LATTE: A Native Table Engine on NVMe Storage

Jiajia Chu, Yunshan Tu, Yao Zhang, and Chuliang Weng*
East China Normal University

***Abstract*—Most database systems rely on complex multi-layer and compatibility-oriented storage stacks, which results in sub-optimal database management system (DBMS) performance and significant write amplification. A heavy storage stack can be tolerated in the slow disk era because its storage overhead is completely overlapped by hardware delay. However, with advances in storage technologies, emerging NVMe devices have reached the same level of latency as software, which in turn has caused the storage stack to become a new bottleneck. On the other hand, NVMe devices not only improve I/O efficiency but also introduce distinctive hardware features that require software modifications to take advantage of them.**

To fully exploit the hardware potential of NVMe devices, we propose a lightweight native storage stack called Lightstack to minimize the software overhead. The core of Lightstack is an efficient table storage engine, LATTE, which abstracts the essential data service of the database's 2D table. LATTE is designed from the ground up to use NVMe devices efficiently. It directly accesses NVMe devices to reduce single I/O latency and utilizes a parallel scheduling strategy to leverage multiple deep I/O queues and CPU cores. Besides, undo logging on heterogeneous storage is proposed to mitigate the write amplification further. We also implement a working prototype and evaluate it with standard benchmarks on the Intel Optane DC P4800X NVMe SSD and the DC P3608 Series NVMe SSD. Experimental results show that LATTE has up to 3.6-6.5× the throughput of MySQL's InnoDB and MyRocks engines, with latency as low as 28% in the same hardware environment.

I. INTRODUCTION

With the development of storage technologies and non-volatile memory express (NVMe) standard [1], many advanced NVMe products have emerged (e.g., NVMe SSD and Intel Optane Memory [2]) to replace traditional storage devices. Compared with hard disk drives (HDDs) and solid state drives (SSDs), NVMe devices have an order of magnitude decrement in latency and a 5-10× increase in bandwidth. Such drastically reduced I/O overhead makes it possible for database systems to break through existing I/O bottlenecks.

However, legacy storage systems are destined to utilize emerging storage hardware insufficiently due to the compatibility-oriented layered software [3], [4]. In general, traditional databases run on file systems, and their storage stacks include the database layer, file system layer, operating system layer, and device driver layer from top to bottom shown as Fig. 1(a). Read and write requests need to be transferred between these layers and eventually reach the storage device. To simplify the deployment of applications, the software layers are generally stacked on demand, so similar and redundant functions between layers (e.g., buffering, caching, copying, and queuing) are inevitably placed together in the storage stack, causing duplicative overhead. The unnecessary, redundant overhead can be arbitrarily ignored in the rotational storage era compared to millisecond-level hardware latency. When NVMe devices reduce I/O hardware latency to the same level as software, the legacy software which takes up about 70% of the total delay (see Section II-B) will dominate the overall latency. At this time, if those unnecessary overheads are still ignored, the system can not achieve extreme runtime performance. Furthermore, NVMe devices support multiple deep I/O queues, which brings unlimited possibilities for improving the parallelism of native storage. Unlike a single hardware queue in an SAS/SATA device, NVMe devices support up to 64K hardware queues. Each queue can be scheduled in parallel by the upper layer, and the maximum queue depth can reach 64K. Nevertheless, traditional databases which born in the era of SAS/SATA HDDs were not aware of this, thereby definitely cannot combine these features to achieve higher I/O parallelism.

To fully leverage the capabilities of NVMe devices, there are three main challenges. Firstly, the storage stack on the NVMe devices is supposed to be minimized. It requires removing unnecessary software and redundant overhead to achieve short-path I/O without affecting the minimum set of application requirements. In this regard, some research efforts have focused on cross-layer integration and optimization. For example, NoFTL [5] integrates the FTL into the database. Aerie [6] supports direct access to storage hardware without kernel interaction. DevFS [7] embeds the file system in the storage device, which ultimately simplifies the storage stack from the file system layer to the device. All of them are instructive, but they give optimizations on a single software layer or a partial layers of the traditional storage stack. Secondly, the storage stack is expected to achieve higher parallelism by combining multiple CPU cores and multiple deep I/O queues. Consequently, some works co-designing storage systems with new features of emerging devices have been proposed [8]–[10], but a lightweight storage system designed for NVMe devices is currently lacking. Last but not least, the software on the NVMe SSD needs to mitigate the write amplification to reduce wear [11], [12]. Since NVMe devices cost more than hard disks and regular SSDs, it motivates us to reduce storage footprint and device wear without sacrificing data consistency and performance.

In this work, we propose a lightweight native storage stack called Lightstack for NVMe devices to minimize the storage overhead. The core of Lightstack is an efficient table storage engine, LATTE, which abstracts the storage and access management of the database's 2D table. To the best of our

*Chuliang Weng is the corresponding author, clweng@dase.ecnu.edu.cn knowledge, it is the first native table storage engine designed for NVMe devices. The contributions made in this paper are summarized as follows:

- **Lightweight Native NVMe-Based Storage Stack.** We propose the Lightstack framework, a lightweight storage stack for NVMe devices. It integrates and simplifies the layers of the database, file system, and operating system to shorten the I/O path to achieve ultra-low latency.
- **Efficient Table Storage Engine.** Following Lightstack, we build a table storage engine, LATTE, which first provides efficient data service in user space without data conversion between tables and files. The experimental result shows that LATTE performs much better than MyRocks [13] and InnoDB engines.
- **Parallel Scheduling for I/O Queues.** In LATTE, we combine multiple deep I/O queues and CPU cores to promote I/O parallelism. Specifically, it includes four binding modes. We analyze the pros and cons of each mode and compare their runtime performance in detail.
- **Undo Logging on Heterogeneous Storage.** In LATTE, we also accelerate the undo logging with NVDIMM to mitigate the issue of write amplification without sacrificing performance and data consistency.

II. Background and Motivation

A. NVMe Storage and NVDIMM

NVMe is an extensible host controller interface protocol that provides efficient access to storage devices [1]. It is designed to optimize the data transfer between the host system and peripherals from the perspective of the transport protocol, enabling storage devices to have higher input/output operations per second (IOPS) and lower I/O latency. Before the popularity of the NVMe protocol, traditional SAS/SATA SSDs tended to use the Advanced Host Controller Interface (AHCI) protocol inherited from HDD. The AHCI protocol was born in an era of hard drives. It only supports one I/O queue, and the maximum queue depth is 32, which is sufficient for rotational storage but limits the performance improvement of the Peripheral Component Interconnect Express (PCIe) bus. Moreover, a single instruction, according to AHCI, requires multiple accesses to the register, increasing the access latency. These make the AHCI protocol challenging to get the most out of the modern SSDs [14].

Different from a single I/O queue in the AHCI protocol, the NVMe protocol supports multiple deep queues to facilitate parallelism in the data processing. The number of queues can be up to 64K, and the maximum depth can reach 64K. Queues in the NVMe protocol are divided into two types, admin queues, and I/O queues. Each NVMe controller corresponds to one pair of admin queues and multiple pairs of I/O queues. The admin queue pair is used to process admin commands, including initialization of I/O queues and command control. The I/O queues are used to receive and handle I/O commands from the host system. Each pair of queues includes a submission queue (SQ) and a completion queue (CQ). An SQ or CQ is

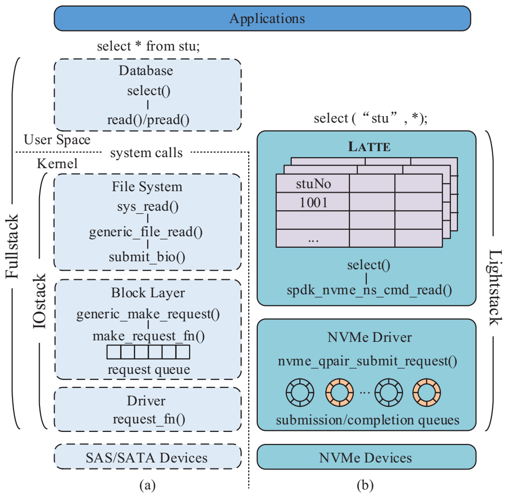


Fig. 1: **Typical Fullstack vs. Our Lightstack.**

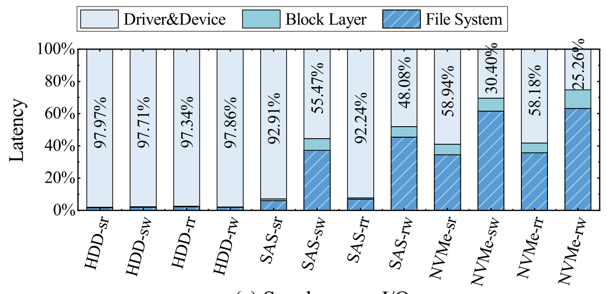
a ring buffer which is a memory area in host systems. An I/O command is first inserted into an SQ. When the NVMe controller completes this I/O, the completion message is added to the associated CQ. Multiple deep queues can execute I/O commands in parallel, considerably increasing the bandwidth of NVMe SSDs.

