CloudJump: Optimizing Cloud Databases for Cloud Storages

Zongzhi Chen, Xinjun Yang, Feifei Li, Xuntao Cheng, Qingda Hu, Zheyu Miao, Rongbiao Xie, Xiaofei Wu, Kang Wang, Zhao Song, Haiqing Sun, Zechao Zhuang, Yuming Yang, Jie Xu, Liang Yin, Wenchao Zhou, Sheng Wang
Alibaba Group
{zongzhi.czz, xinjun.y, lifeifei, xuntao.cxt, qingda.hqd, zheyu.mzy, xierongbiao.xrb, jacketwoo.wxf, gary.wk, songzhao.sz, haiqing.shq, zechao.zzc, yuming.yym, bayan.xj, allen.yinl, zwc231487, sh.wang}@alibaba-inc.com

ABSTRACT

There has been an increasing interest in building cloud-native databases that decouple computation and storage for elasticity. A cloud-native database often adopts a *cloud storage* underneath its storage engine, leveraging another layer of virtualization and providing a high-performance and elastic storage service without exposing complex storage details. It helps reduce the maintenance cost and expedite development cycles for the database kernels. We have observed that there are significant differences between the local and the cloud storage that invalid many designs inside existing databases when they are ported to the cloud storage. In this paper, we analyze the challenges and opportunities of both B-tree and LSM-tree-based storage engines when they are deployed on a cloud storage. We propose an optimization framework that guides database developers to transform on-premise databases into their cloud-native counterparts. We use a B+-tree-based InnoDB as a demonstration vehicle where we have implemented a suite of optimizations using the proposed framework and extend such efforts to the LSM-tree-based RocksDB. On both engines, our evaluations show significant performance improvements on the cloud storage.

PVLDB Reference Format:
Zongzhi Chen, Xinjun Yang, Feifei Li, Xuntao Cheng, Qingda Hu, Zheyu Miao, Rongbiao Xie, Xiaofei Wu, Kang Wang, Zhao Song, Haiqing Sun, Zechao Zhuang, Yuming Yang, Jie Xu, Liang Yin, Wenchao Zhou, Sheng Wang. CloudJump: Optimizing Cloud Databases for Cloud Storages. PVLDB, 15(12): 3432 - 3444, 2022.
doi:10.14778/3554821.3554834

1 INTRODUCTION

Cloud storage has become the *de facto* choice to build a cloud-native database. A cloud storage is an elastic and distributed block-based storage which can serve as a shared-everything storage layer for multiple database (computational) instances, providing QoS (Quality of Service) guarantees, large capacities, elasticity, and pay-as-you-go pricing models. For most cloud providers and cloud users, having a cloud storage service is a much more appealing choice than building and maintaining bare metal SSD clusters. Thus, instead of building and optimizing a dedicated storage service for a cloud-native database or optimizing on-premise databases on local SSDs, harnessing existing cloud storage services for building a cloud-native database is a popular and viable choice. Moreover, as cloud storage services have been nearly standardized across various clouds, engineering efforts made based on one cloud service provider could potentially be transferred to others, making it easier to offer a universal cloud-native database service with the same level of QoS on multiple clouds to avoid vendor lock-in.

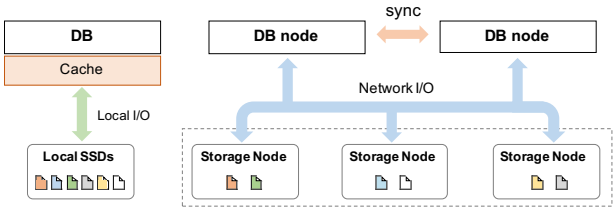

Figure 1: On-premise database v.s. cloud-native database

Figure 1 compares the architecture of an on-premise database on local SSDs with a cloud-native database using a cloud storage. The cloud storage supports multiple database nodes with a distributed but shared storage tier. Amazon Aurora [34, 35] pioneers such migration towards cloud-native databases from on-premise ones. It decouples the database into a storage tier and a computation (i.e., query and transaction processing) tier, and can scale each tier independently. It further customizes the storage tier to allow redo logs to be applied on data pages, so that data pages no longer have to be transferred between the two tiers. Although this design eliminates the heavy network overhead from transferring data pages, it comes with a non-standard storage service in the cloud that could only be used by Aurora's computation tier (i.e., a customized MySQL kernel). This hinders its storage service from becoming a universal and optimized one for other cloud-native databases to achieve a necessary level of performance and QoS.

In this paper, we propose CloudJump, a framework that we derive from our experiences in optimizing an industrial database storage engine, InnoDB, as a standardized cloud storage service, and extend to general cases including both B-tree and LSM-tree-based storage engines. To this end, we analyze the differences between cloud storage and local SSD storage in terms of bandwidth, latency, elasticity, capacities, and their impacts on the designs of a database.

We identify the following technical challenges: (1) high I/O latency in accessing the cloud storage service caused by its remote and distributed storage cluster, (2) the often underutilized aggregated I/O bandwidth, (3) conventional designs that work well on a single machine with its local storage but need to be adapted for

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554834

the cloud storage, for example, page caches, and (4) low isolations among various database I/O operations (e.g., flushing of massive logs contends with page read I/Os). On top of this I/O landscape in the cloud, the very large tables (e.g., tens of TB) that users are encouraged to use instead of sharding, taking advantage of the cloud storage, amplifies these challenges. For example, the underutilized aggregated I/O bandwidth and I/O contentions can have severe negative impacts on a long-range query performed on a very large table, causing it to violate performance QoS guarantees.

To address these challenges, CloudJump introduces a suite of design considerations and optimizations, including thread-level parallelism, task-level parallelism, prefetching, fine-grained locking / lock-free structure, scattering access among distributed nodes, bypassing caches and scheduling prioritized I/O tasks. For each optimization, we have applied it to two storage engines: B-tree-based InnoDB and LSM-tree-based RocksDB [13, 14], and illustrate and analyze how it performs on cloud-native databases with a general and standard cloud storage service. Note that the B-tree and LSM-tree families represent the most popular database systems that are either moving towards or already available as a cloud-native service, such as Aurora, PolarDB, TiDB, ClickHouse, etc.

Our implementation of CloudJump has been running online in more than 15,000 database clusters consisting of more than 45,000 database nodes in Alibaba Cloud since 2019. In this paper, we evaluate CloudJump on top of InnoDB and RocksDB respectively in two case studies. Results show that CloudJump has achieved significant performance improvements on cloud storage, compared with the baseline design optimized for local SSDs.

Our main contributions are summarized as follows:

(1) We have studied the performance characteristics of a typical cloud storage, and compared it with local SSDs that most database storage engines are designed and optimized for. These efforts lead to a systematic summary of the challenges of moving a database towards cloud storage (for building a cloud-native database), the design knobs that developers need to revisit, and the underlying problems that must be addressed in both update-in-place B-tree and append-only LSM-tree-based storage engines.
(2) We have proposed a framework CloudJump with detailed design considerations and optimizations to address the challenges posed by a cloud storage, and implemented the framework in InnoDB (B-tree based), as a demonstration vehicle, and extended it to RocksDB (LSM-tree based). For each system, we provide a dedicated case study to analyze the gains and costs with evaluations.
(3) CloudJump and its implementation on InnoDB has been running online since 2019. It now serves more than 15,000 database clusters with stable performance and a high level of QoS. We have discussed ways to extend CloudJump to more databases that are migrating to use a cloud storage.

This paper is organized as follows. We introduce characteristics of the cloud storage and their impacts on both B-tree and LSM-tree based databases in Section 2. Then, we introduce our design consideartions and optimizations in Section 3 and its implementations on InnoDB and RocksDB in Section 4 and Section 5 respectively, with evaluations. We discuss extensions of CloudJump to more databases in Section 6, the related work in 7, and conclude in Section 8.

Table 1: Specifications of cloud block storages.

	Network	Volume[1]	Bandwidth[1]	IOPS[1]	Latency[2]
SSD P3700	×	2 TB	1.9 GB/s	100K	31 / 6 us
Storage X	TCP/IP	2 / 64 TB	1.0 / 4 GB/s	64.8K/256K	868 / 194 us
ESSD	RDMA	2 / 32 TB	1.2 / 4 GB/s	100K/1000K	287 / 113 us
PolarStore	RDMA	2 / 100 TB	2.4 / 4 GB/s	190K/1000K	229 / 88 us

[1] List purchased / maximum specifications per volume for cloud storage.
[2] For 16 KB writes tested @ 80% BW, steady state.

