CLeveldb: Coalesced Leveldb for Small Data

Khikmatullo Tulkinbekov, Mehdi Pirahandeh and Deok-Hwan Kim*
Department of Electronic Engineering, Inha University
Incheon, South Korea
tillo@inha.edu, mehdi@inha.ac.kr, deokhwan@inha.ac.kr

***Abstract*—** This paper proposes a coalesced key-value pairs generation technique for CLeveldb. The proposed method employs an additional buffer cache and Upper Meta file which increases write performance of CLeveldb by merging multiple small data into a larger chunk and provides consistency by logging each transaction in Upper Meta file. The preliminary experimental results show that, the proposed method improves the writing performance of small data by 6 times as compared with Leveldb as input key-value item size becomes smaller.

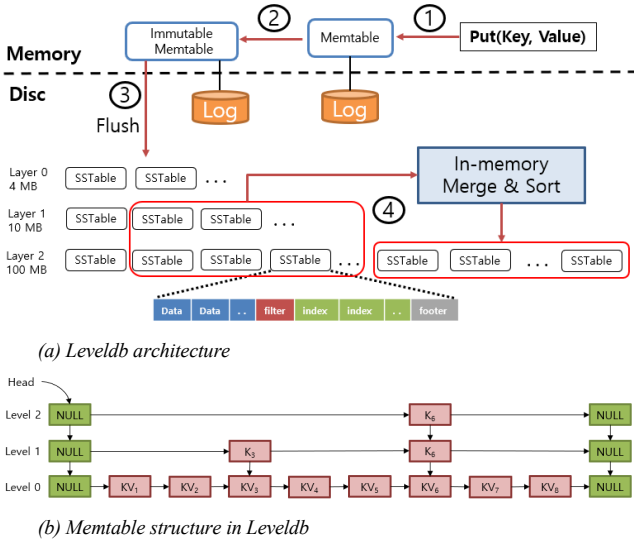
Keywords—Key-Value Store; LSM-Tree; Leveldb; IoT data

I. INTRODUCTION

In recent years, Key-Value (KV) stores became more popular for its flexibility and speed over traditional relational databases. Compared to relational databases, KV stores do not rely on SQL queries but provides public APIs for data transactions.

LSM-Tree [1] is one of the most popular data structures used in key-value stores and benefits by implementing a multilevel architecture. Google's Leveldb [2] is one of the most obvious and simple implementations of LSM-Tree as a KV store. Fig. 1(a) shows the Memtables and String Sorted Tables (SSTable) in the Leveldb architecture. Memtable is a memory resident Skip-Tree [3] as shown in Fig. 1 (b), which sorts the data using the key once it is inserted. SSTable is a byte-stream file where the data, filter, index and footer blocks are written sequentially in sorted order. KV pairs are located in data blocks while each index block keeps a reference for the corresponding data block. Filter block is used to search the KV pairs from SSTable and the footer indicates the end of file. In Fig. 1(a), put process consist of four steps. In the first step, put process begins with allocating KV items into Memtable and logging meta information to provide transaction consistency. When the Memtable is full, it becomes immutable and a new Memtable is created to load incoming data in the second step. KV pairs in Immutable Memtable are flushed into SSTable and the log file is deleted in the third step. LevelDB constructs several layers of disk components, each layer being 10 times larger than the previous layer and sorted by key except layer 0. When data is flushed to layer 0, so there might be overlapping and unordered keys. In step 4 of put process, SSTables of the upper layer is compacted into lower layer until it reaches the lowest layer. In compaction process, two neighbored layers are selected and read into memory. After merging and sorting the data, new SSTables are rewritten to lower layer.

