSize = std::pow(1024, unitI);
150
151     double sizeInUnit = size / ((double) unitSize);
152     int leftOfDecimal = std::log(sizeInUnit) / std::log(10);
153
154     std::stringstream ss;
155     ss << std::setprecision(leftOfDecimal + 2) << sizeInUnit <<
units[unitI];
156     return ss.str();
157 }
```

## Listing 3.2: Code from “utils.cpp”

Listing 3.2 refers to the necessary imports (lines 1-25) and initializes a Mersenne Twister pseudo-random number generator [33] with a seed from a random device (lines 26-27).

Next, the code defines a function `randBlob` (line 29) that generates a random blob of binary data of a given size (lines 30-33). It uses a uniform distribution (lines 30-32) to generate random characters and stores them in a string (line 33). Additionally, an overloaded version of `randBlob` (lines 37-39) takes a range of sizes (line 37) and generates a random blob within that range (line 38). The `ClobGenerator` class constructor initializes the object by caching the file list of text files in a specified folder, storing their paths and sizes in a vector (lines 41-49).

The code then defines an overloaded function call operator for the `ClobGenerator` class. The function takes a `size_t` parameter `size` and generates a string of the specified size by reading content from multiple files (lines 51-61). It determines a random start position within the total size of the files (line 62) and then reads content from the files to construct the string (lines 62-67). It reads the content of the files in chunks, starting from the calculated start position and stopping when the specified size is reached (lines 69-78).

Then, the function call operator is overloaded for the `ClobGenerator` class. It takes a `Range<size_t>` object `size` as input, representing a range of sizes (line 80). It then generates a random size within this range using `randInt(size.min, size.max)` and calls the function call operator again with this randomly generated size to generate a string (line 81). Finally, it returns the generated string (line 81).

The next portion of the code takes a long long integer `i` and converts it to a hexadecimal string with a specified width. The function `intToHex` uses a `stringstream`, or a stream class to operate on strings, to convert the integer to hexadecimal format (line 85) and pads the



result with zeros to match the specified width (line 86). Finally, it returns the hexadecimal string (line 87). The final portion of this code segment generates a random hash of a specified size (line 90) by creating random nibbles (4-bit hexadecimal numbers) and converting them to hexadecimal characters. The `randHash` function takes an integer size as input (line 93) and returns a string representing the random hash of the specified size (line 96).

The next portion has the `genKey` string, which generates a SHA-1 hash (lines 100-101) from the input `size_t i` concatenated with an arbitrary salt value of 136 (line 102). It then converts the hash digest into a string representation (lines 103-106) and returns the first 32 characters of the resulting string (line 109). The next function measures the time taken for a given function `func` to execute by capturing the start and stop time using the `chrono` library in C++ (line 113). The function `timeIt` takes a `std::function<void()>` parameter `func`, representing the function to be timed. It then executes the function `func`, captures the start (line 114) and stop (line 116) time, calculates the duration between them, and returns the duration in nanoseconds (line 117). The `diskUsage` function calculates the disk usage of a specified file or directory by invoking the “`du`” command, which returns the disk usage, with specific arguments and capturing its output (line 123). The output is then processed to extract the total disk usage value (lines 127-129), which is converted to a long long integer and returned (line 131). The next function, `getPeakMemUsage`, returns the peak memory usage of the current process in kilobytes (line 137). It achieves this by using the `getrusage` function with the `RUSAGE_SELF` parameter to get resource usage statistics for the current process (line 136) and then returning the `ru_maxrss` field from the `struct rusage`, which represents the maximum resident set size in kilobytes (line 137). This code opens the corresponding file (line 141) and writes the number “5” to it (line 142), which clears the memory references. The final function, `prettySize`, takes a size in bytes as input (lines 146-147) and converts it to a human-readable format with units like B, KiB, MiB, or GiB, depending on the size (lines 148-152). It calculates the appropriate unit (e.g., KiB for sizes

between 1024 and 1048575 bytes) and formats the size accordingly (lines 154-158).

### 3.3 Taking Measurements

This study compares four embedded key-value databases: SQLite, LevelDB, RocksDB, and Berkeley DB. It treats SQLite as a key-value database with a primary key column and a value column. The databases use two storage strategies: flat and nested. Records in flat storage are kept in one directory, while nested storage uses a hierarchical file structure. Each level of the nested storage strategy has two hexadecimal characters, and we opted for three classes because the depth of 3 was chosen to hold up to 16,777,216 records while keeping around 256 leaf nodes. SQLite and Berkeley DB C interfaces return invalid char\* pointers after database operations, specifically wrappers copy buffer memory, adding overhead. This could be avoided if developers only need data briefly, but most common use cases involve copying buffer memory to avoid hanging pointers and memory safety issues. Our study measures the time taken for fundamental key-value operations, creating, reading, updating, and deleting, recording average, minimum, and maximum values in nanoseconds. Peak memory usage was also measured for each combination. Space efficiency was also measured by comparing the amount of data put in versus the actual space taken up on disk. Embedded databases can add storage overhead or reduce size, while file systems can waste space.

### 3.4 Monitoring Performance

This study measured the time taken for fundamental key-value operations: create, read, update, and delete, running each operation 1,000 times and recording average, minimum, and maximum values in microseconds. The file system, Berkeley DB, RocksDB, LevelDB, SQLite3, on record sizes that are less than a KiB, between 1 and 10 KiB, between 10 and

100 KiB, and between 100 KiB and 1 MiB, and also 100 records, 1000 records, 10000 records, 100000 records, and 1000000 records. This process was also done with a Google Chrome tab watching a YouTube video running in the background to determine which is fastest. After this process is finished. The code is then exported to a CSV file and moved to a table.

# Chapter 4

## Results

Experiments for the benchmark were executed on both a hard drive and a solid-state drive. The results shown in this chapter are divided based on whether the YouTube video was running on Google Chrome when the benchmark was being used. In case the YouTube video was operating in the background while the benchmark was being run, it was labeled as “disturbed”; otherwise, it was labeled as “undisturbed”.

### 4.1 SSD Undisturbed Results

Figure 4.1 refers to the undisturbed SSD results of the lowest average of 1,000 averages of each operation with either 100, 1,000, 10,000, 100,000, or 1,000,000 records. The results show that Berkeley DB remained among the fastest for the insert and update operations under compressible and incompressible data. As the number of records increased and the record sizes increased, both the folder stores (Nested and Flat Folder stores) were quicker. LevelDB was better for smaller record sizes for the get operations, and RocksDB and the folder stores were better for larger record sizes. The 10,000 record results for the larger sizes favor the folder stores, especially the remove operation. SQLite was the most efficient

for smaller record amounts, and RocksDB was the most efficient for larger record amounts.

