Implementing a Persistent Key-Value Storage Engine to Improve the Efficiency of Sequential Data Reading

Aleksandra Tkachenko
Institute of Computer Science and Cybersecurity
Peter the Great St.Petersburg Polytechnic University
St.Petersburg, Russia
tkachenko3.aa@edu.spbstu.ru

Nikita Voinov
Institute of Computer Science and Cybersecurity
Peter the Great St.Petersburg Polytechnic University
St.Petersburg, Russia
voinov@ics2.ecd.spbstu.ru

***Abstract*—The article is devoted to development of a persistent key-value storage engine which implements a new approach to sequential data reading. The main idea of the proposed approach is clustering recently updated keys at the beginning of their data file so that the operating system would cache only this part of the file instead of caching the whole file improving the efficiency of sequential data reading. Experimental results of the developed storage engine implementing the proposed approach proved that it can be particularly efficient in case of a limited set of keys with a fixed size of values when fast sequential search is required.**

Keywords—key-value, storage engine, sequential data reading, efficiency, database

I. Introduction

Today the increasing volume of data and its critical importance for a business or organization emphasize the need of efficient data storage systems [1]. Data storage becomes a fundamental aspect since unsaved facts and information are unavailable for analysis and useless to the enterprise. The presence of systems capable of processing, storing and managing large volumes of data as well as extracting valuable knowledge from them, becomes an integral factor for making management decisions and optimizing business processes [2]. In the context of growing demand, fast data retrieval becomes especially important. Companies shall make decisions quickly based on up-to-date data as delays in accessing information can seriously affect the efficiency of the enterprise and its processes. To avoid such delays and ensure quick access to data, it is necessary to have an optimized data storage system [1].

There are many different data storage technologies including traditional relational databases, NoSQL databases, distributed file systems and cloud data warehouses [3]. Traditional relational databases can be a suitable choice for structured data and tasks that require ACID compliance, offering strong data schema and providing complex operations using SQL queries. However, the explosive growth of unstructured data in recent years has become one of the main reasons why relational databases no longer meet the needs of some companies facing with performance and scalability limitations [4]. To work with such data NoSQL solutions are actively used. They do not have a data schema and provide horizontal scaling [5]. Specific key-value storages can be used to store data that does not require complex processing but needs high access speed. In this case using an embedded persistent key-value storage engine that runs directly within the application can be especially efficient. This can reduce the overhead of communication with a remote database server and delays on data access as well as improve the performance of the entire system.

Solutions for fast data processing and disk space management, which can serve as analogues of the proposed approach, include optimization within file systems (FAT, NTFS, ext4, etc.), where specific methods are used to increase the speed of disk space processing such as clustering similar files, using metadata structures to efficiently retrieve data, and using caching mechanisms like LRU to reduce disk I/O operations. Also, indexing techniques are used in databases to speed up data retrieval by creating index structures: B-trees, hash indexes and bitmap indexes organize data to minimize the need for random disk access (which is similar to clustering frequently used keys). Finally, deduplication can be used. However, this work is specifically focused on creation of a data storage subsystem.

The main goal of the work is to improve the efficiency of sequential data reading by developing a persistent key-value storage engine based on a new approach of clustering recently updated keys at the beginning of the data file. The developed solution can be used within optimization of existing data storage and processing systems or construction of new ones to achieve the following expected benefits:

- improvement of the overall performance since sequential reading can be significantly more efficient than random access [6,7];
- reduction of overhead costs due to the fact that the storage engine works directly within the application, which reduces the overhead costs of communication with a remote database server, reduces delays when accessing data and increases system responsiveness;
- simplified deployment and maintenance because there is no need to install and configure a separate database server, which reduces infrastructure complexity and simplifies data recovery, which is an important factor for ensuring data security in applications.

II. Overview of Existing Solutions

Based on the exploring of relevant web-resources such as DB-Engine, Database of Databases (DoDB), Thoughtworks Technology Radar and Google Trends which provide rankings and compare popularity of miscellaneous software

tools, the following solutions were selected for further analysis: LevelDB [8], RocksDB [9], LMDB [10], Oracle Berkeley DB [11], BoltDB [12] and WiredTiger [13]. All these subsytems for persistent data storage have key-value and schema-free data model. This means that data is stored as key-value pairs where each key has a corresponding value. The absence of a data schema allows storing a variety of data types without first defining their structure or schema. This approach provides flexibility in storing and processing heterogeneous data and simplifies adding, changing or deleting data without the need to update the entire schema [4].

