Hailstorm: Disaggregated Compute and Storage for Distributed LSM-based Databases

Laurent Bindschaedler
EPFL
laurent.bindschaedler@epfl.ch

Ashvin Goel
University of Toronto
ashvin@eecg.toronto.edu

Willy Zwaenepoel
University of Sydney
willy.zwaenepoel@sydney.edu.au

Abstract

Distributed LSM-based databases face throughput and latency issues due to load imbalance across instances and interference from background tasks such as flushing, compaction, and data migration. Hailstorm addresses these problems by deploying the database storage engines over a distributed filesystem that disaggregates storage from processing, enabling storage pooling and compaction offloading. Hailstorm pools storage devices within a rack, allowing each storage engine to fully utilize the aggregate rack storage capacity and bandwidth. Storage pooling successfully handles load imbalance without the need for resharding. Hailstorm offloads compaction tasks to remote nodes, distributing their impact, and improving overall system throughput and response time. We show 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×. In workloads with range queries, Hailstorm provides up to 22× throughput improvements. Hailstorm also enables cost savings of 47-56% in OLTP workloads.

CCS Concepts • **Information systems → Distributed storage**; **Key-value stores**; **Relational parallel and distributed DBMSs**; **Physical data models**; • **Computer systems organization → Secondary storage organization**.

Keywords Hailstorm, disaggregation, compute, storage, distributed, database, key-value store, skew, compaction offloading, RocksDB, MongoDB, TiKV, TiDB, YCSB, TPC-C, TPC-E

ACM Reference Format:

Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. 2020. Hailstorm: Disaggregated Compute and Storage for Distributed LSM-based Databases. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3373376.3378504

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378504

1 Introduction

Distributed databases such as MongoDB [1], Couchbase Server [32], or Apache Cassandra [2] have become the new standard for data storage in cloud applications. Internet companies use them to power large-scale services such as search engines [34, 39], social networks [2, 3, 23, 24, 31, 69], online shopping [1, 41], media services [4], messaging [28], financial services [5, 33], graph analytics [6], and blockchain [7, 45, 56]. As distributed databases become the *de facto* storage systems for distributed applications, ensuring their fast and reliable operation becomes critically important.

Distributed databases shard data across multiple machines and manage the data on each machine using embedded storage engines such as RocksDB [31], a Log-Structured Merge-tree [60] (LSM) key-value (KV) store. These databases can suffer from unpredictable performance and low utilization for two reasons. First, skew occurs naturally in many workloads and causes CPU and I/O imbalance, which degrades overall throughput and response time [47, 52, 55, 71]. Current LSM-based databases address skew by resharding data across machines [1, 2, 8–10] but this operation is expensive because it involves bulk migration of data, which affects foreground operations. Second, 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 [20, 22, 39, 42, 51]. These problems are hard to address in existing systems because the storage engines operate independently of each other and thus are unaware of resource usage and background operations on other machines. As a result, these databases experience significant imbalance in terms of CPU and I/O load, and storage capacity.

This paper presents Hailstorm, a lightweight distributed filesystem specifically designed to improve load balance and utilization of LSM-based distributed databases. Figure 1 shows the high-level architecture of a generic distributed database running with Hailstorm. Hailstorm is deployed under the storage engines running on each machine.

The key idea in Hailstorm is to *disaggregate* compute and storage, allowing each to be load balanced and scaled independently, thus improving overall resource utilization.

Hailstorm improves storage scaling by pooling storage within a rack at a fine granularity so that each database storage engine can seamlessly access the aggregate rack storage bandwidth. The data for each database shard is spread

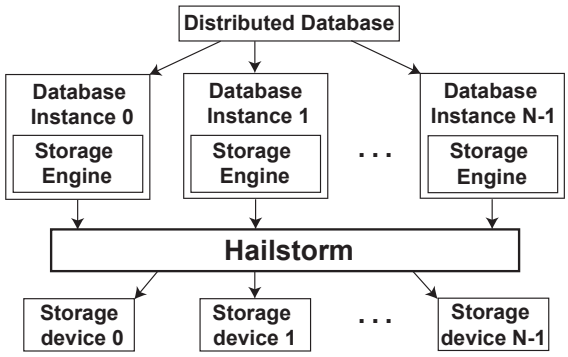

Figure 1. High-level architecture of a distributed database with Hailstorm. Storage engines access storage devices through the Hailstorm filesystem which pools all storage devices within a rack.

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 improves CPU scaling by offloading expensive background compaction tasks to other less utilized nodes, leveraging uniform, fine-grained storage pooling. Our approach reduces the CPU impact of compactions on overloaded nodes, frees up CPU cycles for user requests and lowers the memory footprint, thereby improving throughput and query response time.

We evaluate the performance of Hailstorm with MongoDB [1], a widely-used distributed database with a key-value store interface, running over Mongo-Rocks [8], an adapter for the popular RocksDB [31] storage engine. For our benchmarks, we use the reference Yahoo Cloud Serving Benchmark (YCSB) workloads [38] as well as two production workloads from Nutanix. We also experiment with TiDB [9], a state-of-the-art distributed database that supports SQL ACID transactions and bundles RocksDB as its storage engine. With TiDB, we evaluate the benefits of Hailstorm on industry-standard TPC-C [62] and TPC-E [35] benchmarks.

Hailstorm provides throughput improvements for skewed YCSB workloads running on MongoDB of 60% on average and up to 22× for scan workloads. It also reduces tail latency by 4-5× in skewed write workloads. On the production traces with skew, Hailstorm achieves 3× higher and stable throughput. With TiDB, Hailstorm improves throughput by 56% on TPC-C and 47% on TPC-E.

We make the following contributions in this paper:

- We present Hailstorm, a system that disaggregates storage and compute for distributed LSM-based databases (§3).
- We demonstrate how the specialized, distributed Hailstorm filesystem for LSM storage engines (§3.2) uses pooled storage and fine-grained spreading of data across machines to scale storage within a rack (§3.3).
- We leverage our filesystem design to provide a mechanism to scale CPU resources by seamlessly offloading expensive background tasks to less utilized nodes (§3.4).
- We show that Hailstorm's approach is the proper way to mitigate skew in various workloads and databases, and that the system successfully achieves load balance in both storage and compute (§5).

The rest of the paper describes our approach. Section 2 provides background information and discusses challenges in current systems. Section 3 presents the design of Hailstorm. Section 4 covers implementation. Section 5 evaluates the performance of Hailstorm. Section 6 discusses and compares Hailstorm to related work, and Section 7 provides our conclusions.

2 Background & Challenges

We discuss the issues and challenges involved in dealing with skew and I/O bursts in distributed databases and LSM stores, which we support with empirical evidence.

2.1 Skew in Distributed Databases

Distributed databases [1, 2, 4, 5, 10, 23, 34, 39, 41, 69] store data on many nodes, and are designed for large-scale data storage and low-latency data access. Although different distributed databases offer query abilities ranging from simple key-value semantics to SQL transactions, they all require *sharding*, i.e., partitioning data across multiple database instances, in order to store large datasets. Data is usually partitioned by collection or table using range-based or hash-based key partitioning.

The database engine translates user queries into individual queries that are routed to one or multiple database instances for execution. The set of database instances accessed by a query can vary significantly from one query to another and depends on the sharding policy. For example, while many reads and writes typically access a single instance, range queries and transactions may involve many instances.

Skew occurs naturally in many distributed workloads [47, 52, 55, 71], because some keys are more popular than others. As a result, sharding inevitably leads to data imbalance across the nodes that make up a distributed database. This uneven key distribution can cause capacity problems if a node has too much data to store locally. More importantly, uneven key distribution results in uneven accesses that cause load imbalance as some nodes perform more operations.

2.2 Compaction in LSM KV Stores

Log-Structured Merge-tree (LSM) KV stores [11, 31, 45, 63] are a popular way to provide persistent storage in single-node production environments, especially for write-heavy

workloads. They provide key-value semantics using an in-memory buffer that is periodically *flushed* to disk when it becomes full. In case of failure, LSM KV stores recover the data in the in-memory buffer using a write-ahead log.

LSM KV stores organize data on disk in files sorted by key, called Sorted String Tables (*sstables*). The sstables are maintained in a tree-like data structure, with higher levels of the tree containing larger sstables. The key ranges of different sstables in a given level L_i do not overlap, except for the first level, L_0 which corresponds to flushed in-memory buffers. LSM KV stores preserve the tree structure using background operations called *compactions* that merge sstables in L_i with overlapping sstables in L_{i+1}, while discarding duplicates.

Compactions run in separate background threads as they are typically expensive operations in terms of both CPU and I/O. Executing a compaction at L_i requires reading all overlapping sstables in L_{i+1} to perform an external merge sort before writing back the new sstables. Since sstables are larger as we go to higher levels, this results in high write amplification, where a single, small sstable causes multiple larger sstables in the higher level to be read and written. The larger the sstables involved are, the longer the compaction and the more resources are required. For example, if the highest LSM level contains hundreds of gigabytes, a single compaction involving that level can take minutes to hours.