| Operation        | Record Size  | Incompressible       |                       |                       |                      |                      | Compressible          |                       |                       |                        |                      |
|------------------|--------------|----------------------|-----------------------|-----------------------|----------------------|----------------------|-----------------------|-----------------------|-----------------------|------------------------|----------------------|
|                  |              | 100                  | 1,000                 | 10,000                | 100,000              | 1,000,000            | 100                   | 1,000                 | 10,000                | 100,000                | 1,000,000            |
| insert           | <1 KIB       | Berkeley DB (2 ns)   | Berkeley DB (5 ns)    | LevelDB (6 ns)        | Berkeley DB (8 ns)   | Berkeley DB (9 ns)   | Berkeley DB (4 ns)    | Berkeley DB (9 ns)    | Berkeley DB (8 ns)    | Berkeley DB (8 ns)     | Berkeley DB (10 ns)  |
|                  | 1-10KIB      | Berkeley DB (8 ns)   | Berkeley DB (9 ns)    | Berkeley DB (10 ns)   | Berkeley DB (14 ns)  | NestedFolder (28 ns) | Berkeley DB (9 ns)    | Berkeley DB (12 ns)   | Berkeley DB (14 ns)   | Berkeley DB (12 ns)    | NestedFolder (26 ns) |
|                  | 10-100KIB    | Berkeley DB (59 ns)  | Berkeley DB (59 ns)   | FlatFolder (685 ns)   | Berkeley DB (916 ns) |                      | Berkeley DB (51 ns)   | Berkeley DB (46 ns)   | FlatFolder (60 ns)    | NestedFolder (1333 ns) |                      |
|                  | 100KIB-1 MIB | Berkeley DB (590 ns) | Berkeley DB (819 ns)  | Berkeley DB (4023 ns) |                      |                      | FlatFolder (364 ns)   | Berkeley DB (482 ns)  | NestedFolder (998 ns) |                        |                      |
| update           | <1 KIB       | Berkeley DB (1 ns)   | Berkeley DB (3 ns)    | Berkeley DB (5 ns)    | Berkeley DB (6 ns)   | Berkeley DB (107 ns) | Berkeley DB (3 ns)    | Berkeley DB (7 ns)    | Berkeley DB (9 ns)    | Berkeley DB (7 ns)     | Berkeley DB (10 ns)  |
|                  | 1-10KIB      | Berkeley DB (8 ns)   | Berkeley DB (8 ns)    | Berkeley DB (11 ns)   | Berkeley DB (10 ns)  | RocksDB (537 ns)     | Berkeley DB (31 ns)   | Berkeley DB (8 ns)    | Berkeley DB (10 ns)   | Berkeley DB (12 ns)    | LevelDB (21 ns)      |
|                  | 10-100KIB    | Berkeley DB (61 ns)  | Berkeley DB (68 ns)   | Berkeley DB (95 ns)   | RocksDB (259 ns)     |                      | Berkeley DB (53 ns)   | Berkeley DB (60 ns)   | Berkeley DB (58 ns)   | RocksDB (220 ns)       |                      |
|                  | 100KIB-1 MIB | Berkeley DB (696 ns) | Berkeley DB (752 ns)  | Berkeley DB (808 ns)  |                      |                      | Berkeley DB (600 ns)  | Berkeley DB (3020 ns) | Berkeley DB (4184 ns) |                        |                      |
| get              | <1 KIB       | Berkeley DB (1 ns)   | LevelDB (2 ns)        | Berkeley DB (2 ns)    | Berkeley DB (4 ns)   | Berkeley DB (5 ns)   | Berkeley DB (1 ns)    | LevelDB (2 ns)        | Berkeley DB (4 ns)    | Berkeley DB (2 ns)     | Berkeley DB (4 ns)   |
|                  | 1-10KIB      | LevelDB (2 ns)       | Berkeley DB (4 ns)    | Berkeley DB (4 ns)    | Berkeley DB (6 ns)   | LevelDB (130 ns)     | LevelDB (1 ns)        | Berkeley DB (4 ns)    | Berkeley DB (4 ns)    | Berkeley DB (5 ns)     | RocksDB (160 ns)     |
|                  | 10-100KIB    | RocksDB (7 ns)       | RocksDB (12 ns)       | FlatFolder (16 ns)    | Berkeley DB (51 ns)  |                      | RocksDB (9 ns)        | RocksDB (13 ns)       | NestedFolder (14 ns)  | Berkeley DB (30 ns)    |                      |
|                  | 100KIB-1 MIB | LevelDB (81 ns)      | LevelDB (91 ns)       | NestedFolder (134 ns) |                      |                      | FlatFolder (88 ns)    | FlatFolder (85 ns)    | NestedFolder (129 ns) |                        |                      |
| remove           | <1 KIB       | Berkeley DB (2 ns)   | Berkeley DB (4 ns)    | Berkeley DB (5 ns)    | Berkeley DB (6 ns)   | Berkeley DB (6 ns)   | Berkeley DB (2 ns)    | Berkeley DB (6 ns)    | Berkeley DB (8 ns)    | Berkeley DB (7 ns)     | Berkeley DB (9 ns)   |
|                  | 1-10KIB      | Berkeley DB (6 ns)   | Berkeley DB (9 ns)    | Berkeley DB (11 ns)   | Berkeley DB (10 ns)  | RocksDB (27 ns)      | Berkeley DB (5 ns)    | Berkeley DB (8 ns)    | Berkeley DB (8 ns)    | Berkeley DB (10 ns)    | FlatFolder (30 ns)   |
|                  | 10-100KIB    | RocksDB (31 ns)      | NestedFolder (118 ns) | FlatFolder (36 ns)    | NestedFolder (47 ns) |                      | FlatFolder (30 ns)    | Berkeley DB (42 ns)   | NestedFolder (29 ns)  | Berkeley DB (117 ns)   |                      |
|                  | 100KIB-1 MIB | FlatFolder (113 ns)  | FlatFolder (33 ns)    | FlatFolder (149 ns)   |                      |                      | NestedFolder (120 ns) | FlatFolder (414 ns)   | NestedFolder (687 ns) |                        |                      |
| space efficiency | <1 KIB       | SQLite3 (100%)       | SQLite3 (56%)         | RocksDB (77%)         | RocksDB (91%)        | RocksDB (92%)        | SQLite3 (50%)         | SQLite3 (56%)         | LevelDB (85%)         | RocksDB (91%)          | RocksDB (113%)       |
|                  | 1-10KIB      | FlatFolder (96%)     | SQLite3 (83%)         | SQLite3 (86%)         | RocksDB (97%)        | RocksDB (99%)        | SQLite3 (79%)         | SQLite3 (83%)         | SQLite3 (86%)         | RocksDB (135%)         | RocksDB (141%)       |