Although the performance of NVMe devices has been dramatically improved, its latency is still an order of magnitude higher than DRAM latency. The emergence of the non-volatile dual in-line memory module (NVDIMM) bridges the performance gap between NVMe SSD and DRAM. The NVDIMM is a type of non-volatile memory (NVM) [15] that is packaged using standard DIMM and communicates over a standard double data rate (DDR) bus. It has a broad application prospect in expanding memory capacity and being used as a cache for SSDs. Besides, the JEDEC standardization organization defines three NVDIMM implementations [16], including NVDIMM-N, NVDIMM-F, and NVDIMM-P. In this paper, in addition to leveraging hardware features of NVMe devices, we also use heterogeneous storage of NVMe SSD and byte-addressable NVDIMM (NVDIMM-N or NVDIMM-P) to achieve a better trade-off between performance, space, and consistency in emerging hardware environments.

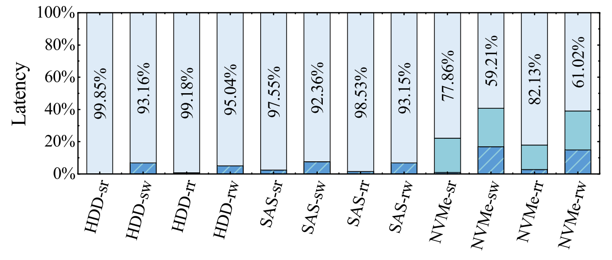
B. Overview of Storage Stack

For traditional database systems, user requests (e.g., select) go through the database layer, file system layer, operating system layer (mainly block layer) and device driver layer from top to bottom, and finally, arrive at the hardware device, as shown in Fig. 1(a). The series of software programs stacked upon the hardware make up the software stack of the DBMS, which work together to achieve functionalities. We define the software stack from the database layer to the device driver layer as Fullstack, where the file system layer to the device driver layer is IOstack.

To analyze the data processing time of the Fullstack, we use the Fio [17] to generate 4 KB I/O operations and the Blktrace tool [18] to trace the processing paths. Given the variety of



(a) Synchronous I/O.



(b) Asynchronous I/O.

Fig. 2: **Break-Down of IOstack Overhead.**

user programs at the database layer, and the Blktrace tool performing I/O tracing in kernel mode, we then record the overhead ratio for each layer in the IOstack. To more comprehensively evaluate I/O latency, we configured the Fio with both synchronous and asynchronous modes for experiments. With Fio and Blktrace, we obtain the processing time after the request is submitted to the kernel (`clat`), from the request entered the block layer until the I/O is completed (`Q2C`), and from the request is submitted to the driver until the I/O complication (`D2C`). Then, we get the time spent on the driver and device (`D2C`), the processing time in the block layer (`Q2C-D2C`) and the time of the file system (`clat-Q2C`). As shown in Fig. 2, whether we use synchronous I/O or asynchronous I/O, the software on the NVMe SSD accounts for a larger proportion. The overhead of writing even reaches 70% through synchronous I/O. The larger proportion of software overhead cannot be overlooked and motivates us to rethink the layered storage stack. Though the traditional storage stack is powerful and convenient, it has following potential issues.

1. Switch between user mode and kernel mode. When the read()/write() is called, it must be toggled between user mode and kernel mode twice. The system first enters the kernel from user space and moves from the user stack to the kernel stack. After executing the kernel code, the process returns to user mode. During kernel mode, the system may experience process switching due to resource preemption.

2. File locks limit parallelism. Based on the file system, the database usually stores the table data and the index data in the files, and all operations on the table data are finally converted into file operations. Many DBMSs (e.g., MySQL and SQLite) tend to use file locks to remove the overhead of acquiring and releasing each fine-grained lock. These file locks restrict individual files from being modified by multiple processes simultaneously, which means that changes to different data within the same file are mutually exclusive. Lock conflicts can be mitigated by merge and batch operations of file writes, but the access latency will inevitably increase.

TABLE I: **Space Utilization in InnoDB.**

	50 MB	100 MB	500 MB
ibdata1 (MB)	64	128	576
ib_logfiles (MB)	65	125	616
total (MB)	129	253	1292

3. Multi-level caching and data copying. The cache in the traditional Linux storage stack mainly includes the user-space buffer, the page cache in the operating system, and the buffer cache between the file system and the storage device. Accordingly, the data in the storage stack is cached redundantly. To reduce the data copies, the 2.4 Linux kernel merges the page cache and the buffer cache. The mmap() maps the page cache to user space. The technology of direct I/O with the O_DIRECT parameter is applied to bypass the page cache. These methods mitigate the redundancy to some extent.

4. Data transmission and queuing. After leaving the file system layer, the request is attached to the processing queue in the block layer. Before the 3.17 Linux kernel, there was only one processing queue, and thereby all parallel requests were converted to sequential processing in that layer. Considering that modern SSDs can support multiple hardware dispatch queues, the 3.17 Linux kernel redesigns a multi-queue block layer [19] to match it, but it still cannot avoid the data movement from the processing queue to the hardware dispatch queue, as well as queuing time in the block layer.

5. Interrupt/polling. The Linux kernel typically uses interrupts to interact with hardware devices [20]. With the interrupt mode, a signal is issued each time the hardware completes the operation. Then, the interrupt controller will capture the signal and submit the interrupt to the CPU for processing. The polling mode always polls the processing results. Compared with the interrupt mode, although the polling mode consumes more CPU resources, it avoids context switching, so that the processing result can be perceived as soon as possible.

The impact of these issues will be magnified and even become a new performance limitation for emerging fast storage devices. Therefore, we propose the Lightstack framework, a lightweight storage stack for NVMe devices to mitigate these issues and fully exploit the hardware performance.

C. Write-Ahead Logging in Databases

The write-ahead logging (WAL) mechanism, which ensures atomicity and durability, is widely adopted in disk-oriented databases. It requires all logs of the update operations to be safely saved before committing the relevant transactions. Since the data modification incurs random I/O overhead, and the log file can be updated by batch appending, the system supporting WAL can convert random updates into sequential logging. Although logging sequentially avoids the overhead of random access, it wastes more storage space and exacerbates the issue of write amplification due to additional logging.

To analyze the space occupied by the undo and redo logs, we perform experiments on the InnoDB engine of MySQL, where the WAL mechanism is enabled. A single data file and multiple redo log files are used in experiments. The file `ibdata1` records all of the actual data and undo logs. Files named `ib_logfile*` store redo logs. We insert different amounts of data and monitor sizes of `ibdata1` and `ib_logfile*`, respectively. Results of the experiment are shown in Table I. We find the space consumption of an insert operation is on average $2\times$ as much as that of actual data. In other words, the write amplification ratio at this time is about 2.

Although the storage space occupied by redo logs can be released by the checkpoint mechanism, the write amplification caused by redo logs still exists, resulting in device wear. Indeed, WAL is a workable choice for write-intensive applications which run on HDDs, but it is not efficient for fast storage devices because the gap between random and sequential access latency has gradually become less pronounced [21], [22]. Therefore, we rethink the WAL mechanism and discuss the undo logging on heterogeneous storage to balance the performance, storage overhead, and data consistency.

III. Design

A. *Lightweight Storage Stack*

Emerging storage devices, with lower and lower hardware latency, correspondingly make the upper software overhead account for a larger proportion and become a new performance bottleneck. To better adapt to NVMe devices, we propose a new kind of native storage stack called Lightstack. It avoids blindly stacking software without compromising application requirements, and has the following considerations.

- **Efficiency.** The Lightstack locates in user space to avoid the overhead of mode switching. It provides a shorter I/O path for data processing and preferably adopts the polling mode to achieve timely responding.
- **Non-Redundancy.** It arranges each layer elaborately and integrates the repetitive functions between layers to eliminate functional redundancy and unnecessary overhead.
- **Parallelism.** As the emerging hardware also supports new features to achieve higher parallelism, the Lightstack needs to be improved to fit it accordingly.
- **Compatibility.** Considering that the 2D table is a fundamental data structure in the databases, Lightstack provides a standard set of table APIs to be fully compatible with operations in the databases.

We rethink and integrate the traditional Fullstack, and build the storage stack of `LATTE+Driver` following the above features, as shown in Fig. 1(b). As analyzed earlier, the software has become a new performance limiting factor, which is optimized by LATTE in our Lightstack instance. Therefore, the rest of this paper will focus on LATTE.

B. LATTE *Architecture*

LATTE is a heterogeneous and efficient table storage engine designed for NVMe devices. It has three design goals, lower latency, higher throughput, and minimal storage footprint.

