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

Jesse Hines
School of Computing
Southern Adventist University
Collegedale, USA
jessehines@southern.edu

Nicholas Cunningham
School of Computing
Southern Adventist University
Collegedale, USA
nicholascunningham@southern.edu

Germán H. Alférez
School of Computing
Southern Adventist University
Collegedale, USA
harveya@southern.edu

***Abstract*—A common scenario when developing local PC applications such as games, mobile apps, or presentation software is storing many small files or records as application data and needing to retrieve and manipulate those records with some unique ID. In this kind of scenario, a developer has the choice of simply saving the records as files with their unique ID as the filename or using an embedded on-disk key-value database. Many file systems have performance issues when handling large numbers of small files, but developers may want to avoid a dependency on an embedded database if it offers little benefit or has a detrimental effect on performance for their use case. Despite the need for benchmarks to enable informed answers to this design decision, little research has been done in this area. Our contribution is the comparison and analysis of the performance for the insert, update, get, and remove operations and the space efficiency of storing records as files vs. using key-value embedded databases including SQLite3, LevelDB, RocksDB, and Berkeley DB.**

***Index Terms*—databases, file systems, database performance**

1. Introduction

A common scenario when developing local desktop applications is the need for persisting many small files or records as application data and needing to retrieve and manipulate those records with some unique ID, essentially forming a key-value store. For example a game developer may need to store records for each game entity or game level or note-taking software would need to store a large number of small text records.

In this kind of scenario, a developer has two main choices: leveraging the file system for storage or using an embedded key-value database. If the developer chooses to use the file system to store the records, they can simply save each record to disk with its unique ID as the filename. This has the advantage of being simple to implement and adding no extra dependencies.

However, file systems can have space and performance issues when handling large numbers of small files [1] [2]. As an alternative to the file system, the developer can choose to use an embedded database. However, adding a dependency on an embedded database adds a fair amount of technical overhead to a project and increasing overall system complexity. If the embedded database is statically linked with an application, it will typically increase compilation time and executable size [3]. If it is linked dynamically, it can complicate installation of an application as external dependencies will need to be installed. Therefore, a developer will likely want to avoid adding a dependency on an embedded database if their use case is not significantly benefited by it.

Despite this being a common scenario, little research has been done comparing simple file system options to embedded key-value databases. Our contribution is the analysis and comparison of the performance of four popular open source embedded databases – SQLite3, LevelDB, RocksDB, and Berkeley DB – with storing records on the file system. This research will enable developers to make informed decisions on what tools are best for their scenario

This paper is organized as follows. Section 2 presents the existing research work on file system and embedded database performance. Section 3 presents the background knowledge behind our approach. Section 4 presents the methodology of the research and Section 5 presents the evaluation of the results. Section 6 presents directions for future research, and Section 7 presents our conclusions.

2. State of the Art

There is an extent of existing research comparing various databases [4], [5]. Additionally, there is existing research comparing the performance of storing blob data in SQL servers vs. the file system [6], [7]. However, most the existing research is not recent, and is mostly focused on database servers. Minimal research has been done on this question in the realm of local desktop applications and comparing the file system to embedded key-value databases such as LevelDB, RocksDB, and Berkeley DB.

One of the few research works on comparing key-value databases to file system is the work done by Patchigolla et. al [8] in 2011. This research performed detailed benchmarks on database IO performance on mobile devices vs. file IO.

The research was specifically focused on Android devices, and the benchmarks compared SQLite and Perst embedded databases. This research concluded that if the number of records is less than 1,000, then using files is a good decision, otherwise an embedded DB is a better option. We believe extending this research into the realm of local desktop applications and taking embedded key-value databases into account will be valuable to future developers.

