Developing a LSM tree Time Series Storage Library in Golang

Nikita Tomilov[0000−0001−9325−0356]

ITMO University, Saint-Petersburg, Russia
mr@itmo.ru
http://micsecs.org/

Abstract. Due to the recent growth in popularity of the Internet of Things solutions, the amount of data being captured, stored and transferred is also significantly increasing. The concept of edge devices allows buffering of the time-series measurement data to help mitigating the network issues. One of the options to safely buffer the data on such a device within the currently running application is to use some kind of embedded database. However, those can have poor performance, especially on embedded computers. That is why in this paper an alternative solution, which involves the LSM tree data structure, was advised. The article describes the concept of an LSM tree-based storage for buffering time series data on an edge device within the Golang application. To demonstrate this concept, a GoLSM library was developed. Then, a comparative analysis to a traditional in-application data storage engine SQLite was performed. This research shows that the developed library provides faster data reads and data writes than SQLite as long as the timestamps in the time series data are linearly increasing, which is common for any data logging application.

Keywords: Time Series · LSM tree · SST · Golang.

1 Theoretical background

1.1 Time Series data

Typically, a time series data is the repeated measurement of parameters over time together with the times at which the measurements were made [1]. Time series often consist of measurements made at regular intervals, but the regularity of time intervals between measurements is not a requirement. As an example, the temperature measurements for the last week with the timestamp for each measurement is a time series temperature data. The time series data is most commonly used for analytical purposes, including machine learning for predictive analysis. A single value of such data could be both a direct measurement from a device or some calculated value, and as long as it has some sort of timestamp to it, the series of such values could be considered a time series.

1.2 IoT data and Edge computing

Nowadays the term "time-series data" is well-known due to the recent growth in popularity of the Internet of Things, or IoT, devices, and solutions, since an IoT device is often used to collect some measure in a form of time-series data. Often this data is transferred to a server for analytical and statistical purposes. However, in a large and complex monitoring system, such as an industrial IoT, the amount of data being generated causes various problems for transferring and analyzing this data. A popular way of extending the capabilities of IoT system is to introduce some form of intermediate devices called "edge" devices [2]. The traditional architecture of organizing a complicated IoT system is shown in Figure 1.

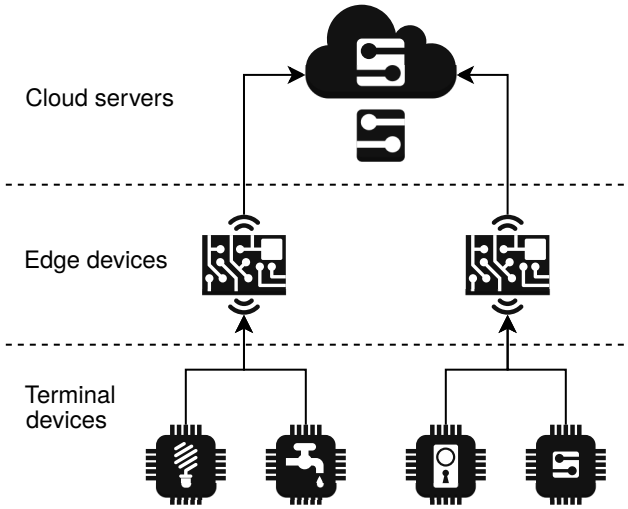


Fig. 1: An architecture of a complicated IoT system.

This architecture provides flexibility in terms of data collection. In case of problems with the network connection between the endpoint device and the cloud, data can be buffered in the edge device and then re-sent to the server when the connection is established. Therefore, an edge device should have the possibility to accumulate the data from the IoT devices for those periods of the network outage. However, often an edge device is not capable to run a proper database management system apart from other applications. So there has to be a way to embed the storing of time-series data to an existing application. From this point an assumption will be made that the application is written in a Go programming language because this language maintains the balance between complexity, being easier to use than C/C++, and resource inefficiency, being less demanding than Python.