|                  | 10-100KIB    | SQLite3 (49%)        | SQLite3 (97%)         | SQLite3 (98%)         | SQLite3 (98%)        |                      | SQLite3 (97%)         | LevelDB (106%)        | LevelDB (139%)        | RocksDB (159%)         |                      |
|                  | 100KIB-1 MIB | FlatFolder (73%)     | FlatFolder (100%)     | SQLite3 (100%)        |                      |                      | FlatFolder (100%)     | RocksDB (115%)        | LevelDB (152%)        |                        |                      |

|   |
|---|
| Legend  |
| <span style="display: inline-block; width: 15px; height: 10px; background-color: #00b0f0; margin-right: 5px;"></span> FlatFolder <span style="display: inline-block; width: 15px; height: 10px; background-color: #008000; margin-left: 10px; margin-right: 5px;"></span> SQLite3 <span style="display: inline-block; width: 15px; height: 10px; background-color: #cccccc; margin-left: 10px; margin-right: 5px;"></span> RocksDB <span style="display: inline-block; width: 15px; height: 10px; background-color: #ffff00; margin-left: 10px; margin-right: 5px;"></span> Berkeley DB <span style="display: inline-block; width: 15px; height: 10px; background-color: #800000; margin-left: 10px; margin-right: 5px;"></span> LevelDB <span style="display: inline-block; width: 15px; height: 10px; background-color: #000080; margin-left: 10px;"></span> NestedFolder |

Figure 4.1: The results of the undisturbed SSD

## 4.2 HDD Undisturbed Results

Figure 4.2 refers to the undisturbed HDD results of the lowest average of 1,000 averages of each operation with either 100, 1,000, 10,000, 100,000, or 1,000,000 records. The flat folder was better on some record amounts in the insert operation on the hard disk drive than on the solid-state drive. The flat folder was more efficient in the small record counts, and RocksDB was more efficient in the larger record counts.

| Operation        | Record Size   | 100                 | 1,000                 | 10,000                | 100,000              | 1,000,000            | 100                   | 1,000                 | 10,000                | 100,000              | 1,000,000           |
|------------------|---------------|---------------------|-----------------------|-----------------------|----------------------|----------------------|-----------------------|-----------------------|-----------------------|----------------------|---------------------|
| insert           | <1 KIB        | Berkeley DB (2 ns)  | Berkeley DB (5 ns)    | Berkeley DB (5 ns)    | Berkeley DB (8 ns)   | Berkeley DB (9 ns)   | Berkeley DB (6 ns)    | Berkeley DB (9 ns)    | Berkeley DB (8 ns)    | Berkeley DB (8 ns)   | Berkeley DB (10 ns) |
|                  | 1- 10KIB      | Berkeley DB (8 ns)  | Berkeley DB (8 ns)    | Berkeley DB (10 ns)   | FlatFolder (11 ns)   | Berkeley DB (12 ns)  | Berkeley DB (9 ns)    | Berkeley DB (12 ns)   | Berkeley DB (14 ns)   | Berkeley DB (12 ns)  | Berkeley DB (16 ns) |
|                  | 10- 100KIB    | FlatFolder (45 ns)  | FlatFolder (38 ns)    | FlatFolder (685 ns)   | Berkeley DB (916 ns) |                      | FlatFolder (38 ns)    | Berkeley DB (46 ns)   | FlatFolder (60 ns)    | Berkeley DB (96 ns)  |                     |
|                  | 100KIB- 1 MIB | FlatFolder (285 ns) | FlatFolder (965 ns)   | FlatFolder (302 ns)   |                      |                      | FlatFolder (364 ns)   | FlatFolder (288 ns)   | NestedFolder (998 ns) |                      |                     |
|                  |               |                     |                       |                       |                      |                      |                       |                       |                       |                      |                     |
| update           | <1 KIB        | Berkeley DB (1 ns)  | Berkeley DB (3 ns)    | Berkeley DB (5 ns)    | Berkeley DB (6 ns)   | Berkeley DB (107 ns) | Berkeley DB (3 ns)    | Berkeley DB (7 ns)    | Berkeley DB (9 ns)    | Berkeley DB (7 ns)   | Berkeley DB (10 ns) |
|                  | 1- 10KIB      | Berkeley DB (8 ns)  | Berkeley DB (8 ns)    | Berkeley DB (15 ns)   | LevelDB (27 ns)      | RocksDB (537 ns)     | Berkeley DB (31 ns)   | Berkeley DB (8 ns)    | Berkeley DB (10 ns)   | LevelDB (17 ns)      | LevelDB (21 ns)     |
|                  | 10- 100KIB    | Berkeley DB (61 ns) | Berkeley DB (68 ns)   | Berkeley DB (95 ns)   | RocksDB (259 ns)     |                      | Berkeley DB (53 ns)   | Berkeley DB (60 ns)   | Berkeley DB (60 ns)   | Berkeley DB (78 ns)  |                     |
|                  | 100KIB- 1 MIB | FlatFolder (162 ns) | NestedFolder (121 ns) | Berkeley DB (808 ns)  |                      |                      | Berkeley DB (600 ns)  | Berkeley DB (1423 ns) | Berkeley DB (3154 ns) |                      |                     |
|                  |               |                     |                       |                       |                      |                      |                       |                       |                       |                      |                     |
| get              | <1 KIB        | Berkeley DB (1 ns)  | LevelDB (2 ns)        | Berkeley DB (2 ns)    | Berkeley DB (4 ns)   | Berkeley DB (5 ns)   | Berkeley DB (1 ns)    | LevelDB (2 ns)        | Berkeley DB (4 ns)    | Berkeley DB (4 ns)   | Berkeley DB (4 ns)  |