Lower Latency—Short-Path Direct I/O. Following the Lightstack, LATTE implements a shorter path for data storage

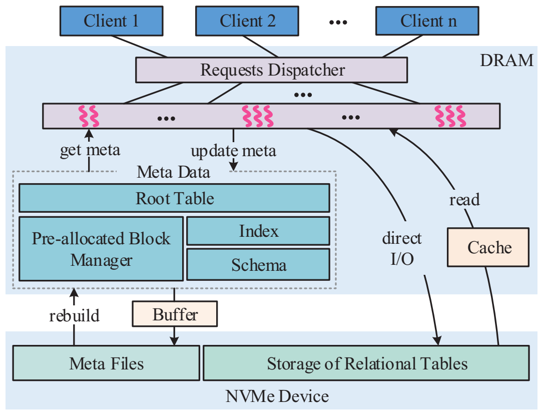


Fig. 3: **The Architecture of LATTE.**

in user space. It bypasses the traditional Fullstack and accesses the NVMe device directly without multi-level caching, and software queuing in the block layer. It also polls the completion queues to receive completed requests immediately.

Higher IOPS—Parallel Queues Scheduling. LATTE proposes a parallel queues scheduling strategy to leverage the hardware parallelism fully. It allows different I/O queues to manipulate different tables in parallel and minimizes the write conflicts by pre-allocating physical storage space for each table. It allocates the queue with the lightest load for a worker thread to balance tasks across various I/O queues. It also binds I/O queues to CPU cores and supports four different CPU affinity modes to improve performance further.

Reduced Storage Footprint—Undo Logging. LATTE adopts undo logging method with NVDIMM to offer a better choice to balance the performance, storage overhead, and data consistency. It stores table data in NVMe SSDs while preserving the undo logs in NVDIMM to take full advantage of the respective strengths of hybrid storage hardware.

Fig. 3 illustrates the overall architecture of the LATTE. Client requests are received and scheduled by the requests dispatcher. Then, they will be assigned to idle worker threads. LATTE runs a worker thread for each CPU core and uses locking methods to resolve conflicts in concurrent operations. For an insert request, a worker thread first accesses and gets the corresponding metadata. Second, it finds a block available to store tuples. It then flushes the tuple directly to the block in the NVMe SSD while preserving the undo log without the redo log. When the insert operation is completed, the worker thread will update the metadata and respond to the client. LATTE also utilizes a single-level block cache to promote query efficiency. The update operation in LATTE needs to read the corresponding data block into memory first, and then write the updated data block back to the NVMe device. For a long transaction with many updates, LATTE first merges the update results of the same primary key, and then, calls `update()` in parallel to refresh the data corresponding to the different primary keys in the NVMe SSD. To ensure the atomicity and consistency of the data, LATTE records the undo log before overwriting the data block. To take advantage of the heterogeneous storage hardware, LATTE temporarily stores

TABLE II: **Major APIs Defined in LATTE.**

Interface	Description
tableID ← `createTable` (tableName, segmentType, schema)	Create a table.
status ← `drop` (tableName)	Remove a table persistently.
status ← `insert` (tableName, primaryKey, tuple)	Insert a row data into related table.
status ← `erase` (tableName, primaryKey)	Delete a row from related table.
status ← `update` (tableName, primaryKey, columnName, columnValue, size)	Update a column value of a row.
tuple ← `select` (tableName, primaryKey)	Select a tuple pointed by a primary key from a table.
cell ← `select` (tableName, primaryKey, columnName)	Read a cell from a specific row.
tuples ← `scan` (tableName, columnName, keyRange)	Scan rows whose cells are in a specific range.

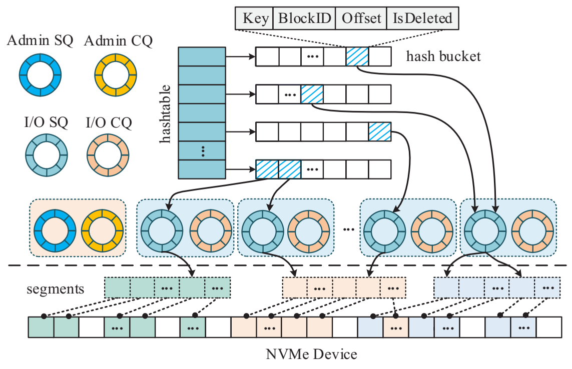


Fig. 4: **Parallel Scheduling of I/O Queues in LATTE.**

the undo logs in the NVDIMM. After the related transaction is committed, the undo logs are removed in batches. The erase operation has two phases, marking and deleting. LATTE always marks a specific tuple as deleted along with writing a tombstone record into the metafiles before responding to the client, and erases the data actually by the garbage collection periodically. For scan operations, LATTE returns the result set in units of 100 tuples at a time, and the client program calls next() until the returned result is NULL.

All the APIs provided by LATTE are summarized in Table II. These interfaces supporting table-level basic operations are similar to those in traditional DBMSs. The above designs deliver ultra-low latency and higher overall performance, making LATTE more suitable for performance-sensitive table-based applications such as GPS track recording, real-time traffic updates, as well as balance and billing queries.

IV. Parallel Queues Scheduling

As described in Section II-A, the significant difference between NVMe devices over slow disks and SAS/SATA SSDs is the introduction of multiple deep I/O queues. This section will detail how LATTE schedules these queues in parallel.

A. Regionalized Table Storage

LATTE divides the NVMe device into massive data blocks, which are the smallest read and write units. To facilitate the management of data in tables, we build a logical layer on top of the physical data blocks. That is, we define a certain number of physical data blocks as a segment, as shown in Fig. 4. Each segment belongs to only one table. When creating a table, LATTE assigns a segment to it, which is the storage space pre-allocated for the table. The data blocks in the segment are maintained and managed by the segment manager. For each data block in the segment, the segment manager updates its information of <BlockID, Offset, IsOccupied> in real time. The BlockID is the number corresponding to the physical data block, the Offset records the current offset of the valid data in the block, and the IsOccupied indicates whether the data block is occupied by a worker thread. When LATTE allocates data blocks for insert operations, it needs to access the segment manager to obtain available data blocks. The data block is available if and only if the remaining physical space can accommodate the tuple to be inserted and are not occupied by any worker thread. A tuple is stored in one data block or across multiple data blocks. LATTE maintains index information for each tuple of the table using a hash-based primary index. The item of the index is a quad of <Key, BlockID, Offset, IsDeleted>. Where the Key is the value of the primary key processed by the MurmurHash function, the BlockID is the number of the block in which the tuple is located, the Offset is the starting offset of the tuple within the block, and the IsDeleted indicates whether the tuple is deleted. We can obtain the distribution of each tuple on the NVMe device by accessing the hash index. With the segments, LATTE enables regionalized storage and management of tables. It makes data access between different tables independent of each other. When the space consumption of the segment exceeds a certain threshold, the segment manager will apply for a new segment.

Algorithm 1: Queue Allocation

```
1 $Q_{min} \leftarrow qpair[0]$;
2 $S_{min} \leftarrow \sum_{j=1}^{qstate[0].length} (\lambda_j \cdot Size_j)$;
3 for $i$ = 1 to SYSTEM_IO_QUEUES do
4     $score \leftarrow \sum_{j=1}^{qstate[i].length} (\lambda_j \cdot Size_j)$;
5     if $score < S_{min}$ then
6         $S_{min} \leftarrow score$;
7         $Q_{min} \leftarrow qpair[i]$;
8 append the I/O task to $Q_{min}.SQ$;
9 $Q_{min}.length \leftarrow Q_{min}.length + 1$;
```

LATTE also leverages multiple deep I/O queues to manipulate different data blocks for higher parallelism, as shown in Fig. 4. When LATTE receives a write request, it first obtains the information of BlockID and Offset of an available block. Next, it hashes the primary key to get a specific hash bucket. Then, LATTE constructs a quad of <Key, BlockID, Offset, IsDeleted> and appends it to the corresponding hash bucket. The data is finally submitted as an I/O task to an idle SQ to wait for writing to the NVMe device. For read operations, LATTE firstly looks at the cache. If it encounters cache misses, the corresponding index item is gotten by hashing the primary key. Then, the I/O task including the information of BlockID and Offset is appended to an SQ to wait for reading required data from the NVMe device. Since different I/O queues do not directly interfere with each other, the I/O tasks on them can be processed in parallel.

TABLE III: **The Strengths and Weaknesses of Each CPU Affinity Mode.**

	Strengths			**Weaknesses**	
Naïve	Providing more balanced use of CPU resources.			Increasing the cache miss and thread migration.	
Pair-Binding	Reducing the cache miss and thread migration.	\	\	Some CPU cores are unbound and idle.	SQ and CQ bound to the same CPU core compete for CPU resources.
Separated-Binding		Reducing CPU competition between SQ and CQ.	\		Consuming more CPU resources for polling.
Skew-Binding			Saving more CPU resource for calculations and data processing.		All CQs occupy only one CPU core for polling, so the check for each I/O task completion is slightly slower than that of Separated-Binding.