1.3 In-app data storage

Traditionally, either some form of in-memory data structures or embedded databases are used to store data within the application. However, if data is critical and it is important not to lose it in case of a power outage, in-memory storage doesn't fit. An embedded database, or other forms of persistent data storage that is used within the application, reduces the access speed compared to any in-memory storage system. This article describes an LSM-based storage system. This type of system is providing the best of both worlds - persistent data storage with fast data access.

2 Implementation

2.1 LSM tree

An LSM tree, or log-structured merge-tree, is a key-value data structure with good performance characteristics. It is a good choice for providing indexed access to the data files, such as transaction log data or time series data. LSM tree maintains the data in two or more structures. Each of them is optimized for the media it is stored on, and the synchronization between the layers is done in batches [3].

A simple LSM tree consists of two layers, named C_0 and C_1. The main difference between these layers is that typically C_0 is an in-memory data structure, while C_1 should be stored on a disk. Therefore, C_1 usually is bigger than C_0, and when the amount of data in C_0 reaches a certain threshold, the data is merged to C_1. To maintain a suitable performance it is suggested both C_0 and C_1 have to be optimized for their application. The data has to be migrated efficiently, probably using algorithms that may be similar to merge sort.

To maintain this merging efficiency, it was decided to use SSTable as the C_1 level, and B-tree for the C_0. SSTable, or Sorted String Table, is a file that contains key-value pairs, sorted by key [4]. Using SSTable for storing time series data is a good solution if the data is being streamed from a monitoring system. Because of this, they are sorted by the timestamp, which is a good candidate for the key. The value for the SSTable could be the measurement itself. SSTable is always an immutable data structure, meaning that the data cannot be directly deleted from the file; it has to be marked as "deleted" and then removed during the compaction process. The compaction process is also used to remove the obsolete data if it has a specific time-to-live period.

2.2 Commitlog

Any application that is used to work with mission-critical data has to have the ability to consistently save the data in case of a power outage or any other unexpected termination of the application. To maintain this ability, a commit log mechanism is used. It is a write-only log of any inserts to the database. It

is written before the data is appended to the data files of the database. This mechanism is commonly used in any relational or non-relation DBMS.

Since the in-app storage system has to maintain this ability as well, it was necessary to implement the commit log alongside the LSM tree. In order to ensure that the entries in this commit log are persisted on the disk storage, the fsync syscall was used, which negatively affected the performance of the resulted storage system.

2.3 Implemented library

In order to implement the feature of storing time-series data within the Go application, the GoLSM library was developed. It provides mechanisms to persist and retrieve time-series data, and it uses a two-layer LSM tree as well as a commit log mechanism to store the data on disk. The architecture of this library is represented in Figure 2.

Since this library was initially developed for a particular subject area and particular usage, it has a number of limitations. For example, it has no functions to delete the data; instead, it is supposed to save the measurement with a particular expiration point, after which the data will be automatically removed during the compaction process. The data that is being stored using GoLSM should consist of one or multiple measurements; each measurement is represented by a tag name, which could be an identifier of a sensor or the measurement device, origin, which is the timestamp when the measurement was captured, and the measurement value, which is stored as a byte array. This byte array can vary in size. It makes the storage of each measurement a more complicated procedure.

As seen, the storage system consists of two layers, in-memory layer and persistent storage layer. The in-memory layer is based on a B-tree implementation by Google [5] . It stores a small portion of the data of a configurable size. The storage layer consists of a commit log manager and an SSTable manager. The commit log manager maintains the two commit log files; while one is used to write the current data, another one is used to append the previously written data to the SSTable files, which are managed by SSTable Manager. Each SSTable file contains its own tag, and it also has a dedicated in-memory index, which is also based on a B-tree. This index is used to speed up the retrieval of the data from the SSTable when the requested time range is bigger than what is stored on an in-memory layer.