Figure 2 shows the throughput fluctuation over a one-hour time period for RocksDB [31], running YCSB A, a workload consisting of 50% reads and 50% writes. The throughput remains generally between 18 KOps/s and 32 KOps/s with an average of 22.4 KOps/s, but repeatedly drops to ~2.7 KOps/s. The performance degradation is due to compaction threads competing for CPU and I/O resources with the threads servicing client requests. 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. As more data is stored, compactions become more expensive, throughput drops to as low as ~0.6 KOps/s around 01:00, and compaction duration quadruples to 9 seconds.

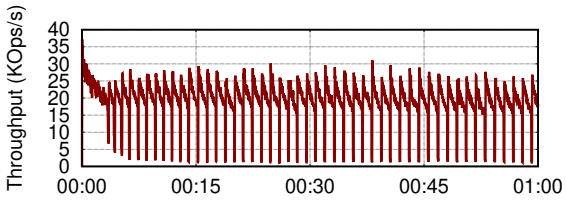

Figure 2. Embedded RocksDB throughput over time (HH:mm) on a node equipped with an Intel S3500 Series SSD using the YCSB A [38] workload (50% reads and 50% writes).

Table 1 shows the average and tail latencies in the same experiment. Latency spikes coincide with the execution of compaction tasks. Compaction tasks not only limit the available CPU and I/O bandwidth for read operations and writes to the write-ahead log, but also slow down flushing of the in-memory buffer when it becomes full, preventing the system from accepting more writes. Upon profiling, we find that writer threads are stalled 48.4% of the time due to flushing.

Mean	P50	P99	P99.9	P99.99	Max
1.6ms	0.7ms	35.1ms	72.1ms	181.3ms	69.5s

Table 1. RocksDB latency profile. YCSB A, 50% reads - 50% writes.

Several solutions have been proposed to reduce the impact of background operations in LSMs on individual nodes, but they do not take advantage of spare resources and capacity on other database nodes [11, 26, 63].

2.3 Distributed Databases with LSM Storage Engines

Many distributed databases rely on embedded LSM KV stores for local storage on each of their database instances in order to benefit from their high performance [2, 8–10, 69].

This two-layer architecture can, however, lead to situations where both layers interfere with each other, combining the undesired effects of skew (§2.1) and expensive background operations (§2.2) and causing severe performance degradation for database users. Skew causes some nodes to experience higher load from user requests, causing overload, and background operations become more intensive since these nodes manage more data. If the requests served by the database have dependencies or high fan-out, these overloaded nodes become stragglers, and performance across the whole database collapses.

MongoDB, like many other distributed databases addresses skew by resharding, i.e., sharding again, to remove hotspots and improve load balance. Resharding operations, performed manually or automatically at the database layer, include adding a new shard, splitting one shard into multiple shards, or merging multiple shards into one. Resharding involves migrating existing data from one shard to another, often located on a different database instance. Resharding is expensive because it introduces background operations that compete for resources with regular operations. Unlike resharding in B-tree-based databases, which is usually performed by splitting B-tree nodes [21, 68], resharding in LSM-based databases is more complex because data is stored in files with overlapping key ranges. As database instances strive to maintain their LSM data structure in the presence of additions and deletions due to migrations, additional flushing and compaction tasks are required. These compactions cause significant amplification of I/O and CPU usage. Furthermore, resharding decisions are usually taken by the distributed database based on its own load metrics, without regard to I/O load on individual instances and their current background operations.

Figure 3 compares the throughput fluctuation over time for 8 MongoDB instances using RocksDB for storage. We run

YCSB A, a write-intensive workload, YCSB C, a read-only workload, both with single-key queries, as well as YCSB E, a read-write workload with range queries, with both uniform and skewed (Zipfian) distributions.

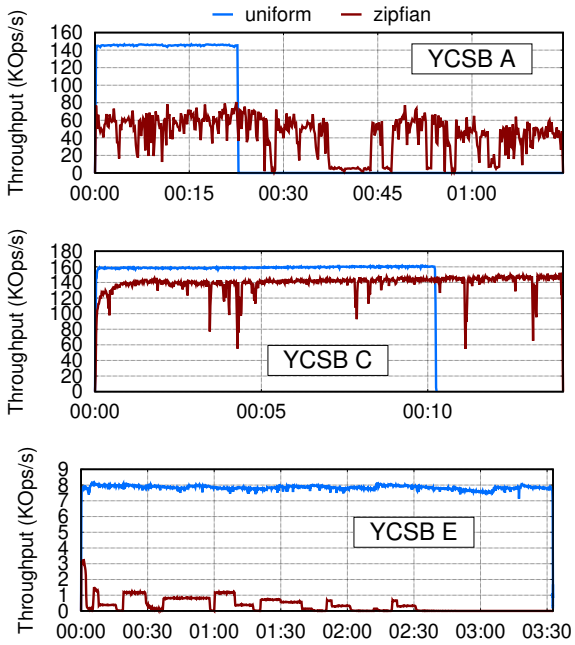

Figure 3. MongoDB aggregate throughput on 8 instances over time (HH:mm) for the YCSB workloads A, C, and E with uniform and Zipfian key distribution. The database is first populated with 100 GB of data before executing each workload with an additional 100 GB. Each MongoDB instance is configured to use the LSM-based RocksDB [31] storage engine.

In all three cases, the throughput is degraded with Zipfian request distribution whereas the uniform throughput is higher and more constant. Skewed workloads therefore take longer to finish: YCSB A, C, and E respectively run 3.3×, 1.4×, and 8.5× slower due to skew.

In YCSB A with Zipfian distribution, the per-instance throughput is much higher on one node than it is on the others. That node serves ~75% of requests and experiences high CPU usage and I/O load. This highlights the problem when skew in the workload and background operations are combined. During the entire execution, MongoDB attempts to recover from this hotspot by rebalancing shards and redistributing a total of 25.4 GB of data across other database instances, thus reducing data skew from 9× to 5×. However, this data migration causes more expensive flushes and compactions as the offloaded keys are deleted from one instance and written to another, and results in additional performance degradation and longer throughput drops, e.g., at 00:37.

In YCSB C, the per-instance throughput in the skewed scenario also suffers from significant imbalance, with one node serving almost 3× more requests than the others. MongoDB's shard rebalancer runs constantly during this workload and migrates data from the node experiencing high load to the others, leading to an increase in throughput from ~140 KOps/s to ~150 KOps/s in 15 minutes. We observe flushes and compaction tasks on each database instance as a result of data migration, causing occasional sharp drops in throughput, e.g., at 00:04. These results demonstrate that even read-only workloads can suffer throughput degradation and exhibit similar characteristics as write-heavy workloads due to resharding.

In YCSB E, the throughput in the Zipfian case is significantly degraded and frequently drops close to 0 as range queries are stalled by compactions competing for I/O bandwidth and CPU resources on overloaded nodes. We profile system resource usage and find that the most overloaded node oscillates between 100% CPU and 100% I/O usage for 2 hours. While the uniform workload completes after 3.5 hours, the skewed workload completes in 30 hours, with 1/8th the throughput and 99-percentile latency 5 orders of magnitude higher. After 2.5 hours of execution, the combination of resharding overhead and increasingly expensive compactions cause near zero throughput for over an hour.

Overall, the MongoDB shard rebalancer is unable to address imbalance in the face of skew and high request rate. Resharding often comes too late and is too slow to be really effective. In addition, data migrations trigger additional background operations on database instances which impact the foreground tasks and make the overload worse.

2.4 Summary

We have shown that skew has significant impact on application performance, and storage engines suffer from I/O bursts due to background operations such as flushing and compaction. When skew and I/O bursts are combined, performance can collapse. In addition, resharding rarely improves performance, especially when run during peak loads and in the presence of hotspots. By rebalancing shards, distributed databases attempt to solve three orthogonal problems: CPU, I/O load, and I/O capacity imbalance. These challenges motivate a more synergistic approach, taken in Hailstorm, where we disaggregate resources to address load balancing at the database and the storage layers independently.

3 The Hailstorm Design

Figure 4 expands on Figure 1 to show the detailed Hailstorm architecture corresponding to a typical deployment. Each datacenter node runs the high-performance, distributed Hailstorm filesystem, consisting of a client that provides a filesystem interface to storage engines, a server that stores data, and a Hailstorm agent that schedules and outsources compaction tasks on behalf of the local storage engine. Clients and servers are provisioned independently and can run on

separate, possibly dedicated nodes. In the rest of this paper, we make the assumption that clients and servers are co-located. Database instances use the Hailstorm filesystem instead of local storage for data persistence. Hailstorm pools all storage devices within the same rack together to provide its storage service.

