201801030_Term_Paper

by Nikhil Mehta

Submission date: 10-Apr-2021 11:18AM (UTC+0530)

Submission ID: 1554456174

File name: 201801030_Term_Paper.pdf (81.63K)

Word count: 1812

Character count: 9177

SlimDB: Efficient space optimized key value storage for semi sorted data

Term paper

Members: Nikhil Mehta (201801030)

Introduction

Topic

In this paper, we would be discussing SlimDB, a particular implementation of Log-structured merge(LSM) trees and why SlimDB is better than its counterparts like LevelDB.

Motivation

In today's era, key-value store is the most preferred way for many high write and read intensive applications. One such write-optimized index which is widespread is the LSM tree. The existing implementations of it have drawbacks such as high write-amplifications, it is poor at read operations,and works poorly in write intensive applications. The main motto of this study is to propose a space efficient storage i.e. SlimDB. Moreover the ratio of read and write operations will be in proportion making it a better choice.

Statement of Purpose

This paper will primarily focus on the following questions:

1) Can we use less memory for filters and indexes with LSM.
2) Can we replace the traditional filters and indexes in LSM to make it more efficient in terms of memory cost, lookup, insertion cost etc.

Description

About LSM trees

LSM is used over B+ trees for optimized indexes because they use excessive buffering and sequential access patterns for writing while in B+ trees, even for small write operations, many random writes are performed in turn leading to wastage of storage. But still the existing implementations of LSM are challenging in case of high performance and scalability. It is because nowadays, small entries are widespread and they have large metadata[1] which leads to excess memory usage. Many applications support key value orders like sorted order or hash order. But not all applications need sorted key order and so in that case we need not iterate through all entries, rather the primary key for such key-value orders is broken in fragments(i.e prefix x and suffix y) and then we iterate only on one fragment which is known as semi-sorted workload. Most write operations performed are non-blind writes. So it's necessary that we've a store engine which is optimized for read and write operations in proportion. SlimDB is preferred as a storage engine with semi-sorted keys since it exploits the above mentioned techniques to optimize its read,write, and store performance.

LevelDB implementation of LSM

LSM tree contains Sorted String(SS)tables[2], which only append data in sequential manner in a sorted order. These SS tables are arranged in multiple levels in an LSM tree. It has been found that sequential writes are much faster than random writes and so LSM exploits this in a way that it has excessive buffering and it performs sequential writes in it. After the current level's limit is reached, the new entry is then migrated to the next level and this process is known as compaction[3]. The data in the new level is 'r' times the data in the previous level. 'r' ranges from 8 to 16. In an SS table, the compaction strategy is such that when the data limit of an SS table is reached, the data is then migrated to the next level by combining arranging the SS tables of the two levels.

Stepped-Merge Algorithm

Stepped-Merge is likewise a variation of LSM which utilizes an alternate compaction methodology. In this the SS tables at each level are divided in 'r' sublevels and while compacting a level to the next level, the SS tables are not merge-sorted with the next level rather, they're 'r' way merge-sorted in the same level and then inserted into the next level. This reduces the write amplification.

Bloom filters and indexes

Even though merging SS tables would not reduce the read latency everytime and so to make read operations more efficient, each SS table is embedded with bloom filters and bloom indexes. Bloom filter is a probabilistic data structure[4] which tells if a particular entry is present in the data block or not. Bloom index contains the largest keys of all data blocks.

Design and Implementation of SlimDB

SlimDB uses *three level* block index and *multi-level* cuckoo filter in place of block indexes and bloom filters respectively. Three level block indexes are uniquely upgraded for semi-sorted data and multi level cuckoo filter returns the latest sub-level which contains the entry. Multi level cuckoo filter is also a probabilistic data structure meaning it can give false results. In SlimDB's implementation, various combinations of different kinds of filters and indexes and their own pros and cons. If we combine a multi-level cuckoo filter with stepped-merge, it has low write amplifications but utilizes more memory resources.

Space-efficiency of Sorted String table indexes

In LevelDB, if there is no caching present, a read operation basically takes two block reads: first loading the block index and second reading the data block. The block index in LevelDB has around 8 bits reserved for a particular key. So to optimize it and without changing the LSM organization, SlimDB supports entry for semi-sorted data. This allows us to trade a few CPU cycles for fewer storage and high cache hit rate. SlimDB uses prefix or suffix of the semi-sorted data to index the entry and for that Entropy Coded Trie(ECT) is used. Compressing the prefix and suffix and storing it in ECT reduces the index memory consumed by SlimDB in comparison to LevelDB by at least 4 times.

Entropy-Coded Trie

ECT are radix tree data structures in which each leaf node represents one key and the edges represent the longest common prefix for the subtree following the edge. ECT just keeps the information which is sufficient to differentiate amongst different keys. ECT maps each input key of an array which is sorted in hash order and assigns it a rank in [0,n-1].

