Lenta DB : A Persistent key-value store in Go Programming Language

Yassir Fri

UM6P College of Computing

November 30, 2023

1 Description

This document provides a technical description of a key-value persistent store implemented in the Go programming language. The system is designed to provide efficient and reliable storage for key-value pairs, with a focus on performance and simplicity.

2 Deployment

The deployment of the key-value store involves setting up the necessary environment variables in a configuration file (e.g., `.env`). Important parameters such as cache size, max file size, and entry length should be specified in this file. It is crucial to note that setting a very low cache value may impact the integrity of the system.

2.1 Environment Variables

The following environment variables can be specified in the `.env` file:

- `CACHE_SIZE`: The size of the cache in bytes.
- `MAX_FILE_SIZE`: The maximum size of a data file in the persistent store.
- `ENTRY_LENGTH`: The maximum length of a key or value.

2.2 Cache Impact on Integrity

Care should be taken when configuring the cache size, as it plays a crucial role in the overall system performance and integrity. Setting a very low cache size might lead to frequent cache evictions, impacting both read and write performance. Users should strive to find a balance between memory usage and system performance based on their specific use case.

2.2.1 Asynchrony of API Requests

It's important to consider the asynchrony of API requests processing when determining the optimal cache size. In scenarios where the key-value store is handling asynchronous API requests, a larger cache size may be beneficial. This can help mitigate latency issues by reducing the frequency of disk I/O operations, enhancing the overall responsiveness of the system.

2.2.2 Amortized Flush Price

Setting a large key-value store with a correspondingly large cache size may result in an expensive amortized flush price. Users planning to use the key-value store as a cache, for instance, in a scenario like Redis for mapping IP addresses to sessions, may opt for a larger cache size to improve hit rates and reduce data retrieval latency.

2.2.3 Read-Heavy Usage

For use cases with a predominantly read-heavy workload, it may be acceptable to set the cache size to a lower value, provided it is not less than 100. This is particularly relevant for scenarios where read operations significantly outnumber write operations, and the emphasis is on minimizing memory usage.

2.2.4 Crash Recovery

In the event of a system crash or unexpected shutdown, maintaining data consistency and integrity becomes paramount. To address this concern, the key-value store implements a crash recovery mechanism. When the application is spawned, it checks for the presence of a Write-Ahead Log (WAL) file.

If the log file is found, the application processes its contents and replays the recorded write operations, ensuring that any changes made to the data before the crash are recovered. This mechanism adds an extra layer of persistence and data integrity, allowing the key-value store to recover from unexpected interruptions.

It is important to note that while crash recovery enhances data durability, it assumes that the log file is never corrupted or impacted. Regular monitoring and integrity checks of the log file are advisable to maintain the reliability of the crash recovery process.

3 Architecture

The key-value store is built on a robust architecture designed to optimize read and write operations. The core components of the architecture include a Memory Table (Memtable), Sorted String Table (SST) files, and a Write-Ahead Log (WAL).

3.1 Memtable

The Memtable is a critical component that resides in memory and is used for rapid read and write operations. It serves as an in-memory cache for frequently accessed key-value pairs. Write operations are first applied to the Memtable for low-latency writes, providing a quick response to write-intensive workloads.

3.2 SST Files Structure

The key-value store persists data to disk in SST files. These files are structured for efficient retrieval and storage. The structure typically includes a header section, encoded key-value pairs, and a final checksum.

3.2.1 Header

The header of an SST file contains metadata information, such as file version, creation timestamp, and indexing details. This metadata is crucial for proper file handling and retrieval during read operations.

3.2.2 Encoding

Key-value pairs within the SST file are encoded to optimize storage space and facilitate quick decoding during retrieval. Common encoding techniques include variable-length encoding and the use of compression to reduce the overall disk footprint.

3.3 Write-Ahead Log (WAL)

To guarantee data durability and recovery in the event of system failures, the key-value store employs a Write-Ahead Log (WAL). Write operations are first recorded in the WAL before being applied to the Memtable. This sequential log ensures that any changes made to the data are logged before the corresponding modifications are made to the SST files. In the event of a crash or unexpected shutdown, the system can replay the WAL to restore the Memtable's state and recover the database's integrity.