B. Load Balancing of I/O Queues

When assigning I/O queues to worker threads, it is convenient to assign them randomly or sequentially. Due to differences in data volume and operation type, different I/O tasks may vary greatly in completion time. Simply using random or sequential assignments may cause load imbalances between queues, further causing long-term I/O tasks on certain queues to starve subsequent tasks. To balance tasks between I/O queues, LATTE monitors the completion of tasks for each I/O queue in real time and always assigns the SQ with the lightest load to the associated worker threads.

A specific queue allocation method in LATTE is shown in Algorithm 1. Depending on the latency shown in Fig. 2, we assign different weights to different I/O types and access patterns (e.g., λ_r = 1, λ_{sw} = 2, λ_{rw} = 3). Since the completion time of an I/O task is mainly affected by the amount of data ($Size_j$) and the type of operation (λ_j), LATTE scores each SQ to get $\arg\min_i\,(\sum_{j=1}^{qstate[i].length}(\lambda_j \cdot Size_j))$ (lines 3-7). Then, LATTE submits the I/O task to this relatively idle SQ (lines 8-9). It is LATTE's default allocation strategy. We could set priorities for different applications, enabling latency-sensitive applications to monopolize some queues.

C. Binding of I/O Queues and CPU Cores

LATTE supports four CPU affinity modes, namely Naïve, Pair-Binding, Separated-Binding and Skew-Binding. The Naïve mode uses Linux default CPU scheduling, and each I/O queue pair is not explicitly bound to a CPU core. In the Pair-Binding mode, each queue pair is bound to a specific CPU core. Different from the Pair-Binding with the queue pair as the binding unit, the Separated-Binding binds the SQ and the CQ of one queue pair respectively to different CPU cores. In the Skew-Binding mode, SQs are bound to different CPU cores, but all the CQs are bound to only one CPU core. We use the Pair-Binding mode as an example to analyze the process of CPU binding and queue detection. The execution of other modes is similar to Pair-Binding, except that the correspondence at the time of binding is slightly different. As shown in Algorithm 2, LATTE first initializes a "qpair" object for each queue pair which is used to manage the I/O tasks submitted to its SQ (line 3). Next, LATTE allocates the memory space for the queue pair and initializes the queue length (lines 4-5). Then, LATTE creates a polling thread for the current CQ and initializes the polling function to Completion() (line 6). Finally, LATTE binds the polling thread to one unoccupied CPU core (line 7), that is, the CPU core specifically serves the tasks of the current queue pair. While the system is running, the polling thread constantly checks the task completion of the corresponding queue pair and responds immediately (lines 9-10).

Algorithm 2: Pair-Binding Mode

```
1  Function Binding()
2      for $i$ = 0 to SYSTEM_IO_QUEUES do
3          $qpair[i] \leftarrow$ initialize an I/O queue pair object;
4          $qbuf[i] \leftarrow$ allocate memory space;
5          $qstate[i].length \leftarrow$ 0;
6          initialize a thread $poller(\text{Completion}, i)$;
7          set cpu affinity for the $poller$ and cpu $i$;
8  Function Completion($qid$)
9      while true do
10         process any outstanding completions for I/O submitted
             on the $qpair[qid]$;
```

The pros and cons of four modes are summarized in Table III. In the Naïve mode, each CPU core can be used in a balanced manner. Considering that the number of I/O queues supported by NVMe devices and the number of host CPU cores are usually unbalanced, using this mode can avoid wasting resources of unbound CPU cores. On the other hand, the Naïve mode increases the cache miss and thread migration. The Pair-Binding mode mitigates this issue by setting CPU affinity for queue pairs, but the SQ and the CQ bound on the same CPU core will compete for CPU resources. Since the CPU core bound to the CQ will poll continuously to check the completion of I/O requests, it will interfere with the submission and processing of I/O tasks on the SQ. The Separated-Binding mode separates the binding for SQs and CQs. It reduces CPU competition between the SQ and the CQ, but half of the CPU cores in the host continue to poll, resulting in lower CPU utilization. Skew-Binding mode binds all CQs to only one CPU core, allowing more CPU resources in the host for data calculation and processing. However, using a single CPU core to check the completion of all CQs is inevitably slightly slower than that of the Separated-Binding mode.

V. UNDO LOGGING ON HETEROGENEOUS STORAGE

As pointed in II-C, the WAL mechanism exacerbates the write amplification and wastes additional unnecessary storage space. Given the cost of NVMe devices, we no longer want to use more redundant logs to exchange performance and data consistency like cheap disks. On the other hand, there is no significant difference between random access and sequential access for NVMe devices, which motivates us to update the data on the peripherals without resorting to additional logs.

Directly updating the data on the NVMe device will introduce data inconsistency. To avoid the torn write, NVMe device provides some degree of atomicity at the hardware level, to ensure that inconsistent data writes do not occur within

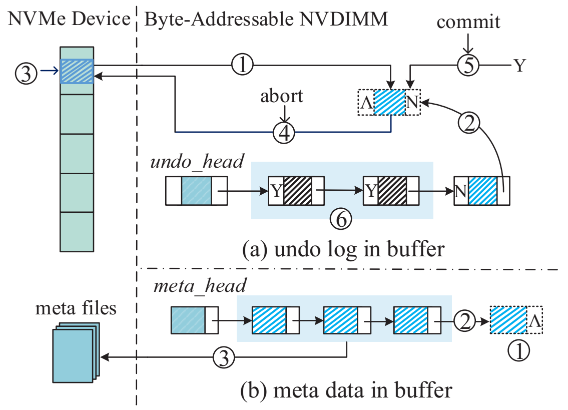


Fig. 5: **The Non-Volatile Buffer.**

a limited size [23]. Considering the diversity of application requirements (e.g., an update spanning multiple blocks), the atomic boundaries [24] supported by hardware may not be sufficient. Therefore, we need to use the undo logging to ensure atomic writes [25]. The steps for updating based on undo logging in LATTE are as follows.

(1) Receive and parse a write request.
(2) Persist the relevant undo log. If the previous value does not exist, it is recorded as NULL.
(3) Update the corresponding data in the NVMe device.
(4) Commit the request and respond to the client.

However, the above steps have two I/Os before responding to the client. Although NVMe SSDs have high bandwidth and ultra-low latency, they are block devices essentially, and their performance is still far behind memory. Since the I/O operation caused by the step (3) is unavoidable, we consider minimizing the impact of persisting undo logs. Therefore, LATTE uses heterogeneous storage of *NVMe SSD + NVDIMM*. It stores the table data on the NVMe device (as described in Section IV-A) and temporarily keeps the undo logs in the byte-addressable NVDIMM [26] to speed up. After the transaction is committed, the corresponding undo logs can be deleted, so a small-capacity NVDIMM is sufficient as a non-volatile buffer.

The undo log management in non-volatile buffer is shown in Fig. 5(a). When a write request arrives, an undo log will be written into the non-volatile buffer by the `CLWB` and `MFENCE` instructions [27] (①). Next, the undo log will be appended to the end of the log list (②). The associated tuple data in the NVMe devices will then be updated (③). If the update fails, the modification will be aborted and the associated tuple data will be rolled back using the undo log in the buffer. After that, the undo log is marked as expired ('Y' in Fig. 5(a)) (④). If the updated data has been committed, the corresponding undo log will also be marked as expired (⑤). When a write request spans multiple blocks, it can not be committed until all related blocks have been successfully updated. Finally, the garbage collection thread will release the expired undo logs in batches (⑥). When the use of NVDIMM space reaches a certain threshold (e.g., 60%), LATTE also triggers space release. In extreme cases, once the NVDIMM space is exhausted, subsequent undo logs will have to be written to the NVMe SSD.

Correctness. With the undo logging on heterogeneous storage, the undo logs are kept in NVDIMM before the tuple data is modified, which ensures that any aborted transaction can be rolled back. It also forces the data update on the NVMe device before responding to the client to ensure data durability. When the system restarts, there are two types of data. The committed tuples have no controversy due to the data durability, while the uncommitted tuples can be safely restored based on the undo logs.

VI. OTHER IMPLEMENTATIONS

Cache. Although NVMe SSDs have high read IOPS, their read latency is still at the microsecond level. To promote query efficiency, LATTE implements a single-level block cache for hot data. The cache follows the least recently used (LRU) replacement policy. It is storage-friendly because it has the same unit size as the underlying physical storage so that it can read blocks from the device without block splitting.