|                  | 1- 10KIB      | LevelDB (2 ns)      | Berkeley DB (4 ns)    | Berkeley DB (4 ns)    | Berkeley DB (6 ns)   | LevelDB (130 ns)     | LevelDB (1 ns)        | Berkeley DB (4 ns)    | Berkeley DB (4 ns)    | FlatFolder (8 ns)    | RocksDB (160 ns)    |
|                  | 10- 100KIB    | RocksDB (7 ns)      | RocksDB (12 ns)       | FlatFolder (16 ns)    | Berkeley DB (51 ns)  |                      | RocksDB (9 ns)        | RocksDB (13 ns)       | NestedFolder (14 ns)  | FlatFolder (16 ns)   |                     |
|                  | 100KIB- 1 MIB | LevelDB (61 ns)     | LevelDB (91 ns)       | NestedFolder (134 ns) |                      |                      | FlatFolder (88 ns)    | FlatFolder (85 ns)    | NestedFolder (129 ns) |                      |                     |
|                  |               |                     |                       |                       |                      |                      |                       |                       |                       |                      |                     |
| remove           | <1 KIB        | Berkeley DB (2 ns)  | Berkeley DB (4 ns)    | Berkeley DB (5 ns)    | Berkeley DB (6 ns)   | Berkeley DB (6 ns)   | Berkeley DB (2 ns)    | Berkeley DB (6 ns)    | Berkeley DB (8 ns)    | Berkeley DB (7 ns)   | Berkeley DB (9 ns)  |
|                  | 1- 10KIB      | Berkeley DB (6 ns)  | Berkeley DB (9 ns)    | Berkeley DB (11 ns)   | Berkeley DB (10 ns)  | RocksDB (27 ns)      | Berkeley DB (5 ns)    | Berkeley DB (8 ns)    | Berkeley DB (8 ns)    | FlatFolder (22 ns)   | FlatFolder (30 ns)  |
|                  | 10- 100KIB    | RocksDB (31 ns)     | NestedFolder (118 ns) | FlatFolder (36 ns)    | NestedFolder (47 ns) |                      | FlatFolder (30 ns)    | Berkeley DB (42 ns)   | NestedFolder (29 ns)  | NestedFolder (32 ns) |                     |
|                  | 100KIB- 1 MIB | FlatFolder (113 ns) | FlatFolder (33 ns)    | FlatFolder (149 ns)   |                      |                      | NestedFolder (120 ns) | FlatFolder (414 ns)   | NestedFolder (687 ns) |                      |                     |
|                  |               |                     |                       |                       |                      |                      |                       |                       |                       |                      |                     |
| space efficiency | <1 KIB        | SQLite3 (50%)       | SQLite3 (53%)         | LevelDB (77%)         | RocksDB (91%)        | RocksDB (92%)        | SQLite3 (50%)         | SQLite3 (56%)         | LevelDB (85%)         | RocksDB (91%)        | RocksDB (113%)      |
|                  | 1- 10KIB      | FlatFolder (96%)    | SQLite3 (83%)         | SQLite3 (86%)         | LevelDB (97%)        | RocksDB (99%)        | SQLite3 (79%)         | SQLite3 (83%)         | SQLite3 (86%)         | RocksDB (135%)       | RocksDB (141%)      |
|                  | 10- 100KIB    | FlatFolder (97%)    | SQLite3 (97%)         | SQLite3 (98%)         | LevelDB (99%)        |                      | FlatFolder (98%)      | LevelDB (106%)        | RocksDB (147%)        | RocksDB (159%)       |                     |
|                  | 100KIB- 1 MIB | FlatFolder (100%)   | FlatFolder (100%)     | FlatFolder (100%)     |                      |                      | LevelDB (112%)        | LevelDB (125%)        | RocksDB (152%)        |                      |                     |
|                  |               |                     |                       |                       |                      |                      |                       |                       |                       |                      |                     |

|        |            |         |         |             |         |              |
|--------|------------|---------|---------|-------------|---------|--------------|
| Legend | FlatFolder | SQLite3 | RocksDB | Berkeley DB | LevelDB | NestedFolder |
|--------|------------|---------|---------|-------------|---------|--------------|

Figure 4.2: The results of the undisturbed HDD

### 4.3 SSD Disturbed Results

Figure 4.3 refers to the results while a YouTube video is continuously running in the background on Google Chrome. While many of the leaders in speed for the operations were the same, SQLite3 became more efficient in the disturbed results.

| Operation        | Record Size   | Incompressible       |                       |                       |                      |                      | Compressible          |                       |                       |                        |                      |
|------------------|---------------|----------------------|-----------------------|-----------------------|----------------------|----------------------|-----------------------|-----------------------|-----------------------|------------------------|----------------------|
|                  |               | 100                  | 1,000                 | 10,000                | 100,000              | 1,000,000            | 100                   | 1,000                 | 10,000                | 100,000                | 1,000,000            |
| insert           | <1 KiB        | Berkeley DB (2 ns)   | Berkeley DB (5 ns)    | Berkeley DB (4 ns)    | Berkeley DB (8 ns)   | Berkeley DB (9 ns)   | Berkeley DB (4 ns)    | Berkeley DB (9 ns)    | Berkeley DB (8 ns)    | Berkeley DB (8 ns)     | Berkeley DB (10 ns)  |
|                  | 1- 10KiB      | Berkeley DB (8 ns)   | Berkeley DB (9 ns)    | Berkeley DB (10 ns)   | Berkeley DB (14 ns)  | NestedFolder (28 ns) | Berkeley DB (9 ns)    | Berkeley DB (12 ns)   | Berkeley DB (14 ns)   | Berkeley DB (12 ns)    | NestedFolder (26 ns) |
