Client Implementation and Performance Analysis of LevelDB

Sneh Raval
MTech (ICT)
Dhirubhai Ambani Institute of Information and Communication Technology, Gandhinagar, Gujarat, India
202311013@daiict.ac.in

Rahul Mundra
MTech (ICT)
Dhirubhai Ambani Institute of Information and Communication Technology, Gandhinagar, Gujarat, India
202311047@daiict.ac.in

P M Jat
Guide
Dhirubhai Ambani Institute of Information and Communication Technology, Gandhinagar, Gujarat, India
pm_jat@daiict.ac.in

***Abstract*—This study presents a detailed analysis of LevelDB, a fast and lightweight key-value storage library developed by Google. Here, we implement a client application that utilises LevelDB by incorporating its header files and implementing the PUT and GET functions to insert key-value pairs and retrieve values based on specific keys. By executing these operations, we aim to gain insights into the internal logging mechanisms of LevelDB and evaluate its performance. Specifically, we measure the time taken for PUT and GET operations under various conditions to understand the efficiency and robustness of LevelDB. Our analysis provides valuable data on the performance characteristics of LevelDB, which can help make informed decisions when selecting storage solutions for their applications.**

***Index Terms*—LSM Tree, LevelDB, SSTables, Memtable, Cmake, Bloom Filter.**

I Motivation

As we studied LevelDB in our study of the LSM tree [2], we wanted to observe its workings practically. The primary motivation behind this study is to implement and observe LevelDB in a live environment, thereby gaining practical insights into its functionality and performance. By developing a client application that employs LevelDB for key-value storage, we aim to understand the core of its internal operations, particularly focusing on its logging mechanisms and the efficiency of PUT and GET operations. Through our study, we seek to analyse LevelDB's structure and evaluate its practical differences in real-world scenarios. Analysing these aspects will help us comprehend the practical performance of LevelDB. By sharing our findings, we aim to contribute to the collective knowledge and offer guidance to those considering LevelDB as a storage solution, ensuring they can make informed decisions based on real-world performance metrics.

II Introduction

This study is about implementing a client using LevelDB [1] to perform PUT and GET operations. We try to extract values for the time taken for PUT and GET operations. We begin by looking into the process of developing a client for LevelDB in section III, detailing the steps involved in building LevelDB from source, programming the client, and integrating key functionalities.

Before discussing the experiment, we review several key terminologies that are fundamental to understanding LevelDB's performance characteristics and optimization strategies in section IV. Following this, we describe the experimental setup, including the configurations tested, the dataset used, and the methodologies for measuring performance in section V. This section provides a comprehensive overview of how different configurations affect LevelDB's PUT and GET operations, highlighting the effects of write buffer size, block size, and compression methods. In section V-A we explore the internal logging mechanisms of LevelDB, explaining how logs are managed and utilised within the database.

Finally, We present the quantitative results of our experiments in a detailed table, followed by a discussion of key observations in section VI. We discuss some possible future work in section VIII.

Overall, this study thoroughly examines LevelDB's performance and operational characteristics through practical experiments, offering valuable insights.

III Implementation of a Client of LevelDB

This section details the steps and tools used to develop a client for LevelDB. It includes the processes of building LevelDB, creating the client, utilising a JSON parser for managing key-value pairs, and running the client to perform PUT and GET operations. This practical approach provides a comprehensive understanding of effectively integrating and using LevelDB in a real-world scenario.

A. Making a build of LevelDB

We first compiled LevelDB from its source code on a Linux system using CMake. It involved configuring the build

environment and executing the necessary build commands to generate the LevelDB binaries and libraries. Details of which can be found at [3].

B. Programming a client for LevelDB

The PUT operation program inserts key-value pairs into the LevelDB database. The program reads data from JSON files and inserts them into the database while utilising a Bloom filter policy to optimise future read operations. To manage key-value pairs, we integrated the nlohmann/json.hpp parser [4] into the client application. This parser facilitated the addition of key-value pairs by reading JSON formatted data. We parsed two JSON files, intentionally including duplicate keys to observe how LevelDB handles such scenarios. This structured approach allowed for easy data input and consistent management of key-value pairs.