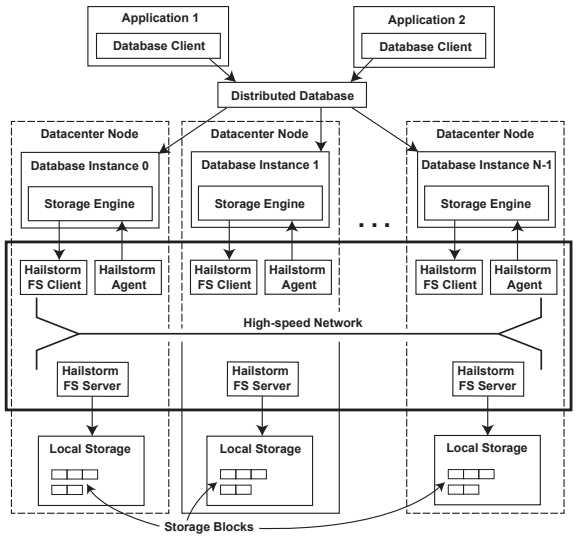

Figure 4. Distributed database deployed on a Hailstorm architecture. Hailstorm spreads data uniformly for each storage engine across all pooled storage devices within the rack.

The distributed database operates the same way as in traditional deployments as it is oblivious to the fact that storage engines are using Hailstorm. User queries are served as usual, but individual storage engines now perform storage operations across the network using the Hailstorm pooled storage service.

In the remainder of this section, we present an overview of the Hailstorm design. We first discuss and motivate the filesystem architecture. We then describe the storage architecture, including some optimizations and handling of fault tolerance. Finally, we demonstrate how our approach supports efficient compaction offloading.

3.1 Hailstorm Design Principles

Hailstorm is built on the following design principles:

1. **Resource disaggregation.** Disaggregating resources is a proven way to scale them independently and provide better utilization. In Hailstorm, we disaggregate compute from storage, enabling each resource to be scaled and load balanced independently.
2. **Storage pooling.** By introducing below the LSM storage engines a filesystem layer that pools together all storage devices within a rack, we gain the ability to mitigate storage hotspots and relieve nodes with taxed storage resources by spreading the I/O load.
3. **Fine-grained spreading of data.** By splitting data into small blocks and spreading them uniformly across all storage devices, we remove I/O hotspots and single-disk capacity issues altogether. The introduction of this new layer of indirection can be thought of as a storage-level sharding layer that guarantees uniform data placement and fine-grained storage load balance.
4. **Compaction offloading.** Compaction tasks are necessary to maintain the LSM structure and to provide good read performance, but use significant resources which causes interference on nodes with a high load. Pooled storage allows Hailstorm to efficiently move compactions to nodes with spare CPU and memory.

3.2 Filesystem Architecture

Hailstorm exposes a subset of the standard POSIX filesystem interface as required by storage engines. This filesystem interface serves as a drop-in replacement for the local filesystem used by storage engines for data persistence. The Hailstorm filesystem uses a traditional client-server architecture, where each client can access and store data on all servers within the same rack, thereby allowing the shards of one storage engine to span multiple physical storage devices.

Why a Filesystem? We choose to expose a filesystem interface, instead of providing a block-level interface, because it provides the desired visibility into the operations of storage engines. In particular, we require knowledge of the sstable files used by the LSM store to perform compaction offloading. In addition, the filesystem interface provides support for operations such as `mmap()` that are commonly used by storage engines. File-level visibility also allows us to perform more informed prefetching. Another significant benefit of using a standard POSIX file interface is that it requires minimal modifications to the storage engine code.

What About Using Existing Filesystems? Unlike existing distributed filesystems [12, 40, 46, 67, 70], Hailstorm is specialized for LSM KV stores. This specialization obviates the need for Hailstorm to implement many complex features found in traditional distributed and cluster filesystems. Since sstable files are not modified in place and only shared across storage engines for compaction offloading, Hailstorm does not require any support for fine-grained file sharing. Therefore, Hailstorm keeps most file metadata locally, avoiding the need for centralized metadata management. We use smaller block sizes (e.g., 1 MB) than most distributed filesystems to keep I/O latency low. Since LSM KV stores already use journaling, Hailstorm does not need to implement journaling to ensure filesystem consistency.

Hailstorm can leverage its specialization to optimize for efficient data access, in particular fast sequential operations on sstables, which is necessary for good compaction performance. We perform aggressive prefetching on behalf of the compaction tasks, and provide remote, in-memory caching

for large data sets using the page cache-backed nature of our implementation. Optionally, we allow write-ahead logs to remain on fast, local storage, facilitating failure recovery.

3.3 Storage Architecture

The first step to provide compute-storage disaggregation is to enable storage to scale within a rack. Hailstorm pools storage from all machines within a rack using a client-server filesystem approach.

Pooling Storage Hailstorm decouples logical from physical storage and splits the data into small blocks. Each Hailstorm client exposes a filesystem interface to its co-located storage engine and stores the data at block granularity on all storage devices, thereby spreading the load and enabling fast data access from any client.

This makes it possible to absorb storage load in the presence of peaks on database instances by spreading I/O operations to all Hailstorm servers within the rack. LSM storage engines running on Hailstorm can therefore sustain reads and writes during compactions, and avoid flush stalls [27]. It also enables efficient compaction offloading since the sstables for a particular storage engine can be accessed from any client and thus storage does not become a bottleneck. Pooling storage also has an important consequence for capacity: small and large shards can cohabit within a rack without requiring additional provisioning or expensive rebalancing.

Since we are operating at rack-level, we need not worry about locality considerations: as long as the top-of-rack switch provides full bisection bandwidth and the network is fast enough, accessing remote storage incurs virtually no penalty (barring a negligible increase in latency of a few microseconds). Hailstorm clients access data blocks using an efficient and low-latency decentralized scheme, thus avoiding the need for a separate namespace server. We ensure high storage utilization by prefetching data using a batch sampling strategy, as described in more detail below.

Data Placement and Access A Hailstorm client splits all files into small blocks (typically 1 MB) and spreads blocks uniformly in a pseudorandom cyclic order across servers. This scheme makes it easy to locate any block of data within a file. Given a file F, block size B, number of servers N, and a pseudorandom mapping function $M_F : \{0...N-1\} \rightarrow \{0...N-1\}$, the byte at offset I in F is on server $M_F(\lfloor \frac{I}{B} \rfloor \bmod N)$ in block number $\lfloor \frac{I}{BN} \rfloor$. When files are accessed sequentially, e.g., when reading sstables for compaction, the client automatically prefetches several blocks on behalf of the storage engine.

Since data placement is deterministic, there is no need for coordination between clients to access data, which reduces latency. Furthermore, each client can read from and write to its files independently.

Metadata Hailstorm identifies files by a universally unique identifier (uuid) and stores them in a flat namespace across servers. As mentioned earlier, Hailstorm divides files in blocks. Each server stores its blocks for the file sequentially in a file, identified by the uuid, on local storage. Each Hailstorm client keeps the mapping between file path and uuid locally, ensuring clients can only see and modify their files, and not the files created by others. Clients also keep all other metadata (file path, size, timestamps, permission, etc.) locally, since the file is typically only accessed by a single database instance. When file sharing across clients is necessary, such as with compaction, sharing uuids and metadata is sufficient to provide access.

Storage Load Balancing Clients can independently request any block in a file from any server, and they can add or replace blocks at any server. This decentralized approach reduces latency as it avoids the need for a centralized data directory, but can lead to load imbalance.

In the absence of any coordination between clients accessing different servers, we minimize the chances of causing load imbalance across servers in two ways. First, the pseudorandom mapping M is a function of the file path (as denoted previously by M_F), 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. This is inspired by recent work [29, 54, 59, 64]. 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. This ensures that the probability that all servers receive at least one request is at worst $1-(1-\frac{1}{N})^{KN}$ [29]. For a value of $K = 3$, this probability is 95% and for $K = 5$, we get over 99%.

In order to ensure that each server always has K outstanding requests ready to service and not in transit, we amplify the request window size at clients by a factor Φ, so that each client always has ΦK pending requests. Φ is necessary to account for network delays and message processing (serialization, etc.). It can be estimated as $\Phi = 1 + \frac{d_{network}}{d_{storage}}$ where $d_{network}$ is the application-level round-trip time on the network and $d_{storage}$ is the time for storage to service one request [64].

Read Optimizations LSM KV stores must often access multiple sstables to find the value for a key. If reads across the network were to use the same block granularity as writes, this may cause long delays. Hailstorm optimizes for this scenario by having reads from foreground threads execute at smaller block granularity, thereby reducing block access latency. Flushing and compaction use the default block granularity to maximize I/O performance. In order to guarantee high storage utilization with smaller granularity requests, we use a larger amplification factor Φ value for reads.

Asynchronous I/O Hailstorm performs most I/O operations asynchronously with the exception of `fsync()`. Storage engines rely on `fsync()` to guarantee that storage is in sync with the in-core state, so we use a blocking implementation to ensure correct `fsync()` semantics.

Fault Tolerance Distributed databases rely on replication to provide fault tolerance. They replicate data across replica sets that are located on different machines, different racks, different availability zones within a datacenter, and possibly across datacenters. LSM storage engines ensure durability by using a write-ahead log (WAL).

In the event of a crash, Hailstorm primarily relies upon the failure recovery mechanisms implemented by LSM storage engines and distributed databases. Replication is a concern for the distributed database layer, and is better implemented there than at the filesystem level where there is insufficient visibility into the entire database. Hailstorm allows databases to perform replication transparently, but requires that replicas be placed in different racks, so they do not all become unavailable at the same time due to failures in a storage pool.