Transactional Isolation. LATTE provides three isolation levels, namely `Read Committed`, `Repeatable Read`, `Serializable`. At the `Read Committed` level, LATTE applies an exclusive lock on the write operation, ensuring that the client only reads the committed data. The `Repeatable Read` level guarantees that the results of multiple reads in the same transaction are the same. All tuples to be operated are locked before the transaction begins, and the read operation blocks the write operation. `Serializable` is the strictest level, and each query requires a table-level shared lock, which is blocked by both reads and writes.

Metadata Persistence. Metadata (including schema and management information) is the basis for table operations. It is used to recover the system and thereby needs to be safely preserved before the request is committed. To improve persistence efficiency, LATTE uses an NVDIMM device as a buffer, as shown in Fig. 5(b). When the request has been committed, its corresponding metadata in the buffer will be periodically written back to the metafiles in batches (③). After LATTE restarts, it traverses the linked list and flushes all metadata entries into the metafiles. It then loads and parses the metafiles to construct the metadata in DRAM.

VII. EVALUATION

In this section, we perform experiments on the working prototype of LATTE with typical benchmarks, in order to evaluate the following issues.

- How does LATTE perform against the state-of-the-art storage engines?
- Whether LATTE can run on multiple storage devices supporting the NVMe protocol with superior performance?
- Does LATTE consume less storage space and mitigate the issue of write amplification?
- What is the impact of using an undo logging method on heterogeneous storage?

The experiments are tested on a machine with a 3.2 TB Intel DC P3608 Series NVMe SSD, a 375 GB Intel Optane DC P4800X NVMe SSD, dual ten-core Xeon E5-2630 v4 CPUs

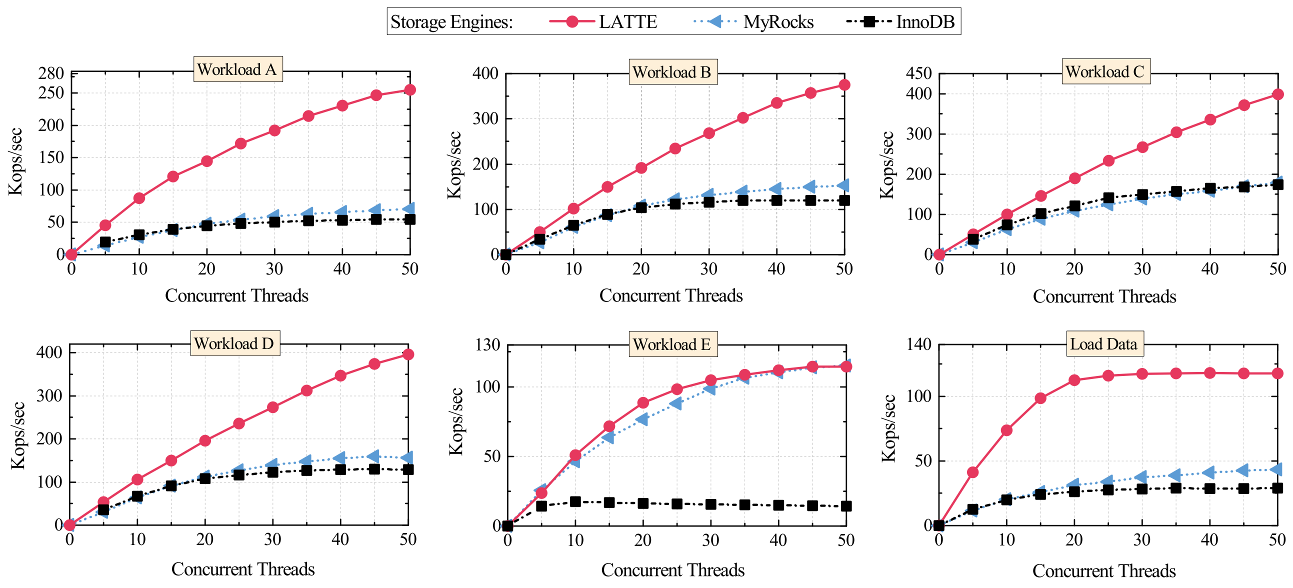


Fig. 6: **Throughput Comparison of LATTE, MyRocks and InnoDB.**

TABLE IV: **Core Workloads in YCSB.**

Workload	Read	Update	Scan	Insert	Distribution
A (Update heavy)	50	50	-	-	Zipfian
B (Read heavy)	95	5	-	-	Zipfian
C (Read only)	100	-	-	-	Uniform
D (Read latest)	95	-	-	5	Latest
E (Short ranges)	-	-	95	5	Zipfian
Load (Insert only)	-	-	-	100	Uniform

and 224 GB of RAM. The system runs the CentOS 7 Linux distribution with Linux 3.10.0 kernel. Given that we have no NVDIMM device available, so we emulate it using the DRAM with additional delay similar to prior works [28]–[30].

A. *Throughput*

We choose the database's typical storage engines, InnoDB and MyRocks [31], as well as the SPDK-based RocksDB (spdk-v5.6.1) [32] for performance comparison. Considering that mainstream database systems mainly include the relational database and the key-value store, storage engines of databases are also divided into two categories accordingly. The InnoDB engine and the MyRocks engine in MySQL are practical and efficient implementations of these two types, so we compare them with LATTE. Both InnoDB and MyRocks run on top of the local EXT4 file system. By comparing LATTE, InnoDB, and MyRocks under standard benchmarks, we could observe the performance improvement of the Lightstack-based storage engine compared to the mainstream Fullstack-based storage engines. In addition, LATTE utilizes the SPDK's NVMe driver library during the implementation. To isolate the impact of the library on runtime performance, we compare LATTE with RocksDB (spdk-v5.6.1). The latter utilizes the SPDK functionality, but is still in the layered software stack, including the NVMe driver, Blobstore and BlobFS [33].

First, we use the Yahoo! Cloud Serving Benchmark (YCSB) [34] to evaluate throughput. YCSB has five different core workloads, each representing a specific type of application [35], as shown in Table IV. In addition to the five explicit workloads, we add `Load` workload, which is the basic phase of other experiments. The default data set is 10 GB, and the size of each row in the table is 505 bytes, which includes a 255-byte primary key and five 50-byte columns. We adopt a host server equipped with a P3608 NVMe SSD to deploy LATTE, InnoDB, and MyRocks respectively. In order to increase the fairness of the experimental comparison, we make YCSB call the stored procedures in MySQL to avoid the overhead of SQL parsing. To prevent the competing of CPU resources between clients and the host, we deploy five YCSB clients on other servers and connect them to the host server through a 10-gigabyte bandwidth network. We set the write latency of 600 ns for the NVDIMM and the same read latency as DRAM according to the prior work [28]. All three storage engines use their default system configuration. In particular, LATTE adopts the Naïve affinity mode to verify the performance improvements of the Lightstack design. Fig. 6 shows that LATTE outperforms InnoDB and MyRocks, and has up to 6.57× the throughput of InnoDB and 3.62× the throughput of MyRocks. LATTE is able to perform well in a variety of application scenarios, including read-intensive and write-intensive workloads. The significant performance improvement is mainly attributed to LATTE's parallel queues scheduling strategy. It leverages the higher bandwidth and the multiple deep I/O queues of NVMe devices, so read and write operations can be performed as parallel as possible. Moreover, shorter I/O paths reduce the processing time of each I/O task, making the queue utilization more efficient.

Second, we compare LATTE with the SPDK-based RocksDB to further analyze the merit of the lightweight storage stack. Since the BlobFS in RocksDB (spdk-v5.6.1) is only integrated with the db_bench benchmark, we implement a similar module in LATTE to perform the comparison.