The program begins by setting up the necessary LevelDB options, including creating the database if it does not already exist and enabling the Snappy compression to optimize storage. A Bloom filter policy is applied with a specified bits-per-key value to enhance read performance by reducing false positive rates.

The data is read from specified JSON files, and each key-value pair is inserted into the database using the Put method. The total time taken for the insertion process is measured and displayed to the user. Finally, the database connection is closed, and the filter policy is cleaned up to free resources.

In addition to the PUT operation, the program supports GET operations to retrieve values based on specific keys. After the insertion process, the program allows users to input keys and perform GET operations. Each GET operation measures and displays the time to retrieve the value from the database. This is done to evaluate the read performance of LevelDB under different configurations.

The program registers a signal handler to ensure that the database connection is closed gracefully in case of an interrupt signal (SIGINT), preventing data corruption and ensuring a clean shutdown. Overall, the client program provides a comprehensive testing framework for analysing LevelDB's performance in both write and read operations under various conditions. All the code details for the client implementation can be found at [3].

IV Terminology

Before discussing the experiment, it is essential to understand key terminologies associated with LevelDB. These terminologies provide a foundational understanding of the various components and mechanisms that underpin LevelDB's operation. By familiarizing ourselves with these terms, we can better appreciate the nuances of the performance metrics and results discussed in the subsequent sections.

Here, we discuss these terminologies very briefly. All the core structural information and internal workings of LevelDB are discussed in our work [2].

The write buffer size in LevelDB is the size of the **MemTable**, an in-memory data structure where data is first written before being flushed to disk as an SSTable. The default write buffer size is typically 4 MB, but it can be configured to be larger or smaller depending on the application's specific needs. A larger write buffer size can improve write performance by reducing the frequency of flushes to disk, but it also consumes more memory.

Block Size in LevelDB refers to the size of the data blocks stored within SSTables. Each block is a contiguous chunk of data that is compressed and written to disk. The default block size is 4 KB, but it can be adjusted. Smaller block sizes can lead to more efficient use of cache but may result in higher overhead due to increased metadata and more frequent disk I/O operations. Conversely, larger block sizes can reduce overhead but may lead to less efficient caching.

SSTable(Sorted String Tables) are immutable data files where LevelDB stores data after being flushed from the MemTable. Each SSTable contains a sequence of key-value pairs sorted by keys. SSTables are designed for fast sequential reads and efficient range queries. They play a crucial role in LevelDB's architecture, enabling fast-reliable data retrieval.

A **Bloom Filter** is a probabilistic data structure used in LevelDB to test whether an element is a set member. It is particularly useful for reducing the number of disk reads for non-existent keys. When a key is queried, the Bloom filter can quickly determine if the key is not present in an SSTable, thereby avoiding unnecessary disk access. This significantly enhances read performance, especially for workloads with many lookups for non-existent keys.

Compaction in LevelDB is merging SSTables to reduce the number of files and reclaim space. During compaction, overlapping key ranges in different SSTables are combined into new SSTables, which helps to eliminate deleted data and merge fragmented data. Compaction ensures that the database remains efficient and prevents the accumulation of excessive disk space due to deleted or obsolete data.

LevelDB includes an LRU (Least Recently Used) **Cache** to store frequently accessed data in memory. This cache helps to speed up read operations by keeping hot data readily available, reducing the need to access the disk for every read request.

V Experiment

In our experiment, we aimed to evaluate the performance of LevelDB, focusing on the effects of different configurations on PUT and GET operations. We used a generated 3 million key-value pairs dataset to conduct our experiments. Our main variables were write buffer size, block size, and compression method. Additionally, we investigated the impact of Bloom filters on GET operations, especially for invalid keys.

We conducted experiments on a Linux machine with 8 GB RAM to handle in-memory operations. We went through several iterations for accurate results.

The LevelDB configurations we tested included three different write buffer sizes (4 MB, 2 MB, and 512 bytes) and three corresponding block sizes (4 KB, 1 KB, and 512 bytes). We also compared the performance of Snappy compression against that of no compression. We set the database options for each test scenario to reflect these configurations.

