Flexible I/O for Database Management Systems with xNVMe

Emil Houlborg
Andreas Nicolaj Tietgen*
IT University of Copenhagen
ehou/anti@itu.dk

Simon A. F. Lund
Samsung
simon.lund@samsung.com

Marcel Weisgut Tilmann Rabl
Hasso Plattner Institute,
University of Potsdam
marcel.weisgut/tilmann.rabl@hpi.de

Javier González Vivek Shah
Samsung
javier.gonz/vi.shah@samsung.com

Pınar Tözün
IT University of Copenhagen
pito@itu.dk

ABSTRACT

Today, NVMe SSDs cover a diverse family of devices (e.g., Zoned Namespaces, Flexible Data Placement, and Key-Value SSDs) and offer high performance (μsec-scale latency). To leverage the capabilities of these devices, a variety of I/O paths are available (e.g., libaio, io_uring, and SPDK). On the other hand, to avoid the challenges and unpredictability that comes with writing code to target such diversity, most data systems today still rely on the conventional filesystem APIs (POSIX) and synchronous IO. While (maybe) increasing programmer productivity, this choice leads to sub-optimal utilization of the modern NVMe storage.

To unify the diverse I/O storage paths and make them more accessible to a wider-scale of programmers, Samsung built `xNVMe` that exposes a single message-passing API with minimal overhead. This paper takes the next step and integrates `xNVMe` into a state-of-the-art database system, DuckDB, by creating a new filesystem extension, *nvmefs*, that interacts with blocks on disk instead of files. We demonstrate that `xNVMe` integration allows DuckDB to utilize IO Passthru, SPDK, and Flexible Data Placement. Using these modern I/O methods, compared to DuckDB's default sync I/O, *nvmefs* achieves either comparable performance for non-I/O-intensive cases or up to 50% lower query times on I/O-intensive queries.

1 INTRODUCTION

NVMe SSDs have achieved widespread deployment as commodity and enterprise hardware today thanks to their high performance and rich feature set [15, 31, 47]. Despite the open specifications by the NVMe (Non-Volatile Memory Express) group [35], there is a complex labyrinth of software abstractions to program the underlying hardware [30].

The POSIX storage abstractions (e.g., `pread, pwrite`) have been the holy grail of programmability, stability, and portability to hide the underlying hardware complexity. However, the need for a low-overhead, asynchronous programming model to leverage modern NVMe hardware has created multiple complex alternative I/O storage stacks, such as `libaio` and `io_uring` interfaces [20, 30] in the Linux kernel and the user-space library `SPDK` [46]. These interfaces differ widely in their API, semantics, and performance. The landscape of interfaces gets even more complicated if one factors in different OS (e.g., Linux, FreeBSD, Windows, and MacOS) and SSD types (e.g., Zoned Namespaces (ZNS) [3], Flexible Data Placement (FDP) [1], and Key-Value (KV) [9] SSDs). This, in turn, complicates software development [43] and hinders the mass adoption and evolution of the NVMe ecosystem.

The reluctance for the adoption of the full capabilities of NVMe can also be seen in the landscape of database management systems. Despite the myriad of work optimizing database components for modern storage [6, 18, 25–27, 37, 38] and showing the bottlenecks of the POSIX abstractions [14, 15, 20], most real-world deployments still, conservatively, rely on `pread` and `pwrite`.

To make both the hardware and software landscape for NVMe SSDs more accessible, Samsung built `xNVMe` [45] that exposes a single message-passing API to support both asynchronous and synchronous communication with NVMe devices and diverse storage I/O paths in different OS and userspace libraries with minimal overhead. The capabilities and high performance of `xNVMe` have already been shown using microbenchmarks composed of read and write requests [30], while running `xNVMe` standalone using the fio tool [11]. Integrating `xNVMe` into a database management system, however, have been largely unexplored to the best of our knowledge.

This paper integrates `xNVMe` into DuckDB [7, 40] without invasive changes to its internals and usage. DuckDB is a state-of-the-art analytics system that supports out-of-core query execution [23, 24] but uses POSIX I/O. In contrast to Ottosen et al. [36] that co-designs DuckDB's buffer manager with xNVMe, we design a DuckDB extension for portability over raw performance. More specifically,

- We create a new filesystem extension for DuckDB, *nvmefs*, that follows the file abstraction in DuckDB and storage I/O API of `xNVMe` and underneath maps data to logical block addresses directly instead of files.
- We demonstrate how the `xNVMe` integration with *nvmefs* gives DuckDB easy access to `io_uring_cmd`, `spdk`, and FDP SSDs.
- Our evaluation shows that DuckDB with *nvmefs*, and without any manual tuning, executes I/O-intensive queries up to 50% faster for an aggregation benchmark [23] and TPC-H [42], and for non-I/O-intensive cases exhibits similar performance.

The rest of the paper is organized as follows. Section 2 and Section 3 provide a brief background for `xNVMe`, before Section 4 describes its integration into DuckDB. Section 5 evaluates the performance of *nvmefs*. Finally, Section 6 discusses next steps for `xNVMe` integration into data-intensive systems, and Section 7 concludes.

*Equal work distribution among the first two authors.