*Corresponding Author: Deok-Hwan Kim *(deokhwan@inha.ac.kr)*



(a) Leveldb architecture

(b) Memtable structure in Leveldb

Fig. 1. Leveldb architecture and Memtable structure

Because of compaction technique, lots of SSTable files are read and rewritten in disc storage. Since the single SSTable contain certain number of KV items, its size is strongly dependent of input KV item size. In other words, small sized KV items will generate a lot of small SSTable files. In this paper we assume that the SSD is used as a persistent disk storage. As it is obvious, the SSD's small file I / O performance is low. There are bunch of works done to improve Leveldb performance [4], [5] and to improve disk speed for small files [6], [7]. But all works still have long latency issues when processing small files. In this paper, we propose CLeveldb to create an additional in-memory Skip-Tree-based buffering technique that merges several small KV items into one big item. This method significantly reduces the number of SSTables in a disk component that avoids the small sized SSTable files generated by LevelDB and reduces compression latency. To perform the grouping mechanism, we introduce group based Memtable and SSTable architecture. CLeveldb also creates an Upper Meta (UMeta) file that guarantees the reliability of coalesced data. We set the database to 100 thousand KV items and changed the KV item size from 50 bytes to 16KB and did experiments on Leveldb and CLeveldb. According to our experimental results, CLeveldb outperforms Leveldb up to 6 times as the KV item size gets smaller. Also, for bigger sized KV items, CLeveldb performs almost as same as Leveldb.

Our paper is organized as follows. In section 2 we will talk about the background of the research and main motivations to develop CLeveldb. In section 3 we will give a detailed explanation on CLeveldb architecture and section 4 provides an evaluation and conclusions of the proposed technique.

II. Background And Motivations

Leveldb is one of the most popular LSM-based open source KV store. Int this paper, we have tested Leveldb write performance. We set the workload to 100 thousand KV pairs and the size of a single KV pairs varied from 50 bytes to 16KB. In Fig. 2, experimental results show that the write throughput of Leveldb increases from 4.5MB/s to 18MB/s in the first interval where the input KV pair size varied from 50 bytes to 4KB. In addition, the Leveldb has a steady write throughput of from 18MB/s for the second interval where the KV pairs size varied from 4KB to 16KB.

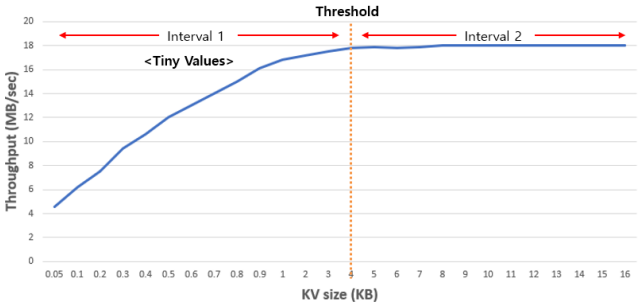


Fig. 2. Leveldb write throughput on different workload

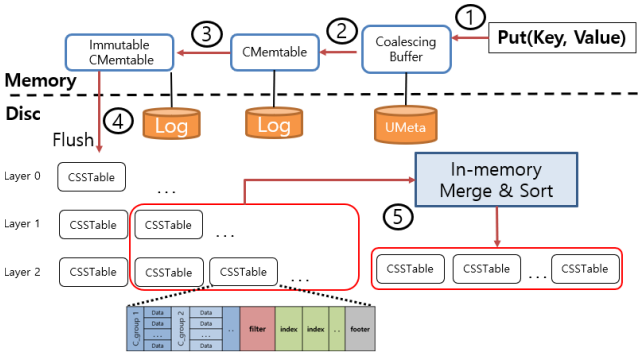
This investigation motivated us to consider about threshold value in implementing CLeveldb. The threshold is a size of input KV item that stabilizes the write throughput at one point in Leveldb. From Fig. 2, we have chosen the point where two intervals meet which is 4KB as threshold. This threshold serves as a coalesced KV items' group size.