A common feature for all mentioned solutions is immediate consistency. It provides a simple data consistency model where written data is immediately available for reading and therefore has low latency since it does not require additional coordination or waiting operations. The main advantage is that the implementation of this type of consistency in the data storage engine is relatively simpler compared to other types which allows the use of simple mechanisms for writing and reading data. This consistency is useful for applications where data relevance is a priority while writing conflicts and concurrency are unlikely or unimportant.

Durability is ensured by Write-Ahead Logging (WAL). This mechanism guarantees data integrity, improves performance and ensures transaction stability. Data is written to the log first and then merged asynchronously with persistent storage which reduces write latency and increases speed [14]. WAL is also easy to implement and allows restoring the state of data in case of a system failure.

The main feature of the considered solutions is key-value data model. Data can be organized into key-value pairs that are stored on disk in corresponding data files. An index file, on the other hand, is used to quickly find keys in a data file. Typically the index file contains a data structure that maps the keys to their corresponding offsets in the data file where the corresponding values are located. That is, when searching for a value by key, the index file is first accessed to find the offset in the data file where the corresponding value is located. The value is then retrieved from the data file and returned to the user.

Choosing an index data structure to implement a storage engine, such as B-tree, Log-Structured Merge (LSM) tree or hash table, involves a number of trade-offs that depend on the specific use case and database requirements. LSM is well suited for write-intensive and read-intensive scenarios. B-tree is well suited for scenarios with a steady flow of writes and reads, where high performance support for data lookups and updates is required. A hash table is well suited for scenarios with high key search intensity and relatively low data update intensity, in addition, it excludes range queries and may be less efficient when working with large volumes of data.

For storages implemented on a B-tree or LSM sequential read performance is lower than for simple reading a file from an SSD [6]. This is because B-tree and LSM stores require additional operations to find keys and values in the data structure such as traversing tree levels or merging sorted lists. In addition, LSM requires sorting all the data which can take considerable time when adding new records. While simply reading a file from an SSD does not require additional operations to look up keys and values and can be performed sequentially without additional I/O operations. This provided higher speed of sequential reading.

With a large flow of input operations LSM trees may encounter the problem of write amplification, which leads to increased disk load due to frequent writes and merging of fragments. If the data is poorly compressed, LSM trees may require more disk space, which will also affect read and write operations. Additionally, when reading large amounts of data, LSM trees can have performance issues due to the need to scan multiple levels of the tree to retrieve data, while optimized serial I/O solutions can perform faster reads due to linear data access. Also, with a large number of requests to update or delete data, LSM trees may encounter a fragmentation problem, which will lead to performance degradation due to the need to perform additional merge operations. None of the storage subsystems can provide the same sequential read speed as simply reading a file from an SSD. Most storage engines are based on LSM, which requires all data to be sorted and does not provide an optimal way to cluster recently updated keys together. Although some keys are frequently updated and queried, LSM-based stores require all of their information to be cached within the operating system in a file cache.

Using simple disk operations instead of multi-level caching and restructuring makes sense in a number of specific scenarios. For example, a solution based on sequential I/O disk operations is easier to implement and support, especially in the case of small projects or applications with a limited development budget. If the data is rarely changed, using multi-level caching and restructuring may be overkill. Simple disk operations can be more efficient in this case, since they do not require additional effort to update caches.

So the analysis of existing solutions revealed that none of them can provide the same sequential reading speed as a simple reading a file from an SSD. Most storage engines are based on LSM which requires all data to be sorted and does not provide an optimal way to cluster recently updated keys together. Although some keys are frequently updated and queried, LSM-based storages require all of their information to be cached within the operating system in a file cache.

To solve this issue a new approach is proposed which consists in clustering recently updated keys at the beginning of their data file so that the operating system would cache only this part of the file instead of caching the whole file improving the efficiency of sequential data reading.

III. Implementation of the Proposed Approach

The developed persistent key-value storage engine StorageDB [15] implementing the proposed approach is based on the following functional features:

- a fixed key and value size (which allows the use of an offset-based indexing strategy);
- the data is stored in a flat file with synchronization markers every few records;
- when the database is opened, an index is created through sequential scanning (the index contains information about the key and its last location in the file);

- when new data is written, it is only added to the end of the file and the index is updated accordingly;
- when a database is opened, an index is created through a sequential scan. The index contains information about the key and its last location in the file;
- for sequential data scanning, a reverse pass through the record log (WAL) occurs first and then a direct pass through the data file; since the newest data is at the end of the WAL, the traversal is performed in reverse order which allows sequential access to the data without affecting the index (less CPU load);
- for random access the data is located by offset;
- when implementing the consolidation strategy, data from the old file is overwritten to the new one while the newly updated keys are collected at the beginning of the new file.