A related question to the comparison of embedded databases and the file system, but within the scope of SQL database servers, is whether to store large pieces of data such as images in the database directly or on the file system. On SQL servers, the the two typical methods for this are to either store the data directly in an SQL BLOB column or to only store the file path in the database and save the data as a file. A widely cited paper on this question is the work done Sears et. al. [6] which analyzed the performance of storing files in SQL Server as BLOBs vs. files. They concluded that data smaller than 256KB is more efficiently handled by database BLOBs, while the file system is better for data larger than 1 MB. Stancu-Mara and Baumann [7] did detailed benchmarking on BLOB handling in various SQL database servers, including PostgreSQL and MySQL. They benchmarked read and write for blobs of various sizes, concluding that MySQL was the fastest at handling BLOBs. However, both of these benchmarks were done in the late 2000s, and their conclusions may no longer be applicable to modern systems.

There are many benchmarks comparing NoSQL databases. For instance, Gupta et. al. compare the performance and functionality of 4 NoSQL databases with widely different focuses: MongoDB, Cassandra, Redis, and Neo4j [4]. Puangsaijai et. al. [5] compare the key-value database Redis with the relational database MariaDB for insert, update, remove, and select operations.

Related to measuring the performance gains of using key-value databases is research on using key-value databases to improve performance. Tulkinbekov et. al. [9] solve the problem of write amplification with key-value databases. The problem of write amplification is when writes are repeated multiple times which can lead to performance issues and extra wear and tear on the memory. Their fix is to introduce their own key-value store CaseDB, which not only solves write amplification but also outperforms LevelDB and WiscKey, a LSM-tree-based key-value store. Similarly, Chandramouli et. al. introduced FASTER [10], a new embedded key-value database system to improve performance in large scale applications. Their research was focused on improving concurrent performance on large scale applications with millions of accesses a second.

An alternative to using key-value databases to replace the file system, is to use key-value databases to improve the file system itself. Chen et al . [11] applied key-value database technology to improve performance of the file system. They created a file system middleware named FILT which uses key-value database technology to achieve up to 5.8x performance over existing file systems.

3. Underpinnings of our Approach

This section presents the fundamental concepts of our approach as shown in Figure 1.

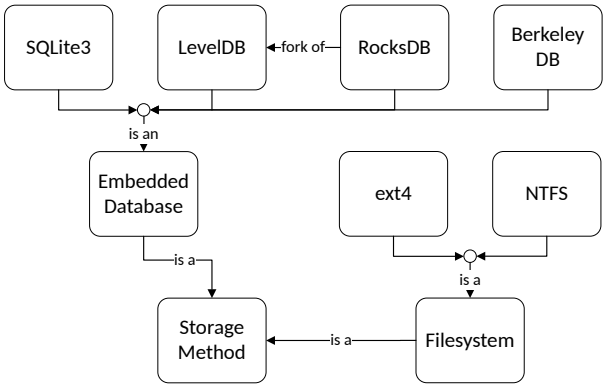

Figure 1. Underpinnings of our approach

3.1. Embedded Databases

Embedded databases are databases that are included in an application rather than as a separate server [12]. Since they are embedded in the application itself, they do not require a separate server or internet access to use. Embedded databases are meant for scenarios where an application is storing data that is only needed on the local machine.

Some embedded databases, such as SQLite, are full relational databases. However, simpler NoSQL alternatives have become popular such as key-value databases. Key-value databases only allow efficient access of each entry by a unique key, and do not support more complex queries. While this is restrictive, it is sufficient for many use cases and allows the database to be much simpler and optimized.

SQLite3[1] is one of the most popular embedded databases. SQLite3 is an open source C library that implements an embedded SQL database engine. It it used extensively in mobile devices, Python apps, and browsers, among many more applications[2]. While SQLite is a fully featured SQL database, for our purposes we treat it as key-value store by just having one table with a primary key.

LevelDB[3] is a open-source, key-value embedded database sponsored by Google. As a key-value database, it maps arbitrary binary string keys and values and does not support SQL queries. LevelDB offers features such as high performance, compression of data, and "snapshots" to provide consistent read-only views of the data. LevelDB is used in many applications including Chrome's IndexedDB[4], Bitcoin Core[5], and Minecraft Bedrock Edition[6].