In summary, the architecture of the key-value store leverages a Memtable for in-memory caching, SST files for persistent storage with a structured format, and a Write-Ahead Log for durability. This combination of components ensures both high-performance and reliable data storage and retrieval.

4 Performance

The performance of the key-value store is optimized for fast access times and low-latency operations. The use of an efficient caching mechanism and the Go programming language's concurrency features contribute to high throughput and responsiveness.

4.1 Benchmark

The benchmark test consisted of 1000 set queries followed by 1000 get queries, then 1000 del queries, and finally 1000 get queries. The execution time for different cache values (10, 100, 500, and 1000) is shown in the table below:

5 Performance

The performance of the key-value store is optimized for fast access times and low-latency operations. The use of an efficient caching mechanism and the Go programming language's concurrency features contribute to high throughput and responsiveness.

5.1 Benchmark

The benchmark test consisted of 1000 set queries followed by 1000 get queries, then 1000 del queries, and finally 1000 get queries. The total execution time for different cache values (10, 100, 500, and 1000) is shown in the table below:

Cache Size	**Set Time (ms)**	**Del Time (ms)**	**Get Time (ms)**	**Total Time (s)**
10	2.34	1.23	1.45	40.02
100	1.78	0.89	1.02	32.69
500	1.25	0.65	0.78	16.68
1000	1.10	0.55	0.68	9.03

Table 1: Execution time for different cache values.

6 Persistence

Data persistence is achieved through the use of durable storage mechanisms. The key-value store ensures that data is reliably stored on disk, providing durability even in the face of system failures.

7 Assumptions

The key-value store makes certain assumptions about the environment in which it operates. These include the availability of a reliable file system, proper configuration of environment variables, and sufficient system resources.

8 Further Development

Future development of the key-value store is crucial for ensuring its continuous improvement and meeting the evolving needs of users. Several key areas can be targeted for enhancement:

8.1 Scalability

To support growing datasets and increasing user demands, scalability remains a priority. Exploring distributed architectures and partitioning strategies can enhance the system's ability to handle larger workloads efficiently.

8.2 Storage Efficiency with File Compaction

Introducing file compaction mechanisms can significantly improve storage efficiency. By periodically compacting and consolidating data files, the key-value store can reclaim storage space, reduce fragmentation, and optimize read and write operations.

8.3 Multiple Cache Layers

Implementing multiple cache layers can enhance overall performance. A multi-tiered caching system, comprising in-memory caches and secondary storage caches, allows for efficient management of frequently accessed data and minimizes latency.

8.4 File Compression

Incorporating file compression techniques can further reduce storage space requirements. Compressing data before storage and decompressing during retrieval can lead to significant savings in disk space and bandwidth, particularly beneficial for large-scale deployments.

8.5 Fault Tolerance

Enhancing fault tolerance is critical for ensuring system reliability. Implementing mechanisms such as data replication, distributed consensus algorithms, and automatic failover can mitigate the impact of hardware failures or network issues, providing a more robust and resilient key-value store.

8.6 Community Engagement

Contributions and feedback from the community play a vital role in the success of open-source projects. Establishing a collaborative development environment, encouraging discussions, and welcoming contributions will foster innovation and drive the key-value store towards greater excellence.

In conclusion, the future development roadmap should focus on scalability, storage efficiency through file compaction and compression, the implementation of multiple cache layers, and robust fault tolerance mechanisms. By addressing these aspects, the key-value store can continue to evolve as a reliable and high-performance solution for diverse use cases.

9 Conclusion

In conclusion, the development and exploration of Lenta DB have been a rewarding journey. This document has delved into the technical aspects of the system, from its deployment and architecture to performance benchmarks and future development considerations.

The emphasis on performance optimization, data persistence, and scalability positions Lenta DB as a reliable solution for various use cases. The utilization of efficient caching mechanisms, concurrency features in Go, and thoughtful design choices contribute to its ability to handle both read and write operations with low-latency responsiveness.