When deploying distributed databases on top of Hailstorm, a single disk failure or machine crash may cause all shards within the rack to become unavailable since data for each shard is spread uniformly. Hailstorm mitigates single disk failures using standard techniques to ensure redundancy, e.g., RAID [61]. In addition, the system supports optional primary-backup replication at the block level to further protect data durability and filesystem availability. File metadata is persisted locally and replicated. All other state in Hailstorm is soft state and can be lost without affecting correctness.

3.4 Compaction Offloading

The second step to provide compute-storage disaggregation is to enable compute tasks to scale out to other nodes within the rack. Hailstorm improves response time on overloaded nodes by outsourcing compaction tasks to other machines. Compaction offloading therefore helps alleviate CPU load, while storage pooling helps alleviate pressure on storage.

Compaction Mechanism Hailstorm runs a lightweight agent alongside each client and database instance to monitor resource usage. Agents intercept all automatically triggered compaction jobs on their co-located LSM storage engine. If the agent believes that the local machine is overloaded, it pauses the compaction threads and attempts to offload the compaction to another node in the rack with lower load. Otherwise, it allows the compaction job to proceed locally. If the agent decides to offload the compaction job, it extracts the relevant parameters (e.g., which sstable files should be compacted), and contacts a peer on another node to run the compaction on its behalf. The agent informs its peer of the details of the compaction job and transfers the associated file metadata. The remote agent spawns a new LSM storage engine process on its node with the sole purpose of running an manual compaction job equivalent to the one that was offloaded. Since compaction does not modify the files in place, no additional synchronization between the two agents is necessary. When compaction completes, the remote agent notifies the agent on the original node with the list of newly created sstable files and their associated file metadata, and wakes up the paused compaction threads. This allows the original LSM storage engine to take ownership of the new sstables and install the compaction locally.

Overload Detection If a database instance is already experiencing significant load, the additional execution of background tasks using significant resources such as compaction can lead to request queuing and stall flushing of the in-memory buffer, thereby causing longer tail latencies and degradation in throughput. Since our design pools secondary storage and the network is fast at the rack level (as is the case in many deployments), compaction tasks primarily lead to CPU and/or memory bottlenecks in Hailstorm.

Compaction Policy Database operators can implement various compaction policies based on their service-level objectives (SLOs), such as running dedicated compaction nodes.

By default, Hailstorm uses a simple heuristic to determine whether to try and offload a local compaction task. Each Hailstorm agent measures CPU utilization periodically and maintains an exponential moving average (EMA) with weight α that is shared with other agents on the same rack. Whenever an agent intercepts a local compaction task, it offloads compaction to the node with the lowest EMA value, provided that the difference between its EMA value and the target node's is larger than a customizable threshold θ. This scheme balances CPU load within a rack over time.

In practice, we find that values of $\alpha = 0.5$ (with 1-second CPU sampling period) and $\theta = 0.2$ work fairly well. Disabling compaction offloading can be achieved by setting $\theta \geq 1$.

4 Implementation

Hailstorm is implemented in about 1,000 lines of C++ code. We use FUSE [13] to provide a filesystem interface to storage engines and use about 2,000 lines of Scala code to implement distribution, client-server communication, and Hailstorm agents. We choose FUSE to simplify development, but alternative approaches such as Parallel NFS [14] are also possible. We interface Scala with our C++ FUSE module using the Java Abstracted Foreign Function Layer [15] for high performance and low overhead. We use the Akka toolkit [16] for high performance concurrency and distribution. For simplicity, we use the local ext4 [53] filesystem to store blocks on servers. We find that the overhead of using a filesystem on the server side is negligible with our block sizes.

Supported databases Hailstorm does not require any modifications to storage engines or databases when used for storage pooling. We have successfully tested Hailstorm with

RocksDB [31], as well as API-compatible variants of LevelDB [45], including PebblesDB [63] and HyperLevelDB [11].

Hailstorm has been tested for deployment under MongoDB [1] using MongoRocks [8] as its storage engine, as well as TiDB [9], whose KV store, TiKV [9], uses an embedded RocksDB engine. It should in principle be possible to deploy Hailstorm in any distributed database environment which uses compatible LSM-based storage.

Compaction Offloading For compaction offloading, we intercept compaction tasks and invoke Hailstorm agents in order to execute these compactions remotely, and therefore need to make small changes to RocksDB (~70 lines of code). In addition, to perform remote compaction, we spawn a RocksDB process modified to remove some checks that would otherwise prevent compaction to run (6 lines of code commented out).

I/O Granularity and Batch Sampling Hailstorm uses a block size of 1 MB. Our sensitivity analysis indicates that block sizes ranging from 100 KB to 4 MB provide similar performance. 1 MB provides a good balance between performance and remote access latency, incurs minimal impact from random accesses to disk, and helps us minimize FUSE overhead by reducing the number of transitions to kernel mode. We pick a block size of 64 KB for client reads, which provides a good balance between latency and overhead from I/O accesses and FUSE. Each Hailstorm client concurrently has $\Phi K = 10$ pending requests for 1 MB blocks and $\Phi K = 100$ for 64 KB blocks (see Section 3.3), which we have empirically determined to work best in our cluster environment.

5 Evaluation

5.1 Goals

We evaluate Hailstorm using synthetic and production workloads on popular storage engines and distributed databases. Our evaluation sets out to answer the following questions:

1. How do distributed databases perform when deployed on Hailstorm in terms of throughput and latency, especially in the presence of skew? (§5.3)
2. Does resharding help in traditional deployments? How does it compare with Hailstorm? (§5.3)
3. Can databases supporting distributed SQL transactions benefit from using Hailstorm? (§5.4)
4. What is the impact of different features of Hailstorm on performance and how does it compare with other distributed filesystems such as HDFS? (§5.5) Do configuration values affect performance? (§5.6) Can Hailstorm improve throughput for B-trees? (§5.7)

5.2 Experimental Environment

Hardware We run this evaluation on up to 18 dedicated 16-core machines (2 CPU sockets with Xeon E5-2630v3). Each machine has 128 GB of DDR3 ECC main memory and an SSD providing a read bandwidth of 420 MB/s and a write bandwidth of 320 MB/s, as reported by fio [17]. The machines are connected to a 40GigE top-of-rack switch that provides full bisection bandwidth.

LSM KV stores We evaluate the performance of Hailstorm using RocksDB [31] (version 6.1), a popular LSM-based single-node KV store.

Distributed databases We use two distributed databases with different designs and characteristics:

- MongoDB [1] (version 3.6), a popular and widely used database with a key-value store interface. MongoDB offers a powerful JSON-based document model and query language, many configuration options, and supports a multitude of storage engines. In this evaluation, we run MongoDB with Mongo-Rocks integration [8] (version 3.6), a RocksDB-based storage engine developed by Facebook and Percona.
- TiDB [9] (version 3.0), a distributed database supporting SQL ACID transactions built on top of TiKV [10], a scalable distributed KV store whose design is inspired by Google Spanner [39] and HBase [69]. TiKV instances embed RocksDB for storage. TiDB requires the use of separate placement drivers responsible for metadata, load balancing, and scheduling.

Deployment Unless otherwise specified, we always run 8 database instances with embedded storage engines (MongoDB shard servers or TiKV instances and placement drivers) on 8 dedicated machines. When using Hailstorm, we co-locate Hailstorm clients and servers on the same machines as the 8 database instances. Different deployments, such as using separate nodes to run Hailstorm servers are possible and supported.

When evaluating MongoDB, we deploy 2 configuration servers co-located with 2 routing nodes (mongos) on 2 additional machines. When running with Hailstorm, we turn off MongoDB's shard balancer for the entire experiment. We run 8 YCSB load generators on 8 separate, dedicated machines, which we have empirically determined to be sufficient to saturate MongoDB. For TiDB, we deploy 8 TiDB servers (responsible for receiving and processing requests) on 8 additional machines. We run the TPC-C and TPC-E load generators on a separate, dedicated machine, which we confirmed was sufficient to saturate TiDB. We do not enable data replication, and clear the buffer cache as well as the database caches before each experiment.

System configuration We limit the total physical memory available to 32 GB of main memory on MongoDB shard servers and TiKV instances to ensure that all workloads are served from both main memory and secondary storage. We allocate up to 8 GB of main memory out of the available 32 GB for Hailstorm, and use default block sizes.

We refer to deployments of RocksDB, MongoDB or TiDB using local storage only as *Baseline* (BL). When deployments rely on Hailstorm to pool storage, but not on compaction offloading, we refer to them as *Storage Pooling* (HS-SP), and we call them *Hailstorm* (HS) when they rely on both.

5.3 Distributed Database: MongoDB

We evaluate Hailstorm in a distributed database setting with MongoDB [1] using synthetic and production workloads, and show how our approach benefits such deployments.