We inserted key-value pairs from two JSON files into the database for PUT operations. We utilised the second JSON file to insert duplicate keys. We measured the time taken for these insertions using the chrono library [5] in C++, recording the durations in microseconds to capture fine-grained performance differences. Similarly, we measured the time for GET operations to retrieve values from the MemTable, newly created SSTables, and the cache. We also evaluated the effectiveness of Bloom filters by measuring GET times for non-existent keys.

Table I showcases all the results we collected, providing a detailed comparison of LevelDB's performance across various configurations and operations. Next, we discuss some important internal logging mechanisms implemented by LevelDB.

A. Internal Logs in LevelDB

During our experiment, we dived deep into the internal workings of LevelDB by examining the logs generated for the MemTable and SSTable files. These logs, stored in binary form, contain detailed information such as the value type, offset, data size, and actual value. To interpret these logs, we utilised the leveldbutil functionality provided by LevelDB, which allowed us to read the contents of both the MemTable and SSTable files effectively.

Our observations revealed that LevelDB initially logs write operations in a log file. These logs serve as a temporary storage mechanism until the data is flushed to disk as SSTable files. The SSTable files, which have the .ldb extension, store the persistent data. Additionally, LevelDB maintains a set of current and manifest files that track the current state and database versions. Each time the database is reopened, LevelDB starts a new log file by default, ensuring that recent write operations are logged efficiently before being flushed to disk.

These insights into the internal file management of LevelDB were crucial in understanding the performance characteristics and behaviours observed during our experiments. We noted that the log files play a critical role in ensuring data durability and efficient write operations, while the SSTable files facilitate fast read access. Using the leveldbutil tool was instrumental in uncovering these details, enabling us to gain a deeper understanding of how LevelDB manages its data structures and logs.

VI Observations

In our experiments, we made several key observations that provide insights into the performance characteristics of LevelDB under different configurations. We discuss these observations in detail, as well as the underlying reasons and implications.

A. *Smaller Block Sizes Take More Time for PUT Operations*

Smaller block sizes significantly increase the time required for PUT operations. This is because smaller blocks lead to more frequent flushing of data to disk, increasing the number of I/O operations. Each flush operation incurs overhead, resulting in slower overall performance. Consequently, larger block sizes, such as 4 KB, are generally more efficient for write-heavy workloads as they reduce the frequency of these costly operations.

B. *Merge Skipping*

Merge skipping is a technique LevelDB employs to optimise compaction processes. We observe this by drilling into the LevelDB code. We added many informative logs to the LevelDB code. The logs gave us all the important function calls and level-wise data ranges. After inserting some data, we observed that when the compaction is done for level 0, it skips a few levels while flushing the newly created tables. When data is written to the database, it is initially stored in the MemTable and then flushed to SSTables on disk. During compaction, LevelDB can skip merging certain SSTables if it determines that merging would not significantly reduce space or improve read performance. This optimisation helps to maintain write performance by avoiding unnecessary I/O operations.

For example, Compaction is triggered in level k, and no overlapping tables are found in the next n levels; the newly created tables are skipped to the k+n level. Fig 1 shows how merge skipping works in LevelDB.

Test	Compression type	MemTable size Default (4 MB) Block size (4 KB) (unit µs)	MemTable size (2 MB) Block size (1 KB) (unit µs)	MemTable size (512 Bytes) Block size (512 Bytes) (unit µs)
PUT	Snappy compression	25896364	24951833	36060092
	No compression	25263918	26786487	38719695
GET from MemTable	Snappy compression	53	32	19
	No compression	30	21	13
GET from new SSTable	Snappy compression	148	141	166
	No compression	460	316	180
GET from Cache	Snappy compression	72	84	100
	No compression	267	158	93
GET for non existent key (with Bloom filter)	Snappy compression	72	83	107
	No compression	94	105	97
GET for non existent key (without Bloom filter)	Snappy compression	2696	2376	2081
	No compression	2108	2180	2290

TABLE I
RESULTS OF LEVELDB PERFORMANCE TESTS

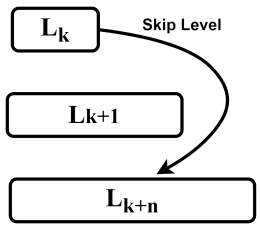

Fig. 1. Merge Skipping

C. *LRU Cache (Added to Cache Before Flushing to Disk)*

LevelDB uses an LRU cache to keep frequently accessed data in memory. Before flushing data from the MemTable to disk, it is added to the LRU cache. This ensures that recent writes are quickly accessible from memory, reducing the need for disk reads and improving read performance. The LRU cache effectively mitigates the performance impact of frequent disk I/O by prioritising recently accessed data.