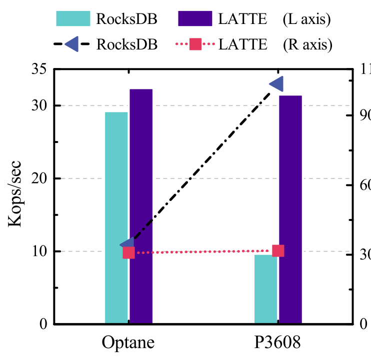


Fig. 7: **Comparison of LATTE and BlobFS-based RocksDB.**

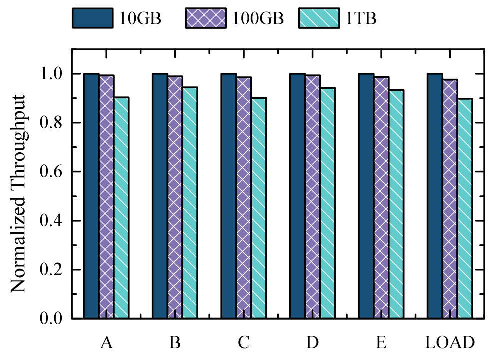


Fig. 8: **Different Sizes of the Dataset.**

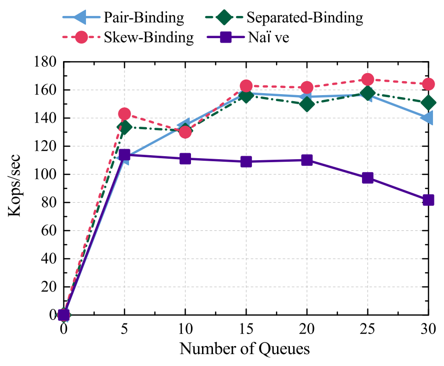


Fig. 9: **Throughput of Four CPU Affinity Modes with Different I/O Queues.**

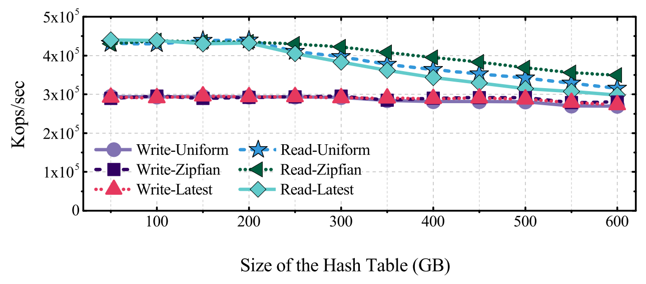


Fig. 10: **Increased Sizes of the Hash Table.**

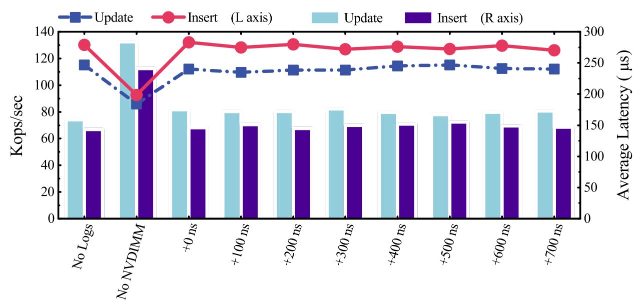


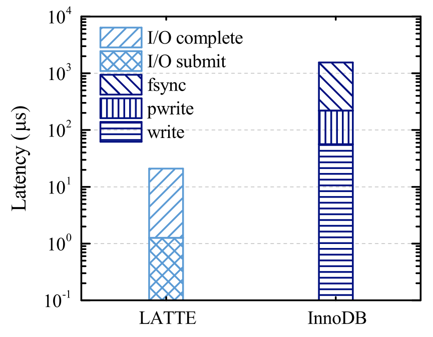
Fig. 11: **Different Configuration of Log Buffer.**

Considering the impact of caching on read and write, we use the `fillsync` workload of the db_bench, which tests 100%-write-sync operations to evaluate the effort of those two systems upon the NVMe driver of SPDK. Besides, to isolate the throughput improvement which is gained by leveraging the hardware parallelism, we use a single client thread for comparative testing. Fig. 7 shows the experimental result of the LATTE and SPDK-based RocksDB. and it is observed that LATTE performs better than RocksDB, which demonstrates that the Lightstack could further reduce the overhead. Since the Optane NVMe SSD has excellent IOPS, the throughput of LATTE and RocksDB when running on the P3608 is lower than that on the Optane, and the latency is higher. Furthermore, the throughput and latency of LATTE are more stable upon different devices compared with RocksDB. Besides, it is noted that the table-level operations in LATTE are more elaborate and time-consuming than put and get operations in RocksDB.

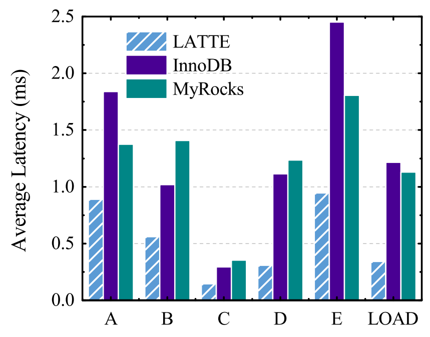
We also evaluated the performance of different data volumes. The experiment was performed on a 3.2TB P3608 Series NVMe SSD. We use YCSB to test LATTE under 10GB, 100GB, and 1TB datasets. To avoid performance interference caused by virtual memory, we turned off Linux swap. The experimental result is shown in Fig. 8. Compared with the result of 10GB and 100GB, the overall performance of 1TB shows the degradation of 5%-15%. Since LATTE schedules I/O queues in parallel, the I/O performance of the insert and update operations are not greatly impacted as the amount of data increases. As the number of available data blocks in the Segment is reduced, the efficiency of block allocating is reduced, so the write performance is slightly degraded. Due to the existence of the cache, the performance of the query is not significantly decreased, especially the workloads `A`, `B`, `D`, `E` of a large amount of hot data.

B. Functional Analysis

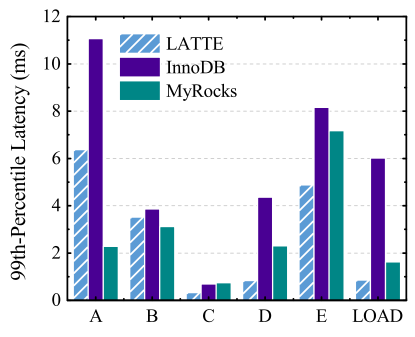
Fig. 9 depicts the peak performance of the insert operation under four CPU affinity modes in LATTE as the number of I/O queues increases. Among them, the performance of the Naïve mode shows a downward trend with the number of queues increases further. It is due to the fact that the Naïve does not fully combine the CPU cores and the I/O queues, thus introducing more thread migration and higher cache misses. The performance is highest in Skew-Binding mode because it binds all CQs to the same core, which mitigates the scramble for CPU resources so that more CPU resources are used to serve user requests. When we bind the CPU core to the I/O queues, two factors affect the overall throughput. One is binding the queue to the CPU core drops the cache miss rate and avoids migration of threads between CPU cores, which brings performance increasing. The other is the performance degradation caused by competing for CPU time slices for polling. Since the Pair-Binding mode is that both the SQ and the CQ are bound to the same core, the above two factors have a similar impact on each bound core. Hence, as the number of queues increases, the performance in this mode tends to increase monotonically. The Skew-Binding and Separated-Binding modes bind each pair of SQ and CQ to the CPU core respectively, and the influence of the above two factors for each CPU core are different. As the number of bound cores increases, the impact of the two factors on performance



(a) Break-Down of an Insert.



(b) YCSB Average Latency.



(c) YCSB Tail Latency.

Fig. 12: **Latency Comparison of LATTE, InnoDB and MyRocks.**

increases and decreases are canceled out. Therefore, Skew-Binding mode and Separated-Binding mode show a short-term performance degradation and then increase again.

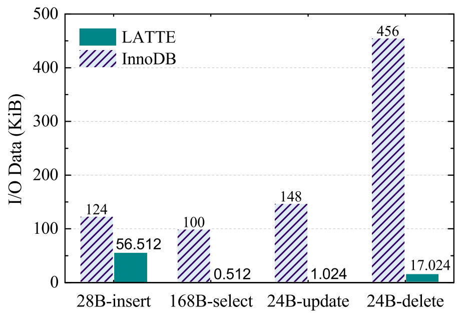
We also evaluated different sizes of the hash table for write and read under the Uniform, Zipfian, and Latest request distributions. We set the memory usage threshold of the hash table to 50%. After the threshold is exceeded, items of the hash table will not be written into the memory and will only be appended into the metadata files. The experimental result is shown in Fig. 10. We observed that as the hash items increase, the write throughput is stable, while the read throughput drops slightly after the size of the hash table exceeds the memory threshold. Since LATTE flushes metadata in batches through the NVDIMM (see Section VI) regardless of the request distribution, the performance of writing hash items is almost unaffected as hash table increases. The performance degradation of reads is mainly due to the need to retrieve some hash items from the metadata file. Especially in the Latest distribution, read performance suffered greatly. It is because recently written data is in the metadata files after the memory threshold is exceeded, so the read under the Latest distribution has a higher probability of inducing I/O. Under the Zipfian distribution, 80% reads are concentrated on 20% of hot data, which leads to a lower likelihood of performance impact.

We now perform experiment to analyze further whether undo logging will hurt overall throughput and latency. In this experiment, we add different memory delays to simulate different byte-addressable NVDIMM devices, and utilize the experimental result of writing without the undo logging method as a baseline (labeled as "No Logging"). We also added the result of persisting undo logs into the NVMe SSD without resorting to the NVDIMM (labeled as "No NVDIMM"). In Fig. 11, the performance difference of using the NVDIMM and not using the NVDIMM is about 30.6%-42.5%, and the latency reduction is 35.7%-38%. Therefore, the NVDIMM buffer is adopted to mitigate the performance reduction. Further, Fig. 11 also shows that as the NVDIMM's write delay increases, the throughput and average latency do not change significantly from the baseline, whether it is an insert or update operation. Since the latency of byte-addressable NVDIMMs is on the order of hundred nanoseconds, the software processing delay for insert or update operations is also at the microsecond level, which is an order of magnitude difference. Therefore, even on the critical path, keeping undo logs does not become a performance bottleneck. As a result, undo logging in LATTE has no significant negative impact on overall performance.