 16th Annual Conference on Innovative Data Systems Research (CIDR '26). January 18-21, Chaminade, USA

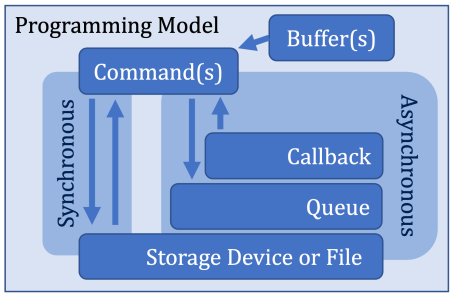


Figure 1: The xNVMe programming model. [44]

2 XNVME

xNVMe is an open-source framework providing a C API and a cross-platform user-space library (`libxnvme`) implementing this API for NVMe devices. The API advocates a command-centric programming model designed around a message passing interface that is exposed through a queue abstraction to provide both asynchronous and synchronous API. On top of the `xNVMe` API are the NVMe command set-specific APIs that help with command construction and packaging of NVMe semantics into the general `xNVMe` command infrastructure. These NVMe command sets are utilized in building command-line tools for interacting with NVMe devices. In this section, we provide a brief exposition of the programming model of `xNVMe`. For more details, the reader can consult [30, 45].

Programming Model. `xNVMe` revolves around expressions for shipping commands. To do so, one needs a device handle to send commands to and a buffer to carry the command-payload (if any). With these things in place, one can choose to send the command either in a synchronous, *blocking*, manner or in an asynchronous, *non-blocking*, manner via a *queue*. Figure 1 illustrates the model.

Reading an LBA. We illustrate using the xNVMe API with a C-style pseudo-code example. The example shows how to read a logical block address (LBA) from an NVMe device.

A *device-handle* (`xnvme_dev` in Listing 1) is obtained by providing an identifier such as the OS device path (`/dev/nvme0n1`), a PCIe-address (`0000:03:00.0`), or an NVMe-transport endpoint (`10.11.12.42:4200`). In addition to the device identifier, options are selected via `xnvme_opts`.

The selection of an I/O interface is done by `libxnvme` which decides based on the capabilities of the I/O storage paths available to the library at runtime. This default behavior is overridden by configuring options, e.g., setting `opts.be.async = "io_uring"`. In this case `libxnvme` will use `io_uring` or fail in creating the device handle if the library is not present.

```
struct xnvme_opts opts = xnvme_opts_default();
struct xnvme_dev *dev;
dev = xnvme_dev_open("/dev/nvme0n1", &opts);
if (!dev) exit(errno);
```

Listing 1: Opening an NVMe device.

With a device-handle in place, one can start preparing the payload buffers for the data to be read (Listing 2). `struct xnvme_geo` describes the device characteristics for the payload buffer allocation. In our example, we are interested in the size of an LBA. The buffer-allocator, `xnvme_buf_alloc()`, provides a pointer to virtual memory, which is aligned according to the I/O constraints of the provided device (`dev`) and any potential requirements of the I/O interface, such as direct memory access (DMA) capability.

```
struct xnvme_geo *geo = xnvme_dev_get_geo(dev);
size_t nbytes = geo->lba_nbytes;
void *buf;
buf = xnvme_buf_alloc(dev, nbytes);
if (!buf) exit(errno);
```

Listing 2: Preparing payload buffers.

A command-context (`xnvme_cmd_ctx`) encapsulates the command (`ctx.cmd`), the command-completion-status (`ctx.cpl`), the device to submit the command to (`ctx.dev`), or the queue to submit the command via (ctx.queue). The most significant command-field is the opcode (`ctx.cmd.opcode`) because it is the encoding of the operation to be performed. The command is submitted in a synchronous manner in Listing 3, with a pre-allocated queue in the background.

```
struct xnvme_cmd_ctx ctx = {
    .dev = dev,
    .mode = XNVME_CMD_SYNC,
    .cmd.opcode = XNVME_SPEC_NVM_OPC_READ};
err = xnvme_cmd_pass(&ctx, buf, nbytes, 0x0, 0);
if (err || xnvme_cmd_ctx_cpl_status(&ctx)) exit(-err);
```

Listing 3: Submitting a command syncronously.

To perform the operation in asynchronous fashion, a callback function and a queue must be set up as shown in Listing 4.

```
static void callback(struct xnvme_cmd_ctx *ctx, void *
    cb_arg) {
    if (xnvme_cmd_ctx_cpl_status(ctx)) exit(EIO);
}
struct xnvme_queue *queue = NULL;
err = xnvme_queue_init(dev, 32, 0x0, &queue);
if (err) exit(-err);
xnvme_queue_set_cb(queue, callback, NULL);

struct xnvme_cmd_ctx *ctx;
ctx = xnvme_queue_get_ctx(queue);
ctx->cmd.opcode = XNVME_SPEC_NVM_OPC_READ;
submit:
xnvme_cmd_pass(ctx, buf, nbytes, 0x0, 0);
```

Listing 4: Submitting a command asyncronously.

I/O Path Support. `xNVMe` currently supports a wide array of I/O storage paths such as POSIX aio, `libaio`, Windows IOCP, Windows IORING, `io_uring`, `io_uring` command, psync (`pread,pwrite`), block-layer IOCTLs, NVMe driver layer IOCTLs. These storage paths are supported on compatible OSes, such as Linux, FreeBSD, Windows, and MacOS. To add to the OS managed storage paths are the userspace NVMe drivers from `SPDK` and `libvfn`. The same unchanged C API examples outlined previously will work with all the storage I/O paths outlined above. In case the API is invoked on a non-NVMe device, then a fallback shim layer maps the NVMe commands to appropriate OS managed I/O operations. The popular Linux I/O benchmarking tool `fio` also provides `xNVMe` as an I/O engine [10] in its options.

