Klaytn Improvement Reserve Proposal
Improving the Read Performance of KV Database in Klaytn

Date 2021.12.01

Prof. Hyeonsang Eom's team
Department of Computer Science and Engineering
Seoul National University
Seoul, Korea (Rep. of)

Summary

As the scales of blockchain systems become larger, the amount of data processed in the blockchain system increases exponentially. Due to a huge amount of data and lack of improvement in the database system, issues of I/O performance degradation in such a blockchain environment become serious. The types and characteristics of data read/written in the blockchain system are different from those of other systems, and therefore optimization that targets block processing is demanded.

In this proposal, we aim to solve the I/O performance degradation. To achieve this, we plan to conduct two main tasks: analysis of I/O performance of KV (Key-Value) database in Klaytn and design of an optimized algorithm and a data structure for Klaytn. First, we plan to analyze the I/O performance of the database by using multiple types of storage devices to identify performance bottleneck points, evaluate compaction operations, and examine the data structure implemented in the database. Furthermore, we will compare the current database with another database to find optimized features that can be applied to the new design. Finally, we aim to propose our new design and implementation of the database that effectively resolves the issues of read performance bottleneck.

Team Introduction

This project is to be carried out by a team from Distributed Computing Systems Lab, Department of Computer Science and Engineering, Seoul National University. This team is led by prof. Hyeonsang Eom. The main research domain of the team includes high performance storage/computing, blockchain, distributed computing, security, and GPU-based acceleration.

Below is the detailed information of team members.

Position	Name	Detail
Principal Investigator	Eom, Hyeongsang	B.S in Computer Engineering, Seoul National University M.S/Ph.D. in Computer Science, University of Maryland at College Park, USA Professor, Seoul National University
Co-Principal Investigator	Kim, Sunggon	B.S in Computer Science, University of Wisconsin - Madison, USA Ph. D in Computer Science and Engineering, Seoul National University Post-Doc researcher, Seoul National University
Investigator	Bang, Jiwoo	B.S in Computer Science, Seoul National University Ph. D student, Seoul National University
Investigator	Shin, Hyunil	B.S in Computer Science, Seoul National University Ph. D student, Seoul National University
Investigator	Kim, Chungyong	B.S in Computer Engineering, University of Seoul Ph. D student in Computer Science and Engineering, Seoul National University
Investigator	Han, Jongbeen	B.S in Computer Engineering, Hansung University M.S in Computer Science and Engineering, Seoul National University Ph. D student, Seoul National University
Investigator	Sung, Dong Kyu	B.S in Computer Science, University of Minnesota Twin Cities, USA Ph. D student, Seoul National University
Investigator	Song, Mansub	B.S in Computer Engineering, Yeungnam University Ph. D student, Seoul National University
Investigator	Seo, Yunhyeong	B.S in Computer Science, Kyungpook National University M.S student, Seoul National University
Investigator	Ban, Jihoon	B.S in Computer Science, Indiana University at Bloomington, USA Intern, Distributed Computing Systems Lab, Seoul National University

Motivation

The blockchain has been widely adopted in many industries due to its secure, decentralized manner. As the scales of blockchain systems become larger, many issues are revealed. One of them is the I/O performance bottleneck, especially from blockchain platforms such as Ethereum that store data regarding blocks into databases. Klaytn is a global blockchain platform used by many highly reputable brands which is derived from Ethereum. Therefore, it suffers from the same problem. As a high number of clients use the blockchain system, a huge amount of data is generated. For processing transactions, the system needs to write or read the corresponding data; due to the high I/O rate on the database, it causes I/O performance degradation, especially that of read.

Ethereum uses a KV store as its database to provide data storage. A large number of global block data entries across the platform are stored and retrieved in/from the internal KV storage. Since keys are hash values of data and alphabetical ordering of hashed keys by the database cannot lead to keeping relevant data together, it causes I/O performance issues. This even results in overall performance degradation of the system. To resolve this issue, we need to examine the I/O characteristics of workloads executed in the system and find out the tendency of write/read operations.