|                  | 10- 100KiB    | Berkeley DB (59 ns)  | Berkeley DB (59 ns)   | FlatFolder (685 ns)   | Berkeley DB (916 ns) |                      | Berkeley DB (51 ns)   | Berkeley DB (46 ns)   | FlatFolder (60 ns)    | NestedFolder (1333 ns) |                      |
|                  | 100KiB- 1 MiB | Berkeley DB (590 ns) | Berkeley DB (819 ns)  | Berkeley DB (4023 ns) |                      |                      | FlatFolder (364 ns)   | Berkeley DB (482 ns)  | NestedFolder (998 ns) |                        |                      |
| update           | <1 KiB        | Berkeley DB (1 ns)   | Berkeley DB (3 ns)    | Berkeley DB (5 ns)    | Berkeley DB (6 ns)   | Berkeley DB (107 ns) | Berkeley DB (3 ns)    | Berkeley DB (7 ns)    | Berkeley DB (9 ns)    | Berkeley DB (7 ns)     | Berkeley DB (10 ns)  |
|                  | 1- 10KiB      | Berkeley DB (8 ns)   | Berkeley DB (8 ns)    | Berkeley DB (11 ns)   | Berkeley DB (10 ns)  | RocksDB (537 ns)     | Berkeley DB (31 ns)   | Berkeley DB (8 ns)    | Berkeley DB (10 ns)   | Berkeley DB (12 ns)    | LevelDB (21 ns)      |
|                  | 10- 100KiB    | Berkeley DB (61 ns)  | Berkeley DB (68 ns)   | Berkeley DB (95 ns)   | RocksDB (259 ns)     |                      | Berkeley DB (53 ns)   | Berkeley DB (60 ns)   | Berkeley DB (58 ns)   | RocksDB (220 ns)       |                      |
|                  | 100KiB- 1 MiB | Berkeley DB (696 ns) | Berkeley DB (752 ns)  | Berkeley DB (808 ns)  |                      |                      | Berkeley DB (600 ns)  | Berkeley DB (3020 ns) | Berkeley DB (4184 ns) |                        |                      |
| get              | <1 KiB        | Berkeley DB (1 ns)   | LevelDB (2 ns)        | Berkeley DB (2 ns)    | Berkeley DB (4 ns)   | Berkeley DB (5 ns)   | Berkeley DB (1 ns)    | LevelDB (2 ns)        | Berkeley DB (4 ns)    | Berkeley DB (2 ns)     | Berkeley DB (4 ns)   |
|                  | 1- 10KiB      | LevelDB (2 ns)       | Berkeley DB (4 ns)    | Berkeley DB (4 ns)    | Berkeley DB (6 ns)   | LevelDB (130 ns)     | LevelDB (1 ns)        | Berkeley DB (4 ns)    | Berkeley DB (4 ns)    | Berkeley DB (5 ns)     | RocksDB (160 ns)     |
|                  | 10- 100KiB    | RocksDB (7 ns)       | RocksDB (12 ns)       | FlatFolder (16 ns)    | Berkeley DB (51 ns)  |                      | RocksDB (9 ns)        | RocksDB (13 ns)       | NestedFolder (14 ns)  | Berkeley DB (30 ns)    |                      |
|                  | 100KiB- 1 MiB | LevelDB (81 ns)      | LevelDB (91 ns)       | NestedFolder (134 ns) |                      |                      | FlatFolder (88 ns)    | FlatFolder (85 ns)    | NestedFolder (129 ns) |                        |                      |
| remove           | <1 KiB        | Berkeley DB (2 ns)   | Berkeley DB (4 ns)    | Berkeley DB (5 ns)    | Berkeley DB (6 ns)   | Berkeley DB (6 ns)   | Berkeley DB (2 ns)    | Berkeley DB (6 ns)    | Berkeley DB (8 ns)    | Berkeley DB (7 ns)     | Berkeley DB (9 ns)   |
|                  | 1- 10KiB      | Berkeley DB (6 ns)   | Berkeley DB (9 ns)    | Berkeley DB (11 ns)   | Berkeley DB (10 ns)  | RocksDB (27 ns)      | Berkeley DB (5 ns)    | Berkeley DB (8 ns)    | Berkeley DB (8 ns)    | Berkeley DB (10 ns)    | FlatFolder (30 ns)   |
|                  | 10- 100KiB    | RocksDB (31 ns)      | NestedFolder (118 ns) | FlatFolder (36 ns)    | NestedFolder (47 ns) |                      | FlatFolder (30 ns)    | Berkeley DB (42 ns)   | NestedFolder (29 ns)  | Berkeley DB (117 ns)   |                      |
|                  | 100KiB- 1 MiB | FlatFolder (113 ns)  | FlatFolder (33 ns)    | FlatFolder (149 ns)   |                      |                      | NestedFolder (120 ns) | FlatFolder (414 ns)   | NestedFolder (687 ns) |                        |                      |
| space efficiency | <1 KiB        | SQLite3 (100%)       | SQLite3 (56%)         | RocksDB (77%)         | RocksDB (91%)        | RocksDB (92%)        | SQLite3 (50%)         | SQLite3 (56%)         | SQLite3 (65%)         | RocksDB (91%)          | RocksDB (113%)       |
|                  | 1- 10KiB      | SQLite3 (98%)        | SQLite3 (83%)         | SQLite3 (86%)         | RocksDB (97%)        | RocksDB (99%)        | SQLite3 (79%)         | SQLite3 (83%)         | SQLite3 (86%)         | RocksDB (135%)         | RocksDB (141%)       |
|                  | 10- 100KiB    | SQLite3 (49%)        | SQLite3 (97%)         | SQLite3 (98%)         | SQLite3 (98%)        |                      | SQLite3 (97%)         | LevelDB (106%)        | LevelDB (139%)        | RocksDB (159%)         |                      |
|                  | 100KiB- 1 MiB | FlatFolder (73%)     | FlatFolder (100%)     | SQLite3 (100%)        |                      |                      | FlatFolder (100%)     | RocksDB (115%)        | LevelDB (152%)        |                        |                      |

|        |            |         |         |             |         |              |
|--------|------------|---------|---------|-------------|---------|--------------|
| Legend | FlatFolder | SQLite3 | RocksDB | Berkeley DB | LevelDB | NestedFolder |
|--------|------------|---------|---------|-------------|---------|--------------|

Figure 4.3: The results of the disturbed SSD

## 4.4 HDD Disturbed Results

Figure 4.4 refers to the results while separate programs ran in the background. While many of the leaders in speed for the operations were the same, SQLite3 became more efficient in the disturbed results.