Language Support. `xNVMe` is implemented in C for low-level control for system programming. C++ can directly consume the C API, and `xNVMe` provides bindings for Python and Rust. It is planned to make these bindings idiomatic to reduce the impedance in integrating the bindings in different programming languages.

3 WHY XNVME?

In this section, we provide the reasons that prompted the birth of xNVMe in 2019 and shaped its evolution over time to highlight its fit for database systems.

3.1 Fracture of the POSIX Interface

The POSIX API was envisaged as the holy grail of contracts between application developers and the operating system, enabling the creation of portable and efficient applications through the system call interface. The POSIX file-based abstractions for storage (e.g., `pread`, `pwrite`) have long been the de facto storage programming primitives, consistent with the UNIX philosophy of **"Everything is a file"**. However, the interface was developed nearly 50 years ago during an era of slow, small storage devices and limited application diversity. It has fractured over time across different OS in both its API and semantics due to the following reasons.

Fast Hardware and the Quest for High Performance. Storage hardware has undergone a sea change over the last five decades, transitioning from tape to HDDs to SSDs, with increasing capacities and bandwidth at diminishing costs. The advent of NVMe SSDs, almost fifteen years ago shifted the performance cost tradeoff between memory and storage toward the storage end [2, 13]. Not only have capacity and bandwidth improved dramatically, but latency has also decreased significantly, leading to higher performance demands from applications. This shift has drawn considerable attention to the performance overheads of kernel storage APIs [4] compared to userspace I/O libraries, notably SPDK [19], NVMeDirect [21], and NVMeVirt [22].

Over time, SPDK has emerged as the storage I/O library preferred by applications that require high storage performance, but its API and design represent a departure from the POSIX storage APIs. Recently, Linux has also taken note of the performance differences offered by storage I/O libraries. This has accelerated the mainstream adoption of `io_uring` and NVMe I/O Passthru work in the kernel [20]. This line of work borrows several ideas from the userspace storage library stack, such as shared memory between userspace and the kernel, asynchronous queuing mechanisms for requests and responses, and polling, to name a few. Similar divergences for performance can be observed in other OS as well.

The Need for Asynchronous Storage Interfaces. The POSIX storage APIs were designed to be synchronous, where the calling code is blocked until the system call completes. Modern low latency NVMe SSDs require low-overhead decoupling of the I/O submission and completion paths to efficiently utilize CPU cycles [15, 19]. This has driven the demand for asynchronous storage APIs. Early attempts at asynchronicity relying on the `epoll`/`select` calls only provided partial asynchronicity and were cumbersome to use. This led to the introduction of the POSIX asynchronous I/O API (`aio_*` system calls). However, these APIs never achieved their goals of portability and widespread adoption and suffered from performance overheads compared to userspace libraries. More recently, `io_uring` has gained attention as a programmable asynchronous I/O abstraction in Linux for both userspace applications and kernel subsystems. However, it incurs higher overheads than SPDK and requires in-depth knowledge of the Linux kernel to achieve high performance.

Evolution of NVMe SSDs. Although the NVMe protocol standardizes the programmability contract with SSDs, its command sets continue to evolve through various technical proposals aimed at leveraging advances in SSD technology. Emerging types of SSDs such as FDP SSDs [1] and computational storage SSDs [16], and upcoming proposals for QoS (Quality of Service) [17] and NVMe over CXL (Compute Express Link) [12, 33] offer functionality that extends beyond the traditional read and write abstractions of the POSIX API. Wrapping these capabilities within a POSIX abstraction not designed for such use cases inevitably sacrifices their benefits. Furthermore, experimental extensions to long-established OS abstractions often face resistance from maintainers who prioritize stability over innovation. As a result, storage vendors must expose new functionality either (1) through low-level NVMe device drivers within the OS or (2) via userspace I/O libraries such as SPDK. Both approaches pose programmability and deployment challenges that limit usability and adoption. Regardless of the chosen approach, the POSIX API is effectively abandoned.

3.2 xNVMe and Unification of Storage I/O Paths

This divergence from the unified POSIX API is unfortunate and has created significant portability and programmability challenges. The existence of multiple I/O stacks, each with its own assumptions, syntax, and semantics, introduces a steep learning curve and creates architecture lock-in that propagates into applications. xNVMe was introduced nearly six years ago to preserve the motivation behind the POSIX API of providing a unified interface for NVMe devices that can flexibly accommodate the fragmentation of storage paths described earlier. Its design principles are as follows:

(1) Provide an efficient and scalable asynchronous API across various storage I/O paths.
(2) Achieve native performance of the underlying storage path while using the xNVMe API.
(3) Provide clear API and semantics rather than generic conceptual abstraction for minimalism and extensibility.

3.3 What is xNVMe Good For?

Given the maturity and stability of the project, xNVMe is now well-positioned to serve a wide range of use cases, namely:

- Storage systems, such as key-value stores, database systems, and userspace filesystems, which require portability and high performance across I/O paths but face complexity due to diverse I/O stacks.
- Virtualization systems, including virtual machine monitors and hypervisors (e.g., QEMU, Rust-VMM), which can benefit from a unified library and API to efficiently manage and switch between multiple storage I/O paths.
- Storage hardware testing, where hardware specifications are implemented and verified across different programming languages (e.g., C for firmware, Python for testing), necessitating flexible tooling without language constraints.

In this paper, we integrate xNVMe with a state-of-the-art database system, DuckDB, to demonstrate its potential for database systems and guide its integration into other data-intensive systems and use cases mentioned above.