3 Comparison against SQLite

3.1 Test methodology

To compare the LSM solution with SQLite, a simple storage system was developed. It has a database that consists of two tables. The first table is called Measurement and it is used to store the measurements. Each measurement is represented by its key, timestamp and value, while the key is the primary ID of

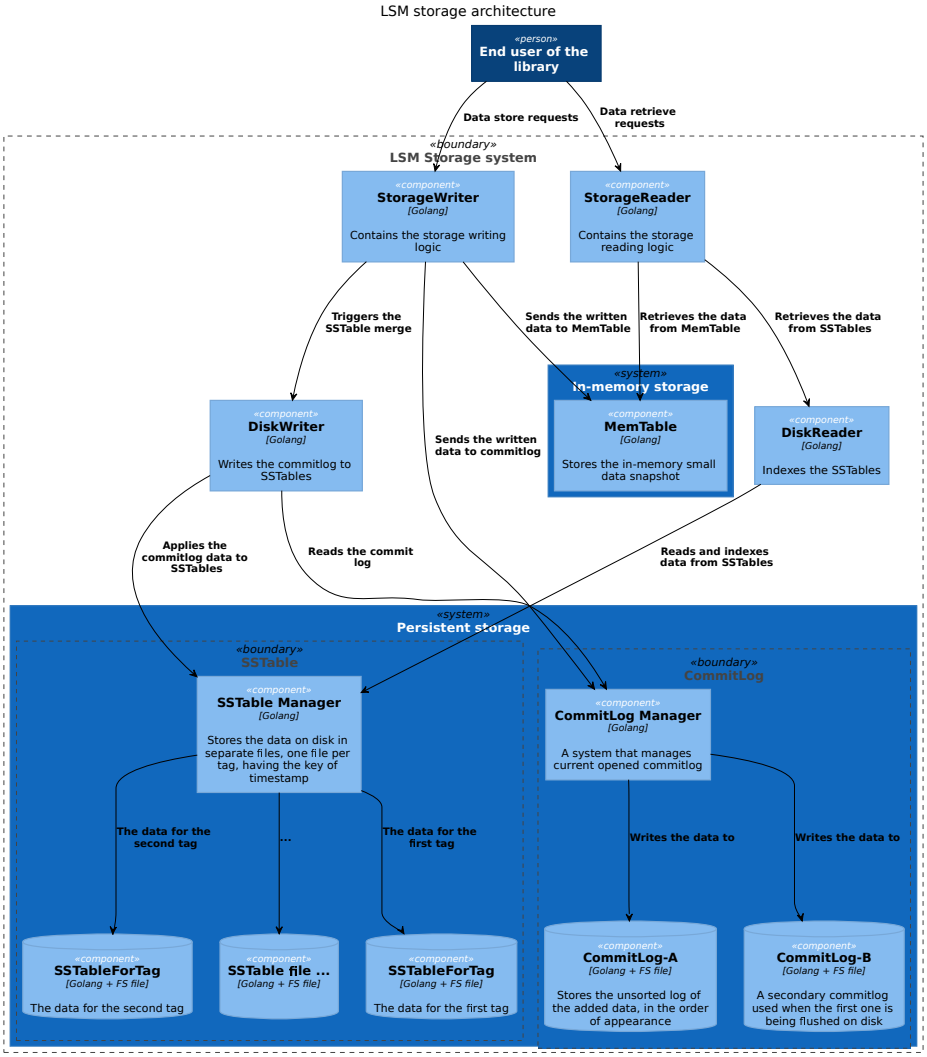


Fig. 2: An architecture of a GoLSM library.

MeasurementMeta entity, that is stored in the second table. This entity stores the tag name of the measurement; therefore it is possible to perform search operations filtering by numeric column instead of a text column. The Measurement table has indexes on both the key and the timestamp columns.