Workloads We use the Yahoo! Cloud Serving Benchmark (YCSB) [38] workloads as well as two production workloads from Nutanix. A summary of workloads used in this section is shown in Table 2, including their profiles (write:read:scan ratio) and the item sizes. YCSB provides 6 synthetic benchmarks covering a wide range of workload characteristics. For completeness, we consider an additional YCSB benchmark consisting of 100% inserts which we refer to as *YCSB I*, and which corresponds to the load execution mode in YCSB. This write-only workload is added to provide a more complete spectrum. 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.

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

Table 2. MongoDB workloads description and characteristics.

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).

5.3.1 Synthetic Benchmark (YCSB)

Throughput Figure 5 compares the average throughput achieved by MongoDB for Baseline and Hailstorm for all YCSB workloads using uniform and Zipfian distributions.

Deploying MongoDB over Hailstorm allows the database to maintain good throughput even in the presence of high skew. Throughput is better with Hailstorm than with Baseline for write workloads (YCSB A, F, and I) thanks to storage pooling and compaction offloading. In particular, the throughput for YCSB A and I improves by ~2.2× and ~2.3×. Read-intensive workloads (YCSB B, C, and D) mostly take advantage of storage pooling and their throughput improves by 46%, 15%, and 5% respectively. Scan-intensive workloads (YCSB E) improve by over 22× with Hailstorm when there is skew in the workload. These benefits stem from offloading compactions which lowers the load on the overloaded node. Range queries almost always involve all MongoDB instances, and the presence of a single overloaded instance is sufficient to degrade performance dramatically. Range queries are commonplace in real deployments, and so Hailstorm will have significant benefits in these environments.

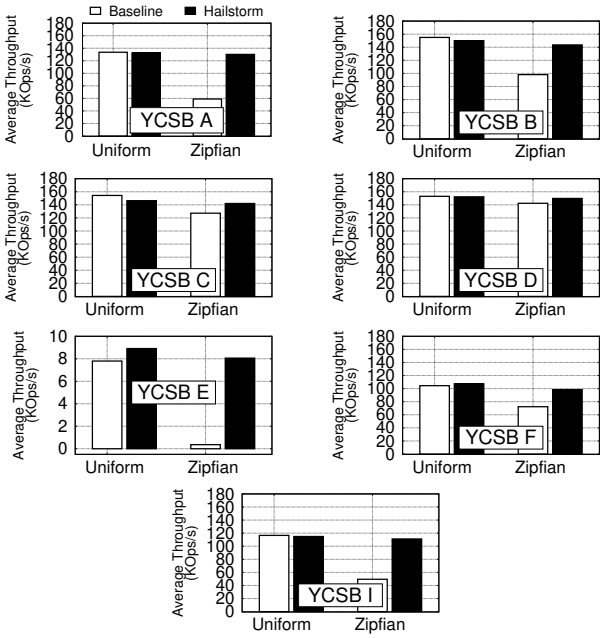

Figure 5. MongoDB average throughput for Baseline and Hailstorm for YCSB workloads with uniform and Zipfian key distributions. Hailstorm maintains high throughput on all workloads.

Some workloads take a small throughput penalty in the uniform case when running on top of Hailstorm, due to FUSE and network overheads. However, these overheads are more than compensated if the workload has skew.

Throughput over time Figure 6 shows the throughput over time on all YCSB workloads with Zipfian key distribution for Baseline and Hailstorm.

These results demonstrate the ability of Hailstorm to maintain high and relatively constant throughput in the presence of skew. In particular, the throughput for write-intensive workloads (YCSB A and I) is lower for Baseline compared to Hailstorm as a result of skew, since one MongoDB instance is absorbing most of the load and can only do so using its local storage. Even with a smaller proportion of writes in YCSB B and F, the Baseline throughput is lower. In addition,

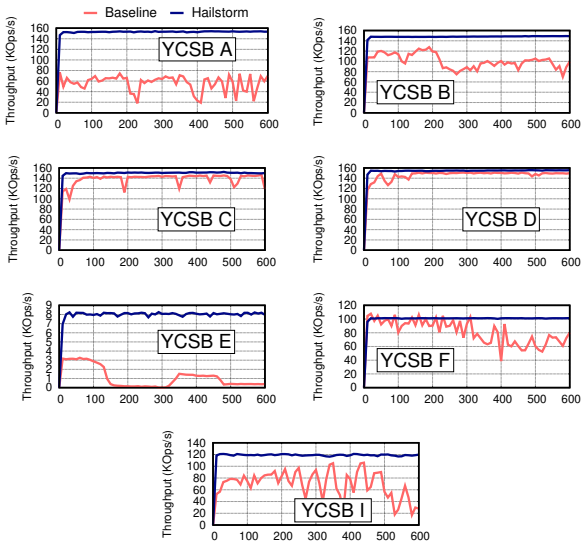

Figure 6. MongoDB per second throughput timelines for Baseline and Hailstorm with YCSB workloads. Hailstorm manages to keep the throughput relatively constant throughout the execution.

the throughput for the Baseline falls repeatedly by ~3× for YCSB A and ~4× for YCSB I, and to 0 in YCSB E as a result of a series of particularly costly compaction tasks. Throughput for YCSB C and D (100% reads) remains similar in both cases, as the database is not large enough to cause significant amounts of local I/O in the Baseline case.

Latency Figure 7 presents the mean and tail client response times for the MongoDB Baseline and Hailstorm for both request distributions using 3 representative workloads: YCSB A, C, and I. Hailstorm significantly improves response times under skew, especially for write workloads. For example, it reduces the mean response time by 37%, 18%, and 29%, as well as tail latencies by ~4×, ~30%, and ~5× for YCSB A, C, and I respectively. In addition, Hailstorm does not adversely affect the mean response times in the uniform case.

5.3.2 Production Traces

Figure 8 compares the average throughput achieved by MongoDB for Baseline and Hailstorm and for both production workloads from Nutanix.

Hailstorm provides consistent throughput for both Nutanix 1 (uniform) and Nutanix 2 (skewed), whereas the MongoDB Baseline suffers from a 3× performance degradation on Nutanix 2 resulting from a hotspot on one of the database instances. Hailstorm therefore does not add significant overhead on uniform workloads and successfully maintains performance close to uniform when the workload is skewed.

5.3.3 Large Database

Until now we have shown the significant benefits of Hailstorm when workloads have skew. In this experiment, we show that Hailstorm benefits uniform workloads as well. Figure 9 shows the throughput over time for Baseline and

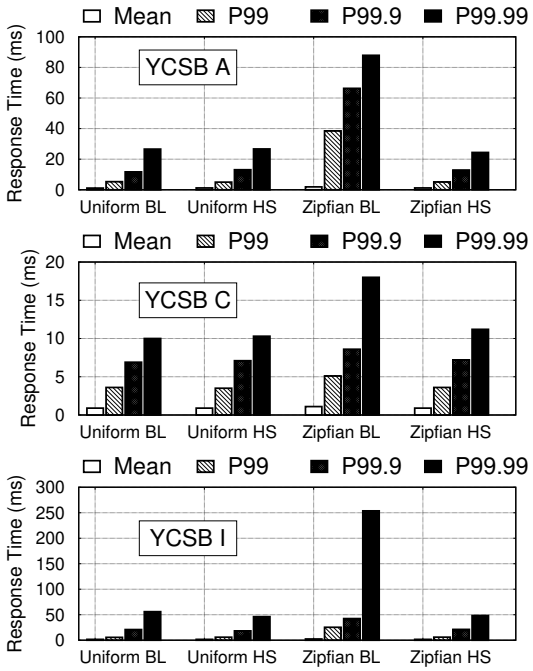

Figure 7. MongoDB mean and tail response times for Baseline (BL) and Hailstorm (HS) for YCSB A, C, and I with uniform and Zipfian requests distributions.

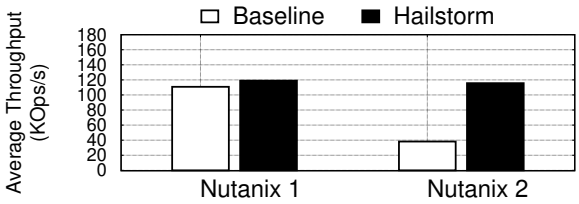

Figure 8. MongoDB average throughput for Baseline and Hailstorm and for Nutanix's workloads. Hailstorm improves throughput in both cases, especially for workload Nutanix 2.

Hailstorm with YCSB A on a uniform distribution starting with a large 1 TB database. Baseline performance suffers from sudden drops that increase over time as a result of larger compactions taking place, while Hailstorm has consistent performance over time. This experiment shows that even with uniform workloads, I/O bursts due to background operations causes skew across storage engines.

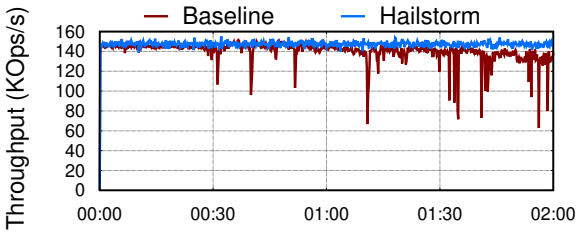

Figure 9. MongoDB per-second throughput for Baseline and Hailstorm on a large 1 TB database with YCSB A and uniform distribution. Baseline experiences drops over time as larger compactions occur, causing load imbalance across storage engines.