|                  |              | Incompressible      |                       |                       |                      |                      | Compressible          |                       |                       |                      |                     |
|------------------|--------------|---------------------|-----------------------|-----------------------|----------------------|----------------------|-----------------------|-----------------------|-----------------------|----------------------|---------------------|
| Operation        | Record Size  | 100                 | 1,000                 | 10,000                | 100,000              | 1,000,000            | 100                   | 1,000                 | 10,000                | 100,000              | 1,000,000           |
| insert           | <1 KIB       | Berkeley DB (2 ns)  | Berkeley DB (5 ns)    | Berkeley DB (5 ns)    | Berkeley DB (8 ns)   | Berkeley DB (9 ns)   | Berkeley DB (6 ns)    | Berkeley DB (9 ns)    | Berkeley DB (8 ns)    | Berkeley DB (8 ns)   | Berkeley DB (10 ns) |
|                  | 1-10KIB      | Berkeley DB (8 ns)  | Berkeley DB (8 ns)    | Berkeley DB (10 ns)   | FlatFolder (11 ns)   | Berkeley DB (12 ns)  | Berkeley DB (9 ns)    | Berkeley DB (12 ns)   | Berkeley DB (14 ns)   | Berkeley DB (12 ns)  | Berkeley DB (16 ns) |
|                  | 10-100KIB    | FlatFolder (45 ns)  | FlatFolder (38 ns)    | FlatFolder (685 ns)   | Berkeley DB (916 ns) |                      | FlatFolder (38 ns)    | Berkeley DB (46 ns)   | FlatFolder (60 ns)    | Berkeley DB (96 ns)  |                     |
|                  | 100KIB-1 MIB | FlatFolder (285 ns) | FlatFolder (965 ns)   | FlatFolder (302 ns)   |                      |                      | FlatFolder (364 ns)   | FlatFolder (288 ns)   | NestedFolder          |                      |                     |
| update           | <1 KIB       | Berkeley DB (1 ns)  | Berkeley DB (3 ns)    | Berkeley DB (5 ns)    | Berkeley DB (6 ns)   | Berkeley DB (107 ns) | Berkeley DB (3 ns)    | Berkeley DB (7 ns)    | Berkeley DB (9 ns)    | Berkeley DB (7 ns)   | Berkeley DB (10 ns) |
|                  | 1-10KIB      | Berkeley DB (8 ns)  | Berkeley DB (8 ns)    | Berkeley DB (15 ns)   | LevelDB (27 ns)      | RocksDB (537 ns)     | Berkeley DB (31 ns)   | Berkeley DB (8 ns)    | Berkeley DB (10 ns)   | LevelDB (17 ns)      | LevelDB (21 ns)     |
|                  | 10-100KIB    | Berkeley DB (61 ns) | Berkeley DB (68 ns)   | Berkeley DB (95 ns)   | RocksDB (259 ns)     |                      | Berkeley DB (53 ns)   | Berkeley DB (60 ns)   | Berkeley DB (58 ns)   | Berkeley DB (78 ns)  |                     |
|                  | 100KIB-1 MIB | FlatFolder (162 ns) | NestedFolder (121 ns) | Berkeley DB (808 ns)  |                      |                      | Berkeley DB (600 ns)  | Berkeley DB (1423 ns) | Berkeley DB (3154 ns) |                      |                     |
| get              | <1 KIB       | Berkeley DB (1 ns)  | LevelDB (2 ns)        | Berkeley DB (2 ns)    | Berkeley DB (4 ns)   | Berkeley DB (5 ns)   | Berkeley DB (1 ns)    | LevelDB (2 ns)        | Berkeley DB (4 ns)    | Berkeley DB (2 ns)   | Berkeley DB (4 ns)  |
|                  | 1-10KIB      | LevelDB (2 ns)      | Berkeley DB (4 ns)    | Berkeley DB (4 ns)    | Berkeley DB (6 ns)   | LevelDB (130 ns)     | LevelDB (4 ns)        | Berkeley DB (4 ns)    | Berkeley DB (8 ns)    | FlatFolder (8 ns)    | RocksDB (160 ns)    |
|                  | 10-100KIB    | RocksDB (7 ns)      | RocksDB (12 ns)       | FlatFolder (16 ns)    | Berkeley DB (51 ns)  |                      | RocksDB (9 ns)        | RocksDB (13 ns)       | NestedFolder (14 ns)  | FlatFolder (16 ns)   |                     |
|                  | 100KIB-1 MIB | LevelDB (81 ns)     | LevelDB (91 ns)       | NestedFolder (134 ns) |                      |                      | FlatFolder (88 ns)    | FlatFolder (85 ns)    | NestedFolder (129 ns) |                      |                     |
| remove           | <1 KIB       | Berkeley DB (2 ns)  | Berkeley DB (4 ns)    | Berkeley DB (5 ns)    | Berkeley DB (6 ns)   | Berkeley DB (6 ns)   | Berkeley DB (2 ns)    | Berkeley DB (6 ns)    | Berkeley DB (8 ns)    | Berkeley DB (7 ns)   | Berkeley DB (9 ns)  |
|                  | 1-10KIB      | Berkeley DB (6 ns)  | Berkeley DB (9 ns)    | Berkeley DB (11 ns)   | Berkeley DB (10 ns)  | RocksDB (27 ns)      | Berkeley DB (5 ns)    | Berkeley DB (8 ns)    | Berkeley DB (8 ns)    | FlatFolder (22 ns)   | FlatFolder (30 ns)  |
|                  | 10-100KIB    | RocksDB (31 ns)     | NestedFolder (118 ns) | FlatFolder (36 ns)    | NestedFolder (47 ns) |                      | FlatFolder (30 ns)    | Berkeley DB (42 ns)   | NestedFolder (29 ns)  | NestedFolder (32 ns) |                     |
|                  | 100KIB-1 MIB | FlatFolder (113 ns) | FlatFolder (33 ns)    | FlatFolder (149 ns)   |                      |                      | NestedFolder (120 ns) | FlatFolder (414 ns)   | NestedFolder (687 ns) |                      |                     |
| space efficiency | <1 KIB       | SQLite3 (50%)       | SQLite3 (53%)         | LevelDB (77%)         | RocksDB (91%)        | RocksDB (92%)        | SQLite3 (50%)         | SQLite3 (56%)         | SQLite3 (88%)         | SQLite3 (91%)        | RocksDB (113%)      |
|                  | 1-10KIB      | SQLite3 (87%)       | SQLite3 (83%)         | SQLite3 (86%)         | LevelDB (97%)        | RocksDB (99%)        | SQLite3 (79%)         | SQLite3 (83%)         | SQLite3 (86%)         | LevelDB (144%)       | LevelDB (155%)      |
|                  | 10-100KIB    | SQLite3 (95%)       | SQLite3 (97%)         | SQLite3 (98%)         | LevelDB (99%)        |                      | FlatFolder (98%)      | LevelDB (106%)        | RocksDB (147%)        | RocksDB (159%)       |                     |
|                  | 100KIB-1 MIB | FlatFolder (100%)   | FlatFolder (100%)     | LevelDB (100%)        |                      |                      | LevelDB (112%)        | LevelDB (125%)        | RocksDB (152%)        |                      |                     |

|        |            |         |         |             |         |              |
|--------|------------|---------|---------|-------------|---------|--------------|
| Legend | FlatFolder | SQLite3 | RocksDB | Berkeley DB | LevelDB | NestedFolder |
|--------|------------|---------|---------|-------------|---------|--------------|

Figure 4.4: The results of the disturbed HDD



## 4.5 Discussion

Key-value store performance is a complicated problem that depends heavily on a particular workload, as seen by this benchmark. Developers should generally use an embedded database, especially one with compression, rather than the file system if space efficiency is a priority. Berkeley DB is among the best performers on our benchmark if speed is the primary consideration; it is the fastest or very close to it in most scenarios and operations, specifically inserting and updating operations.

### 4.5.1 Insert Operation

Berkeley DB, with an average of roughly 1,025 ns per operation and 403 ns for incompressible data, was the quickest average of the Insert operation categories for the SSD and HDD, both with and without a background program running. For both compressible and incompressible data, the nested and flat folder solutions came in second and third, respectively. The incompressible data was around 105% faster than the compressible data, and the data that remained unchanged regardless of the background program was also slower. Incompressible data was around 104% faster than compressible data, with the largest speed difference between the two being less than 1 KiB.