2 BACKGROUND AND MOTIVATION

2.1 Cloud Storage

We refer to cloud storage as an elastic and distributed block-based storage that is able to serve as a shared-everything storage tier for multiple database (a.k.a. computation) nodes. Table 1 lists the specifications of selected block-based storage in the cloud, including Amazon Elastic Block Store [3], Alibaba Cloud Enhanced SSDs [2] and other distributed shared storages [8, 11]. Such cloud storage has been the backbone for many mission-critical and cloud-native databases such as AWS Aurora and Alibaba PolarDB. Other types of storage services in the cloud, e.g., object-based storage, are left out of the scope of this paper.

In Figure 1, we compare the typical architectures of an on-premise database on the left and a cloud-native database on the right. The on-premise database usually uses local disks, e.g., SSDs, where all its data are stored locally and easily cached in the main memory for fast access. In this architecture, the computation and storage capacity of a database is tightly bounded together as local resources (e.g., CPUs, SSDs) and can not be easily scaled to match dynamic workloads. To address these limitations, a cloud-native database built on cloud storage can scatter its computation and storage among multiple nodes. The cloud storage service manages the addition or removal of nodes and workload balancing elastically in a way that is transparent to databases and users.

While the cloud-native architecture achieves elasticity and allows on-demand usages, there are several technical challenges buried in the underlying architecture. Table 2 summarizes such challenges as well as their impacts on the design of databases. Latency and bandwidth differences are the first two major challenges. This is due to accessing a remote node via the network takes a longer time. And, a cluster of nodes offers higher total bandwidth in an aggregated fashion. Database designs optimized for local disks are neither sufficient to address the prolonged latency nor efficient at utilizing (or saturating) the high aggregated bandwidth. Moreover, there are other challenges caused by the distributed architecture nature of a cloud storage. Firstly, there is an expensive overhead caused by maintaining consistent caches or buffers for all database nodes. Secondly, I/O stragglers in a cluster may cause the database to respond to queries with unacceptable latencies. To address it, we highlight the importance of isolating I/O tasks and scheduling I/Os accordingly. In Figure 2 and 3, we offer a glimpse of the differences between the local SSDs and the cloud storage. In section 2.2, we introduce the impacts of these challenges on the database design.

Table 2: Challenges of utilizing the cloud storage, their impacts on the design knobs of databases and our design considerations.

Challenges	Design knobs	Problems		Design Considerations
		Update-in-place	Append-only	
Local accesses v.s. remote accesses	Write-ahead log	Slow serial logging		Thread-level parallelism
	Log replay	Applying logs to multiple pages	Bulk writing and flushes of memtables	Task-level parallelism
	Data read	Loading dependent remote pages	Read amplifications	Prefetching
	Synchronization	Blocking writes while updating pages	Compactions with amplified writes and low I/O utilization	Fine-grained locking and lock-free data structures
Local bandwidth v.s. aggregated bandwidth	Data write	Low bandwidth utilized accessing a single page		Scattered accesses among distributed noes
	Data read		Slow and amplified reads	
Consistency among multiple database nodes	Data cache and buffer	With cache: high consistency overhead, Without cache: amplified I/O with no buffers		Bypassing a centralized cache
I/O isolation	I/O scheduling	Concurrent and extensive log and data I/Os cause unpredictable performance		Differentiating prioritized I/O tasks

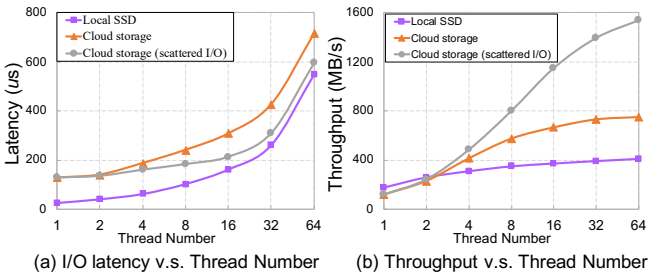

Figure 2: Single-file write performance of local and distributed storages (*both steady state*).

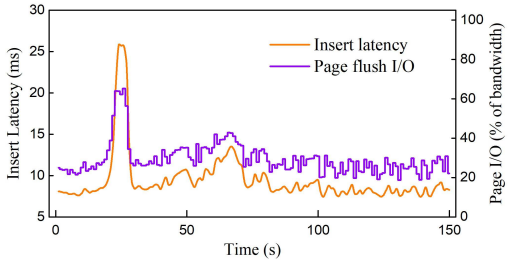

Figure 3: Influence of I/O isolation.

Figure 2 reports the average latency and bandwidth measured while a single database node accesses a single database file on local SSDs or on a cloud storage with and without scattering the I/O, respectively. The cloud storage has higher latencies (up to 3.3x in the worst case) and throughput (up to 3.8x in the base case) than local SSDs in general. With multiple threads scattering their I/Os across multiple nodes, the latencies observed only suffer from a marginal slowdown compared with local SSDs. Bandwidth-wise, scattering I/Os pays back nicely. With few threads doing the job (e.g., 1 or 2), the largest latency slowdown and the lowest bandwidth improvement happen at the same time, forming the worst case. This shows that it is very likely to achieve a higher level of performance on the cloud storage with necessary optimizations on the level of concurrency, compared with local SSDs.

In Figure 3, we have done a simple experiment mixing a constant level of insert queries with a single batch of flushing dirty pages, at around 20 seconds into the experiment, on the cloud storage. The flushing I/O almost immediately consumed 80% of the total bandwidth, leaving the rest for insert queries to compete for and raising the latency of which by 311%.

2.2 Update-in-place and append-only storage engines

Update-in-place and append-only are two major methods for updating a record or an entry in a data structure. The former usually guarantees that the position of a record can be deterministically determined by the primary key itself. All updates and deletes on records happen in place. The latter treats the data store as a log-like structure, at the tail of which new records or changes to existing ones are appended. To retrieve a record, its "base" has to be merged with all its deltas to form its latest and complete version. This design reduces the write complexity at the cost of that of reads. Usually, asynchronous compaction has to be called periodically to merge pieces of the same record to bound the worst lookup performance and storage cost.

B-tree and LSM-tree, including their variants, are two popular designs that implement these update-in-place and append-only methods, respectively. They are the backbone for many crucial data structures (e.g., indexes, KV stores) in many modern databases (e.g., MySQL, RocksDB, ClickHouse) [4, 9, 14, 18, 29]. In MySQL's default storage engine, InnoDB, each table has a clustered index, in which the primary keys of this table are stored in a B+ tree (a variant of B-tree) with all attributes stored in the leaf. And, both leaf and non-leaf nodes are sorted in pages. In RocksDB, a LSM-tree-based storage engine, each table has in-memory memtables for all recent changes and an on-disk tiered storage for the "base" of existing data.

Memtables are flushed to the disk and merged with all those tiers over time. Both systems share a similar process for recoveries, in which redo or undo actions are derived from durable WALs (write ahead logging) and replayed.

In a B-tree, the target data page has to be retrieved synchronously for updates, and inserts, which may block other simultaneous operations to avoid conflicts. While looking up a key, we only know which page to load from the storage after looking up the key in the current page. Such data and logical dependencies expose the access latency in the query response time. And, because records that are adjacent logically are stored together physically in the update-in-place method, queries and transactions tend to have a strong spatial locality.

The LSM-tree [13, 14] removes the synchronization overhead as in the update-in-place method but introduces new issues such as extensive flushes of data (memtables) from the main memory to disks, frequent and heavy compactions, and amplifications for reads and writes. These operations suffer from the prolonged latencies in the cloud storage.

To build a database storage engine using these two data structures, there are some common components added to the system, the designs of which also need to be revisited when using a cloud storage. The write-ahead logging (WAL) mechanism [27], storing changes into the durable storage for recovery purposes, incorporates the long latency into the critical path of transaction processing. Building a cache among those distributed nodes in the cloud storage suffers from the consistency overhead that grows with more nodes added to the cluster. Without caches, more accesses have to go remote, compared with a database on local SSDs. There is also a strong requirement to differentiate logs and data for predictable performance, as shown in Figure 3.

Table 2 summarizes these impacts on the design of databases. To each impact and challenge, we have listed our design considerations and optimizations in CloudJump.

3 DESIGN CONSIDERATIONS

In this section, we introduce our design considerations as well as the rationale leading to them for the challenges and problems listed in Table 2. For each consideration, we introduce its implementation details in the two case studies presented in Section 4 and 5.