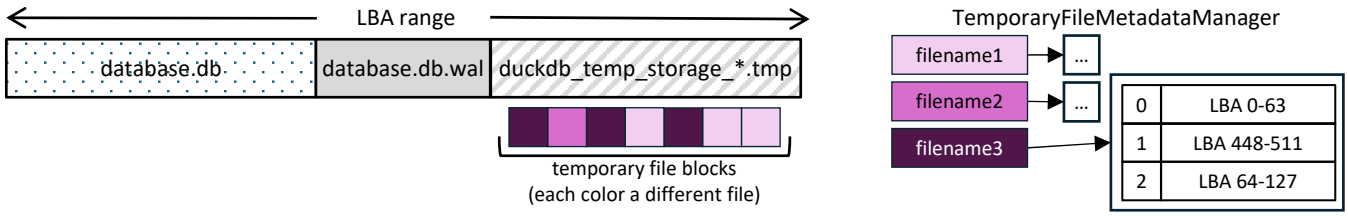


Figure 2: An example of how *nvmefs* maps the different file types to Logical Block Addresses (LBAs). The LBA ranges for the *database* and *write-ahead-log* consist of a single file. The temporary storage LBA range can contain multiple files. LBAs to file blocks mapping for the temporary files is maintained by the TemporaryFileMetadataManager in *nvmefs*.

4 INTEGRATION INTO DUCKDB

DuckDB is an in-process database management system optimized for analytical processing [7, 40]. As a design principle, DuckDB keeps its core codebase as simple and dependency-free as possible to be lightweight and allow easy adoption on a variety of platforms. Therefore, for integrating xNVMe into DuckDB, we choose to create a DuckDB community extension [8][1], which we call *nvmefs*[2].

4.1 DuckDB Files & Storage I/O

DuckDB maintains three types of files: database, write-ahead-log (WAL), and temporary. *Database* file keeps the database records. *WAL* is for the transaction log, to be used for recovery if needed. *Temporary* files are responsible for storing the intermediate query results that are spilled to disk due to lack of space in the buffer pool. The movement of the data to or from main memory is maintained by the buffer pool for both the database and temporary files [23].

Each DuckDB file is composed of blocks, where the default block size is 256 KiB. For *temporary* files, DuckDB uses different sizes of blocks, ranging from 32 KiB to 256 KiB. A different file is created for each block size. The different block sizes help with performing I/O at a granularity that minimizes fragmentation depending on the size of the data being persisted [34].

DuckDB provides a `FileSystem` abstraction to support different file system implementations. When an `OpenFile` call is issued, DuckDB's `VirtualFileSystem` determines which supported filesystem is to be used. The default is the `LocalFileSystem`, which uses OS filesystem abstractions and synchronous I/O. This also helps with DuckDB's goal of supporting a wide variety of platforms. However, it also falls short in utilizing modern NVMe storage for I/O-intensive applications.

4.2 *nvmefs*: The NVMe File System

nvmefs, the NVMe File System, offers an alternative way to manage data on disk for DuckDB. It inherits from DuckDB's `FileSystem` for compatibility and is implemented as a DuckDB community extension. To indicate a file should be handled by *nvmefs*, the user needs to add the file path prefix `nvmefs://`.

Setting the Storage I/O Backend. *nvmefs* implements storage I/O following `xNVMe`'s API (Section 2), hence, supporting all the storage backends `xNVMe` supports. The user provides the choice for the backend together with the storage device path in a configuration file. This file is persisted using DuckDB `SECRET`s manager, following the design of other community extensions. At initialization, the I/O backend choice is read from this configuration information and set by `nvme_opts`, as shown in Section 2. If not given, `xNVMe` chooses a backend based on what is available at runtime.

LBA Mapping. Since some storage I/O backends bypass OS layers, such as I/O Passhtru (`io_uring_cmd`) [20] and `spdk` [46], *nvmefs* directly keeps track of the LBAs assigned to each DuckDB file. As Figure 2 illustrates, each database, WAL, and temporary file has its LBA range set. Since there is one database and WAL file, keeping the start LBA and the current LBA for writes is sufficient. For temporary files, however, the LBA ranges for the many files may be interleaved. Hence, *nvmefs* keeps track of the LBAs assigned to temporary files through a metadata manager, called `TemporaryFileMetadataManager`. It maps each temporary filename to a list of LBA ranges, as shown in Figure 2, where the ranges do not have to be consecutive. A `TemporaryBlockManager` keeps track of which blocks are free or not.

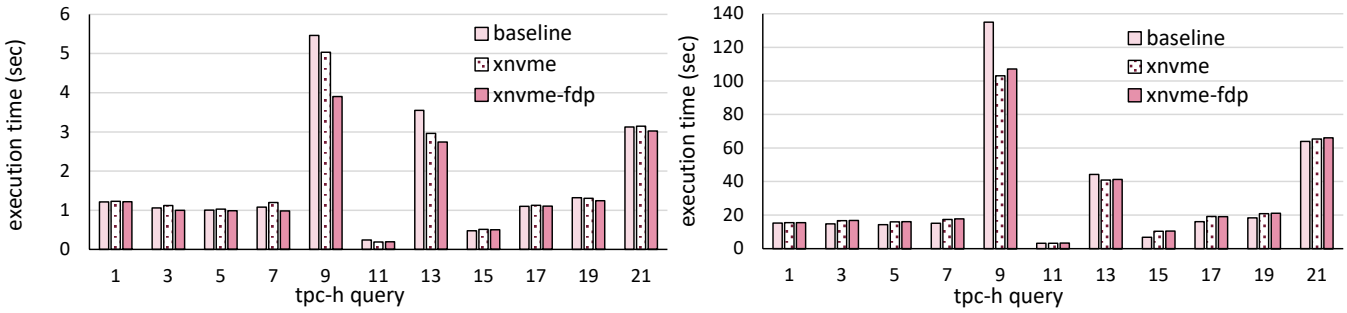
NVMe Queues. I/O backends that use async I/O need NVMe queues to be set (Listing 4). *nvmefs* creates a separate `xnvme_queue` for each DuckDB worker thread. This eliminates the need for synchronization when multiple threads access the same queue and is a common optimization while using NVMe devices [4, 15].