High-level architecture of the developed storage engine is presented in Figure 1.

StorageDBBuilder is an implementation of the Builder pattern for co-building an instance of the StorageDB class. StorageDB is a part of the storage engine and provides functionality for database management. This structural element provides a convenient way to configure and create a StorageDB instance with specified configuration parameters, allowing flexible storage management.

StorageDB [15] is an implementation of a key-value store database. It provides methods for reading, writing and manipulating data in a database. The main purpose of this class is to efficiently store and retrieve key-value pairs, as well as perform database compression and recovery from failures using files to store data, transaction logs and a buffer to temporarily store records before writing them to disk. It uses IndexMap to track the location of each value in a data file or WAL file by mapping keys to data offsets in the files. It also uses a buffer to optimize writes and periodically flushes the buffer to disk. In addition, it implements a background thread for compaction (combining data files to reduce the number of small files and improve read performance) and allows the use of a thread pool to perform background ExecutorService operations.

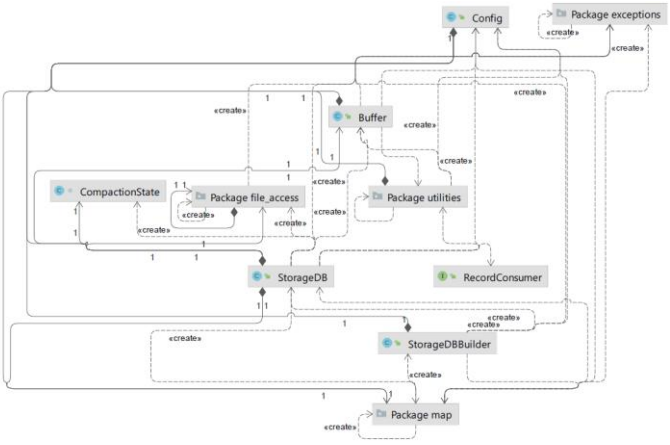

Fig. 1. High-level architecture of the developed storage engine

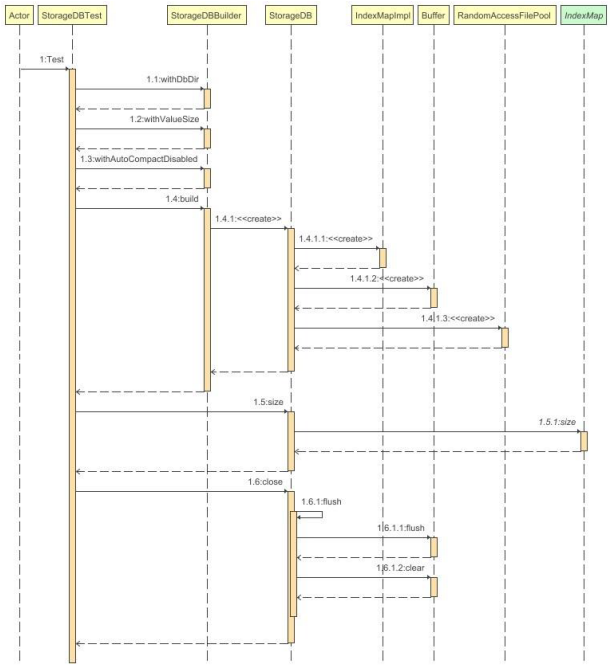

Fig. 2. Use-case diagram with a function call to determine the storage size

Config and CompactionState are needed to set system parameters. Also implemented are utilities for working with files, a number of additional exceptions and utilities for data recovery and address calculations.

Buffer is used to write data to a WAL write log file and then write it to disk. It is designed to work with a record buffer in RAM. The buffer is a logical extension of the WAL file, and if the index points to an offset greater than that of the actual WAL file, then the data is assumed to be in the write buffer. This class performs the following:

- Represents a buffer that is used to temporarily store records before they are written to disk. It serves as intermediate storage for data changes that will later be written to permanent storage (in a WAL file).
- Provides methods for performing read and write operations on a buffer. It allows adding new records with specified keys and values to the buffer, as well as updating existing records by key.
- Manages the allocation and release of memory for storing entries in the buffer. It allocates sufficient memory to store records based on the specified configuration and record sizes.
- Responsible for inserting synchronization markers into the buffer to ensure data integrity when written to disk. Synchronization markers help track record block boundaries and ensure data integrity when read.
- Provides methods for sequentially accessing entries in a buffer. This is convenient for performing traversal operations or processing entries in a buffer.