### 4.5.2 Update Operation

Berkeley DB was the quickest average of the Update operation categories for the SSD and HDD, averaging roughly 2,289 ns per operation for compressible data and 1,289 ns for incompressible data, both with and without a background program. Regarding both compressible and incompressible data, the nested and flat folder alternatives came in second and third. Uncompressed data, regardless of whether a background program was running, performed around 103% slower than data that was compressed. Incompressible data was

roughly 103% faster than compressible data, with the largest speed difference being the less than 1 KiB section.

### **4.5.3 Get Operation**

With a program running in the background or not, the flat folder choice had the fastest average of all the Get operation categories for the SSD and HDD, averaging roughly 1,562 ns per operation for compressible data. However, LevelDB was the fastest average for compressible data on both the SSD and HDD, averaging roughly 472 ns per operation for incompressible data, both with and without an application running in the background. For compressible data, the BerkeleyDB and LevelDB options were second and third, respectively, while the flat and nested file options ranked second and third fastest for incompressible data. Whether or not a background program was running, compressible data performed approximately 120% slower than incompressible data. Data smaller than 1 KiB showed the largest speed disparity between compressible and incompressible data—with incompressible data running almost 205% quicker.

### **4.5.4 Remove Operation**

With an average removal operation time of 360 ns for compressible data and 1,550 ns for incompressible data, the nested folder choice was the fastest of the Remove operation categories for the SSD and HDD, both with and without a background program. For both compressible and incompressible data, the flat folder option and Berkeley DB ranked second and third, respectively. Compressible data performed around 103% slower than incompressible data, and data remained unchanged regardless of the presence of a background program. Between 100 KiB and 1 MiB, there was a notable speed differential between compressible and incompressible data; in this range, incompressible data was around 185% quicker.

### **4.5.5 Space Efficiency**

With an average space efficiency of about 84%, SQLite was the most space-efficient option for incompressible data on both SSD and HDD devices. Berkeley DB came next, while the flat folder option came in third. With an average space efficiency of 90%, LevelDB was the most space-efficient database for compressible data on SSD and HDD devices, followed by SQLite3 and RocksDB. The efficiency of compressible data is 111% higher than that of incompressible data. Between 100 KiB and 1 MiB, compressible data was around 115% more efficient in terms of space efficiency.

### **4.5.6 File Systems vs Embedded Databases**

When determining whether to use an embedded database or a file system, the file system was among the quickest speeds in the insert and remove operations for compressible and incompressible data. However, the fastest speed on average for the insert, update, and remove options was consistently Berkeley DB. Space efficiency was consistent between LevelDB, RocksDB, and SQLite3.

### **4.5.7 Effects of Compression**

Although compression fully slowed down activities, it exchanged speed for space efficiency. Compared to its compressible equivalent, the average insert speed for incompressible data was around 95% faster. The average update incompressible insert speed was about 98% faster than the compressible speed. The average get operation incompressible speed was approximately 155% faster than the compressible speed. Compared to the compressible speed, the average removal operation incompressible speed was around 120% faster. On the other hand, compressible data had an average space efficiency of 109% more efficient than incompressible data.

## Chapter 5

# Conclusions and Future Work

This study compared the following key-value databases on NTFS: SQLite3, LevelDB, RocksDB, Berkeley DB, and the file system. Our results showed that Berkeley DB was the fastest among smaller record sizes among all operations for both compressible and incompressible data in both HDD and SSD. The flat and file folder options, along with RocksDB and LevelDB were the fastest options for larger record sizes among all operations for both compressible and incompressible data in both HDD and SSD. Larger record counts favor the nested folder option, and smaller record counts favor the flat folder option for both incompressible and compressible data in both HDD and SSD.

SQLite is more space-efficient for smaller record counts, while RocksDB is more space-efficient for bigger record counts. Some record counts, specifically 10,000 in the compressed data, notably suit LevelDB for compressible data.

This work differs from our previous research [1] due to modifications to random data storage. The random access data during the previous benchmark's execution varied with each run, potentially affecting the results. Storage modifications made on that work each time the benchmark was performed may also cause file sizes to change, so the earlier work was not compared.

The future work of this project is to create a benchmark for Android devices. Although SQLite is seen as the best database management system for Android devices due to its light weight and efficiency [34], it would be ideal to evaluate its efficiency on Android tablets or phones.

Also, we expect to consider to play a video in the background instead of playing it in YouTube on the experiments. Since the computer performance may be affected by network latency, we believe that playing the video in the background, without Internet, may help focus on the machine performance itself.

Finally, the project could be compared directly on Windows instead of WSL 2, as WSL 2 on Ubuntu has measurable overhead in code compilation performance and real-world scenarios. Michael Larabel's benchmark [35] acknowledges WSL2's competitive performance with Ubuntu 20.04 LTS and Ubuntu 21.10 but notes its slower performance, necessitating future code rewriting for Windows execution.