FDP Integration. Flexible Data Placement (FDP) [1] is an emerging SSD technology that is included in the NVMe base specification. With FDP, the host can indirectly influence where data is stored on the SSD to group the data with similar lifetimes onto the same erase blocks and reduce the extra rewrites during garbage collection. This, in turn, results in lower write amplification, which is the ratio between the amount of data actually written to the device vs the data intended to be written by the application.

We implement FDP support using `xNVMe`'s I/O Passthru backend in *nvmefs*. Without `xNVMe`, this implementation has to use the lower-level `io_uring_cmd` API, which is more cumbersome and error-prone compared to relying on the scaffolding and future-compatibility provided by `xNVMe`.

In DuckDB, FDP is used to separate the database data from the temporary data, as the temporary data has the lifetime of a query, while the database data is stored long term. When *nvmefs* opens an `nvme_device`, it checks whether FDP is enabled on this device using `xNVMe`'s *get-feature* API. If it is enabled, then the write command receives different placement identifiers for the temporary and database data. Otherwise, no such identifier is given.

[1] Extensions in DuckDB are compiled as dynamically linked libraries.
[2] Code and examples can be found at https://github.com/itu-rad/nvmefs. We plan to upstream *nvmefs* as a DuckDB community extension.



(a) TPC-H scaling factor 100. **(b) TPC-H scaling factor 1000.**

Figure 3: TPC-H results with DuckDB's `LocalFileSystem` (*baseline*) vs *nvmefs* using `io_uring_cmd` I/O backend with (*xnvme-fdp*) and without (*xnvme*) Flexible Data Placement. Buffer pool is 20 GB and number of threads is 16. Only queries with odd numbers are shown for brevity. Notice the different y-axes across the graphs.

5 EVALUATION

Our evaluation goals are (1) to showcase the flexibility *nvmefs*, hence `xNVMe`, brings to a real-world data management system, and (2) to investigate the performance impact of modern storage I/O techniques such as I/O Passthru, SPDK, and FDP on data analytics.

To achieve our goal, we compare *nvmefs* with different configurations against DuckDB's default `LocalFileSystem`, based on the OS filesystem and *synchronous* I/O. We use `ext4` for the filesystem. Our comparison covers three I/O backends that give a good overview of the landscape: traditional filesystem, I/O Passthru [20] (as close to kernel bypass as possible while still using the Linux kernel infrastructure), and SPDK (user-space NVMe driver) [46].

For this comparison, we use both the standardized analytics benchmark TPC-H [42] and the aggregation benchmark proposed by Kuiper et al. [23] to evaluate DuckDB's performance on aggregation queries when intermediate results must spill to disk.

While already deployed in datacenters, FDP SSDs cannot be purchased separately yet. Thus, we use a server (Table 1) provided by Samsung Memory Research Center with FDP SSDs.

Finally, we use xNVMe 0.7.5, SPDK 22.09, and DuckDB 1.2.0. The buffer pool size and the number of threads for DuckDB in each experiment will be stated in the corresponding sections. Otherwise, the default options are kept for DuckDB.

CPU (2×) - Intel(R) Xeon(R) Gold 6342	
RAM	500 GB
Cores (Threads)	24 (48)
Clock rate (Turbo)	2.8 (3.5) GHz
L1 (D/I) & L2 per core	48 / 32 KiB & 1.3 MiB
L3 shared cache	36 MiB
Operating system	CentOS Stream 9
Kernel (Linux)	6.13.0 (custom)
FDP SSD	
Capacity & Block Size	3.76 TB & 4 KiB

Table 1: Hardware specifications.

5.1 TPC-H

Figure 3 shows the execution times for TPC-H queries when run with DuckDB's `LocalFileSystem`, *baseline*, and *nvmefs* with the I/O Passthru (`io_uring_cmd`). In addition to a regular run with I/O Passthru, *xnvme*, we evaluate our preliminary support for FDP SSDs in DuckDB, *xnvme-fdp*. As Section 4 describes, this support separates the database and the temporary data on the SSD.

We run TPC-H with scaling factor 100 (Figure 3a) and 1000 (Figure 3b), which results in a database size of 26 GB and 265 GB, in DuckDB thanks to DuckDB's compression. For brevity, we show the results only for the queries with odd numbers, while reporting average and maximum values from the entire result set.

We set DuckDB's buffer pool size to 20 GB and the number of threads per query to 16 in these experiments. This means that the database itself does not fit into the buffer pool, and there will be some intermediate query results creating further I/O pressure. For a sanity check, we ran experiments with smaller TPC-H scaling factors, 1 and 10, that can fit into the 20 GB buffer pool. We did not observe significant performance differences across the *baseline* and *nvmefs* runs for these cases.