1. https://www.sqlite.org
2. https://www.sqlite.org/mostdeployed.html
3. https://github.com/google/leveldb
4. https://chromium.googlesource.com/chromium/src/+/a77a9a1/third_party/blink/renderer/modules/indexeddb/docs/idb_data_path.md
5. https://bitcoindev.network/understanding-the-data
6. https://github.com/Mojang/leveldb-mcpe

RocksDB[7] is another open-source, key-value embedded database. RocksDB is sponsored by Facebook, and is used in much of their infrastructure. RocksDB actually started as a fork of LevelDB, and has a similar API. Facebook found that LevelDB didn't perform well with massive parallelism or datasets larger than RAM, so they created RocksDB with a focus on scaling to large applications.

Berkeley DB[8] is an open source embedded database provided by Oracle. Berkeley DB advertises itself as "a family of embedded key-value database libraries that provides scalable, high-performance data management services to applications." Berkeley DB is very feature rich, and encompasses multiple different database technologies. Our benchmark makes use of the efficient key-value API, but Berkeley DB can also create relational SQL databases, and use multiple different index types, among many other features. Some programs that use Berkeley DB are Bogofilter, Citadel, and SpamAssassin.

3.2. File System Performance

File systems can have issues when handling large numbers of small files. For instance, most modern file systems, including FAT, ext3, and ext4, allocate space for files in a unit of a cluster or block, no matter how small the file is. On large files this is inconsequential, but if there is a very large number of files smaller than the cluster size it can lead to a large amount of space being wasted [1]. Directory lookup can also take a performance hit if a large number of files are directly under a single folder [2].

This file system performance bottleneck has led many applications, including browsers and web caches, to use "nested hash file structures" with files placed into intermediate sub-directories based on their hash [13]. This avoids placing too many files directly under a single folder and can improve performance. As seen in Figure 3, the top level directory may contain folders `00` through `ff`, where each sub-directory contains files whose hash digest starts with those characters. Files would be spread evenly across all 256 sub-directories. This process can be repeated for as many levels of nested as is required to get a reasonable number of files in the "leaf" directories.

4. Methodology

An pseudocode outline of the algorithm used for the benchmark can be seen in Listing 1. The algorithm loops over different combinations of store type, data type (compressible or incompressible), size range, and count range (Line 3). For each combination it benchmarks 1,000 iterations of each of the 4 key-value operations (Lines 11, 16, 20 and 25) using a random access pattern and random data. It then records the peak memory usage and disk space efficiency for each combination (Lines 30 and 33). Finally it outputs the results to a CSV file (Line 35).

7. http://rocksdb.org

8. https://www.oracle.com/database/technologies/related/berkeleydb.html

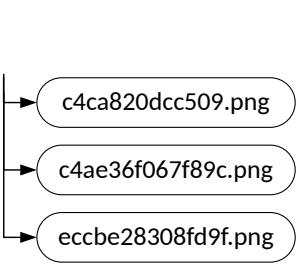

Figure 2. flat structure

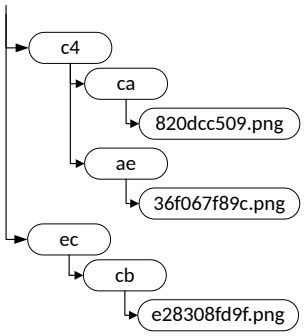

Figure 3. nested structure

```
1  baseMem = getPeakMem()
2
3  for each combination of
4      (storeType, dataType, size, count):
5      if averageRecordSize * count.max > 10GiB:
6          skip
7
8      resetPeakMem()
9      store = new storeType with initial data
10
11     repeat 1000 times:
12         key = newKey(store)
13         value = gen(dataType, size)
14         benchmark store.insert(key, value)
15
16     repeat 1000 times:
17         key = pickKey(store)
18         benchmark store.get(key)
19
20     repeat 1000 times:
21         key = pickKey(store)
22         value = gen(dataType, size)
23         benchmark store.update(key, value)
24
25     repeat 1000 times:
26         key = pickKey(store)
27         benchmark store.remove(key)
28         store.insert(key, gen(dataType, size))
29
30     peakMem = getPeakMem() - baseMem
31     dataSize = getDataSize(store)
32     diskSize = getDiskUsage(store)
33     spaceEfficiency = dataSize / diskSize
34
35     write measurements to file
```