3.1 Thread-level Parallelism

Write-ahead logs (or other types of database logs) carry the information that a database needs to persist before committing a transaction, so that it knows how to recover its state after a crash or failure. In single-machine storage engines, e.g., the conventional InnoDB, changes from multiple small transactions are usually merged together in a global buffer before they are flushed to the disk, to achieve higher I/O performance by writing a larger chunk of data sequentially. Such a global buffer creates a synchronization point in the log flushing process that hinders the opportunity of applying thread-level parallelism (TLP).

In CloudJump, we emphasize the importance of partitioning such global log buffers by page IDs for B-tree-based databases, so that TLP can be applied at the partition level and sequential writes are still achieved for each page in the cloud storage. For LSM-tree-based RocksDB, log buffers can be partitioned by sub-tables (part of a relational table materialized as an LSM-tree) on which the changes they carry are supposed to be applied, or their corresponding log sequence numbers.

For both types of systems, there is very likely a gap between their own partitioning granularity and the optimal I/O job size for better latency and throughput. Moreover, these I/O jobs into and out of the cloud storage need to be aligned with the block sizes to reduce read/write amplification and avoid the need for padding. To address this, we have proposed a two-level design of log flushing. The first level, as introduced above, partitions logs in the main memory; the second level, the log writer, further tunes the I/O job size by slicing each log partition (if needed), and aligns I/O writes with the block size of the cloud storage (with padding if needed [21]). It further adjusts the number of asynchronous thread workers that write these slices to the cloud storage, to achieve a reasonably high level of latency and throughput, as illustrated in Figure 5.

3.2 Task-level Parallelism

In addition to the thread-level parallelism that parallelizes the writing of WALs, we also introduce task-level parallelism that further increases the level of concurrency. For example in B-tree-based InnoDB, the recovery process needs to apply logs to pages from various tables, causing flushes of dirty pages. All of them need to be finished before the database can resume its services. The same applies to LSM-tree-based RocksDB, where logs need to be replayed for all sub-tables, often resulting in flushes of memtables. By executing multiple table-level tasks, the system is able to read from and write to a larger number of nodes for higher throughput, compared with only exploiting thread-level parallelism.

3.3 Prefetching

Prefetching techniques can potentially achieve larger performance gains on the cloud storage compared with those on local SSDs, because they save larger I/O costs on the cloud storage. With update-in-place or B-tree in general, once the first page that satisfies the predicate has been located, its following siblings will be prefetched to the database nodes with a very high probability because records are well sorted in such data structures. With append-only or LSM-tree in general, prefetching also helps when multiple extents (data blocks) from the storage are retrieved and merged for either range queries or compactions.

3.4 Fine-grained Locking and Lock-free Data Structures

The synchronization overheads are amplified by the longer I/O latencies in the cloud storage. Therefore, there are stronger needs for fine-grained locking, or even lock-free indexes, that lock data as little as possible to minimize the chances of contention between simultaneous operations. This is crucial for update-in-place B-trees where adding records into the tree may trigger structure modification operations (SMOs) that need to add exclusive lock on parts of the tree. In append-only LSM-trees, there is no such overhead when appending new records. However, background compactions do need to lock the inputs before materializing the merged outputs.

Thus, many LSM-tree optimizations exercise the copy-on-write techniques that trade memory spaces for higher performance.

3.5 Scattering among Distributed Nodes

While scattering accesses among multiple nodes in the cloud storage could utilize more hardware resources, the opportunities of doing so are determined by the placement of data in the cloud storage at the first place. Assuming the cloud storage manages data at a chunk level, there are at least two strategies for such placement:
Compacted: allocate a new chunk on the same node with the previous allocation as long as there is sufficient space on that node.
Balanced: allocate a new chunk in the node with the largest free space. The compacted strategy uses fewer machines for the same capacity and offers less aggregated bandwidth at the same time.
We have observed that many cloud storage services apply a hybrid approach combining both of these two strategies: use the compacted strategy for small-size allocations and use the balanced strategy as the size of requested spaces increases.

With this in mind, the log write we introduced in Section 3.1 has been designed to be able to flush slices of a single log file into multiple physical files, in an attempt to trigger more chunk allocations in the cloud storage and utilize the resources of multiple nodes for higher bandwidth utilization.

3.6 Bypassing Caches

While caching pages from the cloud storage in the main memory of database nodes helps reduce remote I/Os for hot data, we have to add the maintenance of such caches in the transaction processing pipeline to guarantee that the contents of the caches in each database node is up to date for queries, that is, they are coherent with the latest database state. Importantly, the overhead for maintaining cache coherence increases rapidly with the size of these caches and the number of nodes added into the cluster.

In CloudJump, we argue that bypassing caches to save this overhead is likely preferable: developers can divert the engineering efforts to other techniques, such as prefetching, and can still achieve comparable performance. More importantly, such an approach of bypassing caches allows the database performance to scale nicely with the size of the database cluster when deployed on the cloud storage.

3.7 Scheduling Prioritized I/O Tasks

There are four major types of I/O tasks in a database: writing and reading of logs, writing and reading of data pages. Among these, writing logs is critical for committing an incoming transaction, while the logs are only read for replications or recoveries. Regarding pages, the writing of dirty pages is less critical as its contents have been made durable in logs; the delay in the writing of data pages is tolerable, and, in fact, often preferred for performance considerations. On the other hand, the reading of pages is often on the critical path of query and transaction processing, and its performance strongly impacts the overall performance.

From the perspective of keeping the database's high availability for OLTP workloads, we argue that the writing of logs should be prioritized over the rest to avoid transaction stalls. Page reading, on the other hand, should be carefully tuned for query performance.

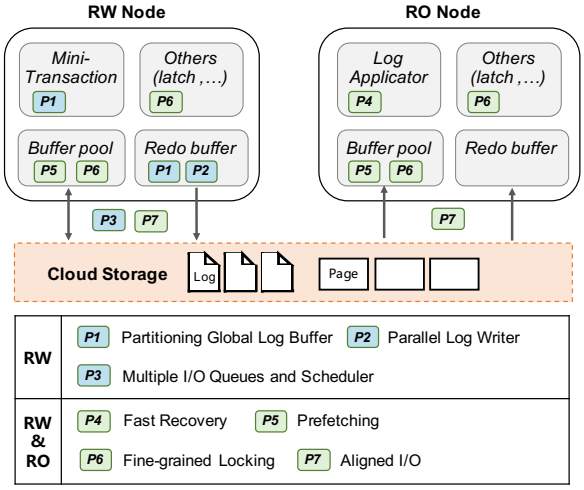

Figure 4: InnoDB Optimizations in PolarDB.

Page flushes can be delayed, and the reading of logs usually only happens during the replication or recovery, and therefore can be de-prioritized. To reflect this design consideration, we introduce in CloudJump an I/O scheduler that isolates different types of I/Os and schedules them according to their priorities.

4 CASE STUDY: POLARDB

PolarDB incorporates MySQL together with its default B-tree-based storage engine, InnoDB, as its computation layer, and applied CloudJump to suit the cloud storage. Figure 4 illustrates our implementation of CloudJump with respect to their associated components. In this cloud-native database architecture, the single read-write (RW) node is differentiated from the rest read-only (RO) nodes. There are up to 15 RO nodes in total, which can be added to the cluster elastically according to user workloads. Only the InnoDB in the RW node is responsible for transaction processing that modifies the state of the database, on which we apply the optimizations on the log write path. On both RW and RO nodes, we apply the other optimizations on the page reads/writes and log reads.

4.1 Accelerating Persisting WAL

4.1.1 Partitioning Global Log Buffer. InnoDB organizes the processing of logs at the granularity of mini-transactions (MTRs) for various benefits in single-machine setups, the processing of which needs to be optimized for cloud storage. The baseline InnoDB divides a transaction into MTRs and limits the number of pages that an MTR can exclusively modify to only one. For each MTR, all its modifications are recorded in its own redo log records. InnoDB keeps a global log buffer that collects redo log records from multiple MTRs and flushes them to the disk for durability. When a transaction is committed, all its log records are appended into the tail of the global centralized redo log buffer and a Global Log Sequence Number (GLSN) is assigned to each log record for page version management.

This design bounds the scale of pages that a transaction exclusively latches during its processing to avoid holding a latch for too

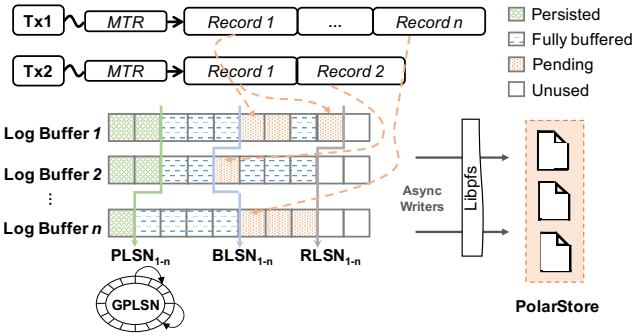

Figure 5: Partitioning global Log buffer.

long. It also reduces small I/Os by merging MTR's small log buffers into a global and sequential one. On the cloud storage, however, such buffer merging hinders the potential of the total bandwidth that can be achieved by scattering I/Os among multiple nodes. We choose to partition the global log buffer to allow the application of the **thread-level parallelism** that can scatter I/Os out. And, we choose to partition it by page IDs because a redo log record, when it is needed, is applied to its target page during recovery.

