# Skew and Compaction Performance Improvement in LSM-based Distributed Databases

Zidi Chen

January 13, 2021

**Abstract**

Distributed LSM-based databases usually face some performance issues due to their structure and mechanism. Skew and compaction could cause severe I/O and CPU performance degradation. In this research paper, we mainly focus on Hailstorm which provides a way that disaggregates computation and storage in distributed databases to reduce the performance degradation. Besides, this paper also researched some other studies that tried to solve the performance issue caused by skew and compaction, which mainly focus on optimizing the structure of LSM-Tree, utilizing specific characteristics of certain hardware or databases structure. The performance of these research works will also be discussed.

## Contents

## 1 Introduction of the Research Field

Distributed databases like MongoDB[1], TiDB[2] have been more widely used in many large-scale service scenarios. Distributed databases shard data across multiple machines and manage the data on each machine using embedded storage engines. But in many cases, distributed databases could suffer performance degradation and low utilization from skew, background operation and compaction.

Skew occurs naturally in many workloads and causes CPU and I/O imbalance, which degrades overall throughput and response time. [3] Current LSM-based databases address skew by resharding data[1, 2] across machines but this operation is expensive because it involves bulk migration of data, which affects foreground operations. Background operations such as flushing and compaction can cause significant I/O and CPU bursts, leading to severe latency spikes, especially for queries spanning multiple nodes such as range queries [4, 5].

This paper researches some soutions to these issues. The main paper *Hailstorm: Disaggregated Compute and Storage for Distributed LSM-based Databases* [6]introduces Hailstorm, a system that disaggregates storage and compute for distributed LSM-based databases. Hailstorm provides a storage pooling mechanism, in which the data blocks are sharded into smaller pieces than traditional LSM-based database systems, which will promote the read / write performance, as well as reducing the bad influence on I/O and CPU performance caused by compaction. It shows that Hailstorm achieves load balance in many MongoDB deployments with skewed workloads, improving the average throughput by 60%, while decreasing tail latency by as much as 5×.

There are also some works that focus on relatively lower level of LSM-based databases. MongoDB provides resharding and monitoring mechanisms to reduce skew problem and ahcieve load balancing. Light-weight LSM-Tree (LWC-Tree) is a variant of LSM-Tree which promotes the compaction performance by changing the structure and compaction mechanism. SMRDB[7] utilizes the characteristics of SMR[1] [7] disks and provides better performance on the database server side. Basic solutions to these works will be discussed in the 3 and 4. However, these works mostly focus on some certain parts of the LSM-Tree structure and lack of versatility. They don't take theutilization of other nodes in consideration. When it comes to different storage engines or environments, they require a lot of changes. Hailstorm provides a different solution from these related works, where it runs as a middleware between database and storage engines, providing more flexibility and versatility.

The rest of the paper will discuss the evaluation of the Hailstorm as well as some effects of other solutions Section 6 provides some discussion of these different approaches. 5 will provide conclusions.

## 2 Basics

### 2.1 LSM-Tree

Traditional disk-based index structures such as the B-tree will effectively double the I/O cost of the transaction to maintain an index such as this in real time, increasing the total system cost up to fifty percent. Hence Log-structured merge tree (LSM-Tree) [8] has been widely used by many distributed databases, as it provides efficient indexing for a key-value store with a high rate of inserts and deletes. The LSM-Tree uses an algorithm that defers and batches index changes, cascading the changes from a memory-based component through one or more disk components in an efficient manner reminiscent of merge sort. During this process all index values are continuously accessible to retrievals (aside from very short locking periods), either through the memory component or one of the disk components.

[1]Shingled Magnetic Recording (SMR) Shingled magnetic recording (SMR) is a magnetic storage data recording technology used in hard disk drives (HDDs) to increase storage density and overall per-drive storage capacity.

## 2.2 Compaction in LSM KV Stores

Compaction[5] is a critical mechanism in a system based on LSM-Tree. Log append method brings high-throughput writes. As the sstable continues to be written, more and more files will be opened by the system, and the accumulated data changes (updates, deletes) operations for the same key will increase. Since sstable is immutable, the data in the memory will reach the upper limit in certain layer. In order to reduce the number of files and clean up invalid data in time, compaction is introduced to optimize read performance and space.