There are many attempts for optimizing KV stores for I/O performance in a general context; however, there are only a few of them in the blockchain environment.

Raju et al. [1] suggests a modified data structure that reduces read and write amplification while maintaining the capability of processing operations. In this study, the researchers tried to resolve the issues on the inefficiency of databases in the blockchain system. However, the impact of the work has not been evaluated or confirmed yet. In other studies [2, 3], the researchers have tried to provide a querying layer in between the application and database layers to enable efficient data retrieval. However, these approaches do not address fundamental issues of how characteristics of workloads in the blockchain environment impact the operation of data storage applications such as that of LevelDB. Currently, there is lack of thorough examination and research on I/O behavior in the blockchain system, and optimization of the I/O layer in platforms such as Ethereum.

In this project, we propose to identify I/O patterns in the Klaytn environment that cause inefficient operations on the database and study the impacts of functionalities of the KV storage on the I/O behavior of blockchain applications in order to suggest an optimized form of algorithm or data structure in the given blockchain environment. To do this, we plan to analyze the I/O performance of Klaytn to identify performance bottleneck points in the I/O layer and design an optimized algorithm and a data structure to improve the performance.

Background

Ethereum uses a data structure called Merkle Patricia trie which provides cryptographically authenticated data storage; data used as a key is hashed with KECCAK 256 hash algorithm, and the corresponding data is encoded with Recursive Length Prefix. Multiple tries are connected from fields in a block and each trie contains data in a single category. There are several types of data that get stored in the tries.

State. State is data regarding accounts and their states. Since it must include all accounts that participate in the system, there is one global trie that handles state data. State trie contains KV pairs and the key is an account’s unique identifier called address. The value for the pair has information such as the balance of the account.

Account Storage. It is additional data that is only relevant to the contract account which is the one that gets deployed when smart contracts between users are created. This information is saved into a trie called storage trie.

Transaction. Transaction is cryptographically signed data generated by clients who send and receive messages [2]. A single block contains multiple transactions and has its own transaction trie. Each node in the trie stores information on the accounts of sender and receiver of a transaction, the amount of value that has been transferred, and so on.

Receipt. As proof of a transaction that has taken place, the transaction information is encoded into a receipt [2]. It can be used as an index when searching for a specific transaction. It is also stored in one called receipt trie.

In order to permanently write data from those tries and retrieve required information, an internal database is used. LevelDB is an open-source KV store developed by Google, and it is a backend data storage in Ethereum. It has been selected as the database solution for several reasons such as its support for batch write, ordered mapping from string keys to string values, and high speed.

However, some features of LevelDB such as maintaining multiple data levels and alphabetical ordering of keys might lead to inefficient I/O operations and unnecessary read or write amplification when blockchain workloads are executed.

Project Description

In Klaytn, tens of thousands of data items are batch-written into the database every few seconds, which requires an enormous amount of data capacity. When blocks are processed, the data already written in the database needs to be read in sequence. However, data to be read is not guaranteed to be located near the data previously searched for. Since a hash value is saved as a key, there is no relevance in terms of order between keys even though the corresponding data can be related; for example, the account data items that are involved in transactions of the same block have no relevance in terms of key ordering.

In order to examine the characteristics and tendency of I/O operations in Klaytn, we plan to carry out experiments to identify key factors that cause I/O performance bottlenecks. In this project, we will perform two main tasks: analysis of I/O performance in Klaytn and design of an optimized algorithm and a data structure for Klaytn.

1. Analysis of I/O performance in Klaytn

1-1. Identifying the performance bottleneck of Klaytn using high-performance devices

In LevelDB, compaction and flush operations are executed normally while KV pairs are inserted or searched for in random due to hashing. It is important to explore whether performance bottlenecks of I/O operations in Klaytn occur at the hardware level or software level. In order to detect the possible impact of hardware performance, we will use high performance storage devices such as NVMe SSD, and RAM Disk, which are theoretically better in performance (compared to SSD and HDD). In this way, we can find out whether I/O performance is bounded by storage devices or CPU.