5.3.4 Resharding Costs

Table 3 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 3. MongoDB average throughput for Baseline and Hailstorm for YCSB A Zipfian distribution with resharding enabled or disabled.

This table presents several interesting results. First, turning resharding off for MongoDB causes throughput to drop by 27%. Clearly, resharding in MongoDB is beneficial in skewed workloads. Second, Hailstorm performs better than Baseline with or without resharding, indicating that storage pooling and compaction offloading are more effective than resharding. Indeed, proper skew mitigation requires synergistic approaches at both the distributed database and storage layers. Finally, resharding causes a 15% throughput drop for Hailstorm due to increased I/O operations. This justifies our decision to disable MongoDB's balancer for experiments with Hailstorm.

5.4 Distributed SQL Transactions: TiDB

We now consider distributed SQL transactions in TiDB [9], a popular horizontally-scalable database compatible with MySQL [44]. TiDB is built on top of TiKV [10], a distributed database with a key-value interface.

Workloads We use both the industry-standard TPC-C [62] benchmark and the more recent TPC-E [35] benchmark. TPC-C models a number of warehouses with orders, entries, payments, monitoring of stock, etc. Multiple transactions execute simultaneously, and the performance metric is the number of new-order transactions per minute (*tpmC*). TPC-E models a broker whose customers generate trades, account balance checks, market analysis, etc., and the performance metric is the number of trade-result (executed trades) transactions per second (*tpsE*). Both benchmarks also include a price per performance metric based on the total cost of ownership of the cluster used for a period of 3 years.

Bench	Model	Tables	Txs	R:W	RW:RO	Sec Idx
TPC-C	Warehouses	9	5	65:35	92:8	2
TPC-E	Brokerage	33	12	91:9	23:76	10

Table 4. TiDB benchmarks description and characteristics.

The main characteristics of each benchmark are shown in Table 4, including the type of business modeled by the benchmark, the number of tables, the number of distinct transactions, the I/O read to write ratio (R:W), the read-write to read-only transaction ratio (RW:RO), and the number of transactions using a secondary index. [36] contains more details and comparisons about these benchmarks.

Benchmark Results Table 5 summarizes the benchmark results for both TPC benchmarks, with Baseline and Hailstorm. We show performance and price per unit of performance (price-performance) metrics for our cluster. We conclude that Hailstorm provides significant performance improvements and cost reduction for distributed databases.

	TPC-C		TPC-E	
Configuration	**tpmC**	**$ / tpmC**	**tpsE**	**$ / tpsE**
Baseline	32,184	3.10	277.3	360.60
Hailstorm	50,178	2.00	408.1	245.05

Table 5. TiDB TPC-C and TPC-E results for Baseline and Hailstorm. Estimated total system cost for our cluster is USD 100,000. Hailstorm improves throughput by 1.56× and 1.47× respectively.

Figure 10 compares the throughput (measured in transactions per second) over a period of time of 1 hour for both scenarios and both benchmarks.

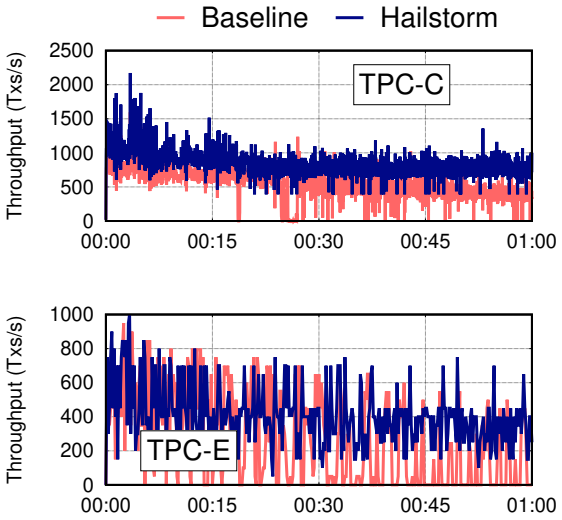

Figure 10. TiDB per 10-second throughput timelines (HH:mm) for Baseline and Hailstorm with TPC-C and TPC-E.

Baseline suffers from unstable throughput and frequent, drastic drops in throughput in both benchmarks. These drops are caused by compactions running on TiKV instances and resharding operations executed by placement drivers trying to remove hotspots. We notice significant amount of data migration due to TiDB's resharding policies. Short bursts of data migration consume as much as 90% of the I/O bandwith for a single TiKV instance and are responsible for prolonged drops at approximately 00:30 and 00:40 for TPC-C. Although TPC-C's request distribution is uniform and TPC-E is only mildly skewed, there is significant imbalance across TiKV instances due to compaction and uneven data placement.

Hailstorm offers more stable and overall higher throughput than Baseline. Compaction offloading helps limit pressure on overloaded instances, and storage pooling removes many I/O bottlenecks.

5.5 Comparison with HDFS

In this section, we compare Hailstorm with HDFS [67], an existing production distributed file system. We perform experiments directly on standalone RocksDB [31], thus avoiding any interference or overheads of using a distributed database. To this end, we design a microbenchmark that uses YCSB and its driver for embedded RocksDB.

Workloads We consider three custom YCSB workloads: a read-only workload, a write-only workload, and a mixed workload consisting of 50% writes and 50% reads. Keys are selected uniformly at random, and values are 1 KB each.

Configurations We run each workload on 8 nodes in parallel using separate RocksDB databases in 4 configurations *i:8* for i values of 8, 4, 2, and 1. An i:8 configuration represents a scenario where 8 nodes are running a RocksDB database, but only i of them are executing the workload with the other nodes remaining idle. *8:8* corresponds to uniform, *4:8* to mild skew, *2:8* to intermediate skew, and *1:8* to high skew.

We execute each of the above 4 configurations 4 times: first with RocksDB using the local ext4 [53] filesystem, thereby establishing a Baseline (BL), then with RocksDB using HDFS with a replication factor of 1 to maximize performance, then with RocksDB using Hailstorm for storage pooling (HS-SP), and finally with RocksDB running on top of Hailstorm with both storage pooling and compaction offloading (HS).

We run this experiment with two workload sizes: 100 GB (storage workload, dataset does not fit in memory), and 20 GB (in-memory workload, dataset fits in memory). The in-memory workload shows Hailstorm performance when the workload is CPU bound. We first discuss the results for the storage workload and then the in-memory workload.

Storage Workload Results Figure 11a shows the aggregate throughput (over all 8 nodes) for each of the 3 workloads in each of the 4 configurations for the 100 GB workload.

Hailstorm in the *8:8* configuration (uniform case) fares comparably with vanilla RocksDB. However, in the presence of skew, as expected from previous experiments, Hailstorm's throughput is much higher than the corresponding vanilla RocksDB (Baseline) throughput. Storage pooling and compaction offloading together enable Hailstorm to keep throughput close to the *8:8* configuration. Hailstorm performance decreases mildly with increasing skew due to remote reads and writes and increased compaction offloading.

In contrast, the throughput of RocksDB over HDFS is lower than the corresponding Baseline case, even though it uses distributed storage. We profile this experiment and find that the low performance stems from synchronous calls to the namenode before accessing data, performing I/O one block at a time, and writing blocks preferentially to the local disk. This shows the need for a specialized filesystem designed to maximize storage bandwidth. As an aside, we also experimented with running RocksDB on two full-featured, distributed filesystems (Ceph [70] and GlusterFS [40]), and unfortunately both filesystems would invariably crash after some time, and RockDB would hang, for unclear reasons.

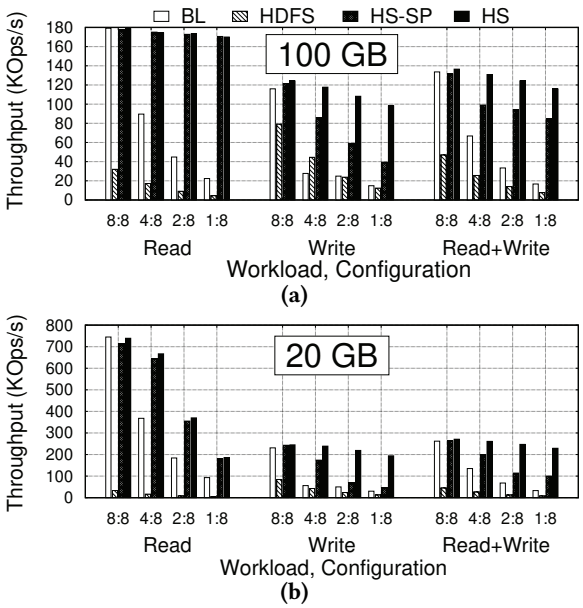

Figure 11. RocksDB aggregate throughput comparison between Baseline (BL), RocksDB over HDFS (HDFS), Hailstorm with storage pooling (HS-SP), and full Hailstorm (HS) on 8 machines. 3 workloads are considered in 4 different configurations with increasing skew, and data sizes of 100 GB and 20 GB.