The system will accumulate more segment files as it continues to run. These segment files need to be cleaned up and maintained in order to prevent the number of segment files from getting out of hand. This is the responsibility of a process called compaction. Compaction is a background process that is continuously combining old segments together into newer segments. Once the compaction process has written a new segment for the input segments, the old segment files are deleted.

Compaction can be implemented in many ways. For example, RocksDB implements Tiered+ Leveled, termed Level compaction[9]. Generally, in LSM-Tree, each level contains multiple partitions of data sorted by keys. When compaction will be triggered by continuously recycling old version data and merging multiple layers into one layer through periodic background tasks. The basic procedure goes as follows:

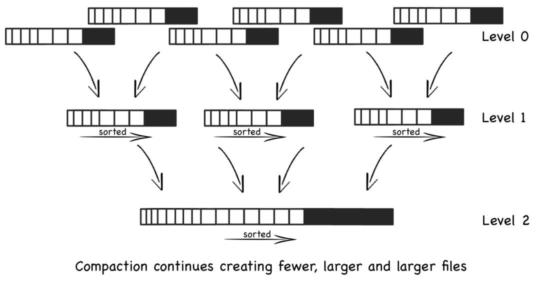

Level 0
Level 1
sorted
sorted
sorted
Level 2
sorted
Compaction continues creating fewer, larger and larger files

Figure 1: LSM-Tree [10]

As is shown in Figure 1, compaction into level N (L(n)) merges data from L(n-1) to L(n), where data is rewrited into the new level. In original paper of LSM-Tree, all data from L(n-1) is merged into L(n). In modern distributed databases such as LevelDB and RocksDB, overlapping data is eliminated, therefore for most time, only some data will be merged into new level.

Generally, LSM-Tree will continuously compact key-value pairs ,and the compaction and disk flushes will trigger big writes, which will cause significantly overheads due to I/O amplifications.

## 2.3 Sharding and Tow-level Sharding

Sharding is widely used in distributed databases[1]. Large data sets or high throughput applications can challenge the capacity of one single server. Each area has the same schema and columns, but each table has completely different rows. Similarly, the data stored in each partition is unique and has nothing to do with the data stored in other partitions. Sharding divides a piece of data into two or more smaller blocks, called logical shards. Then, logical shards are distributed on separate database nodes, called physical shards. Physical shards can hold multiple logical shards. Nevertheless, the data stored in all shards collectively represent the entire logical data set.

A tow-level sharding way is introduced in the main paper, where sharding is seperated into storage-level and database-level. The data for each database shard is spread uniformly across all the storage devices in a rack in small blocks (1 MB). This approach effectively provides a second, storage-level sharding layer that guarantees high storage utilization even in the presence of skew, removes per-node disk space constraints, and eliminates the need for database-level resharding within the rack. Together storage pooling and fine-granularity data spreading allow Hailstorm to improve I/O and storage capacity balance. Hailstorm uses a tow-step data assignment scheme, where data objects are dynamically assigned to different partitions, so that the system could react better to the changes in the workload and perform better load balance. The target LSM-based databases still use the filesystem solution to redistribute data blocks into different nodes uniformly.

### 2.4 Skew in Distributed Databases

Distributed databases [11, 2] are designed for large-scale data storage, which usually run on multiple nodes in the same, or even partitioned networks. Distributed database use sharding to handle larger dataset and handle additional requests by distributing the data among multiple machines. The database engine translates user queries into individual queries that are routed to one or multiple database instances for execution. Therefore, the database may suffer from skew issue, where keys are unevenly distributed to different nodes, which could cause uneven access and some nodes may become the hotsopt. This could lead to globally performance degradation.

## 3 Previous and Related Work

### 3.1 MongoDB's Solution of Reducing Skew

MongoDB introduces a balancer to mitigate imbalanced shards[12]. Some manual operations can also be performed, e.g., moving "hotspot" chunks manually to address the imbalances. In practice, some table related operations will also work, e.g., spliting one table into several sub-tables.