A use-case example demonstrating a function call to determine the storage size is presented in Figure 2.

First, an instance of the StorageDB object is created using the Builder pattern, performing several

configurations (1.1, 1.2, 1.3 in Fig.2). The build() call creates and returns an instance of the StorageDB object with the specified configuration parameters. During the execution of this construct, an implementation of the default index structure (1.4.1.1), WAL (1.4.1.2) and a pool of objects for file management (1.4.1.3) is created. Next, the test calls for the database size (1.5). At the end, db.close() is called to close the database after the test is run. This is important for properly freeing resources and preventing memory leaks. As part of this process, the flush() method (1.6.1) is called, which performs the operation of resetting the write buffer to the permanent record log storage (WAL) and performs some checks and status updates, in which the flush() of the Buffer class is used to write data from the buffer to the output stream and clear() is used to clearing the buffer by setting it to its original state.

The required software [15] was implemented with Java programming language. Maven was used as a build system, SLF4J as a logging framework, Logback-Classic as a logger, JUnit for testing, Mockito and Trove4j for optimized and efficient collections, JMH for measuring performance, JaCoCo for measuring code coverage, Apache Commons Pool 2 for resource pool management.

IV. Results

To assess the efficiency of the developed storage engine its performance was compared with one of the most popular existing solutions - RocksDB, which occupies leading positions in various ratings. It has been noted by experts and data storage specialists as one of the most efficient and productive solutions in its category. Quantitative measurements carried out in the selected use case showed significantly better performance of this solution compared to other analogues [16-19].

LSM trees commonly used in NoSQL databases are designed to efficiently handle keys and values of varying sizes, but the approach proposed in the paper targets a specific use case where fixed key and value sizes are feasible and beneficial for the scenario. While LSM trees are excellent at handling dynamic key and value sizes, they may not always be the most suitable solution for scenarios where predictable performance and optimized disk usage have the highest priority. The proposed approach offers an alternative solution tailored to specific requirements. Comparison with LSM trees was used just to highlight the specific results of the proposed method with no intention to position it as a direct competitor to LSM-based solutions.

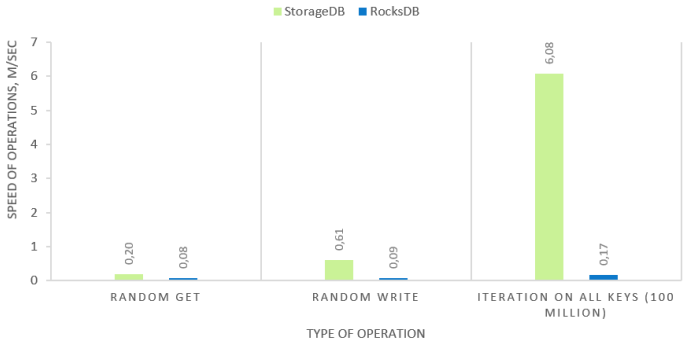

Fig. 3. Performance of the developed storage engine (green) and RocksDB (blue) on the measured metrics

To validate the functionality of the developed solution a singleflow test was conducted that included processing of a dataset consisting of 100 million 4-byte keys paired with corresponding 28-byte values, similar in its structure to the use case described in [20]. To obtain the results JMH (Java Microbenchmark Harness) was used, which is a popular tool for microbenchmarking in Java, as well as the following hardware: AMD Ryzen 5 2500U with Radeon Vega Mobile Gfx, 8 CPU, 8 GB RAM. The test was aimed primarily at verifying the functionality of the developed storage engine rather than providing a comprehensive benchmark against existing solutions. Tests on replace or removal operations has not been executed.

The results are presented in Figure 3 in the form of a histogram that displays the performance of two solutions based on the measurements. It's worth noting that the test was not intended to serve as a thorough comparison against existing solutions. It was rather designed to demonstrate the functionality of the developed storage engine under basic conditions.

V. Conclusion

Described in the paper is the persistent key-value storage engine which implements a new approach to sequential data reading. The main idea of the proposed approach is clustering recently updated keys at the beginning of their data file so that the operating system would cache only this part of the file instead of caching the whole file improving the efficiency of sequential data reading.