Figure 5 shows the above process where log records from transactions Tx_1's and Tx_2's MTRs are written to a series of *Log Buffers*. *Log Buffer*$_i$ is the buffer dedicated to page i. After partitioning, there are three main steps: logging, synchronizing, and writing steps. Firstly in logging, log records are copied into reserved spaces in a log buffer. Next, we track and update the offsets marking four different regions in the synchronizing step. Log records with an LSN smaller than Persisted LSN (PLSN) have persisted. Those between the PLSN and Buffered LSN (BLSN) for a commit have been fully collected and buffered, waiting to be persisted. And, those between BLSN and Reserved LSN (RLSN) are being copied from incoming log records from MTRs. The rest are unused spaces. We derive the global persisted LSN (GPLSN), for page version management, from PLSNs via a lightweight LinkBuf, which is a concurrent data structure extensively used in MySQL 8.0 and allows to track concurrently operations performed out of order [29].

In the final writing step, each log buffer partition is written to the storage nodes by a parallel log writer asynchronously, which scatters (see 4.1.2) and aligns (see 4.6.1) I/O and then writes log records up to the corresponding BLSNs. Once the file write is finished, we update the log states and advance the PLSN for each partition. After that, an async notifier is used to notify all workers of the updates.

In addition to the above design, we implement the RW-to-RO replication of redo logs via the RDMA network, so that scattered redo logs in the cluster do not slow down ROs trying to catch up with the RW. We also enable users to configure various consistency levels to trade RO's real-time visibility for performance.

4.1.2 Parallel Log Writer. Figure 6 shows the design of the parallel log writer, which further splits each log buffer into multiple slices, assigns one worker thread to each slice, and issues asynchronous I/O tasks from these threads concurrently to flush log records to logs files in the cloud storage. In the cloud storage, a file consists of multiple chunks, which may very likely locate in various storage

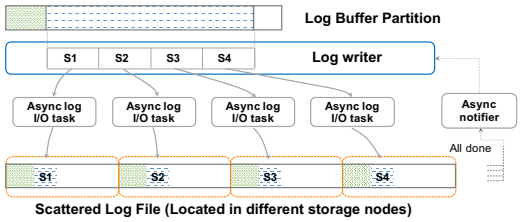

Figure 6: Behavior of parallel log writer.

nodes, according to the allocation strategy for chunks. With this layer of log writers, we are able to split a large I/O task that may be a straggler into multiple slices, for large I/O tasks (*e.g.*, group commit, BLOB update), and reduce the makespan by executing multiple asynchronous I/O tasks at the thread level concurrently.

4.2 Fast Recovery

There are two common scenarios involving log replaying in InnoDB: the crash recovery and the log applying in a RO node of PolarDB. In the traditional InnoDB recovery process, all files are first opened and read for meta-information (*e.g.*, file names, table ids, page ids) and validated for correctness during startup. Then, the uncheckpointed data are recovered via the ARIES-style algorithm [28].

Instead of scanning all files, we record and centralize the necessary meta-information during the lifecycle of databases. At the validating stage, it spends a larger time accessing the remote distributed files when executing the scanning task. To solve this problem, we collectively record necessary meta-information in a superblock (*e.g.*, the first block of a data chunk in the storage node) when creating table files. Note that the superblock is updated when certain meta information is updated, which is low-frequency. Moreover, all necessary metadata for different files is also centrally backed up in one global superblock. When validating, the checker uses the metadata in the superblock instead of getting them from all files. Therefore, instant scanning and validating can be realized with the reduced I/O overhead.

As is shown in Figure 7, the redo log file is divided into multiple partitions as introduced in Seciton 4.1.1, multiple recovery worker threads scan and parse the log records greater than the checkpoint GLSN within one partition. However, the record is always located through the local LSN in its partition. To achieve fast seeking, we record the GLSN mapping information in the block header of the log file. The survived transactions are detected and their redo contents in each partition are saved to the recovery cache (a hash table), it mainly contains two parts: the location information and the specific redo records. Finally, we can identify affected tables and record the involving dirty pages for each partition, and accordingly redo the updates. The incomplete transactions are rolled back in the background by parsing the log records into a separate undo list, and then undoing the updates.

During the recovery, the dependencies and constraints among different partitions need to be considered, *i.e.*, if a transaction is completed, all its redos should be recovered. If one transaction modifies multiple pages, the modifications should be atomic. To synchronize this, we put all the file operation and logic-related redo records into partition 0, and generate corresponding **constraint**

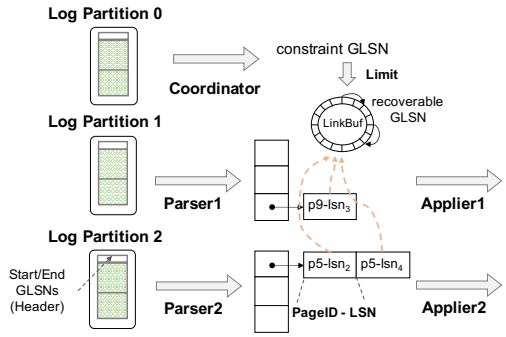

Figure 7: Overview of parallel recovery.

GLSNs to limit the advance of each worker's recovery and guarantee the correctness. The recovery of partition 0 is taken charge of by a coordinator as shown in Figure 7. The coordinator finds all the synchronization points during parsing, and then notifies other workers. All partitions add their maximum consecutive GLSN of scanned redo records into a LinkBuf. Then the coordinator advances the LinkBuf and gets the maximum **recoverable GLSN**. Until the recoverable GLSN achieves the constraint GLSN, all workers can go on to recover the redo records that have larger LSN than the constraint GLSN.

4.3 Prefetching

In databases, prefetching is widely used to improve cache performance or reduce long access latencies [40]. We apply prefetching in PolarDB in addition to traditional read-ahead algorithms in InnoDB.

For a primary index scan, InnoDB performs the prefetch in a logical order (based on the sequence of B-tree leaf nodes), which usually outperforms the physical order (based on the sequence of data in physical files) due to the unordered inserts and updates. For a secondary index scan on non-indexed columns, the worker thread should also look up the primary index for uncovered data. Then, single random data accesses occur with serial long I/Os. In this case, we collect relevant keys in batches when scanning. Keys in a finished batch are accumulated and arranged to the primary index order. Meanwhile, an asynchronous prefetching thread is used to retrieve the corresponding data from the primary index, while the remaining scan is still ongoing.

When performing an index search for updates, we selectively prefetch the logical and physical neighbours in the targeted leaf node's linked list. Due to localities, pages that are frequently queried may also experience frequent structure modification operations (SMOs). In the space cleaner task, if an index page falls below the merge threshold when a row is deleted or an update operation shortens a row, the next page will be needed for page merging. By prefetching the neighbouring page, modifications to the related structure are accelerated.

4.4 Fine-grained Locking

The high I/O latency would cause a longer blocked time for other threads acquiring the same resources. In this section, we introduce the following two representative cases.

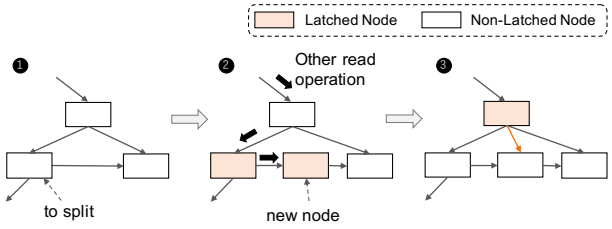

Figure 8: Lock-optimized B-tree Index.

4.4.1 Shadow page. To avoid an operation holding an index's exclusive lock for too long, and blocking other operations, we use shadow pages. In the baseline InnoDB, the thread holds the lock during the entire I/O request. With the shadow page, when the I/O is initiated, we first lock the page in shared exclusive mode and then copy a shadow page of the current page version in the memory Immediately, the lock of this page is released once the shadow page copy is finished. After that, the background I/O thread only writes the copied page into the cloud storage, while the original page can be read and updated. The lock holding time does not last for the entire I/O period but only for one memory copy operation. The time to hold the page's shared exclusive lock is significantly reduced, thereby improving the operation concurrency.