Listing 1. Pseudocode of benchmark

4.1. Comparing Storage Methods

4 different embedded key-value databases were compared, SQLite3, LevelDB, RocksDB, and Berkeley DB. While SQLite is a fully featured SQL database, for our purposes we treat it as key-value database by just having one table with a primary key column and a value column.

Then, the embedded databases were compared with two strategies for storing key-value records on disk, flat and nested. The flat storage strategy places all the records as files with their key as their name under one directory, as

seen in Figure 2. The nested storage strategy uses a nested hash file structure as described in Section 3.2 and Figure 3. 3 levels of nesting were chosen with 2 hexadecimal chars per level. We choose a depth of 3 as it can hold up to 16,777,216 (256^3) records while still keeping around 256 records in the leaf nodes, and our benchmark is only testing up to 10,000,000 records.

As seen in Figure 4, wrappers were created for each of the 6 different storage methods that implemented an abstract interface that could be used by the benchmark. The wrappers had methods for insert, update, get, and remove, and kept an count of how many records were in the database.

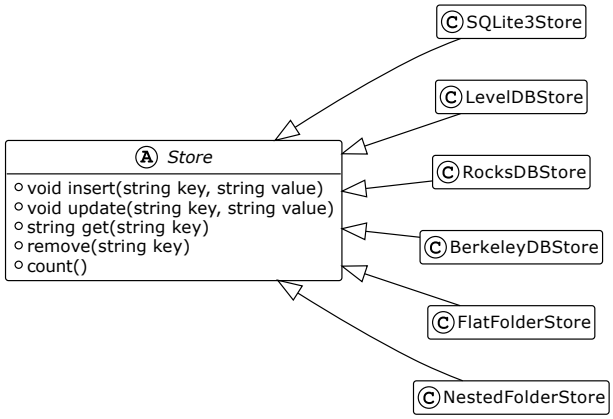

Figure 4. Class diagram

Both the SQLite and Berkeley DB C interfaces return `char*` pointers that become invalid after doing another database operation. The wrappers did a copy of the buffer memory managed by our code. This does add some overhead that could be avoided if a developer only needed the data briefly and did not need to do any other database operations until they were finished with the data. But we assume that the most common use case will do a copy to avoid risking dangling pointers and memory safety issues.

4.2. Comparing Usage Patterns

Multiple different usage patterns for each of the storage methods were compared. 4 record size ranges were compared, less than 1 KiB, 1 - 10 KiB, 10 - 100 KiB, and 100 KiB - 1 MiB, and 3 record count ranges, 100 - 1,000, 10,000 - 100,000, and 1,000,000 - 10,000,000. All combinations of sizes and counts that led to databases smaller than 10 GiB were benchmarked (Line 5).

This benchmark measured random access reads and updates. A valuable direction for future work could be to benchmark different access patterns.

LevelDB and RocksDB implement data compression. Since this could have a significant impact on the size of the records on disk and the cost of IO operations, both compressible and incompressible data were tested. When working with compressible data both LevelDB and RocksDB were configured to use the Snappy[9] compression algorithm. Compression was disabled when working with incompressible data.

4.3. Generating Data

Data for values in insert and update statements was randomly generated. Incompressible data was generated with the Mersenne Twister 19937 algorithm (as in the C++ standard library) to generate pseudorandom bytes. Compressible text was selected from public domain books downloaded from the Gutenberg project[10]. We downloaded 250 English language books from Gutenberg and selected text randomly from them.