1-2. Evaluating the impact of compaction in LevelDB on I/O performance

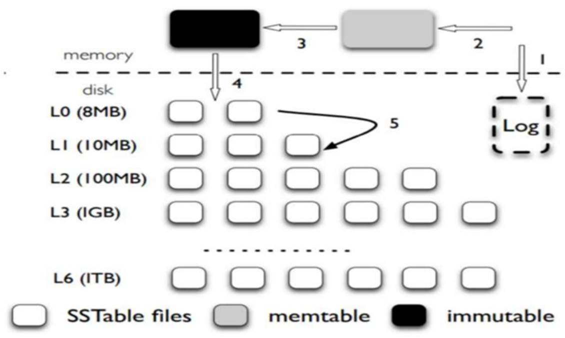

Figure 1. LevelDB architecture

Compaction is a process where overlapping keys between levels are combined and integrated into the lower level. As new data gets written to the database, old data gets stored at a lower level, losing priority when searching takes place. This can lead to read performance degradation.

In this task, we plan to identify the impact of the compaction operation in LevelDB, particularly on the I/O performance of Klaytn. To do this, we will execute blockchain workloads with a large number of PUTs and GETs in order to see how the performance changes depending on different parameter settings that affect compaction operations in the database. These parameters include DefaultCompactionTableSize, DefaultCompactionTableSizeMultiplier, DefaultCompactionTotalSize, and DefaultCompactionTotalSizeMultiplier.

1-3. Analyzing the data access pattern of LevelDB

As mentioned above, I/O operations for processing transactions are random because keys are hashed, and therefore there is no relevance of key orderings. In order to measure how many random reads and writes are in the system, we will utilize an I/O tracing tool. Blktrace is a tool that traces procedures in the block I/O layer [8]. Using blktrace, we are able to see how workloads in Klaytn access storage space used by LevelDB. LevelDB has multiple levels and each level contains a different number of KV pairs. We plan to analyze the write pattern to see how it affects the keys getting compacted into lower levels and the read pattern to check how the write pattern affects read performance.

In addition to analyzing data access, we plan to analyze the impact of horizontal partitioning at levels. Min et al. improved the I/O performance of a database by partitioning data space by users’ namespaces and maintaining per-namespace dedicated LSM (Log-Structured Merge) trees for users [13]. In a similar way, we can partition data space at levels and keep each partition in a separate zone of storage device such as ZNS (Zoned Namespace) SSD [14]; if ZNS SSDs are not available in the Klaytn cloud environment, we may develop the same zone functionality in software. Moreover, we plan to gather data elements that are most likely accessed in sequence into the same zone. By partitioning data and maintaining zones, we believe that bloom filters also can also be optimized by making each of them keep a smaller number of data elements and thus reducing the search space on the storage device. Besides bloom filters, Kipf et al. improved memory usage and query latency by providing variable-sized fingerprints on PostgreSQL by an index structure called cuckoo index [16]. The variable-sized fingerprint can use as smaller number of bits of fingerprints as possible to represent set-membership so that it can reduce time spent on referencing the index structures. We aim to examine the impact of such an index structure other than bloom filter on the read performance of the database in Klaytn.

2. Designing an optimized algorithm and a data structure for Klaytn

For the second main task, we plan to investigate the underlying components of the KV database such as its algorithm and data structure, and finally design an optimized version of the database for Klaytn. To start the task, we will use RocksDB. RocksDB is derived from LevelDB and supports additional features such as multi-threaded compaction and memtable bloom filter [11]. Therefore, RocksDB can provide better performance compared with LevelDB.

In terms of read performance, RocksDB is faster than LevelDB in the case of large-sized data where a lot of query operations are executed. Kwon et al. [12] shows differences of read performance between LevelDB and RocksDB. As shown in the figure below, the databases achieve different degrees of performance in terms of OPS (Operations per second) depending on the number of read operations. Comparing RocksDB with LevelDB, RocksDB shows poor performance when the number of query operations is very small; however, it exceeds LevelDB when the number of query operations is over 10K.