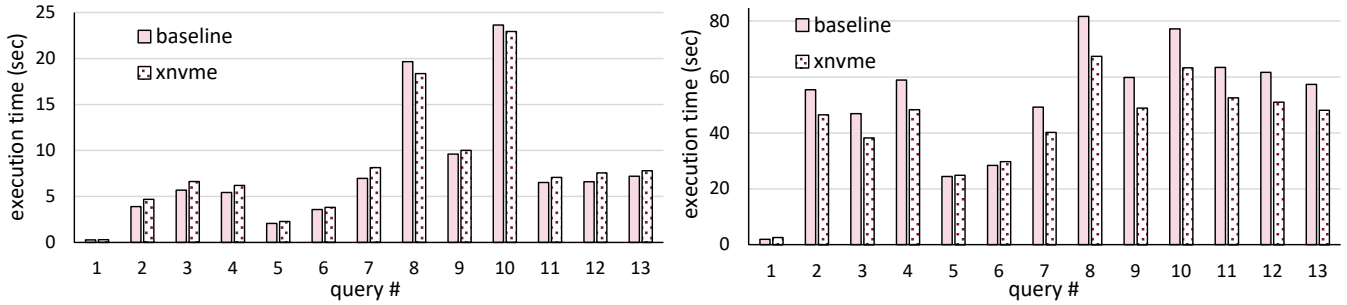
From Figure 3, we see that especially query 9 and 13 benefit from the I/O Passthru backend and FDP. While not shown in Figure 3, the execution time for query 12 gets halved with FDP.

For the other queries, *nvmefs* performs within a ~ 10% variance of the *baseline*. The close performance between *xnvme* and *baseline* for these queries can be explained by the impact of the OS page cache, as also stated by the authors of the I/O Passthru work [20]. While I/O Passthru bypasses the OS layers and provides a more direct I/O path through the kernel, it also bypasses the OS filesystem's data buffering in memory.

Furthermore, all the experiments exhibit a write-amplification of ~1, stemming from the mostly sequential data writing behavior of the analytical workloads we evaluate in this section. Therefore, *xnvme-fdp* performs on-par with the others in most queries.

5.2 Aggregation Benchmark

Figure 4 reports the execution times for the aggregation queries proposed by Kuiper et al. [23], which are performed on the TPC-H's `LineItem` table. The higher the query number, the more results it produces, hence, the more I/O pressure expected. Similar to Figure 3,



(a) Thin: Only select the columns being grouped in the result.

(b) Wide: Select all columns in the result.

Figure 4: Aggregation benchmark [23] with queries performing aggregation on TPC-H `lineitem` table with DuckDB's `LocalFileSystem` (*baseline*) vs *nvmefs* using `io_uring_cmd` I/O backend (*xnvme*). Buffer pool is 20 GB, number of threads is 16, and scaling factor is 128. As we go from left-to-right on the x-axes, the queries produce more tuples in the results. Notice the different y-axes across the graphs.

Figure 4 compares *nvmefs* with `io_uring_cmd` backend, *xnvme*, with DuckDB's `LocalFileSystem`, *baseline*. We omit FDP results, as FDP performs almost the same as the runs with no FDP with *nvmefs*, since the write amplification is also ~1 for this workload.

The aggregation benchmark comes with two variants for each query: thin and wide. The *thin* queries only select the columns being grouped, while the *wide* ones select all columns, creating a heavier memory pressure and higher chances of spilling to disk. We run the benchmark with a scaling factor of 128, which results in a database size of 22 GB, and a buffer pool size of 20 GB. We ran this benchmark with scaling factors (SF) of 32, 8, and 2 as well. While the runs with SF=32 behave similarly to the ones with SF=128, SF=8 and SF=2 do not create enough I/O pressure to observe differences between the *baseline* and *nvmefs*. Finally, the number of threads per query is set to 16 in these experiments as well.

From Figure 4a, we observe that *nvmfs* does not benefit *thin* query variants. These variants do not create as high I/O pressure with SF=128, and the OS page cache can likely handle most of it as discussed in Section 5.1. On the other hand, the *wide* variants benefit from *nvmefs* as shown by Figure 4b, with ~ 20% reduction in query execution times on average.

5.3 SPDK

Finally, Figure 5 demonstrates how *nvmefs*, by being built on *xnvme*, allows DuckDB to utilize `spdk` as an I/O backend as well. We run the same aggregation benchmark from Section 5.2, but with scaling factor 32 that results in a database size of 5.3 GB. To evaluate a scenario with heavier I/O pressure, we set DuckDB buffer pool size to 2 GB and run the wide query variants. The number of threads is set to 1 in this experiment.

Based on Figure 5, we see that *xnvme-spdk* delivers 4-30% improvement for the query execution times over the *baseline*. On the other hand, the `io_uring_cmd` backend, *xnvme*, almost halves the query times for most queries.

While `io_uring_cmd` consistently outperforms `spdk` in Figure 5, we note that a more optimized `spdk` setup in *nvmefs* is possible (e.g., pre-allocating buffers to pass I/O results). Figure 5 simply underlines that for I/O-intensive cases, even the out-of-the-box `spdk` support

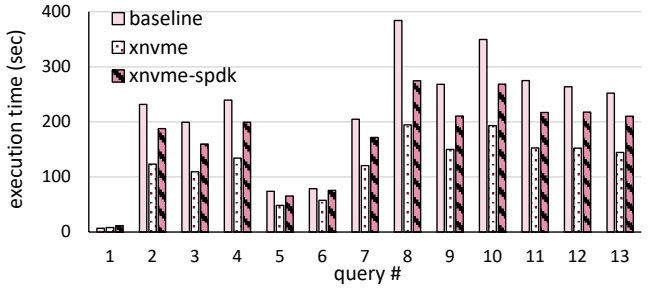


Figure 5: Aggregation benchmark [23] with queries performing aggregation on TPC-H `lineitem` table with DuckDB's `LocalFileSystem` (*baseline*) vs *nvmefs* using `io_uring_cmd` (*xnvme*) and `spdk` (*xnvme-spdk*) as I/O backends. Buffer pool is 2 GB, number of threads is 1, scaling factor is 32, and queries use the *wide* variant. As we go from left-to-right on the x-axis, the queries produce more tuples in the results.

of *xnvme* benefits DuckDB, while releasing the end-users from the configuration hurdles of SPDK [43].