The developed storage engine can be particularly efficient in cases with a limited set of keys with a fixed size of values when fast sequential search is required. The results obtained are important because key-value data is the main tool for many similar systems while their internal implementation is modestly covered in the literature.

While demonstrated results indicate promising performance gains, further benchmarking against existing solutions with similar fixed-length key-value structures would provide a more comprehensive understanding of the developed solution's comparative performance. Futher it is planned to incorporate more robust testing methodologies, including diverse datasets and workload scenarios, to assess the stability and scalability of the proposed approach. Possible improvements to the developed software include less common data structures for data storage and indexing to speed up data access and reduce system load and more complicated testing to detect its bottlenecks.

References

[1] B. Denisenko, M. Tyanutov, I. Nikiforov, and S. Ustinov, "Algorithm for calculating TCO and SCE metrics to assess the efficiency of using a data center," Proceedings of the SPIE, vol. 12564, 1256403, Jan. 2023. DOI: 10.1117/12.2669285

[2] N. J. Ogbuke, Y. Y. Yusuf, K. Dharma, and B. A. Mercangoz, "Big data supply chain analytics: ethical, privacy and security challenges posed to business, industries and society," Production Planning & Control, vol. 33, no. 2-3, pp.123-137, Feb. 2022.

[3] S. Huang, Y. Qin, X. Zhang, Y. Tu, Z. Li, and B. Cui, "Survey on performance optimization for database systems," Science China Information Sciences, vol. 66, no. 2, p.121102, Feb. 2023.

[4] W. Khan, T. Kumar, C. Zhang, K. Raj, A. M. Roy, and B. Luo, "SQL and NoSQL database software architecture performance analysis and assessments – A systematic literature review," Big Data and Cognitive Computing, vol. 7, no. 2, p.97, May 2023.

[5] A. Ali, S. Naeem, S. Anam, and M. M. Ahmed, "A state of art survey for big data processing and nosql database architecture," International Journal of Computing and Digital Systems, vol. 14, no. 1, pp.1-1, May 2023.

[6] M. Kleppmann, "Designing Data–Intensive Applications," O'Reilly Media, 611 p., 2017.

[7] C. F. Andor, "Runtime Metric Analysis in NoSQL Database Performance Benchmarking," in 2021 International Conference on Software, Telecommunications and Computer Networks (SoftCOM), Sep. 2021, pp. 1-6.

[8] LevelDB [Online]. Available: https://github.com/google/leveldb [Accessed: May 06, 2024].

[9] RocksDB [Online]. Available: https://github.com/facebook/rocksdb [Accessed: May 06, 2024].

[10] LMDB [Online]. Available: www.symas.com/symas-embedded-database-lmdb [Accessed: May 06, 2024].

[11] BerkleyDB [Online]. Available: www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html [Accessed: May 06, 2024].

[12] BoltDB [Online]. Available: https://github.com/boltdb/bolt [Accessed: May 06, 2024].

[13] WiredTiger [Online]. Available: https://github.com/wiredtiger/wiredtiger [Accessed: May 06, 2024].

[14] H. Kim, H. Y. Yeom, and Y. Son, "An efficient database backup and recovery scheme using write-ahead logging," in 2020 IEEE 13th International Conference on Cloud Computing (CLOUD), Oct. 2020, pp. 405-413.

[15] StorageDB [Online]. Available: https://github.com/alixchan/storagedb [Accessed: May 06, 2024].

[16] Key-value for metadata storing. Testing embedded databases. [Online]. Available: https://habr.com/ru/companies/raidix/articles/345076/ [Accessed: May 06, 2024].

[17] LMDBJava [Online]. Available: https://github.com/lmdbjava/benchmarks [Accessed: May 06, 2024].

[18] Benchmarking LevelDB vs. RocksDB vs. HyperLevelDB vs. LMDB Performance for InfluxDB [Online]. Available: https://www.influxdata.com/blog/benchmarking-leveldb-vs-rocksdb-vs-hyperleveldb-vs-lmdb-performance-for-influxdb/ [Accessed: May 06, 2024].

[19] Metadata fot rhe cluser: race of key-value heroes [Online]. Available: https://highload.ru/2017/abstracts/2974.html [Accessed: May 06, 2024].

[20] Use case example [Online]. Available: https://github.com/alixchan/storagedb/blob/master/src/main/resources/usage.md [Accessed: May 06, 2024].