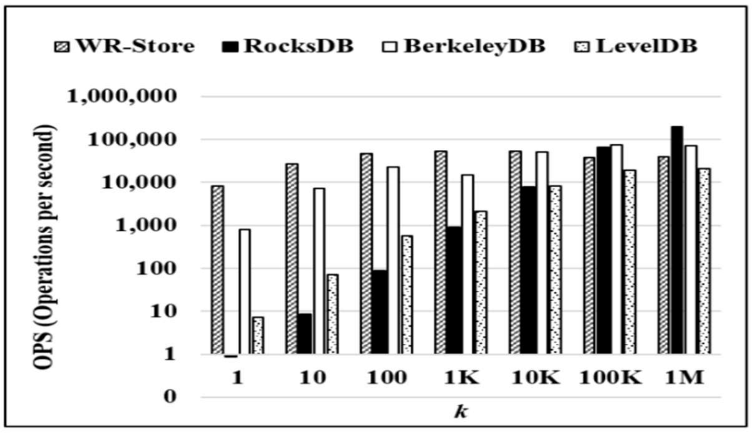

Figure 2. Performance by the number of read operations

This is because RocksDB uses atomic operation to modify reference counters for memtable and SSTs (Sorted Strings Table) instead of mutex lock. Also, by using thread local storage [17], RocksDB can remove locks in the read path. In Klaytn, there is over 150GB of data on each shard and the number of read operations in the system increases as the size of the data becomes larger. Thus, Klaytn can benefit from using RocksDB.

Moreover, Ethereum engine that uses RocksDB generally achieves better performance. There are two main Ethereum clients. Geth uses LevelDB and Parity uses RocksDB as their backend database engines [5]. According to the performance analysis done by Parity community [4], Parity achieves about 3x better performance in block processing which includes checking proof of work, transaction signatures, executing EVM codes, building and updating tries, and so on.

Therefore, we may improve the functionalities of the KV database using RocksDB with the analysis results from Task 1 and propose our optimized design of the KV database for Klaytn. Klaytn is using LevelDB written in Go as the main storage engine. In our project, we plan to use various storage engines such as RocksDB and WiredTiger, and compare the performance impact of various storage engines. As Klaytn communicates with storage engines by calling standard KV APIs (e.g., GET and PUT), we believe that various storage engines can operate with Klaytn, and that it is possible to select and use the one, the performance of which is the best.

2-1. Analyzing the data structure of KV database

In terms of read performance, the LSM tree-based data structure is not the best choice. B+ tree is advantageous over range scan performance [6]. B+ tree is a widely used data structure which is a variant of self-balancing trees. By balancing the tree when a new insert, update, and delete request comes, B+ tree can have a high balancing overhead but can have better read performance compared with LSM tree as the tree is balanced. The balanced tree structure can lead to low overhead due to the sequential tree traversal pattern during the GET operation. Thus, we are planning to evaluate the impact of the B+ tree in the Klaytn environment.

MongoDB is a widely used and mature relational database system. To support MongoDB, WiredTiger KV database is used as the default storage engine [15]. WiredTiger supports two types of data structures with varying characteristics to store KV pairs, which are B+ tree and LSM tree. According to the performance comparison done by the WiredTiger community, B+ tree achieves throughput from 1.5x to 3x larger in read compared with LSM tree [7].

The goal of the project is to improve the read performance of the current database implementation while maintaining the write performance as much as possible. A previous study [6] shows that utilizing the LSM tree and B+ tree by transitioning between the two helps achieve good performance when write and read occur. In this task, we study and examine how the use of B+ tree data structure can improve the read performance in Klaytn and come up with an optimized form of data structure such as the one which permits transitioning between LSM and B+ trees or dual implementation of them.

2-2. Analyzing the compaction algorithm of KV database

In this task, we plan to analyze the compaction operations of RocksDB and suggest an improved compaction algorithm that can lead to better I/O performance of the database in Klaytn.