Three-Level Index Design

In SlimDB, the semi-ordering of keys is important and so to maintain it, we cannot alone use a single ECT, rather we'll have to use two ECTs to compress both the fragments of the key. This compression leads to a three-level index system which in turn helps us in finding the desired data entry from a data block with less consumption of space. Constructing a three level index system is based on the compression of a vanilla block index which contains the first and last key of each data block. The first level is the prefix array which consists only of the prefixes which uniquely determines all the entries from all the data blocks. ECT is used to compress this prefix array and then store it. This level will allow us to map each entry with its corresponding prefix value in the array which will serve as the input to the second level of the three-level file framework. The second level then takes the position from the prefix array and guides it to the SS tables which have data entries with that prefix value. The second level is an integer array which basically stores the last block number which has the corresponding prefix value in it. Because of these two levels, it is possible to find out the SS tables which might contain the desired key's prefix. The third level consists of suffixes which when combined with the results obtained from the first two levels and when performed binary search on the range of blocks gives us the desired data's block.

Multi-Leveled Cuckoo Filter

Data structures used for indexing which are probabilistic in nature, like the bloom filters have a false positive rate and which then leads to increased read latency. Multi-level cuckoo filter is such which can reduce the number of reads in cases when there are chances of false positive answers. Cuckoo filters basically consist of a hash table where each key has two candidate buckets in the table. So if data needs to be inserted in the hash table, the corresponding buckets are checked and then inserted into the bucket if either of them is empty. If the insertion process goes in an infinite loop, the hash functions of the candidate buckets are recalculated [5]. Here the hash tables store the full key which is a tradeoff compared to bloom filters. Rather a fingerprint could be stored which will remove this tradeoff. Cuckoo filter does so by calculating the entry's alternative bucket position and not the key.The two hashes which are created by following the algorithm are very much interdependent and so there are high chances of failure but still with an appropriate fingerprint size and the bucket size,it is more space proficient than the bloom filters. Multi-level cuckoo filters consist of two tables, the primary table and the secondary table. The primary table is a variation of cuckoo filter which along with the fingerprint also stores the level number of the LSM tree where the data can be present. The secondary table is helpful to reduce the read latency in cases where the fingerprint in the primary table for two or more keys collide. In such cases the secondary table along with the fingerprint stores the entire key which avoids unnecessary reads in case of the same fingerprint value and yields the result quicker. Insertion and lookup are performed keeping in mind the consistency of the primary,secondary tables and the LSM trees is maintained and hence the algorithms for lookup and insertion are devised in that manner.

Implementation of SlimDB

The implementation of SlimDB is similar to that of RocksDB, just that there are about 5000 lines of change in the code. SlimDB has SS tables and it maintains iterators in the SS tables to scan the prefixes which is slower than traditional LevelDB.

Conclusion

From the above study, we can conclude that SlimDB is good in many terms as compared to their counterparts like it has improved upon its lookup cost because it didn't resort to the traditional bloom filter and block indexes. SlimDB requires less main memory because of its optimal methods of storage and improved write operations which require less memory. SlimDB is efficient at data storage and hence it scales well when it comes to data scalability.

Future Works

Currently, many of the well known companies like Facebook, Google, use LSM index for their data value storage. Facebook uses RocksDB, Google uses LevelDB. SlimDB has its own advantages and disadvantages in comparison to LevelDB,RocksDb, etc. So SlimDB could be tried to replace LevelDB or RocksDB in a few cases at companies like Google and Facebook and then measure its performance.

References

[1] https://www.pdl.cmu.edu/PDL-FTP/FS/CMU-PDL-19-102.pdf.

[2] https://www.igvita.com/2012/02/06/sstable-and-log-structured-storage-leveldb/

[3]https://www.microsoft.com/en-us/research/uploads/prod/2019/09/SILK-Preventing-Latency-Spikes-in-Log-Structured-Merge-Key-Value-Stores.pdf

[4]https://llimllib.github.io/bloomfilter-tutorial/#:~:text=A%20Bloom%20filter%20is%20a,may%20be%20in%20the%20set.

[5]https://www.geeksforgeeks.org/cuckoo-hashing/

* Ren, Kai, et al. "SlimDB: a space-efficient key-value storage engine for semi-sorted data." Proceedings of the VLDB Endowment 10.13 (2017): 2037-2048.

201801030_Term_Paper

ORIGINALITY REPORT

7%	1%	7%	%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Kai Ren, Qing Zheng, Joy Arulraj, Garth Gibson. "SlimDB", Proceedings of the VLDB Endowment, 2017 Publication	6%
2	Jianchuang Li, Peiquan Jin, Shouhong Wan. "Adaptive Lazy Compaction with High Stability and Low Latency for Data-Intensive Systems", 2020 IEEE International Conference on Big Data (Big Data), 2020 Publication	1%
3	Joonyong Jeong, Jaewook Kwak, Daeyong Lee, Seungdo Choi, Jungkeol Lee, Jungwook Choi, Yong Ho Song, Yong Ho Song. "Level Aware Data Placement Technique for Hybrid NAND Flash Storage of Log-Structured Merge-Tree Based Key-Value Store System", IEEE Access, 2020 Publication	<1%
4	zhu45.org Internet Source	<1%

Exclude quotes	On	Exclude matches	Off
Exclude bibliography	On		