For the following benchmarks, a sample synthetic setup was used. This setup has 10 sensors that emit data at a sampling frequency of 1Hz. For the reading benchmarks, the data for those 10 tags were generated for the time range of three hours. Therefore, the SQLite database had 108000 entries, or points, in total, which means 10800 points per each tag. The LSM library has 108000 entries as

well, splitted across 10 files per each tag, having 10800 points in each one. For the writing benchmarks, the data for those 10 tags is generated for various time ranges and then stored in both storage engines.

In order to benchmark both storage engines, a standard Go benchmarking mechanism was used. It runs the target code multiple times until the benchmark function lasts long enough to be measured reliably [6]. It produces the result that is measured in ns/op, nanoseconds per iteration; for the following benchmarks, iteration is a single storage read or storage write call.

All benchmarks were running on a PC with Intel Core i5-8400 running Ubuntu 18.04, the data files for both storages were stored on an NVMe SSD.

3.2 Reading from full range of data

This benchmark measures the read time for various time ranges, while the operation is performed across full three hours of measurements. The results of this benchmark are available in Figure 3. As seen, the GoLSM is about twice as fast as SQLite engine on read requests. It is worth mentioning that the C_0 level of LSM tree is not playing a significant role in speeding up the data retrieval, since the time range is picked randomly from across all three hours of measurements while the C_0 level only contains the last two minutes of measurements.

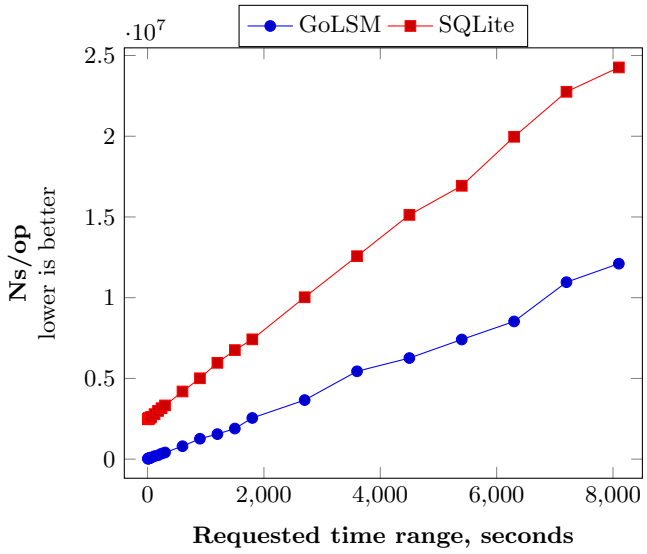


Fig. 3: Reading from full range of data.

As seen, the LSM storage is up to 50 times faster than SQLite on a small request range (51251 ns/op for GoLSM and 2517353 ns/op for SQLite on 25s range) and up to twice as fast on a big request range (12106688 ns/op and

24262132 ns/op on 2h15min range). This big difference may be caused by using an in-memory index over each SST file that performed better than indexing within the SQLite. An efficiency increase for the small request range may be caused by the fact that it is more likely for the small requested range to fit within the C_0 level of the LSM tree, so the slow retrieval from the SST is not called.

3.3 Reading from the last three minutes

This benchmark measures the read time for various time ranges, while the operation is performed across the latest three minutes of measurements. The results of this benchmark are shown in Figure 4. Since the C_0 level of the LSM tree contains the last two minutes of measurements, it is more likely that the request will fit the C_0 level without the need to request data from the C_1 level. Therefore, data retrieval is up to twice as fast as retrieving the data from across all three hours of measurements, as seen in Figure 5.

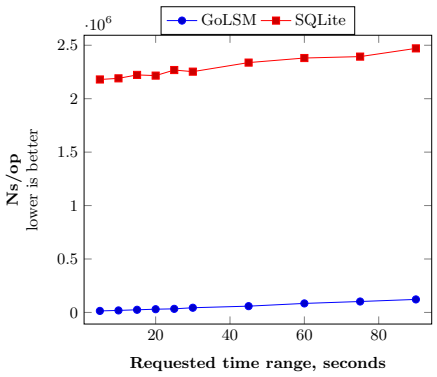


Fig. 4: Reading from full range of data.