RocksDB provides several features such as multi-threaded compaction and compaction filter. As explained above, a large number of data items are written every few seconds in Klaytn. Thus, write operations can interrupt read operations. By using multi-threaded compactions, time for writing data can be decreased and this can reduce the negative effects of write on read. In addition, RocksDB supports a different type of compaction: universal style compaction. Unlike level style compaction which is the basic compaction type of LevelDB, universal compaction can order data in terms of timestamp, and thus this can lead to different performance results in the Klaytn environment.

We aim to analyze compaction operations in RocksDB and the compaction algorithm that is optimized for block processing in Klaytn using the evaluation results obtained in Task 1-2.

2-3. Improving features based on data access pattern

From the analysis of the data access pattern of Klaytn workloads in Task 1-3, we plan to improve the read performance by adding features such as caching. Analyzing and modifying the compaction algorithm (Task 2-2) can improve the read performance by adjusting the localities of data between levels. In addition, using caching features will lead to a chance of improving the read performance depending on how often data is accessed in the Klaytn environment. Recent studies show that caching helps efficient I/O [9, 10]. Wu et al. optimized cache to save data, a pointer to a key, or a block of data to reduce query latency depending on read tendency [9]. Wang et al. presented an enhanced version of LSM tree that stores hot keys in separate spaces to reduce the number of search operations in SST files [10]. We aim to evaluate caching features in the LSM tree-based database and examine the advantages that can be applied to Klaytn depending on the data access pattern.

2-4. Designing an improved version of database for read performance

Based on results from Tasks 2-1, 2-2, and 2-3, we aim to come up with the improved forms of the algorithm, data structure, and features that are efficient in the Klaytn environment. We believe that through analyzing the compaction algorithm and characteristics of data structure such as LSM and B+ tree and data access pattern, we will be able to optimize the current database implementation. In order to propose the best design at the end of the project, we will refer to and utilize the test and analysis results from GroundX. Also, we will try to contribute to the open source with our new design.

Project Milestones and Schedule

Expected project duration: 12 months

Month	Task	Details	Note
Start date + 1 month	Task 1-1	Identifying the performance bottleneck of Klaytn using high performance storage devices - Analyze the I/O bottleneck of LevelDB - Identify whether hardware or software is a bottleneck	Milestone 1
Start date + 2 months	Task 1-2	Evaluating the impact of compaction algorithm of LevelDB on I/O performance - Study how compaction algorithm affects read performance - Study how adjusting parameters can change performance	Milestone 2
Start date + 3 months	Task 1-3	Analyzing the data access pattern of LevelDB - Use I/O traces or workloads from Klaytn to analyze the data access pattern - Identify how write affects read in the current database. - Examine the impact of horizontal partitioning with ZNS SSDs - Improve the accuracy of bloom filter - Utilize an improved index structure to reduce query latency	Milestone 3
Start date + 5 months	Task 2-1	Analyzing the data structure of KV database - Study the characteristics of LSM tree and B+ tree in the database implementation - Suggest an improved form of data structure	Milestone 4
Start date + 7 months	Task 2-2	Analyzing the compaction algorithm of KV database - Identify effective features in the compaction algorithm of RocksDB by comparing LevelDB and RocksDB	Milestone 5
Start date + 9 months	Task 2-3	Improving features based on based on data access pattern - Suggest a caching feature to improve read performance in Klaytn	Milestone 6
Start date + 12 months	Task 2-4	Designing an improved version of database for read performance - Combine the results from analysis and study to propose optimized forms of compaction algorithm, data structure, and features	Milestone 7

Key Deliverables

1. Technical reports
   - Technical report describing our research steps and results for two main tasks

2. Experiment results
   - Data from analysis and experiments conducted in the project including analysis on compaction algorithm, comparison experiments of LevelDB and RocksDB, and so on.

3. Test code
   - Code used for the analysis and experiments mentioned above.

4. Implementation details
   - Implementation details including source code for our modified compaction algorithm or data structure.