Resharding could be carried out manually or automatically to reduce the influence on performance due to skew. Auto-splitting is a mechanism in MongoDB[1] that detect automatically when data overflow happens in chunks, and then split the oversized chunk into smaller pieces. The server that runs MongoDB contains the process ChunkSplitter that runs continuously to detect weather any chunk grows beyond the configured chunk size (maxChunkSizeBytes) due to some insert operation.

The auto split task will be carried out if possible and necessary. ChunkSplitter supports asynchronous auto-split, thus, multiple split operations could be handled concurrently with no overlap. Basic proceduce goes as follows: First, ChunkSplitter will set the flag to the chunks which need resharding, where the estimated data size will be reseted to handle the potiential incoming writing operations. Then the splitVector will perform the split by inserting the split keys to original chunk vector, and returning the split points.

Auto-split may not be enough to handle skew issue, as it requires additional operations to reduce read operation hot spots. So MongoDB uses top chunk optimization to prevent some top chunks becoming potential hot spot. The balancer tries to move the top chunk out of the start when auto-split is carried out. MongoDB also provides auto-balancing mechanism that try to reduce hot spot chunks, where a balancer process runs continuously in the background to monitor the chunk distribution in the cluster. Resharding operation may still cause some performance loss, some manual operations that take the structure of the table in consideration may be costly in time.

### 3.2 Compaction and performance degradation

Compaction may cause severe I/O performance degradation in LSM-based distributed databases, as it merges multiple tables and writes back to disk, which will typically comsue large I/O and CPU computing resources. A benchmark test is conducted in the mainpaper.

Profiling this particular experiment reveals that storage is saturated as I/O bandwidth remains almost constantly close to 320 MB/s, the maximum write bandwidth for our SSD. We also observe peaks of CPU utilization when compaction tasks run.

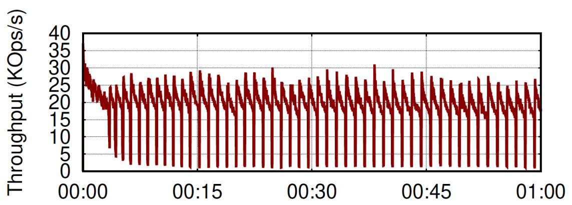


Figure 2: Embedded RocksDB throughput over time. [6]

### 3.2.1 Light-weight LSM-Tree

The light-weight compaction tree(LWC-Tree) [13] is a variant from the original LSM-Tree. The LWC-Tree tries to improve the write performance during compaction by merging a little metadata rather than perform the operation on the while range of data. As is discussed previously, traditional LSM-Tree carries out compaction by read, sort and rewrite all whole tables.

LWC-Tree divides the overlapping data from the original table into small segments, and then appends these segments into the tables on the next level, and then only merge related metadata, which is a very small scale compared to the original scale. As the unit of data management in LWC-tree, the table is defined as a DTable. LWC-Tree organizes the DTable structure based on the change of metadata. LWC-Tree also provides load balancing for the same level DTables to reduce hot spots as much as possible. The balancer moves the overly-full tables data to other tables on the same level by adjusting the key range, which reduces the cost for data movement as much as possible.

### 3.2.2 Reducing Involved Components in Compaction

Skip-tree [14] solve the performance degradation caused by compaction in a quite aggressive way. As is discussed, I/O performance gets severe influence by the compaction due to heavily read and write-to-disk operations. So Skip-tree skips some components in compaction and put the output of the compaction into some larger components. The basic architecture of Skip-tree is shown in Figure 3. Skip-tree introduces a large size memory to store the index for the KV objects as a buffer. It push the KV objects into non-adjacent components via skipping components during the procedure of compaction as many as possible. This would reduce the steps of operations from memory to larger disk components, which reduces the I/O throughput. As a result, there will be fewer compactions before the KV objects reach the destination component.

### 3.2.3 Disk Side optimization

SMRDB [7] is a KV data store that works specially for disks to address the need of utilizing disks by using approaches to leverage their proclivity for sequential writes as much as possible. It aims at providing a new key-value data management for SMR disks to handle the overlapping data in random write and in-place updates operations on the disk. SMRDB works as a KV database engine on the SMR disks.