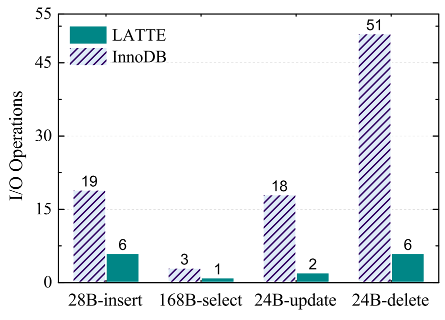
C. Latency

Besides the augment in throughput, we are also concerned about the reduction in response latency. We adopt the strace tool [36] to track the details of a single insert operation in LATTE and MySQL, and the result is shown in Fig. 12(a). Each insert in MySQL with InnoDB engine takes an average of 10^3 μs and can be divided into three phases, `pwrite`, `fsync`, and `write`. The `pwrite` and `fsync` are the stages in which the actual data is written, and the `write` stage is a small part of appending system log files. LATTE only takes 20 μs, and 1 μs is used to submit the request, while the rest is used to wait for the request to complete. Since the `I/O submit` and `I/O complete` are the basic stages of each table-level operation in LATTE, all I/O requests on the LATTE host can be completed in 20-30 μs. The sharp drop in latency is primarily due to LATTE's shorter I/O path in user space, eliminating the need to transfer data between the file system layer and the block layer. In addition, polling the completion queue for checking allows the host to immediately receive the completed I/O requests.

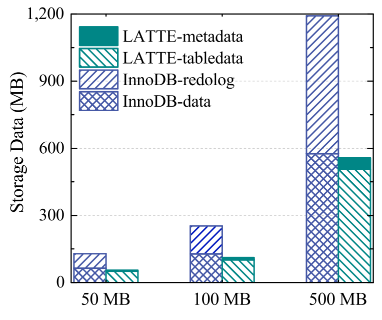
Figs. 12(b) and 12(c) respectively describe the average and tail latency when each YCSB workload reaches peak throughput. It can be seen that the average delay of LATTE is significantly lower than that of MyRocks and InnoDB. Especially for the workload `Load`, LATTE's average latency is only 28% of that of other storage engines. For workloads `A`, `B`, `D` and `E`, LATTE's average latency is about 30%-50% of that of InnoDB and MyRocks. The weakening of the delay difference between them is mainly due to the impact of data caching. In a mixed workload, the more reads and the higher the cache hit ratio, the smaller the delay gap between the three storage engines. Workload `C` contains only read operations, resulting in the lowest latency and the smallest difference between them. Since YCSB clients interact with the host over the network, the response latency includes not only local processing time but also round-trip time (RTT), which causes LATTE's average latency to be on the order of hundreds of microseconds.



(a) I/O Data.



(b) I/O Operations.



(c) Space Utilization.

Fig. 13: **Storage Overhead in LATTE and MySQL.**

Fig. 12(c) shows that the tail delay of LATTE is also lower than that of MyRocks and InnoDB under workloads C, D, E, and Load, but higher than that of MyRocks in workloads A and B. It is mainly because LATTE submits more I/O requests than MyRocks when peak throughput is reached (see Fig. 6) resulting in deeper queue depth. Also, the default workloads A and B use the Zipfian request distribution [35]. Since the Zipfian is an uneven distribution, more lock conflicts of update operations are caused, which limits the I/O parallelism of LATTE. Workload B has a smaller update ratio than A, so the tail delay of three engines under the workload B is less impacted and its difference is smaller.

D. Storage Overhead

In addition to improving runtime performance, LATTE is also working to reduce the storage overhead of NVMe devices. We use the strace tool to record the actual amount of data and total I/O counts generated by LATTE and MySQL with InnoDB engine during insert, select, update, and delete operations, and then compare their storage overhead.

According to Figs. 13(a) and 13(b), we observe that LATTE generates much less I/O data and operations than MySQL. It is thanks to LATTE's lightweight storage stack. Unlike traditional file system-based databases, LATTE directly accesses data blocks of NVMe devices, bypassing the file system layer and the block layer. It eliminates the extra merging and decomposition of I/O requests, which mitigates the write amplification. Since the select and update operations involve only accessing the data blocks without modifying the metadata files, the amount of I/O data and the number of I/Os are especially small. Fig. 13(c) shows that MySQL occupies twice as much storage space as LATTE. In fact, redo logs are the main reason in causing this problem. It is almost stored in a 1:1 ratio with the table data for MySQL, as shown in the experimental diagram. LATTE temporarily stores undo logs in the NVDIMM but does not retain any redo logs, which greatly reduces the storage footprint.

VIII. Related Work

NVMe-Oriented Works. With the popularity of NVMe SSDs, more and more research is focused on fully exploiting its hardware potential. Xu et al. [37] give the performance analysis of NVMe drivers and the inapplicability of traditional software mechanisms. NVMeDirect [38] builds an efficient user-space I/O framework for NVMe SSDs. H-NVMe [10] develops a hybrid framework of the NVMe-based storage system in cloud computing. There are also some useful optimizations for NVMe devices, e.g., the schedule of I/O threads [39], isolating read and write requests [40], reducing the cost of remote access [41], reducing DRAM footprint [42], I/O resource sharing [43], NVMe virtualization [44], optimized journaling and checkpointing schemes [45], improving data plane efficiency [46] and introducing WRR support in Linux NVMe drivers [47]. LATTE also runs on NVMe SSDs. It leverages multiple I/O queues of the NVMe protocol to fully exploit device parallelism.

Approaches Targeting to Lean Storage Stack. To implement lightweight systems, some studies have reduced the storage stack overhead [48]. LightNVM [9] is a Linux subsystem designed for open-channel SSDs. It is located in the Linux kernel and integrated with the traditional I/O stack, while LATTE bypasses the kernel to provide a shorter I/O path in the user space. Unlike LightNVM providing host-based FTL, LATTE focuses on reducing the overhead of the storage stack upon the device driver (e.g., overlapping overhead from the database's storage engine and the file system, task migration between queues of different layers). Arrakis [49] and IX [50] reduce the software overhead of the operating system. Some recent file systems [7], [51]–[53] provide user-level direct accesses to the storage devices. NoFTL [5] integrates Flash management into the database layer. Bankshot [54] proposes a caching architecture which reduces the overhead of operating systems and file systems. FLASHSHARE [55] simplifies the storage stack spanning from the OS kernel to the SSD firmware for Ultra-Low-Latency SSDs. Those designs mainly benefit from lightening a single software layer or a partial software stack. Our work differs from those efforts, it integrates and simplifies the layers of the database, file system, and operating system to shorten the I/O path.

Log Optimization for NVDIMMs. Ouyang et al. [25] design the atomic-write primitive within a log-structured FTL that allows multiple I/O operations to commit or rollback as a group. Ogleari et al. [56] propose an efficient hardware undo+redo logging scheme for NVDIMM. Wang et al. [57] propose a lightweight group commit protocol for NVDIMM-based distributed logging. Write-behind logging (WBL) [58] is a research effort to reduce log volume. It is designed for byte-addressable NVDIMMs. PolarFS [59] uses hybrid storage devices to store logs and data. It stores the chunk data in the NVMe SSDs and persists the write-ahead logs in the faster 3D XPoint SSD. LATTE provides management for undo logs on heterogeneous storage of the NVDIMM and the NVMe SSD. It uses the NVDIMM to store undo logs temporarily, and releases them in batches after the transaction commits.

IX. CONCLUSION

We proposed the Lightstack framework, a lightweight storage stack designed for NVMe devices. The core of Lightstack is LATTE, a native table storage engine with high throughput and ultra-low latency. LATTE accesses NVMe devices directly in user space to reduce single I/O latency and combines multiple deep I/O queues and CPU cores to improve parallelism. It only records undo logs to decrease the storage overhead while ensuring data consistency. Although the traditional storage stack is well-developed, LATTE demonstrates that using a customized lightweight storage stack with corresponding optimization, can make better use of emerging hardware.

ACKNOWLEDGMENTS

This research was supported by National Natural Science Foundation of China (No. 61772204, No. 61732014), ECNU Academic Innovation Promotion Program for Excellent Doctoral Students, and the ECNU Fund for International Conferences (No. 40600-19301-512200). The authors would like to thank the anonymous reviewers for their valuable feedback.