Budget

1. Total budget

Item	Unit Price (USD)	Unit	Total (USD)
Manpower	11,500	12 (months)	138,000
Server machine	20,000	1	20,000
NVMe SSD	2,000	4	8,400
DRAM	100	4	
Travel expense	2,291.6	12 (weeks)	27,500
Meeting fee	245.7	35 (times)	8,600
VAT	27,000	N/A (Total – VAT)*10%	27,000
Operating cost	27,000	N/A (Total – VAT)*10%	27,000
Overhead cost	40,500	N/A (Total – VAT)*15%	40,500
Total			297,000

Table 1. Total budget

2. Labor cost

Classification	Monthly-wage (USD)	Man-month	Total (USD)
Manpower	11,500	50.904	138,000

Table 2. Total man-month

Name	Monthly-wage (USD)	Man-month/month	Total (USD)
Eom, Hyeongsang	9,200	.217	2,000
Kim, Sunggon	4,000	.375	1,500
Bang, Jiwoo	2,500	.4	1,000
Shin, Hyunil	2,500	.4	1,000
Kim, Chungyong	2,500	.4	1,000
Han, Jongbeen	2,500	.4	1,000
Sung, Dong Kyu	2,500	.4	1,000
Song, Mansub	1,800	.55	1,000
Seo, Yunhyeong	1,800	.55	1,000
Ban, Jihoon	1,800	.55	1,000
Total		4.242	11,500

Table 3. Man-month by each member

3. Travel expense

Classification	Destination(s)	Date	Total (USD)
Travel expense	- Lawrence Berkeley Laboratory – Scientific Data Management group, Computational Research Division - Oracle - Google	2022.06 - 2022.08	27,500

Table 4. Travel expense details

Reference

[1] Raju, Pandian, et al. "mlsm: Making authenticated storage faster in ethereum." *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*. 2018
[2] Li, Yang, et al. "EtherQL: a query layer for blockchain system." *International Conference on Database Systems for Advanced Applications*. Springer, Cham, 2017
[3] Peng, Zhe, et al. "VQL: Providing query efficiency and data authenticity in blockchain systems." *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 2019
[4] https://www.parity.io/blog/performance-analysis
[5] Rouhani, Sara, and Ralph Deters. "Performance analysis of ethereum transactions in private blockchain." *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*. IEEE, 2017
[6] Jain, Varun, James Lennon, and Harshita Gupta. "Lsm-trees and b-trees: The best of both worlds." *Proceedings of the 2019 International Conference on Management of Data*. 2019
[7] https://github.com/wiredtiger/wiredtiger/wiki/Btree-vs-LSM
[8] Brunelle, Alan D. "Block i/o layer tracing: blktrace." *HP, Gelato-Cupertino, CA, USA* 57 (2006)
[9] Wu, Fenggang, et al. "AC-key: Adaptive caching for LSM-based key-value stores." *2020 USENIX Annual Technical Conference (USENIX ATC)*. 2020
[10] Wang, Yi, Peiquan Jin, and Shouhong Wan. "HotKey-LSM: A Hotness-Aware LSM-Tree for Big Data Storage." *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 2020
[11] https://github.com/facebook/rocksdb/wiki/Features-Not-in-LevelDB
[12] Kwon, Hyuk-Yoon. "Constructing a lightweight key-value store based on the windows native features." *Applied Sciences* 9.18 (2019): 3801
[13] Min, Donghyun, and Youngjae Kim. "Isolating namespace and performance in key-value SSDs for multi-tenant environments." *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*. 2021
[14] https://zonedstorage.io/introduction/zns/
[15] https://docs.mongodb.com/manual/core/wiredtiger/
[16] Kipf, Andreas, et al. "Cuckoo index: a lightweight secondary index structure." *Proceedings of the VLDB Endowment* 13.13 (2020): 3559-3572
[17] http://rocksdb.org/blog/2014/06/27/avoid-expensive-locks-in-get.html
[18] https://github.com/memsql/dbbench