The basic idea of SMR solution for LSM-Tree is shown in Figure 4. To meet the requirement of large data reorganization in LSM-Tree based databases, a simple SMR disks solution is introduced in this paper. Stitching is used in the compaction process, where it basicly moves part of the compaction data to a new area, but some splited data will still remain in the same place and be stitched with in-coming new data. But it can't be evaluated how this scheme works when compactions are carried in the background.

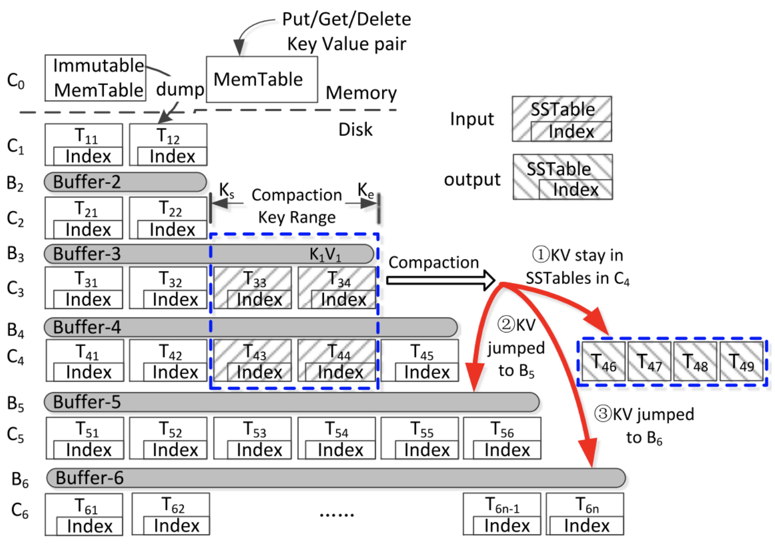


Figure 3: The architecture of Skip-tree. [14]

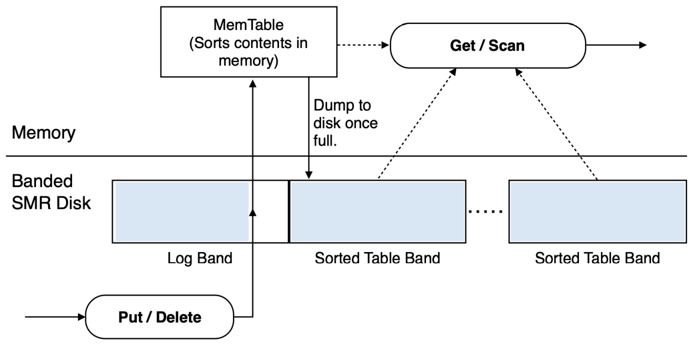


Figure 4: LevelDB based data access management. [7]

## 3.3 Summary

It can be shown from the previous work that skew will cause severe performance degradation, and the database system suffers from I/O brusts due to compaction and disk flushing. MongoDB's solution to reduce skew is basically based on resharding and monitoring mechanism. There are also many solutions that try to slove the I/O and CPU performance degradation. But these three solutions above are mostly aimed at LSM-Tree structure on individual nodes, they don't take the utilization of other nodes in consideration. Hailstrom tries to solve these problems by disaggregate resources to address load balancing at database and storage layers separately.

# 4 Design of Hailstrom: Disaggregate Compute and Storage

Hailstorm is designed to improve load balance and utilization for LSM-based distributed database. It on a higher level to reduce the skew for read and write operations, as well as to reduce the I/O and CPU performance issue caused by compaction behavior. The key idea of Hailstorm is to disaggregate compute and storage.

## 4.1 Basic Architecture

The basic architecture of Hailstorm shown in Figure 5. Hailstorm works like a middleware in the middle of in-memory database storage instances and disk side storage devices. Hailstorm spreads data uniformly for each storage engine across all pooled storage devices within the rack. The Hailstorm filesystem contains three parts:
1. Clients that provide filesystem interface to storage engines,
2. Servers that operate on local storages to store data and relevant operations,
3. Hailstorm agent in the middle to schedule operations of compaction tasks.
The user side operations could still operate on traditional databases.