REFERENCES

[1] "NVM Express Overview," http://www.nvmexpress.org/nvm-express-overview/.
[2] "Intel Optane Memory," http://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html.
[3] P. Ilia, K. Andreas, H. Sergey, and V. Tobias, "Native Storage Techniques for Data Management," in *ICDE workshops*, 2019.
[4] M. Bjørling, P. Bonnet, L. Bouganim, and N. Dayan, "The Necessary Death of the Block Device Interface," in *CIDR*, 2013.
[5] S. Hardock, I. Petrov, R. Gottstein, and A. Buchmann, "NoFTL: Database Systems on FTL-less Flash Storage," *VLDB*, vol. 6, no. 12, pp. 1278–1281, 2013.
[6] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible File-System Interfaces to Storage-Class Memory," in *EuroSys*, 2014.
[7] S. Kannan, A. Arpaci-Dusseau, R. Arpaci-Dusseau, Y. Wang, J. Xu, and G. Palani, "Designing a True Direct-Access File System with DevFS," in *FAST*, 2018.
[8] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang, "SDF: Software-defined Flash for Web-scale Internet Storage Systems," in *ASPLOS*, 2014.
[9] M. Bjørling, J. González, and P. Bonnet, "LightNVM: The Linux Open-Channel SSD Subsystem," in *FAST*, 2017.
[10] Z. Yang, M. Hoseinzadeh, P. Wong, J. Artoux, C. Mayers, D. T. Evans, R. T. Bolt, J. Bhimani, N. Mi, and S. Swanson, "H-NVMe: A Hybrid Framework of NVMe-based Storage System in Cloud Computing Environment," in *IPCCC*, 2017.
[11] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write Amplification Analysis in Flash-Based Solid State Drives," in *SYSTOR*, 2009.
[12] Y. Lu, J. Shu, and W. Zheng, "Extending the Lifetime of Flash-based Storage through Reducing Write Amplification from File Systems," in *FAST*, 2013.
[13] "MyRocks," http://myrocks.io/.
[14] "AHCI and NVMe as Interfaces for SATA Express Devices - Overview," https://www.sata-io.org/sata-express.
[15] Y. Wang, H. Yu, L. Ni, G.-B. Huang, M. Yan, C. Weng, W. Yang, and J. Zhao, "An Energy-Efficient Nonvolatile In-Memory Computing Architecture for Extreme Learning Machine by Domain-Wall Nanowire Devices," *IEEE Transactions on Nanotechnology*, vol. 14, no. 6, pp. 998–1012, 2015.
[16] "JEDEC Announces Support for NVDIMM Modules," https://www.jedec.org/news/pressreleases/jedec-announces-support-nvdimm-hybrid-memory-modules.
[17] "FIO," http://git.kernel.dk/?p=fio.git;a=summary.
[18] "Block I/O Layer Tracing (Blktrace)," https://git.kernel.org/pub/scm/linux/kernel/git/axboe/blktrace.git/.
[19] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems," in *SYSTOR*, 2013.
[20] J. Yang, D. B. Minturn, and F. Hady, "When Poll is Better than Interrupt," in *FAST*, 2012.
[21] J. Arulraj and A. Pavlo, "How to Build a Non-Volatile Memory Database Management System," in *SIGMOD*, 2017.
[22] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems," in *SIGMOD*, 2015.
[23] "NVM express, revision 1.3 c," https://nvmexpress.org/wp-content/uploads/NVM-Express-1_3c-2018.05.24-Ratified.pdf, 2018.
[24] "Intel Solid-State Drive DC P3608 Series," https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-dc-p3608-spec.pdf, 2015.
[25] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda, "Beyond Block I/O: Rethinking Traditional Storage Primitives," in *HPCA*, 2011.
[26] M. Seltzer, V. Marathe, and S. Byan, "An NVM Carol: Visions of NVM Past, Present, and Future," in *ICDE*, 2018.
[27] "Architecture Instruction Set Extensions and Future Features Programming Reference," https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf, 2018.
[28] F. Xia, D. Jiang, J. Xiong, and N. Sun, "HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems," in *USENIX ATC*, 2017.
[29] J. Ou, J. Shu, and Y. Lu, "A High Performance File System for Non-Volatile Main Memory," in *EuroSys*, 2016.
[30] J. Huang, K. Schwan, and M. K. Qureshi, "NVRAM-aware Logging in Transaction Systems," *VLDB*, vol. 8, no. 4, pp. 389–400, 2014.
[31] "RocksDB," http://rocksdb.org/.
[32] "SPDK-RocksDB," https://github.com/spdk/rocksdb/tree/spdk-v5.6.1.
[33] "BlobFS," http://www.spdk.io/doc/blobfs.html.
[34] "YCSB Master Branch," https://github.com/brianfrankcooper/YCSB.
[35] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *SOCC*, 2010.
[36] "Strace," https://strace.io/.
[37] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan, "Performance Analysis of NVMe SSDs and their Implication on Real World Databases," in *SYSTOR*, 2015.
[38] H.-J. Kim, Y.-S. Lee, and J.-S. Kim, "NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs," in *HotStorage*, 2016.
[39] J. Qian, H. Jiang, W. Srisa-an, S. Seth, S. Skelton, and J. Moore, "Energy-Efficient I/O Thread Schedulers for NVMe SSDs on NUMA," in *CCGRID*, 2017.
[40] M. Lee, D. H. Kang, M. Lee, and Y. I. Eom, "Improving Read Performance by Isolating Multiple Queues in NVMe SSDs," in *ICUIMC*, 2017.
[41] Z. Guz, H. H. Li, A. Shayesteh, and V. Balakrishnan, "NVMe-over-Fabrics Performance Characterization and the Path to Low-Overhead Flash Disaggregation," in *SYSTOR*, 2017.
[42] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. Hazelwood, C. Petersen, A. Cidon, and S. Katti, "Reducing DRAM Footprint with NVM in Facebook," in *EuroSys*, 2018.
[43] S. Ahn, K. La, and J. Kim, "Improving I/O Resource Sharing of Linux Cgroup for NVMe SSDs on Multi-core Systems," in *HotStorage*, 2016.
[44] B. Peng, H. Zhang, J. Yao, Y. Dong, Y. Xu, and H. Guan, "MDev-NVMe: A NVMe Storage Virtualization Solution with Mediated Pass-Through," in *USENIX ATC*, 2018.
[45] Y. Son, S. Kim, H. Y. Yeom, and H. Han, "High-Performance Transaction Processing in Journaling File Systems," in *FAST*, 2018.
[46] Z. An, Z. Zhang, Q. Li, J. Xing, H. Du, Z. Wang, Z. Huo, and J. Ma, "Optimizing the Datapath for Key-value Middleware with NVMe SSDs over RDMA Interconnects," in *CLUSTER*, 2017.
[47] K. Joshi, P. Choudhary, and K. Yadav, "Enabling NVMe WRR Support in Linux Block Layer," in *HotStorage*, 2017.
[48] S. Swanson and A. M. Caulfield, "Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage," *Computer*, 2013.
[49] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," *TOCS*, vol. 33, no. 4, pp. 11:1–11:30, 2016.
[50] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A Protected Dataplane Operating System for High Throughput and Low Latency," in *OSDI*, 2014.
[51] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn, "DFS: A file system for virtualized flash storage," *TOS*, vol. 6, no. 3, pp. 14:1–14:25, 2010.
[52] J. Kim, H. Shim, S.-Y. Park, S. Maeng, and J.-S. Kim, "FlashLight: a lightweight flash file system for embedded systems," *TECS*, vol. 11, no. 1, p. 18, 2012.
[53] D. Woodhouse, "JFFS: The Journalling Flash File System," in *Ottawa linux symposium*, 2001.
[54] M. S. Bhaskaran, J. Xu, and S. Swanson, "Bankshot: Caching Slow Storage in Fast Non-Volatile Memory," *ACM SIGOPS Operating Systems Review*, 2014.
[55] J. Zhang, M. Kwon, D. Gouk, S. Koh, C. Lee, M. Alian, M. Chun, M. T. Kandemir, N. S. Kim, J. Kim, and M. Jung, "FLASHSHARE: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs," in *OSDI*, 2018.
[56] M. A. Ogleari, E. L. Miller, and J. Zhao, "Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems," in *HPCA*, 2018.
[57] T. Wang and R. Johnson, "Scalable Logging through Emerging Non-Volatile Memory," *VLDB*, vol. 7, no. 10, pp. 865–876, 2014.
[58] J. Arulraj, M. Perron, and A. Pavlo, "Write-Behind Logging," *VLDB*, vol. 10, no. 4, pp. 337–348, 2016.
[59] W. Cao, Z. Liu, P. Wang, S. Chen, C. Zhu, S. Zheng, Y. Wang, and G. Ma, "PolarFS: An Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database," *VLDB*, vol. 11, no. 12, pp. 1849–1862, 2018.