6 ROAD AHEAD

Section 5 shows (1) how integrating `xNVMe` makes the modern storage I/O more accessible for a state-of-the-art database management system such as DuckDB in a non-intrusive way, and (2) how that access translates to higher performance for I/O-intensive cases. There are further avenues to explore to achieve a more holistic integration between the data-intensive systems in general and modern I/O stack. We list some of them in this section.

Optimizing the Integration. While Section 5 reports very promising results for *nvmefs*, our `xNVMe` integration is a preliminary one, and further code optimizations are possible. For example, when it comes to the synchronization points, it is worthwhile to investigate if all is needed and whether some that relies on mutexes can be redesigned to use atomic operations instead. Furthermore, modern I/O paths have many tuning knobs. While `xNVMe` gives a robust default configuration for the I/O options it supports, with further

experimentation it would be possible to achieve more workload- and hardware-aware configurations. As stated in Section 5.3, for example, the SPDK backend can deliver even higher performance through optimizations such as pre-allocated buffers. In addition, experimenting with different FDP placement options can be an avenue for further optimization.

Other Database Systems. This work proposes a way to integrate xNVMe into a database system in addition to demonstrating the benefits of this integration. We hope that this encourages more people to invest in the time to do such integration for other database systems (e.g., PostgreSQL, CockroachDB, LeanStore). We advocate that integration of xNVMe is a more robust and sustainable solution to leverage the capabilities of modern storage instead of creating separate solutions for the different I/O backends or storage devices.

Other Database Workloads. By integrating xNVMe to other systems, we can also test its performance impact on other popular database workloads, such as online transaction processing (OLTP). OLTP workloads have higher random access patterns compared to the analytical workloads Section 5 evaluates. Hence, they tend to exhibit a higher write amplification. We expect, therefore, OLTP workloads and OLTP-optimized database systems would benefit more from FDP SSDs. We are aware that the LeanStore [26] team has done some work to integrate xNVMe into LeanStore. However, we are not aware of the results or a public codebase.

Accelerator-Centric I/O. Efficient and fast storage access from hardware accelerators, especially GPUs, has recently gained attention with the popularity of the AI workloads. This trend has driven demand for direct, low-cost access to storage with minimal CPU overhead [32, 41]. However, recent efforts [5, 39] have produced ad hoc programming models and APIs that not only require familiarity with low-level hardware interfaces, such as PCIe and NVMe, but also tightly couple applications to specific hardware technologies.

While these solutions leverage advanced hardware capabilities effectively, their diversity echoes the fragmentation seen with POSIX storage APIs following the introduction of newer NVMe SSDs. We view this trend as an opportunity for xNVMe to establish a standardized programming model and API for GPU applications, one that manages underlying hardware seamlessly while enabling efficient, direct storage access. We are actively exploring abstractions and implementations that facilitate data exchange between GPUs and modern NVMe storage devices [28, 29].

7 CONCLUSION

Despite the availability of a rich NVMe feature set and diverse I/O backends, most modern database management systems still rely on the traditional OS filesystem abstractions and synchronous I/O. In this work, we propose to leverage the xNVMe framework to offer a more flexible storage I/O backend for database systems. To evaluate this proposal, we integrate xNVMe into DuckDB by adding a new filesystem to DuckDB, *nvmefs*. We demonstrate that with *nvmefs*, DuckDB can now use state-of-the-art I/O backends such as IO Passthru and SPDK in addition to a wider variety of SSD types such as FDP SSDs, without requiring intrusive code changes. Compared to DuckDB's default filesystem (OS filesystem and synchronous I/O backend), *nvmefs* can even halve the execution times of some queries from the standardized TPC-H benchmark and an aggregation benchmark proposed by the DuckDB team. We hope that our experience with *nvmefs* encourages and helps more in the community to integrate xNVMe into their data-intensive systems to leverage modern NVMe landscape.

ACKNOWLEDGMENTS

We thank the Samsung Memory Research Center (SMRC) for providing access to their hardware; Sung-Jun Park for supporting us with uninterrupted access to SMRC; Karl Bonde Torp for the help with xNVMe; Peter Boncz, Hannes Mühleisen, and Mark Raasveldt for the suggestion of using DuckDB extensions for adding alternative I/O options to DuckDB during early discussions; DASYA and RAD group members at IT University of Copenhagen and GOST members at Samsung for their support; and the CIDR reviewers for their constructive feedback.