4.4.2 Index optimization. We apply the fine-grained latching policy motivated by the B-link tree [23], as shown in Figure 8 to address the following issue. When doing the structure modification operation (SMO) on the B-tree index, the parent node holds a write latch and then all other threads accessing it and its sibling nodes will be blocked. In addition, in the baseline InnoDB, a high-level index latch is used for consistency, which does not allow the concurrent SMOs and brings lower concurrency.

Instead of adding latches for all searched nodes along the way for SMO operations, we allow that only two latches of nodes are owned with the latch coupling method, *i.e.*, the parent node latch is only released when the latch was acquired successfully. Besides, the high-level index latch is removed, and multiple SMOs are also allowed with the following latching policy.

Generally, when a leaf node overflows, there are at least three nodes that should be updated: the splitting node that overflows, its parent node, and the newly created node. To avoid the deadlocks, we propose three locking rules for concurrency control: (1) The locking is only allowed from top to bottom and left to right; (2) Similar to the B-link tree, each non-leaf node also has a pointer to its right neighbour, and the SMO operation is divided into two steps, the first step is to create the new node and migrate data from the splitting node to it (❶ → ❷ in Figure). Note that only the splitting node and the newly created node hold the exclusive latches, remaining the parent node free. At this time, other read operations are allowed, but must first compare the splitting key with its high fence and proceed to read the new right neighbour, if the searched key is higher than the fence key, the procedure as shown in ❷ of Figure 8. After that, the latches are released but the two nodes are marked as the intermediate status since the newly created node is not yet referenced by the parent. The second step is to make a root-to-node traversal to lock the parent node and update it (❷ →

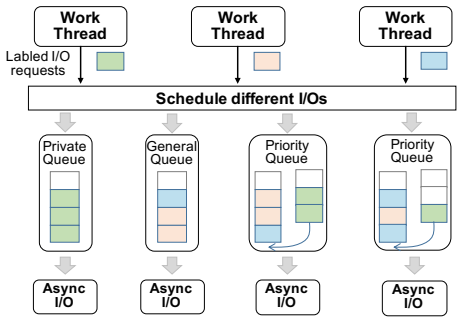

Figure 9: Asynchronous scheduled I/O model.

❸) since the locking order is required as mentioned before. (3) For other threads accessing the nodes with the intermediate status, they need to release their held locks and wait until the SMO operation has been completed, and then restart a root-to-leaf traversal.

4.5 Multiple I/O Queues and Scheduler

4.5.1 Multiple I/O Queues. In PolarDB, we establish multiple concurrent producer-consumer queues for async I/O, as shown in Figure 9, and the length of each queue can be sufficiently expanded. In this way, the overhead of heavy page flushing is amortized for both producers and consumers, and long queues can reduce the impact of network latency fluctuation. Besides, through this model, we can perceive different I/O types and schedule accordingly

4.5.2 I/O Scheduler. Cloud storage cannot distinguish whether the I/O request is data I/O or log I/O, or others, thus the storage layer treats all I/Os in a unified manner. The concern is that the non-critical I/Os may negatively delay the critical I/Os. We handle this problem by perceiving (labeling) and scheduling I/Os in the database layer shown in Figure 9.

Here, we take an example in which there exists log I/Os and data I/Os. As mentioned in 4.5.1, we adopt the asynchronous I/O model to handle the I/O requests from the worker threads, *e.g.*, log writer and dirty data flusher. Each queue is labelled as the private queue for log or the priority queue, and the total queue number is controlled based on the I/O capacity of storage. There is also a general queue shown in Figure 9 that treats requests equally.In each queue, an atomic variable is used to check the total existence of I/Os. When the I/O usage (tracked by existing task number) is low, log I/O requests are all sent to its private queues, and data I/O requests are sent to the priority queue. As the usage increases, log I/O requests are also sent to the priority queue evenly. Data I/Os can not invade the priority queue, since it is less critical but needs long-term high throughput. It can enter and executed in the priority queue only when there are no log I/O requests. Therefore, the redo log I/Os are prioritized over the data I/Os by setting round-robin priority with high reserved priority. In this way, a certain number of I/O resources are reserved and prioritized to serve the log writing. With increasing memory and network, the system can dynamically adjust the dirty data ratio in an extensive range, and the spikes are further flattened.

4.6 Aligned I/O

4.6.1 Aligned Log I/O. For both local SSDs and the cloud storage, the file is read/written with a fixed-size block. If the length or offset is not aligned to the block size, there may be more read-on-write or logical-to-physical mapping, especially without page cache. Usually, the cloud storage has a larger desired block size of a supposed single I/O (*e.g.*, 4-128 KB for PolarFS) than a local file system [7, 21, 31]. The origin redo log aligning strategy is no longer suitable for the stripe boundary in the cloud storage and even degrades I/O performance. On the cloud storage, during the procedure of committing log data into log buffers, we align both length and offset to the minimum desirable size. Before writing to cloud storage, a suitable block size is also selected based on the contents of the buffer and the desirable size of the cloud storage. After that, all log I/Os are aligned in the user space with optimal sizes and offsets, avoiding the aforementioned negative issues.

4.6.2 Aligned Data I/O. Generally, there are two types of data pages: the regular data page that can be aligned to the flush page size easily, and the small compressed page that can cause the unaligned offsets of following I/Os. In either case, we should align both regular and compressed page I/Os. Taking the small compressed page of MySQL as the example, and assuming that the flush page size is 4 KB and the current file offset is 3 KB, If the next I/O is 1 KB, we start writing from 3 KB and end at 4 KB; if the next I/O is 16 KB, we hole the first 3-4 KB, and start writing from 4 KB and end at 20 KB. Then, all data I/Os are aligned expectantly.

Removing Data I/O Merging: In Vanilla MySQL, the page I/O will be merged to form a large I/O request for serial sequential writing. In cloud storage, the concurrent writing into different storage nodes shows superiority, so the merge operation can be removed when the page I/O size is set to the flush page size of cloud storage.

4.7 Experiments and Evaluation

In this subsection, we evaluate: (1) how does PolarDB, with its InnoDB storage engines inside optimized by our implementations of CloudJump, compare with the open-source and baseline MySQL (version 8.0) deployed on local SSDs and the same cloud storage as used by PolarDB, (2) how does the same PolarDB compare with the AWS Aurora sitting on its own cloud storage in the AWS, (3) how does each individual optimization we have implemented contribute to the overall performance and (4) how does PolarDB compare with MySQL on various types of the cloud storage.

4.7.1 Experiment Setup. Local experiments are executed on machines with 32 cores featuring Intel Xeon E5-2682 v4 CPU and 504 GB of memory (DDR4 2666 MHz) with the specifications of P3700 SSDs listed in Table 1. All database instances we compare are configured to run on the same number of CPUs (i.e., 32) and main-memory buffer capacity (i.e., 256GB). The MySQL (local) and MySQL (PolarStore) we refer to are a single stand-alone instance on local SSDs with no replicas and a single MySQL instance on the PolarFS cloud storage, respectively. The PolarDB we use is a single-RW cluster with no ROs, running on the PolareStore, as the evaluation of ROs' elasticity is out of the scope of this paper.

We also evaluate MySQL and PolarDB, both using a single node, on various types of cloud storage: Storage X (a popular block-based

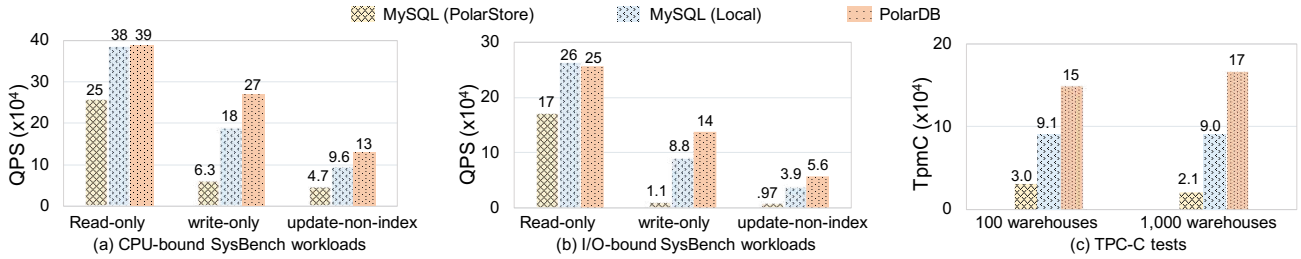

Figure 10: Total performance evaluation of PolarDB.