Keys were generated using a SHA-1 hash of the index, based on order keys were inserted (Line 12). Using this method let us emulate having random keys, but still be able to pick random keys from the store by hashing a random number from the range [0, store.count). This is important to implement random access get (Line 17), update (Line 21), and remove (Line 26) without storing a list of all keys inserted. This strategy does cause a complication when benchmarking the remove operation however. After a random remove we can no longer assume the keys are in a continuous range. This is easily remedied by replacing the key after each removal to keep the keys continuous (Line 28).

4.4. Taking Measurements

For each combination of store and usage patterns we measured multiple attributes. The primary attribute we measured was the time taken for each of the fundamental key-value operations, insert, update, get, and remove. For each combination we ran each operation 1,000 times and recorded the average, minimum, and maximum values in nanoseconds.

Since the memory usage of an application is also an important factor of performance, we measured the peak memory usage for each combination as well (Line 30). To measure peak memory usage we used the Linux `getrusage` syscall[11]. This syscall returns the peak resident set size used by the process in kilobytes. We reset the peak memory between configurations by using the special file `/proc/self/clear_refs`[12]. Writing a "5" to this file resets the process's peak memory usage or "high water mark" measurement and allows us to get the peak memory usage for a specific period of time. We then subtracted a baseline measurement of the memory usage taken at the beginning of the benchmark to show only the memory used by the store.

Another factor besides performance that may be important for developers is the "space efficiency" of a storage solution, especially if they are working with large amounts of

9. https://google.github.io/snappy
10. https://www.gutenberg.org/
11. https://man7.org/linux/man-pages/man2/getrusage.2.html
12. https://man7.org/linux/man-pages/man5/proc.5.html

data or limited space. Embedded databases can add storage overhead or, if they have compression, greatly reduce the size used. And the file system itself can waste space when storing many tiny files. We measured the approximate space efficiency of each store by comparing the amount of data put in versus the actual space taken up on disk (Line 33). We measured space efficiency by counting the number of bytes actually inserted into the store, and then using the `du` command[13] to measure the space taken on disk including wasted block space.

4.5. Implementing the Algorithm

We choose to implement the benchmark in C++. All the embedded databases we compared were written in C or C++ and so could be used in a C++ project. Using C++ allowed us to compile and link the embedded databases giving us more control over their configuration. Additionally, using C++ ensures there is no extra overhead from the language bindings.

The benchmark is open source and the code is available on GitHub[14].

4.6. Choosing Hardware

The benchmark was run on an virtual machine provide by Southern Adventist University. The machine was running Ubuntu Server 21.10 with 8 GiB of RAM and 250 GiB hard drive. TODO get more details of the server.

5. Results

This section describes notable patterns and analysis of the results of the benchmark. A condensed summary of the results is listed in Figure 5, which lists the most efficient storage option for each usage pattern. The operation measured and record size are listed on the vertical axis, while data compressibility and record count are listed on the horizontal. Combinations of record size and count that would result in more 50 GiB of files were skipped and left blank in the figure. The full results of the of the benchmark are available on GitHub[15] in both CSV and Excel format with charts.

5.1. Filesystem vs DBs

The choice of when to use the file system vs. an embedded database is complex and very dependent on the particular usage pattern what types of operations are being performed.

For records less than 1 KiB, an embedded database is faster, in this benchmark, Berkeley DB is the fastes in most scenarios with tiny records.

13. https://man7.org/linux/man-pages/man1/du.1.html
14. https://www.github.com/TODO/host/on/github
15. https://www.github.com/TODO/host/on/github

At the 1 to 10 KiB range, databases are still better in most scenarios, though the file system is faster on get and inserts when there are 100,000 records.

At the 10 to 100 KiB range, file system performance is better than or very close to the database's for get and insert while RocksDB or LevelDB is faster removes with record counts less than 10,000 as long as compression is turned off. Berkeley DB leads on the update benchmarks.

The results with 100 KiB to 1 MiB records is similar, the file system is generally faster for get and insert, though LevelDB without compression is faster for 1,000 records or less. Berkeley DB performs well on updates. RocksDB without compression is fastest for removes until 10,000 then the file system is faster.