III. CLevelDB Architecture

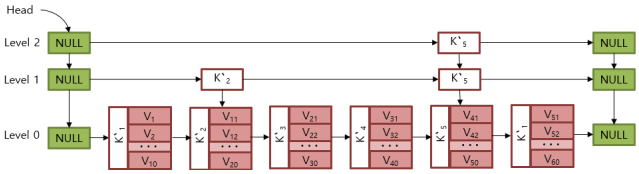
CLeveldb employs a coalescing buffer in memory as shown in Fig. 3(a). The purpose of this buffer is to merge multiple values into one group and generate a unique key for the coalesced KV items. The coalescing buffer is a Skip-Tree-based data structure which sorts KV items and merges them by group key once they are inserted to the buffer. In order to keep data consistency, each transaction in coalescing buffer is saved to UMeta file as shown in Fig. 3(c). UMeta file keeps the original key of the item together with its group key. Also, it keeps the index of KV item in the merged group and its size in order to read the exact data from the group. The grouping technique changes the Memtable and SSTable structure in CLeveldb. CLeveldb uses Coalesced Memtables (CMemtable) and SSTables (CSSTable). The structure of CMemtable and CSSTable is similar to those of Memtable and SSTable in Leveldb whereas format of data block in CLeveldb is different from data block in Leveldb. As shown in Fig. 3(b), the Skip-Tree of CMemtable holds multiple values as one group and uses the generated key for the group of KV items in data transactions. In CSSTable, each data block holds multiple KV items as one group as shown in Fig. 3(a).

A. Write process

CLeveldb put process involves 5 steps as shown in Fig. 3(a). In the first step, incoming KV items are inserted to coalescing buffer. In this step, multiple values are grouped until the group size reaches the threshold value and new unique key for the group is generated. All transactions are saved to UMeta file. The grouped data is moved to CMemtable with the key generated in step 2. The remaining process is as same as in Leveldb except CLeveldb processes the grouped data as a single KV item in flushing and compaction steps.



(a) CLeveldb architecture



(b) CMemtable structure in CLeveldb

<key, c_key, index, size>
<k_1, c_1, 0, 50>
<k_2, c_1, 50, 120>
<k_3, c_1, 170, 70>
. . .

File view

key	c_key	index	size
k_1	c_1	0	50
k_2	c_1	50	120
k_3	c_1	170	70
. . .	. . .	. . .	. . .

Table view

(c) UMeta structure in CLeveldb

Fig. 3. CLeveldb architecture, CMemtable and UMeta structure

B. Read process

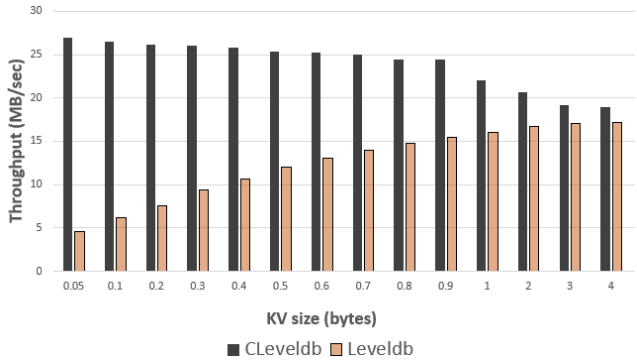
When reading the data from CLeveldb, UMeta file plays the main role. When the get request comes, the requested key is searched from UMeta file. Since the data inside the UMeta file is already sorted by key, we can provide the best search performance on it. If the key is not found from UMeta, it means the searched data is not available in KV store. If the key is available in UMeta, the group where the KV item is located is loaded from CLeveldb according to group id. The exact KV item is extracted from the loaded group according to index and the size.

IV. Preliminary Experiments

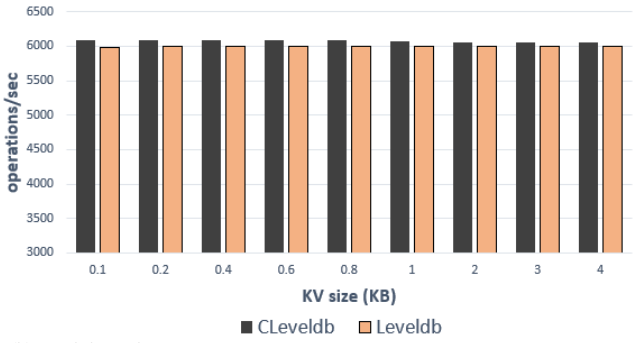
We implemented a coalesced buffer in CLeveldb to group multiple tiny KV items into a threshold value sized items and reduce the number of SSTable files in disc component. In evaluating the proposed method, we have set the system configuration as described in Table 1. Moreover, we have used fixed set workloads in order to check CLeveldb performance over Leveldb.