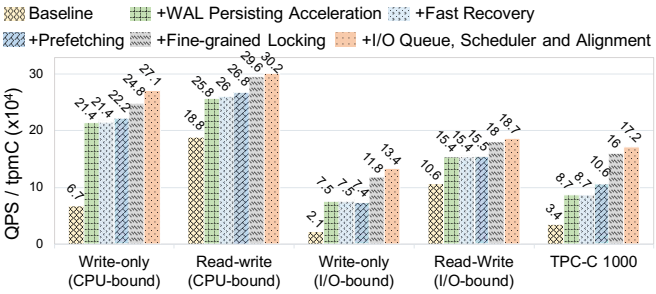

Figure 11: Performance Breakdown.

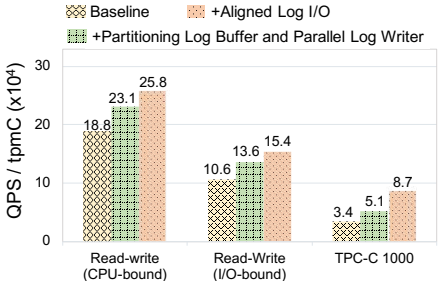

Figure 12: WAL acceleration breakdown.

storage service from another commercial public cloud), Alibaba Enhanced SSD (ESSD), and PolarStore with PolarFS. Table 1 outlines the specifications of these storages.

Benchmarks. We use two benchmarks: the SysBench [22] with read-only, write-only, read-write, and update-non-index workloads, and TPC-C. With SysBench, we generate two workloads: a CPU-bound workload, consisting of 50 2-GB tables (100 GB in total) and a IO-bound workload, consisting of 12 100-GB tables (1.2 TB in total). With TPC-C, we generate two workloads with 100 and 1,000 warehouses, respectively. All SysBench and TPC-C results are reported with the unit of queries per second (QPS) and transactions per minute (tpmC), respectively.

4.7.2 Evaluating Total Performance. 1) Overall throughput. Figure 10 shows the overall throughput with the two benchmarks. PolarDB outperforms all the others in all cases. MySQL (PolarStore), as a baseline for PolarDB with no proposed optimizations, performs the worst in all cases, despite the usage of the cloud storage. PolarDB generally achieves higher speedups in scenarios with more writes that lead to the flushing of WALs to the storage

Table 3: Read replica lag (millisecond).

Writes/sec	PolarDB	MySQL
5,000	0.85	$5E4$
10,000	1.03	$2.0E5$
100,000	3.27	$5.8E5$
200,000	4.92	×

and structure modification operations (SMOs) with locking overhead that we have addressed in CloudJump, compared with other cases. And, PolarDB achieves its largest speedups over MySQL (PolarStore) and MySQL (Local) in the write-only case for both the CPU-bound and the I/O bound workloads. The largest speedup is achieved for the write-only workload in the I/O-bound scenario, which contains the highest level of remote I/Os, where PolarDB is 12.73 times faster than its unoptimized baseline, MySQL (PolarStore). The TPC-C test, shown in Figure 10 (c), shows similar results, where PolarDB is around 5 times and 8.1 times faster than the baseline MySQL (PolarStore) in the case with 100, and 1,000 warehouses, respectively.

2) Replica lag We run the write-only workload on the RW node with varying number of connections. We measure the elapsed time starting from the time changes on a record are committed by a transaction in the RW node till the time a RO sees its corresponding version. Table 3 presents the results, where PolarDB is orders of magnitude faster than the MySQL baseline.

4.7.3 Breakdown Analysis of Performance Gains. Here we break the total performance improvements by individual optimizations that we have implemented. As shown in Figure 11, the baseline, MySQL (PolarStore), has no optimizations, while the last version "+I/O Queue, Scheduler and Alignment" has all optimizations and equals to PolarDB.

"+WAL Persisting Acceleration" has the most profound impact on transaction processing in all these workloads. In the CPU-bound workloads, it has achieved the highest performance improvement (69%). For the other two write-intensive workloads, it contributes about 48% to the total improvement. The benefits of "+Fast Recovery" are not obvious in the write-only and read-write mixed scenario. It has contributed to the reduction of recovery time and replica lag as in Table 3. "+Prefetching" contributes mostly to the CPU-bounded read-only workloads. "Fine-grained Locking" contributes a lot to these workloads (about 25-35%), especially for high contention workloads. Comparing the two methods, the index optimization is about one time more effective than the shadow page. The

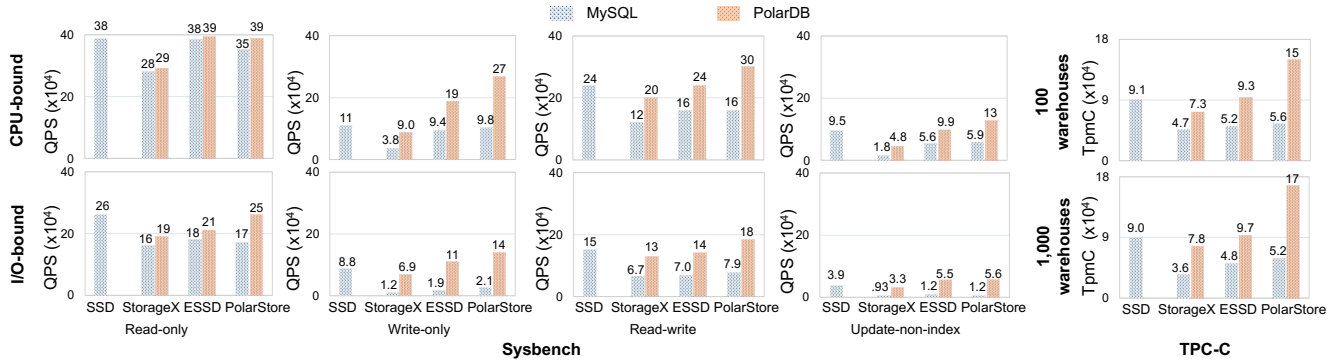

Figure 13: Evaluation for PolarDB when attaching to diverse-type storages.

"+I/O Queue, Scheduler and Alignment" optimization contributes mostly to the performance stability, rather than a high percentage of performance improvement. But it still contributes about 8% on average.

We further analyzed the breakdown of the most effective optimizations on WAL persisting acceleration, including the global log buffer partitioning, paralleled log writer, and also the aligned log I/O, as shown in Figure 12. We put the first two optimizations as a whole in the analysis since both of them partition the log space and parallelize the logging process. "Partitioning Global Log Buffer and Parallel Log Writer" contribute about 64% to the owned improvement of log write while "Aligned Log I/O" contributes the rest 36%.

4.7.4 Various Cloud Storage Types. In this part, we evaluate MySQL and PolarDB on three different types of cloud storage: Storage X, ESSD and PolareStore. As a comparison, we also include the performance achieved on local SSDs.

Storage X. Storage X adopts a traditional TCP/IP network for the netwrok traffics, which has higher I/O latency than RDMA based networks in PolareStore. As shown in Figure 13, PolarDB on Storage X shows the worst performance among all compared storages, although it is still faster than MySQL, especially for the I/O-bound and write-intensive scenarios. In the TPC-C test, PolarDB achieves the highest tpmC (with 55% and 116% improvement) on both settings.

Alibaba ESSD. ESSD is another cloud storage of Alibaba with RDMA network, which can be accessed on [2]. The comparison results for PolarDB and MySQL on the ESSD are also presented in Figure 13. PolarDB, without special customizations, still outperforms MySQL significantly. For the write-only scenario in the I/O-bound SysBench workload, the speedup is as high as 5.8×.

5 CASE STUDY: ROCKSDB

In this section, we report how we apply CloudJump on RocksDB with evaluations. Figure 14 shows the optimization overview including optimizations on WALs, data accesses and I/Os. Among these optimizations, we implement the parallel log writer, the I/O scheduler, and tunes I/Os to the cloud storage, with the rest already implemented by the RocksDB community.

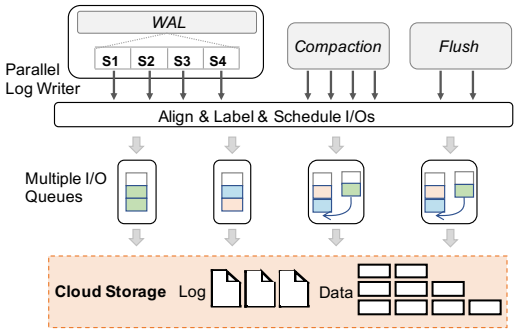

Figure 14: Optimization overview of RocksDB.