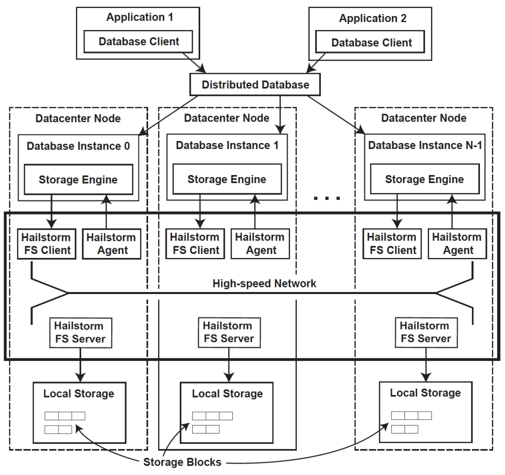


Figure 5: Distributed database deployed on a Hailstorm architecture.

## 4.2 Hailstorm Filesystem Interface

Hailstorm provides a subset of POSIX filesystem interface to storage engines rather than block-level interface. As Hailstorm in the middle of local storage and database nodes, filesystem interface could provide filesystem-level information to sotrage engines. Therefore Hailstorm hides KV store related specialized implementation details, and exposes traditional POSIX interface to nodes.

In order to promote I/O performance, Hailstorm filesystem keeps most metadata locally and use smaller block size compared with traditional LSM-based databases. The filesystem is also designed specially to reduce the influence of compaction, and aggressive prefetching is carried out to compaction tasks.

## 4.3 Storage Architecture Design

In order to provide disaggregation of computation and storage, the storage architecture of Hailstorm is designed to pool all data from nodes in a large rack. In the large rack, data is divided into small blocks (typically 1MB), which is designed in this way to reduce I/O latency and throughput in compaction. In this way, LSM storage could avoid flush stalls in compaction. Another advantage of this approach is that as long as the bandwidth is wide enough, Hailstorm could work on the whole-rack level, therefore there's no need of locality consideration. The splited data blocks are spread uniformly to database servers. In read operation, data could be accessed evenly, therefore there's no need to spend additional time for searching the position of data, as the locations are certain. The client would automatically prefetch blocks to promote the performance.

Load balancing is an important goal of Hailstorm. As data blocks are splited uniformly on different servers, load balancing can be well guaranted as it avoids the centralized data index management. Specifically, imbalance among servers are reduced in 2 ways. First, the pseudo-random mapping function M is a function of the file path, which ensures that different clients working on different files do not operate in lockstep. Second, we use batch sampling to ensure high storage utilization by ensuring there are always multiple pending operations. We assume there are as many clients as servers. Each client concurrently reads and writes from K distinct servers. Given N servers, there will always be KN pending operations within the rack. For a value of K = 3, this probability is %95 and for K = 5, we get over %99. [6] Another goal of Haistorm is to improve the performance. In the part of storage design, this is achieved by prepatching. By targeting rack-scale deployment and specializing our filesystem, Hailstorm can provide high storage utilization, provide optimizations such as prefetching sstables blocks and accessing files at different block granularities. Besides, in order to promote read performance, Hailstorm will have reads from previous threads at smaller block granularity.

### 4.4 Compaction Offloading

In the previous section, compaction is discussed, that by splitting data into small blocks, separating blocks uniformly across servers, and using aggressive pre-fetching mechanism, the performance influence of compaction could be reduced. In this section, specialized mechanisms are applied for compaction offloading. That is, generally, outsourcing compaction tasks to other machines.

A lightweight background agent is running to monitor the data usage situation. The monitor process will detect if I/O or CPU resources are overloaded due to compaction. Hailstorm will then decide heuristicly whether to offload local compaction tasks, and in this way to balance CPU load within a rack over time.