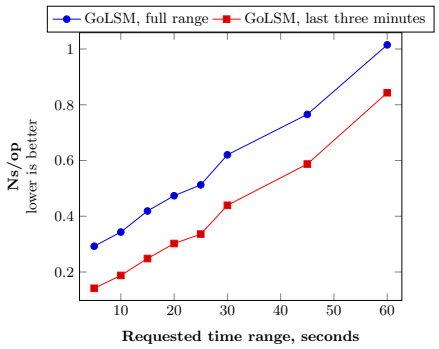
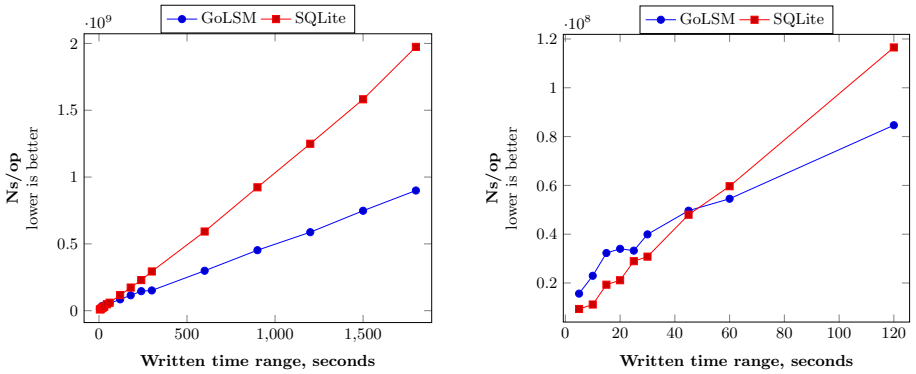


Fig. 5: Difference in GoLSM speed on the latest data.

If the data retrieval pattern is retrieving only the latest data, increasing the capacity of C_0 can drastically improve reading performance and reduce the reading load on the persistent disk storage.

3.4 Writing linear data

This benchmark measures the write time for various time ranges. While the data is linear the minimal timestamp of the next data batch to be written is always more than the maximum timestamp of the previously written batch. It means, for each SST within the GoLSM engine the data is only appended to the end of the SST file without the need to redo the sorting of the SST file. The results of this benchmark are available in Figure 6.



(a) Over big range (b) Over small range

Fig. 6: Writing linear data.

In Figure 6, (a) it is clearly seen that GoLSM writes data twice as fast as SQLite over the big time range that is being written. However, in Figure 6, (b) for the small time range the GoLSM is actually slower. The reason for this is that for LSM was measured the time until the data is actually written to the commit log, and not directly to the SST files; and when the amount of entries in the commit log reaches a certain threshold, it has to be transferred to the SST, causing spikes in write time. However, for big time ranges being written this overhead is not important compared to generally slow batch inserts to SQLite.

It is worth mentioning that for inserting to SQLite it is necessary to construct the SQL Insert statement that involves converting a double value to string. This operation is extremely slow for big batch inserts (a batch of 10 tags for 30 minutes is 18000 points), and using ORM such as GORM does not improve the situation.

3.5 Writing randomized data

This benchmark measures the write time for various time ranges, while the data is random, there are no guarantees that all of the next batch timestamps are bigger than the timestamps of the previously written batch. So the batches could occur before or after one another in terms of their timestamps. It means each time when the data is transferred from the commit log file to SST files the engine has to resort the whole SST file, slowing down the writing process. The results of this benchmark are available in Figure 7. The comparison of writing the data to LSM when the data is either linear or randomized is available in Figure 8.