As expected, the file system had poor space efficiency of about 12% on the tiny records due to the block allocation overhead, while the embedded databases were able to compact multiple records into a block. The baseline storage overhead of the embedded databases appears to be minimal, and both Berkeley DB and RocksDB were more space efficient than the flat folder store even with only 100 records. At large sizes the differences in space efficiency between databases and the file system are minimal, as the wasted space at the end of each block becomes less of a factor. However, in the case of compressible data, LevelDB and RocksDB can use compression to get space efficiency of up to 156% at the cost of more CPU overhead.

5.2. Effects of Compression

For the databases that implemented compression, LevelDB and RocksDB, turning compression on increased space efficiency of course, at the cost of notably decreased performance. The reduced IO does not seem to make up for the extra CPU overhead of performing the compression.

5.3. SQLite3 Patterns

SQLite scales well in most scenarios, for instance with less than 1 KiB records the get operation is only 60% slower at 1,000,000 records than for 100. An interesting deviation from this is at 1,000,000, 1KiB and 10KiB records of incompressible data, where SQLite take 64x longer than at 100,000 records. This spike does not occur when using compressible data however, so perhaps SQLite handles text and binary data differently n certain scenarios. The update operation took about 20-30% long the incompressible binary data than compressible text data. Space efficiency increased as record size and record count increased, reaching nearly 100% at the highest sizes.

However, the constant factor is very high and SQLite is one of the worst performing storage options. This is unsurprising as SQLite is a full relational database and has many more features and overheads than necessary for a simple key-value store.

		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 (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 5. Summary of the best storage options for each usage pattern

5.4. LevelDB Patterns

Compression adds significant overhead to LevelDB. With compression turned on, performance on all operations at large record sizes or large record counts is up to 20x slower than with it off. Even without compression, LevelDB has some major performance degradation for gets starting at 1,000,000 records for less than 1 - 100 KiB records, and 100,000 for 100 KiB to 1 MiB records as well as inserts at the 100 KiB to 1 MiB size range and over 100,000 records.

With compression enabled, space efficiency of text data can reach as high as 155 percent space efficiency.

5.5. RocksDB Patterns

Since RocksDB is a fork of LevelDB, we expected to see similar results, however their performance had quite a few differences. RocksDB performance degrades significantly at 100 KiB to 1 MiB records for all the operations, and the performance degradation for get with large record counts and compression was more pronounced for than in Lev-

elDB. RocksDB did not notable performance degradation for inserts on high record counts, unlike LevelDB.

5.6. Berkeley DB Patterns

Berkeley DB is one of the best performing databases in this benchmark. With the time of the operation increasing slowly as count increases. However, there's some notable spikes. At 1,000,000, 1 KiB - 10 KiB records there was an up to 100x slow down for get, update, remove operations with both incompressible and compressible data.

5.7. File System Patterns

The file system scales to large numbers of folders quite well until about 1,000,000, 1 - 10 KiB records where get time increases 500x for both compressible and incompressible data. This spike does not occur on smaller records. Insert and update time is primarily dominated by the record size factor.

File system operations have several outliers. For instance on the insert, compressible, 1 - 10 KiB, 1,000,000 record benchmark, the nested folder store has a 33x slowdown compared to incompressible. Since the file system does not do any compression, it seems unlikely that text vs binary data would make such a large difference and if it did it should affect both flat and nested, which it doesn't. Looking at the measurements, the total sum of the 1000 repetitions was 776 ms, while the maximum record was 707 ms, one single iteration took 90% of the time. Similar spikes occur elsewhere in file system operations. We suspect that some OS factor is causing intermittent stalls in file system operations.

5.8. Flat Folder vs. Nested Folder

Using the nested directory structure seems to be mostly counter productive. In most cases the nested folder is slower than the simpler flat structure. It was also less space efficient than the flat folder store, unsurprisingly, due to the extra directory overhead. In a few scenarios it offers some benefits, on tje remove operation at 10 - 100 KiB with 100,000 records, the nested structure is about 30% faster than the flat and nested is faster for updates with 10 - 100 KiB sized records and a record count of 100,000.