## 5 Evaluation

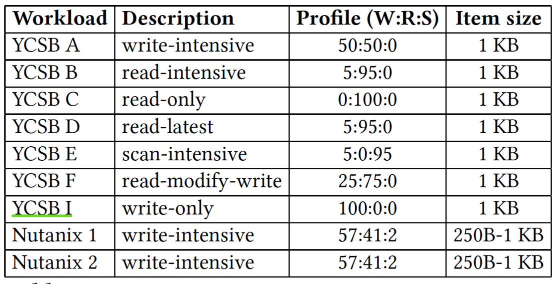
The goal of the evaluation is to judge the performance of Hailstorm on these aspects[6]:
1. How do distributed databases perform when deployed on Hailstorm in terms of throughput, especially in the presence of skew?
2. Does resharding help in traditional deployments? How does it compare with Hailstorm?
3. What is the impact of different features of Hailstorm on performance and how does it compare with other distributed filesystems such as HDFS? Do configuration values affect performance? Can Hailstorm improve throughput for B-trees?

### 5.1 Performance of Throughput

Here we mainly focus on the performance of Hailstorm in sloving skew problem and compaction performance degradation. The benchmark is carried out on MongoDB. The basic workflow to test how Hailstorm performs on solving skew is as follows. Here, Yahoo! Cloud Serving Benchmark (YCSB)[4] is used to run the benchmark. YCSB provides 6 different scenarios from YCSB A to YCSB F, and the paper adds some other scenarios in addition. The test workload table is shown in Figure 6.

To evaluate the impact of skew, we consider uniform and Zipfian key distributions for YCSB workloads. Zipfian key distributions are skewed, simulating the effect of popular keys. Nutanix's workloads are write-intensive workloads from production clusters. Nutanix 1 is more uniform than Nutanix 2, which has some skew.
We first populate the database with 100 GB of data (100 million keys) from each workload before executing the workload with an additional 100 GB of data (100 million keys). We execute Nutanix's workloads with a pre-populated database containing 256 GB of data, and execute each workload with an additional dataset size of 256 GB (approximately 700 million keys).

As is shown in Figure 7, Hailstorm performs better than Baseline in throughput in handling Zipfan data, where there are more skews in the data set. In dealing with Uniform data, Hailstorm performs almost the same with the Baseline, and there are slightly more overheads in some



| Workload | Description | Profile (W:R:S) | Item size |
|---|---|---|---|
| YCSB A | write-intensive | 50:50:0 | 1 KB |
| YCSB B | read-intensive | 5:95:0 | 1 KB |
| YCSB C | read-only | 0:100:0 | 1 KB |
| YCSB D | read-latest | 5:95:0 | 1 KB |
| YCSB E | scan-intensive | 5:0:95 | 1 KB |
| YCSB F | read-modify-write | 25:75:0 | 1 KB |
| YCSB I | write-only | 100:0:0 | 1 KB |
| Nutanix 1 | write-intensive | 57:41:2 | 250B-1 KB |
| Nutanix 2 | write-intensive | 57:41:2 | 250B-1 KB |

Figure 6: MongoDB workloads description and characteristics [6]

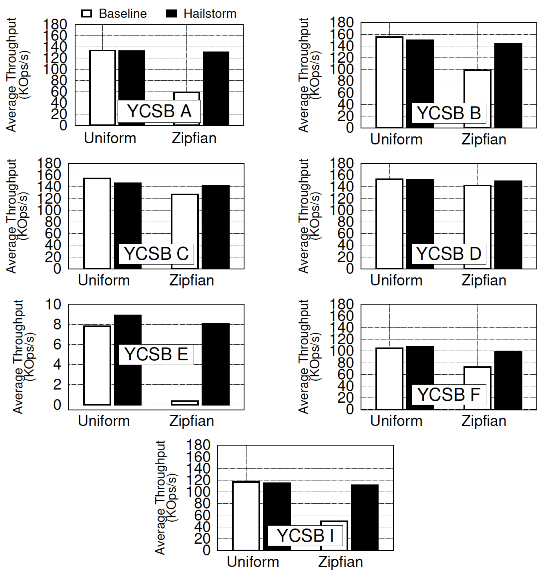


Figure 7: MongoDB average throughput for Baseline and Hailstorm for YCSB workloads with uniform and Zipfian key distributions. Hailstorm maintains high throughput on all workloads. [6]

scenarios than the Baseline. It can be concluded from the result that generally Hailstorm has a better performance in reducing the performance degradation caused by compactions.