As seen in Figure 7, the overhead of resorting the SST files on every data write is so big that it outperforms the slow batch inserts to SQLite. The comparison of the data writes with this overhead to data writes without the overhead is in Figure 8. When the timestamps of the data are always linearly increasing, shows that the inserts are almost up to three times slower on large time range

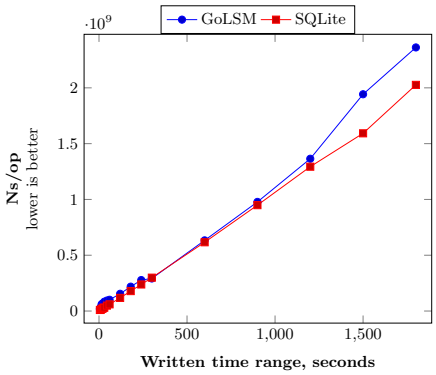


Fig. 7: Writing randomized data.

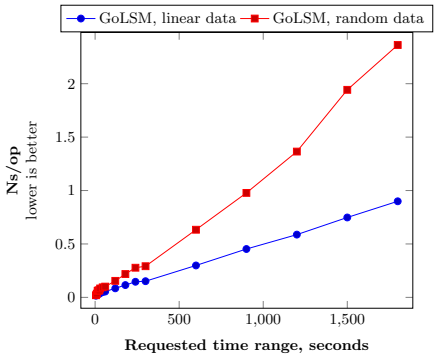


Fig. 8: Difference in GoLSM speed when writing linear and randomized data.

writes (747870163 ns/op, or 747 ms, for writing 25 minutes of linear data, and 1942545058 ns/op, or 1942 ms, for writing 25 minutes of randomized data).

4 Conclusion

As shown in the article, the developed LSM storage library provides faster write and read operations for the time series data than SQLite. Also, it allows keeping data in case of unexpected application crashes due to commit log mechanism. In the best cases, this library is up to 50 times faster than SQLite for read operations, thanks to its C_0 level. It effectively serves as an in-memory cache, and up to 2 times faster than SQLite on write operations, due to the way how writing mechanisms work. However, the writing advantage is only valid if the time series data always have increasing timestamps. So the next written batch of data has bigger timestamp than the already written data for each tag. If this is not guaranteed, the data writing process is slowed down by an order of magnitude, making it slower than using SQLite. However, the benefits of multi-level storage for reading the data are still present. Varying the size of C_0 level can give the end-user the required balance between reading speed and RAM usage.

Using in any data logging use-case scenario, where the application has to store or buffer data that is continuously retrieved from various sensors, the timestamps for this data are naturally increasing. It removes the potential problem with writing randomized data and making the LSM-based storage engine a good alternative to SQLite, improving both read and write speeds.

The potential improvements of the developed library include splitting a single SST file per tag to multiple files, reducing the resorting time when the timestamp-randomized was written, and adjusting the in-memory layer so that its size can be different for different tags, for the use-cases where certain tags are requested more frequently and/or for bigger time ranges.

References

1. Dunning, T., Friedman, E.: Time Series Databases: New Ways to Store and Access Data. O'Reilly Media, CA, USA (2014)
2. B. Chen, J. Wan, A. Celesti, D. Li, H. Abbas and Q. Zhang, "Edge Computing in IoT-Based Manufacturing," in IEEE Communications Magazine, vol. 56, no. 9, pp. 103–109 (2018). https://doi.org/10.1109/MCOM.2018.1701231
3. O'Neil, P., Cheng, E., Gawlick, D. et al. The log-structured merge-tree (LSM-tree). Acta Informatica 33, 351–385 (1996). https://doi.org/10.1007/s002360050048
4. Adhikari, M., Kar, S. NoSQL databases. In Handbook of Research on Securing Cloud-Based Databases with Biometric Applications (pp. 109-152). IGI Global. (2015)
5. BTree implementation for Go, https://github.com/google/btree. Last accessed 31 Oct 2020
6. Go benchmark documentation, https://golang.org/pkg/testing/. Last accessed 1 Nov 2020