5.1 Accelerating Persisting WAL

RocksDB also uses WAL to guarantee process crash consistency, in which a single WAL, written to a single log file in the storage, captures write logs for all column families (separate key spaces in RocksDB). We apply the parallel log writer as in InnoDB to accelerate the processing of WALs.

In RocksDB, an auto-incremented sequence number is assigned to each write of key-value pairs. As there is no update-in-place pages in the LSM-tree, we choose to directly partition the log write-batch in the log writer by sequence numbers, which is similar to the proposal in section 4.1.2. The preallocated WAL file is divided into N segments (*e.g.*, 4 partitions for a 64MB file), and a log writebatch is scattered and delivered into these partitions by N parallel task, as shown in Figure 6.

5.2 Accelerating Data Access

Long remote data access and read amplifications in the LSM-tree reduces the read performance for RocksDB on the cloud storage. Prefetching, filtering and compression mechanisms are the state-of-the-art techniques in RocksDB to accelerate data access. There are native implementations of data prefetching, filtering, and compression in RocksDB [13].

Prefetching can contribute to the accelerations of both lookups and compactions. We recommend deterministic prefetching for compaction. And the read-ahead size is suggested to be the storage granularity size (*e.g.*, 4 MB in our case). If the parallel tasks are not heavy, we consider using block read-ahead. Although bloom filters

Table 4: Data compression and access latency.

	Uncompressed	LZ4	ZSTD
Data size[1]	512 KB	336 KB	256 KB
Compression time	0	546 us	727 us
Flush time	1893 us	1259 us	954 us
Total (write)	1893 us	1805 us	1681 us
Deompression time	0	108 us	126 us
Read Time	1125 us	786 us	561 us
Total (read)	1125 us	894us	687 us

[1] Data request blocks are 16 KB aligned

take significant main memory spaces, applying it to save remote I/Os increases its potential benefits on the performance. We leave the detailed analysis of such trade-offs as future work following this case study.

Compressions save both the storage cost and the network I/O. Table 4 shows a case concerning the relationship between compression and data access latency, tested on our platform (section 5.5.1) with single compression/decompression/flush/read thread. Despite that compressions add overheads in the write path, the total write time is reduced by 33% and 50% using LZ4 and ZSTD, respectively. And, the total read time is reduced by 21% and 49%, respectively. In a conventional TCP network, savings of the network overhead may contribute more to latency reductions.

5.3 Multiple I/O Queues and Scheduler

We introduce a similar multi-queue I/O model with that in Section 4.5, to split I/O tasks and work tasks (*e.g.*, compaction jobs and flush jobs) in RocksDB. Since we split I/O tasks from background flush jobs, there is no need to further increase the number of flush threads, which only align I/O requests and dispatch them. We tune the size of I/O request and data organization (*e.g.*, block and SST) according to the cloud storage, and more precise control is done to make the block size of a filter of an SST file also aligned properly. Taking our case as an example, the storage has a minimum request size of 4 KB (representing the smallest processing unit), a desirable request size of multiple of 16 KB, and a storage granularity size of 4 MB. The SST size and block size are strictly assigned to multiple the storage granularity and the desirable request size respectively. Generally, compared to RocksDB, our proposal can enable data to align with the compression enabled. We do not deliver an I/O request smaller than the minimum request size to the multi-queue I/O model. Instead, we align the small I/O and cache the misaligned suffix in memory for later use. Besides, we do not issue an I/O request larger than the storage granularity. Instead, we issue parallel tasks via the multi-queue I/O model. These not only scatter data on different storage nodes as much as possible but also maximize parallelism to make the most of the bandwidth.

Based on the I/O model, we also propose I/O scheduling consistent with that in Section 4.5.2. RocksDB also provides a scheduling scheme in the foreground, based on thread roles, while our scheduling method is based on the I/O labels.

5.4 I/O Alignment

We align I/O requests before enqueueing them, and this is similar to Section 4.6. In RocksDB, the WAL path has an I/O mode similar to PolarDB. Besides, there can be large and unaligned I/O requests due to LSM-tree structure and data compression. Following the premise in Section 5.3, we give the RocksDB example here. When a job wants to flush 2 KB log data starting at 1 KB, it fills 0-1 KB from the previously cached data and appends 3-4 KB with zeros, and then sends a 4 KB I/O from 0 KB. On our cloud storage (with a minimum request size of 4 KB), this has the same write amplification factor as sending a 2 KB I/O from 1 KB, however, the aligned I/O causes no read-on-write problem. When a job wants to flush 8 MB data, it delivers two I/O requests of 4 MB into two I/O queues, rather than an 8 MB request to one queue. The I/Os workers execute tasks in parallel, maintain the request context, and signal the requester when the two tasks are complete.

5.5 Experiments and Evaluation

5.5.1 Experiment Setup. We build a RocksDB instance on top of the ESSD with parameters shown in Table 1, and the computation node runs on an Alibaba Cloud ECS instance with type of ecs.hfg6.4xlarge (16C64G). We adopt the RocksDB v6.27.0 as the codebase, and optimize it with the proposals in section 5.1, 5.3 and 5.4. We use the *Performance Benchmarks* (based on *db_bench*) [15] as the benchmark toolkit to test performance of the database.

5.5.2 Evaluating Performance. To focus on the impact of remote I/Os, CACHE_SIZE is set to only 16 GB, and direct I/O is enabled for all tests. A maximum write buffer (writable_file_ max_buffer_size) is set to 4 MB, and db_bench thread number is 256 except bulkload test. Default values of other parameters in *Performance Benchmarks* are used except for specified instructions. We build databases with different key-value sizes in the evaluation: 1) database 1 has *1e9* entries with the value size of 400 B; 2) database 2 has *1e7* entries with the value size of 40 KB. Both databases have the key size of 20 B, and the compressed database size is about 200 GB with ZSTD. 4 workloads defined in *Benchmarks* are used 1) *"Bulkload"*: bulk load of random key entries without WAL; 2) *"Readrandom"*: random read test; 3) *"Overwrite"*: random overwrite test; 4) *"Readwhilewriting"*: multi-threaded read and single-threaded write test.

We introduce two optimizations in the prototype: scheduled and aligned multi-queue I/O mode (section 5.3 & 5.4), and parallel log writer (section 5.1). The throughput is adopted to make a head-to-head comparison, and Figure 15 depicts the results of the 4 workloads. *"Bulkload"* disables WAL to load faster and requires client to discover where to restart a load after a crash. The I/O mode optimization improves about 43% and 22% in two databases. The large value size gives a high bandwidth in the baseline (about 900 MB/s), and makes the optimized prototype reach the maximum storage bandwidth, resulting in a smaller percentage of performance improvement for database 2. In *"Readrandom"* test, the I/O mode optimization improves about 32% and 61%, since the scattered I/O can further accelerate large remote access. The parallel log writer has no influence in the two cases since the related path is bypassed. We enable WAL in *"Overwrite"* and *"Readwhilewriting"* tests. The optimized prototype improves two workloads about 42% and 44% in databases 1, and 105% and 74% in databases 2, respectively. For

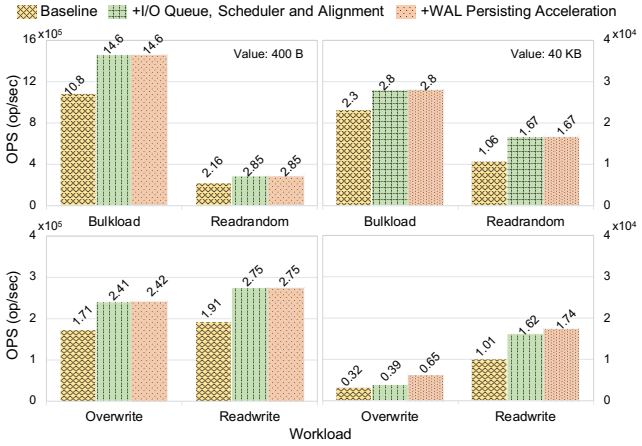

Figure 15: RocksDB Performance.

database 1, almost all improvements come from the I/O mode optimization. It is after that we persist WAL for each operation and small log I/O can not be partitioned. However, from test results of database 2 with large entries, the parallel log writer shows it gains, especially in the *"Overwrite"* test. The I/O mode optimization contributes about 32% and 71% to the total improvement, respectively for the two workloads, while the WAL optimization contributes the rest (68% and 39%).

6 LESSONS LEARNED ON USER WORKLOADS

There are three major types of user data and workloads that have been pushing us to optimize PolarDB and finally come up with CloudJump: (1) very large tables, a single table can be as large as almost 20 TB in the storage; (2) large attributes, a single attribute storing a flattened semi-structured or JSON object may be as large as 100 KB; and (3) tens of thousands of tables that are concurrently accessed by various departments in the same company.