Hailstorm performs better in terms of throughput in the presence of skew than the baseline. Baseline suffers from compaction, as after a period of time, the throughput will decrease continuously due to the operation of compaction on TiKV instances and resharding operations. Hailstorm performs more stablily in the existence of skew. The compaction offloading helps to drop some compaction tasks on certain nodes, and ensures a stable throughput in general.

## 5.2 Resharding Costs

Table 5.2 shows the average throughput in MongoDB for Baseline and Hailstorm for YCSB A with Zipfian distribution and with resharding enabled or disabled.

| | Resharding=OFF | Resharding=ON |
|---|---|---|
| Baseline | 42.9 KOps/s | 58.9 KOps/s |
| Hailstorm | 130.2 KOps/s | 113.0 KOps/s |

Table 1: MongoDB average throughput for Baseline and Hailstorm with resharding enabled or disabled. [6]

It can be concluded from Table 5.2 that Hailstorm and Baseline both perform better with sharding. Turning resharding off for MongoDB causes throughput to drop by 27%. Clearly, resharding in MongoDB is beneficial in skewed workloads. Hailstorm performs better than Baseline with or without resharding, indicating that storage pooling and compaction offloading are more effective than resharding.

## 5.3 Using Hailstorm with B-trees

Here the paper tests how Hailstorm performs on storage engines based on B-trees, e.g., WiredTiger[15]. Figure 8 compares the average throughput achieved by MongoDB with the B-tree-based WiredTiger storage engine for Baseline and Hailstorm for all YCSB workloads using both uniform and Zipfian distributions on 8 machines. Unlike with LSM stores, Hailstorm does not improve performance

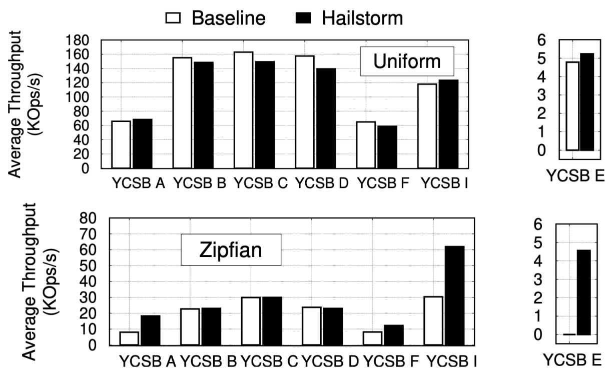


Figure 8: MongoDB with B-Tree average throughput for Baseline and Hailstorm [6]

for reads in the presence of skew because the CPU, not the I/O, is the bottleneck. Hailstorm improves performance for range-based queries in YCSB E as it partially relieves the overloaded node that becomes a straggler.

# 6 Discussion

Hailstorm provides a mechanism that works on a relatively higher level to promote the load balancing and compaction performance than the solutions discussed in the related work section. It acts in the middle of database server and client nodes, providing basic filesystem interface, and disaggregate compute and storage to improve both independently and load balance. Compared with Hailstorm, solutions in related work mainly focus on improvement on a single node or some variant of the basic structure of distributed databases - LSM-Tree. The advantage of Hailstorm is that it provides a way that act as a middleware in the computation part and storage part, rather than changing the low-level structure, which is more general for different kinds of use cases. It can be concluded from the evaluation that it indeed provides better performance in dealing with skew issue and compaction degradation.