D. *Reads Get Slower from Disk with Snappy Compression*

While Snappy compression reduces the data size on the disk, it introduces additional computational overhead during reads. This overhead is due to the need to decompress data before processing. As a result, reading directly from the disk can be slower when using Snappy compression than when uncompressed data is used. However, this trade-off is often acceptable as the reduced data size can lead to fewer disk I/O operations, balancing out the overall performance.

E. *Significantly Faster Reads from Cache*

Reads from the LRU cache is significantly faster than reads from disk. This is because accessing data from memory is orders of magnitude quicker than performing disk I/O. The cache plays a crucial role in enhancing read performance, especially for hot data that is accessed frequently. LevelDB minimises latency and delivers faster read responses by keeping this data in memory.

F. *The behaviour of Bloom Filters*

Bloom filters are a probabilistic data structure LevelDB uses to quickly determine if a key is not present in an SSTable. This reduces the number of disk accesses required for key lookups, especially for non-existent keys. Our experiments showed that enabling Bloom filters greatly improved GET operation times for invalid keys. This is because Bloom filters can efficiently filter out non-existent keys without accessing the disk, thereby reducing read latency and improving overall read performance.

G. *Snappy Compression is More Effective for Bigger Block Sizes*

Snappy compression shows its true potential with larger block sizes. Larger blocks benefit more from compression as the relative overhead of compression and decompression is lower than smaller blocks. Additionally, larger blocks reduce the frequency of I/O operations, further enhancing the effectiveness of compression. Our findings indicate that Snappy compression with larger block sizes results in a more

efficient storage solution, balancing the benefits of reduced disk usage and manageable read latency.

These observations provide valuable insights into the performance trade-offs involved in configuring LevelDB. By understanding these factors, users can better optimise their database configurations to achieve desired performance outcomes based on their specific workload characteristics.

VII Conclusion

Our comprehensive analysis and experimentation with LevelDB have provided significant insights into its performance characteristics and internal workings. By implementing a client application and conducting various tests on PUT and GET operations under different configurations, we have highlighted the impact of factors such as block size, write buffer size, compression methods, and Bloom filters. Our findings underscore the importance of optimising these configurations to enhance performance, especially regarding read and write efficiency. The detailed examination of LevelDB's logging mechanisms and file management has furthered our understanding of its durability and data management strategies. These insights are valuable for our study and serve as a practical guide for researchers leveraging LevelDB for their key-value storage needs. Ultimately, our research contributes to the broader knowledge base, offering actionable data and recommendations to inform future applications and improvements in LevelDB and similar storage solutions.

VIII Suggested Future Work

For future work, we plan to extend our analysis by comparing the performance of LevelDB with traditional data stores such as B+ trees [5]. This comparison will help us understand the strengths and weaknesses of LevelDB relative to other established storage solutions. Additionally, we aim to study and implement potential optimizations in LevelDB to enhance its performance further.

References

[1] S. Ghemawat and J. Dean, "LevelDB," GitHub Repository, https://github.com/google/leveldb

[2] S. Raval, R. Mundra, and P. M. Jat, "Study of LSM Tree", Dhirubhai Ambani Institute of Information and Communication Technology, Gandhinagar, Gujarat, India, unpublished, May 2024. https://shorturl.at/utli2

[3] https://github.com/S17eh/client-of-leveldb

[4] N.Lohmann,"nlohmann/json,"GitHub Repository, https://github.com/nlohmann/json

[5] https://en.cppreference.com/w/cpp/chrono

[6] M. Kleppmann, Designing Data-Intensive Applications, 1st ed. Sebastopol, CA: O'Reilly Media, 2017, pp. 71–85.