TABLE I. System Configuration

Category	*Value*
CPU	AMD Ryzen 7 1700 eight-core
DRAM	DDR3 SDRAM / 4GB
SSD	Samsung SSD 860 EVO / 500 GB
OS	Ubuntu 18.04.1 LTS



(a) Write throughput



(b) Read throughput

Fig. 4. Evaluations of CLeveldb and Leveldb

Workload configuration is changed according to KV item size during evaluation. Since we have implemented the CLeveldb to increase the performance of Leveldb for tiny data, we have assumed the workload only includes files smaller than threshold value and set the number of KV items to in workload to 100 thousand. Firstly, we set the single KV item size to 50 bytes and increased the size up to 4KB which is our threshold value. Then, recorded the CLeveldb and Leveldb put process performance as shown in Fig. 4(a). From preliminary experiment results, we can conclude that with grouping technique, CLeveldb performs at 27MB/sec while Leveldb performs at 4.5MB/sec which is 6 times faster when the single KV item size is equal to 50 bytes. As the input data size increases, this difference factor decreases until the threshold value and Cleveldb performs almost similar to Leveldb at 4KB data.

We have also evaluated reading performance of the CLeveldb and Leveldb and the result of our preliminary experiments results are shown in Fig. 4(b). As we can see from the results, CLeveldb can perform almost same as Leveldb in read operations. Even though reading data from Cleveldb requires more steps in reading UMeta file and extracting the tiny KV item from its group, the coalescing technique of CLeveldb significantly reduces the number of SSTables and retrieving the grouped data from persistent storage can be done faster than Leveldb.

V. Conclusions and Future Work

In this paper, we proposed a coalesced KV items generation technique for CLeveldb. We have also employed a UMeta file to keep consistency of grouped data. Preliminary experiments showed that, the proposed method outperforms the Leveldb put performance by 6 times as the input data becomes smaller.

As a future work, we are planning to reduce compaction latency of Leveldb. By this way, we can increase the put performance of Leveldb significantly not only for tiny, but for all size data.

Acknowledgment

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2018R1D1A1B07042602) and in part by Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korea government(MSIT) (No.2019-0-00064, Intelligent Mobile Edge Cloud Solution for Connected Car, No.2019-0-00240, Deep Partition-and-Merge: Merging and Splitting Deep Neural Networks on Smart Embedded Devices for Real Time Interface)

References

[1] P.O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-Tree)", Acta Informatica, vol 33, pp. 351-385, 1996.

[2] (Jan, 2014). A fast and Lightweight Key-Value Store Library by Google. [Online]. Available: http://code.google.com/p/leveldb

[3] Xavier Messeguer, "Skip Trees, an alternative data structure to skip lists in aconcurrent approach", Theoretical Informatics and Applications (vol. 31, 1997, pp. 251-269).

[4] W. Shang, Y. Xu, Y, Li, Y. Zhang, D. Li, "FlameDB: A key-value store with grouped level structure and heterogeneous bloom filter", IEEEAccess, May, 2018.

[5] H. Sun, W. Liu, Z. Qiao, S. Fu, W. Shi, "DStore: A holistic key-value store exploring near-data processing and on-demand scheduling for compaction optimization", IEEEAccess, November, 2018.

[6] M. Pirahandeh, D.H. Kim, "High performance GPU-based parity computing scheduler in storage applications", Concurrency and Computation: Practice and Experience 29 (4), e 3889, 2017.

[7] M. Pirahandeh, D.H. Kim, "Delta RAID to Enhance Recovery and Small-Write Performance of RAID storages", Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, 2016