However, it should also be noticed that Hailstorm may sacrifice some resources and performance, as it also requires some computation and I/O resources. When facing large scale of data with skew issue, or dealing with severe performance drop caused by compaction, the cost of Hailstrom can be considered acceptable. But in case of dealing with small scale of data, or uniformly distributed data sets, Hailstorm is likely to encounter with some performance issue. Therefore, if Hailstorm causes unacceptable performance issue to the whole system in certain cases, some other solutions could be combined to reduce the influence, e.g., it could be combined with LWC-Tree which is a variant of original LSM-Tree structure that promotes the performance in dealing with basic client side operations. It promotes the performance mainly by merging metadata rather than the whole rack of data in compaction. However, this may require some additional changes to the low-level structure, which sometimes may not be possible. It is also possible to use SMRDB for disks to provide better performance for server side services. Still, in general, Hailstorm does fulfill its goals in most cases.

## 7 Conclusion

This paper discusses the topics concerning about the performance issues in the LSM-based distributed databases due to the skew and the compaction mechanism. We mainly focus the way provided by Hailstorm, which disaggregate the computation and storage in the database system, and act as a middleware to handle the data flow. Hailstorm provides a large scale storage pooling where data blocks are limited to small size, allowing each node to utilize the whole rack evenly, which promotes the overall performance for the read / write operations and reduces the severe performance drop in compaction.

In addition to Hailstorm, some other works also tried to solve the performance issue in different approaches. So some typical solutions in different abstract levels are discussed in the related work. About the skew issue in distributed databases, mechanisms in MongoDB are chosen as an example, where it provides the balancer and monitor to achieve load balance as much as possible. In dealing with compaction, a variant of LSM-Tree structure is discussed, which tries to reduce the computation work in compaction by merging metadata rather the whole rack. On the storage side, SMR disks are introduced, where by using SMRDB to provide better performance on this special disk structure. Still, there are also some relevant topics to be discussed. For example, in the LSM storage engines, shard balancer may not be able to truly address the imbalance in the face of skew and high request rate. Resharding frequency will also influence the overall performance.

## References

[1] MongoDB. Chunk balancing and collection limits. https://github.com/mongodb/mongo/blob/master/src/mongo/db/s/README.md. 1, 2.3, 3.1

[2] PingCAP. Tidb. https://pingcap.com/. 1, 2.4

[3] K. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In *VLDB*, 1991. 1

[4] Veronika Abramova, Jorge Bernardino, and Pedro Furtado. Evaluating cassandra scalability with ycsb. In Hendrik Decker, Lenka Lhotská, Sebastian Link, Marcus Spies, and Roland R. Wagner, editors, *Database and Expert Systems Applications*, pages 199–207, Cham, 2014. Springer International Publishing. 1, 5.1

[5] Muhammad Yousuf Ahmad and Bettina Kemme. Compaction management in distributed key-value datastores. *Proc. VLDB Endow.*, 8(8):850–861, April 2015. 1, 2.2

[6] Willy Zwaenepoel Laurent Bindschaedler, Ashvin Goel. Hailstorm: Disaggregated compute and storage for distributed lsm-based databases. *ASPLOS'20*, page 301–316, March 2020. 1, 2, 4.3, 5, 6, 7, 1, 8

[7] Rekha Pitchumani, James Hughes, and Ethan Miller. Smrdb: Key-value data store for shingled magnetic recording disks. 05 2015. 1, 3.2.3, 4

[8] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996. 2.1

[9] RocksDB. Overview of compaction algorithm. https://github.com/facebook/rocksdb/wiki/Compaction. 2.2

[10] LSM-Tree-compaction. Lsm-tree-fig. https://en.wikipedia.org/wiki/Log-structured_merge-tree/. 1

[11] Mongodb. https://www.mongodb.com/. 2.4

[12] MongoDB. Sharding internals. https://www.mongodb.com/blog/post/sharding-pitfalls-part-iii-chunk-balancing-and. 3.1

[13] T. Yao, Jiguang Wan, P. Huang, Xubin He, Q. Gui, F. Wu, and C. Xie. A light-weight compaction tree to reduce i / o amplification toward efficient key-value stores. 2017. 3.2.1

[14] Y. Yue, B. He, Y. Li, and W. Wang. Building an efficient put-intensive key-value store with skip-tree. *IEEE Transactions on Parallel and Distributed Systems*, 28(4):961–973, 2017. 3.2.2, 3

[15] wiredtiger. wiredtiger. http://www.wiredtiger.com/. 5.3