In-memory Workload Results Figure 11b shows the aggregate throughput results for the 20 GB CPU-bound workload. The read workload results show that Hailstorm improves performance compared to the baseline by roughly a factor of 2 under skew. This benefit results from storage pooling, which allows the initial dataset to be loaded faster from disk. To understand why Hailstorm read performance goes down with increasing skew, we measured the maximum achievable RocksDB random read throughput on a single node using a RAM disk and found that the system becomes CPU-bound at 200 KOps/s. RocksDB spends significant times on binary search to find a random key, decompression, and checksums, which limits Hailstorm performance.

Write and read+write numbers are qualitatively similar for both workload sizes. When comparing with the 100 GB workload, the 20 GB write and read+write throughputs are almost double since the data can be cached in memory. However, unlike the read-only workloads, the throughput cannot go over 2× due to write-ahead logging. In addition, HS-SP throughput suffers from a steep drop from 4:8 to 2:8 because the bottleneck switches from storage to CPU. This is not the case for HS due to compaction offloading.

5.6 Sensitivity Analysis

We perform a sensitivity analysis for the compaction offloading threshold θ using two machines: node1, which receives

full load, and node 2, which receives 10%, 50%, or 80% of the full load. We use the same workloads and configurations as in the previous section (§5.5). For each scenario, we consider three θ values: 0.1, 0.2, and 0.5. Figure 12 reports the average throughput with the read+write workload for each instance in each scenario.

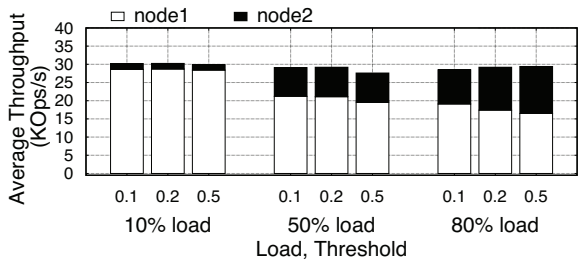

Figure 12. RocksDB average throughput for the read+write workload with different compaction offloading thresholds θ using two RocksDB instances where one node receives 10%, 50%, or 80% load.

Overall, different θ values have little impact. Large values (e.g., $\theta = 0.5$) make compaction offloading less frequent, and thus lead to slightly lower throughput on overloaded nodes.

5.7 Using Hailstorm with B-trees

Although Hailstorm is primarily intended for use with LSM-based storage engines, we expect storage pooling to still provide benefits when storage engines are based on B-trees, e.g., Aerospike [5], Couchbase Server [32], KVell [49], and WiredTiger [18]. B-trees exhibit different access patterns and storage behavior than LSMs, and do not require compaction [37].

Figure 13 compares the average throughput achieved by MongoDB with the B-tree-based WiredTiger [18] storage engine for Baseline and Hailstorm for all YCSB workloads in Table 2 using both uniform and Zipfian distributions on 8 machines. We only use Hailstorm for storage pooling and disable compaction offloading.

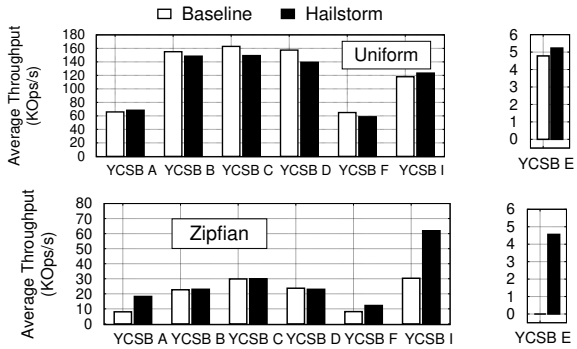

Figure 13. MongoDB with WiredTiger [18] storage engine average throughput for Baseline and Hailstorm for YCSB workloads with uniform and Zipfian key distributions.

Unlike with LSM stores, Hailstorm does not improve performance for reads in the presence of skew because the CPU, not the I/O, is the bottleneck. Hailstorm's storage pooling provides ~2× throughput improvements in the Zipfian case for write workloads YCSB A, F, and I. Hailstorm also improves performance for range-based queries in YCSB E as it partially relieves the overloaded node that becomes a straggler. We expect that offloading B-tree background tasks such as garbage collection in a similar way as we offload compaction tasks in LSMs would further improve write performance.

6 Related Work

Skew in distributed databases Skew is often intrinsic to the application and cannot easily be removed. It is often mitigated manually or automatically using resharding [19, 39]. This is achieved by identifying hotspots in the workload and migrating data to less utilized database instances [24, 50]. However, resharding requires in-depth knowledge of the shape of the data managed [25]. Furthermore, rebalancing data between shards is costly as it triggers additional compaction, flushing, and garbage collection tasks, and is performed too late to be effective. Hailstorm automatically reshards data uniformly across all storage devices, which works in all cases without requiring knowledge of the shape of data.

Distributed databases Distributed databases distribute data across many nodes, often on a global scale. Many recent databases rely on distributed KV stores and store data on local storage [1, 5, 9, 10]. Other databases such as Apache HBase [69] or Google Spanner [39] rely on a distributed filesystem to store data, which they leverage mainly for data replication. Hailstorm is primarily intended for databases using local storage, effectively providing storage-layer resharding to remove I/O hotspots by spreading block data uniformly across storage devices.

LSM KV stores Many systems attempt to solve the write amplification problem in LSMs. HyperLevelDB increases parallelism and modifies the compaction algorithm to reduce compaction costs [11]. PebblesDB combines LSM with skip-lists to fragment data in smaller chunks, thereby avoiding complete rewrites of sstables within a level [63]. TRIAD delays compactions until there is sufficient overlap and pins hot key entries in memory to avoid creating many copies on storage [26]. Silk attempts to opportunistically execute compactions during low load and preempt them at high load [27]. While these approaches provide temporary relief, they often lead to higher costs in the long run as uncompacted or fragmented LSMs suffer from increased read latency, and delayed compactions inevitably trickle down the LSM levels. Furthermore, all these solutions apply to a single node configuration and do not take advantage of distributed storage. Nevertheless, these optimizations are orthogonal to our approach and could be combined with it.

Distributed filesystems Distributed parallel filesystems such as HDFS [67], GFS [46], GlusterFS [40], or Ceph [70] are used in large scale intra-/inter- datacenter deployments [12]. These systems often focus on providing high availability and scalability by spreading blocks of data and replicating them across servers. Hailstorm is designed with a different goal in mind: enabling database instances within a rack to pool their storage resources. By targeting rack-scale deployment and specializing our filesystem, we can provide high storage utilization, provide optimizations such as prefetching sstables blocks and accessing files at different block granularities. Our use cases also exclude concurrent parallel accesses to the same file, greatly simplifying metadata and consistency.

Disaggregated storage Many systems performing storage disaggregation were proposed recently. Flash storage disaggregation aims to improve storage capacity and IOPS utilization by accessing remote flash devices [48]. Hailstorm performs file-level disaggregation as opposed to block-level, allowing us to support high-level operations such compaction offloading. Each file in Hailstorm is sharded at block-level to ensure uniform distribution of data across all storage devices in a rack. Storage disaggregation is a feature of LegoOS [66], an operating system designed specifically for hardware resource disaggregation, but storage is not the main focus of this work and, unlike Hailstorm, LegoOS's approach is not meant to maximize storage utilization. Finally, disaggregated storage has also been successfully used in blob stores [57] and analytics [29, 64]. To the best of our knowledge, Hailstorm is the first disaggregated storage system targeting distributed databases with LSM storage engines.

Distributed in-memory storage Several distributed in-memory storage systems have been proposed recently [30, 43, 58]). These systems are about pooling main memory, not secondary storage. Although Hailstorm leverages the buffer cache when pooling storage devices, its main focus remains on pooling secondary storage.

Two-level sharding Social Hash [65] is a framework running in production at Facebook that aims to optimize query response time and load balance in large social graphs by using two-level sharding. In this scheme, data objects are first partitioned using a graph partitioning algorithm before being dynamically assigned in groups. By dynamically assigning data objects to groups, the system can react to changes in the workload and achieve better load balance. Hailstorm similarly leverages the increased flexibility offered by using a two-step data assignment scheme, but targets LSM-based databases and relies on a filesystem solution to uniformly redistribute data blocks across storage devices.

B-tree load balancing Yesquel's approach to splitting B+ tree nodes improves load balance and reduces contention, but does not achieve uniform distribution across all database instances [21]. Furthermore, this approach may lead to many unnecessary splits if load intensity varies across keys over time. Although Hailstorm provides improvements mainly for write-intensive workloads, our block-level sharding would still improve storage load balance in Yesquel. In addition, although we have not explored this, it should be possible for Yesquel or other similar systems to use Hailstorm for split offloading. MoSQL relies on a B-tree design and keeps all data in main memory for fast access [68]. This reduces contention and load imbalance, but places a hard cap on the total size of the database. Hailstorm has no such limitation since we use secondary storage.