The benchmark did not show the file system performance degrading significantly at large file sizes. In fact file system access time was near constant, varying only by 1-2x on most count ranges, with for notable exception. At the 1 - 10Kb record size range and 1,000,000 count range the file system has a 500x degradation in speed compared to the 100,000 range. Interestingly, this degradation did not occur on any of the other record sizes.

6. Future Work

This benchmark has shown that key-value store performance is a complex issue that is highly depedent on a particular workload. Research examining other factors is needed. This benchmark ran on the ext4 file system, research comparing the performance of different file systems, in particular the wide-spread NTFS, may have different performance profiles. Examining the effects of each embedded database's configuration as well as different access patterns, such as sequential access, would also be valuable avenues for research.

Performance may be affected by complex caching patterns or the embedded databses running background processes such as LevelDBs background compaction[16]. More research in to the exact causes for the various outliers and performance degradations noted in this dataset would be valuable as well.

7. Conclusion

TODO

References

[1] T.-Y. Chen, Y.-H. Chang, S.-H. Chen, N.-I. Hsu, H.-W. Wei, and W.-K. Shih, "On space utilization enhancement of file systems for embedded storage systems," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 3, apr 2017. [Online]. Available: https://doi.org/10.1145/2820488

[2] L. Ruan, Y. Ding, B. Dong, X. Li, and L. Xiao, "Small files problem in parallel file system," in *Network Computing and Information Security, International Conference on*, vol. 2. Los Alamitos, CA, USA: IEEE Computer Society, may 2011, pp. 227–232. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/NCIS.2011.143

[3] C. Collberg, J. H. Hartman, S. Babu, and S. K. Udupa, "Slinky: Static linking reloaded," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USA: USENIX Association, 2005, p. 34.

[4] A. Gupta, S. Tyagi, N. Panwar, S. Sachdeva, and U. Saxena, "Nosql databases: Critical analysis and comparison," in *2017 International Conference on Computing and Communication Technologies for Smart Nation (IC3TSN)*, 2017, pp. 293–299.

[5] W. Puangsaijai and S. Puntheeranurak, "A comparative study of relational database and key-value database for big data applications," in *2017 International Electrical Engineering Congress (iEECON)*, 2017, pp. 1–4.

[6] R. Sears, C. van Ingen, and J. Gray, "To BLOB or not to BLOB: large object storage in a database or a filesystem?" *CoRR*, vol. abs/cs/0701168, 2007. [Online]. Available: http://arxiv.org/abs/cs/0701168

[7] S. Stancu-Mara and P. Baumann, "A comparative benchmark of large objects in relational databases," in *Proceedings of the 2008 International Symposium on Database Engineering & Applications*, ser. IDEAS '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 277–284. [Online]. Available: https://doi.org/10.1145/1451940.1451980

[8] K. Lutes, V. Patchigolla, and J. Springer, "Embedded database management performance," in *Information Technology: New Generations, Third International Conference on*. Los Alamitos, CA, USA: IEEE Computer Society, apr 2011, pp. 998–1001. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ITNG.2011.171

16. https://github.com/google/leveldb/blob/main/doc/impl.md#compactions

[9] K. Tulkinbekov and D.-H. Kim, "Casedb: Lightweight key-value store for edge computing environment," *IEEE Access*, vol. 8, p. 149775–149786, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9167231

[10] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, "Faster," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, p. 1930–1933, Aug 2018.

[11] C. Chen, T. Deng, J. Zhang, Y. Zou, X. Zhu, and S. Yin, "Filt: Optimizing kv-embedded file systems through flat indexing," p. 1203–1204, Nov 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9355804

[12] Techopedia, "What is an embedded database? - definition from techopedia," Dec 2014. [Online]. Available: https://www.techopedia.com/definition/30660/embedded-database

[13] S. Patil and G. Gibson, "Scale and concurrency of giga+: File system directories with millions of files," in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, ser. FAST'11. USA: USENIX Association, 2011, p. 13.