# Bibliography

- [1] J. Hines, N. Cunningham, and G. H. Alférez, “Performance comparison of operations in the file system and in embedded key-value databases,” *Lecture Notes in Networks and Systems*, p. 386–400, Aug 2023. ([document](#)), [1.1](#), [2.2.1](#), [3](#), [5](#)
- [2] C. Loewen, M. Wojciakowski, N. Sharma, Kyle, A. Junker, A. Jenks, G. Álvarez, D. Coulter, and A. Nasim, “Get started mounting a Linux disk in WSL 2,” Jul 2023. [Online]. Available: <https://learn.microsoft.com/en-us/windows/wsl/wsl2-mount-disk> [1.2](#)
- [3] N. Stepanets, “Introduction to file systems,” UFS Explorer, 04 2023. [Online]. Available: <https://www.ufsexplorer.com/articles/file-systems-basics/> [2.1.1](#)
- [4] —, “The file systems of Windows,” UFS Explorer, 04 2023. [Online]. Available: <https://www.ufsexplorer.com/articles/windows-file-systems/> [2.1.1](#)
- [5] J. Gerend and J. Barnett, “NTFS overview,” learn.microsoft.com, 03 2023. [Online]. Available: <https://learn.microsoft.com/en-us/windows-server/storage/file-server/ntfs-overview> [2.1.1](#), [2.1.3](#)
- [6] M. Corporation, *Microsoft BASIC-80 Version 5.0*. Microsoft, 1979. [2.1.1](#)
- [7] S. Cai, J. Kennedy, A. Ashcraft, F. Bristrow, A. Gubin, J. Martinez, K. Sharkey, S. Gupta, and N. Adman, “exFAT file system specification,” learn.microsoft.com, 10

2022. [Online]. Available: <https://learn.microsoft.com/en-US/windows/win32/fileio/exfat-specification> 2.1.1
- [8] R. Harwood, B. Smolen, M. Dhiman, S. Leavitt, K. Downie, K. Wester-Ebbinghaus, and E. Ross, “Resilient file system (ReFS) overview,” learn.microsoft.com, 02 2023. [Online]. Available: <https://learn.microsoft.com/en-us/windows-server/storage/refs/refs-overview> 2.1.1, 2.1.3
- [9] G. Burd, “Berkeleydb,” Dec 2012. [Online]. Available: <https://github.com/berkeleydb/libdb/releases> 2.1.2
- [10] A. Sullivan, “LevelDB,” Jan 2022. [Online]. Available: <https://github.com/google/leveldb> 2.1.2
- [11] I. Meta Platforms, “RocksDB.” [Online]. Available: <https://rocksdb.org/> 2.1.2
- [12] S. Consortium, “SQLite,” Nov 2007. [Online]. Available: <https://www.sqlite.org> 2.1.2
- [13] —, “Most widely deployed SQL database engine,” Dec 2007. [Online]. Available: <https://www.sqlite.org/mostdeployed.html> 2.1.2
- [14] S. Dan-Peng and D. Dan-Li, “ReFS vs NTFS vs FAT32, which one should you use?” EaseUS, 08 2023. [Online]. Available: <https://www.easeus.com/computer-instruction/refs-vs-ntfs-vs-fat32.html> 2.1.3
- [15] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, “The new EXT4 filesystem: current status and future plans,” in *Proceedings of the Linux symposium*, vol. 2. Citeseer, 2007, pp. 21–33. 2.1.3
- [16] M. Tariq, “Differences between the NTFS and FAT32 memory card file systems huawei support global,” consumer.huawei.com. [Online]. Available: <https://consumer.huawei.com/en/support/content/en-us00414645> 2.1.3

- [17] L. Renteria, “Primary Storage vs Secondary Storage: What’s the Difference?” Arcserve, 01 2023. [Online]. Available: <https://www.arcserve.com/blog/primary-storage-vs-secondary-storage-whats-difference> 2.1.4
- [18] V. Kasavajhala, “Solid state drive vs. hard disk drive price and performance study,” *Proc. Dell Tech. White Paper*, pp. 8–9, 2011. 2.1.4
- [19] S. Leeks, *Windows Subsystem for Linux 2 (WSL 2) Tips, Tricks, and Techniques: Maximise productivity of your Windows 10 development machine with custom workflows and configurations*. Packt Publishing, 2020. [Online]. Available: <https://books.google.com/books?id=8RYFEAAAQBAJ> 2.1.5
- [20] C. Loewen, “Announcing wsl 2,” May 2019. [Online]. Available: <https://devblogs.microsoft.com/commandline/announcing-wsl-2/> 2.1.5
- [21] P. J. Fleming and J. J. Wallace, “How not to lie with statistics: the correct way to summarize benchmark results,” *Communications of the ACM*, vol. 29, no. 3, p. 218–221, Mar 1986. 2.2
- [22] J. Gray, *The Benchmark Handbook*. Morgan Kaufmann Publishers, 1993. 2.2
- [23] N. Najafzade and A. Z. Mariezcurrena, “Embedded Databases with Berkeley DB,” *Universite Libre de Bruxelles, Universitr d’Europe*. 2.2.2
- [24] V. N. R. Patchigolla, J. Springer, and K. Lutes, “Embedded database management performance,” in *2011 Eighth International Conference on Information Technology: New Generations*, 2011, pp. 998–1001. 2.2.2
- [25] M. Slabinoha, S. Melnychuk, I. Manuliak, and B. Pashkovskiy, “Comparative analysis of embedded databases performance on single board computer systems using python,”



- in *2022 IEEE 17th International Conference on Computer Sciences and Information Technologies (CSIT)*, 2022, pp. 222–225. [2.2.3](#)
- [26] N. Obradovic, A. Kelec, and I. Dujlovic, “Performance analysis on Android SQLite Database,” *2019 18th International Symposium INFOTEH-JAHORINA (INFOTEH)*, pp. 1–4, 2019. [Online]. Available: <https://api.semanticscholar.org/CorpusID:159043379>  
[2.2.3](#)
- [27] H. B. Hassan and Q. I. Sarhan, “Performance evaluation of relational embedded databases: an empirical study,” *Innovaciencia*, vol. 6, no. 1, pp. 1–8, 2018. [2.2.3](#)
- [28] K. Fraczek and M. Plechawska-Wojcik, “Comparative analysis of relational and non-relational databases in the context of performance in web applications,” in *Beyond Databases, Architectures and Structures. Towards Efficient Solutions for Data Analysis and Knowledge Representation: 13th International Conference, BDAS 2017, Ustroń, Poland, May 30-June 2, 2017, Proceedings 13*. Springer, 2017, pp. 153–164. [2.2.3](#)
- [29] B. Sterniczuk, “Comparison of EXT4 and NTFS filesystem performance,” *Journal of Computer Sciences Institute*, vol. 25, pp. 297–300, 2022. [2.2.4](#)
- [30] K. L. Rusbarsky and K. City, “A forensic comparison of NTFS and FAT32 file systems,” *Marshall Univ*, vol. 29, 2012. [2.2.4](#)
- [31] Ubuntu, “Ubuntu pc operating system — ubuntu,” 2020. [Online]. Available: <https://ubuntu.com/desktop> [3](#)
- [32] B. Stroube, “Literary freedom: Project Gutenberg,” *XRDS: Crossroads, The ACM Magazine for Students*, vol. 10, no. 1, pp. 3–3, 2003. [3](#)
- [33] M. Matsumoto and T. Nishimura, “Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator,” *ACM Trans. Model.*

*Comput. Simul.*, vol. 8, no. 1, p. 3–30, jan 1998. [Online]. Available: <https://doi.org/10.1145/272991.272995> 3.2

[34] K. Dhokale, N. Bange, S. Pradip, and S. Malave, “Implementation of SQL server based on SQLite engine on android platform,” *International Journal of Research in Engineering and Technology*, vol. 3, no. 4, pp. 8–14, 2014. 5

[35] M. Larabel, “Windows 11 WSL2 Performance Is Quite Competitive against Ubuntu 20.04 LTS / Ubuntu 21.10,” Sep 2021. [Online]. Available: <https://www.phoronix.com/review/windows11-wsl2-good> 5