Compaction offloading Using a dedicated remote compaction server for a replicated store has been previously proposed in the context of HBase [22]. In this scheme, the system offloads large compactions to a dedicated remote compaction server relying on replication to provide fast data access. Hailstorm takes a different approach by exploiting rack-locality to create a storage management layer underneath storage engines that allows fast access to data without depending on replication. We offload compaction tasks in a peer-to-peer manner which does not require complex centralized decision making. Finally, our solution works for any distributed database using LSM storage engines.

7 Conclusions

As the scale of distributed databases grows and their performance requirements become more stringent, solutions that can address challenging issues such as the presence of skew at scale become necessary. We have made the case for deploying distributed databases over Hailstorm, a system that disaggregates compute and storage in order to scale both independently and improve load balance. Hailstorm consists of a storage layer that pools storage across the nodes of a rack, allowing each storage engine to utilize rack storage bandwidth. This effectively provides storage-level sharding, which helps mitigate the impact of skew, addresses per-node capacity limitations, and absorbs I/O spikes. Hailstorm leverages its storage layer to perform compaction offloading and reduce CPU pressure on overloaded machines.

In this paper, we focused on deployments of Hailstorm to improve utilization within a single rack. We leave for future work the design of extensions to increase performance across racks, such as a Hailstorm-supported cross-rack data migration mechanism to replace resharding.

8 Acknowledgments

We would like to thank our anonymous reviewers, Michael Stumm, Michael Cahill, Calin Iorgulescu, and Baptiste Lepers, for their feedback that improved this work. We also thank Oana Balmau for providing us with the Nutanix production traces. This work was supported in part by the Swiss National Science Foundation NRP 75 Grant No. 167157.

References

[1] 2019. https://www.mongodb.com/.
[2] 2019. http://cassandra.apache.org/.
[3] 2019. https://myrocks.io/.
[4] 2019. https://github.com/Netflix/dynomite.
[5] 2019. https://www.aerospike.com/.
[6] 2019. https://dgraph.io/.
[7] 2019. https://en.bitcoin.it/wiki/Bitcoin_Core_0.11_(ch_2):_Data_Storage.
[8] 2019. https://github.com/mongodb-partners/mongo-rocks/.
[9] 2019. https://pingcap.com/.
[10] 2019. https://tikv.org/.
[11] 2019. https://github.com/rescrv/HyperLevelDB.
[12] 2019. https://en.wikipedia.org/wiki/List_of_file_systems.
[13] 2019. https://github.com/libfuse/libfuse/.
[14] 2019. http://www.pnfs.com/.
[15] 2019. https://github.com/jnr/jnr-ffi.
[16] 2019. http://akka.io/.
[17] 2019. http://freecode.com/projects/fio.
[18] 2019. http://www.wiredtiger.com/.
[19] 2019. https://github.com/mongodb/mongo/wiki/Sharding-Internals.
[20] Veronika Abramova, Jorge Bernardino, and Pedro Furtado. 2014. Evaluating cassandra scalability with YCSB. In *International Conference on Database and Expert Systems Applications*. Springer, 199–207.
[21] Marcos K Aguilera, Joshua B Leners, and Michael Walfish. 2015. Yesquel: scalable sql storage for web applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 245–262.
[22] Muhammad Yousuf Ahmad and Bettina Kemme. 2015. Compaction management in distributed key-value datastores. *Proceedings of the VLDB Endowment* 8, 8 (2015), 850–861.
[23] J Chris Anderson, Jan Lehnardt, and Noah Slater. 2010. *CouchDB: The Definitive Guide: Time to Relax.* " O'Reilly Media, Inc.".
[24] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. 2018. Sharding the shards: managing datastore locality at scale with Akkio. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 445–460.
[25] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 40. ACM, 53–64.
[26] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. *Proc. of ATC* (2017).
[27] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 753–766. https://www.usenix.org/conference/atc19/presentation/balmau
[28] Cristina Băsescu, Christian Cachin, Ittay Eyal, Robert Haas, Alessandro Sorniotti, Marko Vukolić, and Ido Zachevsky. 2012. Robust data sharing with key-value stores. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE, 1–12.
[29] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. 2018. Rock you like a hurricane: taming skew in large scale analytics. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 20.
[30] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The end of slow networks: it's time for a redesign. *Proceedings of the VLDB Endowment* 9, 7 (2016), 528–539.
[31] Dhruba Borthakur. 2013. Under the Hood: Building and open-sourcing RocksDB. *Facebook Engineering Notes* (2013).
[32] Martin C Brown. 2012. *Getting Started with Couchbase Server: Extreme Scalability at Your Fingertips.* " O'Reilly Media, Inc.".
[33] Josiah L Carlson. 2013. *Redis in action.* Manning Shelter Island.
[34] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 4.
[35] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. 2011. TPC-E vs. TPC-C: characterizing the new TPC-E benchmark via an I/O comparison study. *ACM SIGMOD Record* 39, 3 (2011), 5–10.
[36] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. 2011. TPC-E vs. TPC-C: characterizing the new TPC-E benchmark via an I/O comparison study. *ACM SIGMOD Record* 39, 3 (2011), 5–10.
[37] Douglas Comer. 1979. Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (June 1979), 121–137. https://doi.org/10.1145/356770.356776
[38] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 143–154.
[39] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 8.
[40] Alex Davies and Alessandro Orsaria. 2013. Scale out with GlusterFS. *Linux Journal* 2013, 235 (2013), 1.
[41] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, Vol. 41. ACM, 205–220.
[42] Thibault Dory, Boris Mejías, Peter Van Roy, and Nam Luc Tran. 2011. Comparative elasticity and scalability measurements of cloud databases. In *Proc of the 2nd ACM symposium on cloud computing (SoCC)*, Vol. 11. Citeseer.
[43] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. 401–414.
[44] Paul DuBois and Michael Foreword By-Widenius. 1999. *MySQL*. New riders publishing.
[45] Sanjay Ghemawat and Jeff Dean. 2011. LevelDB. *URL: https://github.com/google/leveldb,% 20http://leveldb. org* (2011).
[46] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. *The Google file system*. Vol. 37. ACM.
[47] Kien A Hua and Chiang Lee. 1991. Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning.. In *VLDB*. Citeseer, 525–535.
[48] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. 2016. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 29.
[49] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 447–461.
[50] Justin Li and Florian Weingarten. 2019. Zero-Downtime Rebalancing and Data Migration of a Mature Multi-Shard Platform. USENIX Association, Dublin.
[51] Martin Maas, Tim Harris, Krste Asanovic, and John Kubiatowicz. 2015. Trash Day: Coordinating Garbage Collection in Distributed Systems.. In *HotOS*.

[52] Holger Märtens. 2001. A Classification of Skew Effects in Parallel Database Systems. In *European Conference on Parallel Processing*. Springer, 291–300.

[53] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, Vol. 2. 21–33.

[54] Michael Mitzenmacher. 2001. The Power of Two Choices in Randomized Load Balancing. *Trans. Parallel Distrib. Syst.* 12, 10 (2001).

[55] Jaeseok Myung, Junho Shim, Jongheum Yeon, and Sang-goo Lee. 2016. Handling data skew in join algorithms using MapReduce. *Expert Systems with Applications* 51 (2016), 286–299.

[56] Satoshi Nakamoto et al. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[57] Edmund B Nightingale, Jeremy Elson, Jinliang Fan, Owen S Hofmann, Jon Howell, and Yutaka Suzue. 2012. Flat Datacenter Storage.. In *OSDI*. 1–15.

[58] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2016. The case for RackOut: Scalable data serving using rack-scale systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 182–195.

[59] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 69–84.

[60] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[61] David A Patterson, Garth Gibson, and Randy H Katz. 1988. *A case for redundant arrays of inexpensive disks (RAID)*. Vol. 17. ACM.

[62] Meikel Poess and Chris Floyd. 2000. New TPC benchmarks for decision support and web commerce. *ACM Sigmod Record* 29, 4 (2000), 64–71.

[63] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 497–514.

[64] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 410–424.

[65] Alon Shalita, Brian Karrer, Igor Kabiljo, Arun Sharma, Alessandro Presta, Aaron Adcock, Herald Kllapi, and Michael Stumm. 2016. Social hash: an assignment framework for optimizing distributed systems operations on social networks. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 455–468.

[66] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. Legoos: A disseminated, distributed {OS} for hardware resource disaggregation. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 69–87.

[67] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. Ieee, 1–10.

[68] Alexander Tomic, Daniele Sciascia, and Fernando Pedone. 2013. MoSQL: An elastic storage engine for MySQL. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, 455–462.

[69] Mehul Nalin Vora. 2011. Hadoop-HBase for large-scale data. In *Computer science and network technology (ICCSNT), 2011 international conference on*, Vol. 1. IEEE, 601–605.

[70] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 307–320.

[71] Zhen Ye and Shanping Li. 2011. A request skew aware heterogeneous distributed storage system based on Cassandra. In *Computer and Management (CAMAN), 2011 International Conference on*. IEEE, 1–5.