Because cloud storage offers a unified view of a very large storage capacity (e.g., 100TB), users are free from the troubles of sharding their large tables into multiple database instances. We have observed single tables to be as large as 10 to 20 TBs. With pages placed across multiple nodes in the storage and I/Os scattered by our optimizations, queries on such large tables are now able to benefit from concurrent accesses and the high aggregated bandwidth of the cloud storage.

Large attributes are common in many industries, e.g., online gaming, where the application developers store their semi-structured data as a whole in a single attribute for easy adaptations to changes in their application logic. In PolarDB, we store the primary key separately from such large attributes, preventing queries dependent mostly on the primary key alone from being slowed down by these large I/Os. And, prefetching helps to accelerate the fetch of such large attributes when queried.

Many SaaS companies share a single PolarDB cluster among multiple departments and scale the number of RO nodes out. We have observed that there can be as many as tens of thousands of tables with non-trivial sizes, on which these departments run independent workloads in parallel. In PolarDB, these workloads are balanced among multiple database and storage nodes, resulting in an isolated performance experience for each department user.

7 RELATED WORK

Databases on decoupled computation and storage. By moving certain functions like the log applicator to the storage layer, Aurora [34, 35] reduces data traffic by moving functions to the storage layer. Socrates [5] follows the same principle and further extracts the log layer from the storage, which separates availability and durability, allowing optimal utilization of different machines. Taurus [12] also separates the compute and storage layers in a similar manner but uses asymmetric replication based on separate persistence mechanisms for database logs and pages. This design is indeed elegant, however, it takes obligatory engineering effort to customize the storage layer to the specific engine, and there is less research on how to leverage general storage service.

RDMA-based distributed file systems. PolarFS [8] uses RDMA to build a reliable block device for the benefits. Octopus [25] is an RDMA-enabled distributed persistent memory file system, closely coupling NVM and RDMA features. Orion [38] is also a distributed file system on RDMA, improving high-performance metadata and data access by a clean slate design. The Ceph community also develops Ceph over Accelio to support RDMA [1, 11]. Generally, high-performance distributed file systems are deployed on block devices exposed, using RDMA to accelerate the data path with dedicated optimizations [17, 26, 32, 39].

Scalable logging and recovery. The WAL protocol codified in ARIES [27, 28] makes databases recover safely from failure. Many database products usually use traditional centralized logs [7, 29]. Recently, many scalable logging mechanisms have been proposed based on byte-addressable NVM technologies with low access latency [19, 20, 36]. Some scholars innovated logging algorithms to reduce consumption, making them better adapt to NVM-based systems [6, 10, 16, 33] or in-memory systems [24, 30, 37].

8 CONCLUSION

In this work, we analysed the performance characteristics of the cloud storage, summarized their impacts on the design of databases and derived a framework CloudJump to guide the optimizations of storage engines on the cloud storage. CloudJump addresses a wide range of design issues in the processing of both database logs and pages. We have used InnoDB as the demonstration vehicle of CloudJump and extended such efforts to RocksDB. Evaluations with the TPC-C benchmark show that CloudJump improves the performance by around 5-8 times, compared with directly porting an on-premise database on a cloud storage. We believe CloudJump sheds lights on how developers should optimize their storage engiens towards the cloud storage for many types of databases that use either update-in-place (B-tree) or append-only (LSM-tree) methods for their internal data structures.

ACKNOWLEDGMENTS

We would like to thank our database solutions architects Guangxing Hu, Bingpeng Wang, Liangzhong Guo, Linping Wang and Wei Wang at Alibaba Group for their contributions on connecting the development team of CloudJump with customers.

REFERENCES

[1] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R Ganger, and George Amvrosiadis. 2019. File systems unfit as distributed storage backends: lessons from 10 years of Ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 353–369.

[2] Alibaba. 2020. Alibaba Cloud Enhanced SSDs. https://www.alibabacloud.com/help/doc-detail/122389.html.

[3] Amazon. 2020. Amazon Elastic Block Store. https://aws.amazon.com/ebs/features/.

[4] Amazon. 2020. MySQL on Amazon RDS. https://docs.aws.amazon.com/AmazonRDS/latest/User-Guide/CHAP_MySQL.html.

[5] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, et al. 2019. Socrates: the new SQL server in the cloud. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*. 1743–1756.

[6] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-behind logging. *Proceedings of the VLDB Endowment* 10, 4 (2016), 337–348.

[7] Hillel Avni, Alisher Aliev, Oren Amor, Aharon Avitzur, Ilan Bronshtein, Eli Ginot, Shay Goikhman, Eliezer Levy, Idan Levy, Fuyang Lu, et al. 2020. Industrial-strength OLTP using main memory and many cores. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3099–3111.

[8] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1849–1862.

[9] ClickHouse. 2021. ClickHouse. https://clickhouse.com/.

[10] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. 2013. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. 197–212.

[11] Cohortfs. 2021. Ceph over Accelio. https://www.cohortfs.com/ceph-over-accelio.

[12] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, et al. 2020. Taurus Database: How to be Fast, Available, and Frugal in the Cloud. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1463–1478.

[13] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB.. In *CIDR*, Vol. 3. 3.

[14] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 33–49.

[15] Facebook. 2021. RocksDB. https://github.com/facebook/rocksdb.

[16] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 877–892.

[17] Nusrat S Islam, Mohammad Wahidur Rahman, Jithin Jose, Raghunath Rajachandrasekar, Hao Wang, Hari Subramoni, Chet Murthy, and Dhabaleswar K Panda. 2012. High performance RDMA-based design of HDFS over InfiniBand. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[18] Varun Jain, James Lennon, and Harshita Gupta. 2019. Lsm-trees and b-trees: The best of both worlds. In *Proceedings of the 2019 International Conference on Management of Data*. 1829–1831.

[19] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2010. Aether: a scalable approach to logging. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 681–692.

[20] Hyungsoo Jung, Hyuck Han, and Sooyong Kang. 2017. Scalable database logging for multicores. *Proceedings of the VLDB Endowment* 11, 2 (2017), 135–148.

[21] Aarati Kakaraparthy, Jignesh M Patel, Kwanghyun Park, and Brian P Kroth. 2019. Optimizing databases by learning hidden parameters of solid state drives. *Proceedings of the VLDB Endowment* 13, 4 (2019), 519–532.

[22] Alexey Kopytov. 2021. Sysbench. https://github.com/akopytov/sysbench.

[23] Philip L Lehman and S Bing Yao. 1981. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems (TODS)* 6, 4 (1981), 650–670.

[24] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. 2017. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 21–35.

[25] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, Dilma Da Silva and Bryan Ford (Eds.). USENIX Association, 773–785. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu

[26] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting Network Support for RDMA. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 313–326.

[27] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)* 17, 1 (1992), 94–162.

[28] C Mohan and Frank Levine. 1992. ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. *ACM Sigmod Record* 21, 2 (1992), 371–380.

[29] Oracle. 2021. MySQL 8.0 Reference Manual. https://dev.mysql.com/doc/refman/8.0/en/.

[30] Kun Ren, Thaddeus Diamond, Daniel J Abadi, and Alexander Thomson. 2016. Low-overhead asynchronous checkpointing in main-memory database systems. In *Proceedings of the 2016 International Conference on Management of Data*. 1539–1551.

[31] Radu Stoica and Anastasia Ailamaki. 2013. Enabling efficient OS paging for main-memory OLTP databases. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*. 1–7.

[32] Shin-Yeh Tsai and Yiying Zhang. 2017. LITE Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 306–324.

[33] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1541–1555.

[34] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*. 1041–1052.

[35] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvilli, et al. 2018. Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*. 789–796.

[36] Tianzheng Wang and Ryan Johnson. 2014. Scalable logging through emerging non-volatile memory. *Proceedings of the VLDB Endowment* 7, 10 (2014), 865–876.

[37] Yingjun Wu, Wentian Guo, Chee-Yong Chan, and Kian-Lee Tan. 2017. Fast failure recovery for main-memory dbmss on multicores. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*. 267–281.

[38] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2019. Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 221–234. https://www.usenix.org/conference/fast19/presentation/yang

[39] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2020. FileMR: Rethinking RDMA Networking for Scalable Persistent Memory. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, Ranjita Bhagwan and George Porter (Eds.). USENIX Association, 111–125.

[40] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. 2020. Leaper: a learned prefetcher for cache invalidation in LSM-tree based storage engines. *Proceedings of the VLDB Endowment* 13, 12 (2020), 1976–1989.