REFERENCES

[1] Michael Allison, Arun George, Javier González, Dan Helmick, Vikash Kumar, Roshan R. Nair, and Vivek Shah. 2025. Towards Efficient Flash Caches with Emerging NVMe Flexible Data Placement SSDs. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*. ACM, 1142–1160. https://doi.org/10.1145/3689031.3696091

[2] Raja Appuswamy, Goetz Graefe, Renata Borovica-Gajic, and Anastasia Ailamaki. 2019. The Five-Minute Rule 30 Years Later and Its Impact on the Storage Hierarchy. *Commun. ACM* 62, 11 (2019), 114–120. https://doi.org/10.1145/3318163

[3] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 689–703. https://www.usenix.org/conference/atc21/presentation/bjorling

[4] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. 2013. Linux Block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems. In *Proceedings of the 6th International Systems and Storage Conference* (Haifa, Israel) *(SYSTOR '13)*. Association for Computing Machinery, New York, NY, USA, Article 22, 10 pages. https://doi.org/10.1145/2485732.2485740

[5] Chia-Hao Chang, Jihoon Han, Anand Sivasubramaniam, Vikram Sharma Mailthody, Zaid Qureshi, and Wen-Mei Hwu. 2024. GMT: GPU Orchestrated Memory Tiering for the Big Data Era. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (La Jolla, CA, USA) *(ASPLOS '24, Vol. 3)*. Association for Computing Machinery, New York, NY, USA, 464–478. https://doi.org/10.1145/3620666.3651353

[6] Niv Dayan, Philippe Bonnet, and Stratos Idreos. 2016. GeckoFTL: Scalable Flash Translation Techniques For Very Large Flash Devices. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 327–342. https://doi.org/10.1145/2882903.2915219

[7] DuckDB. [n. d.]. An in-process SQL OLAP Database Management System. https://duckdb.org/.

[8] DuckDB. [n. d.]. DuckDB Docs Extensions. https://duckdb.org/docs/stable/extensions/overview.html.

[9] Carl Duffy, Jaehoon Shim, Sang-Hoon Kim, and Jin-Soo Kim. 2023. Dotori: A Key-Value SSD Based KV Store. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1560–1572. https://doi.org/10.14778/3583140.3583167

[10] fio. 2025. xNVMe ioengine Part 1. https://github.com/vincentkfu/fio-blog/wiki/xNVMe-ioengine-Part-1.

[11] FIO. [n. d.]. Flexible I/O Tester. https://fio.readthedocs.io/en/latest/fio_doc.html

[12] Bill Gervasi. 2024. NVMe Over CXL. https://files.futurememorystorage.com/proceedings/2024/20240806_DCTR-102-1_Gervasi.pdf.

[13] Goetz Graefe. 2009. The Five-Minute Rule 20 Years Later (and How Flash Memory Changes the Rules). *Commun. ACM* 52, 7 (2009), 48–59. https://doi.org/10.1145/1538788.1538805

[14] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p16-haas-cidr20.pdf

[15] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, and How to Exploit it: High-Performance I/O for High-Performance Storage Engines.

Proc. VLDB Endow. 16, 9 (May 2023), 2090–2102. https://doi.org/10.14778/3598581.3598584

[16] Niclas Hedam, Morten Tychsen Clausen, Philippe Bonnet, Sangjin Lee, and Ken Friis Larsen. 2023. Delilah: eBPF-offload on Computational Storage. In *Proceedings of the 19th International Workshop on Data Management on New Hardware* (Seattle, WA, USA) *(DaMoN '23)*. Association for Computing Machinery, New York, NY, USA, 70–76. https://doi.org/10.1145/3592980.3595319

[17] Dan Helmick. 2025. Unlocking QoS Potential: Optimizing SSD Performance for Emerging Standards. https://nvmexpress.org/wp-content/uploads/03_Helmick-Unlocking-QoS-Potential_Final.pdf.

[18] Kaisong Huang, Darien Imai, Tianzheng Wang, and Dong Xie. 2022. SSDs Striking Back: The Storage Jungle and Its Implications to Persistent Indexes. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org. https://www.cidrdb.org/cidr2022/papers/p64-huang.pdf

[19] Matthias Jasny, Muhammad El-Hindi, Tobias Ziegler, and Carsten Binnig. 2025. A Wake-Up Call for Kernel-Bypass on Modern Hardware. In *Proceedings of the 21st International Workshop on Data Management on New Hardware, DaMoN 2025, Berlin, Germany, June 22-27, 2025*. ACM, 14:1–14:5. https://doi.org/10.1145/3736227.3736235

[20] Kanchan Joshi, Anuj Gupta, Javier González, Ankit Kumar, Krishna Kanth Reddy, Arun George, Simon Andreas Frimann Lund, and Jens Axboe. 2024. I/O Passthru: Upstreaming a flexible and efficient I/O Path in Linux. In *22nd USENIX Conference on File and Storage Technologies, FAST 2024, Santa Clara, CA, USA, February 27-29, 2024*, Xiaosong Ma and Youjip Won (Eds.). USENIX Association, 107–121. https://www.usenix.org/conference/fast24/presentation/joshi

[21] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. NVMeDirect: A User-Space I/O Framework for Application-Specific Optimization on NVMe SSDs. In *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems* (Denver, CO) *(HotStorage'16)*. USENIX Association, USA, 41–45.

[22] Sang-Hoon Kim, Jaehoon Shim, Euidong Lee, Seongyeop Jeong, Ilkueon Kang, and Jin-Soo Kim. 2023. NVMeVirt: A Versatile Software-Defined Virtual NVMe Device. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST'23)*. USENIX Association, USA, Article 24, 15 pages.

[23] Laurens Kuiper, Peter Boncz, and Hannes Mühleisen. 2024. Robust External Hash Aggregation in the Solid State Age. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 3753–3766. https://doi.org/10.1109/ICDE60146.2024.00288

[24] Laurens Kuiper, Mark Raasveldt, and Hannes Mühleisen. 2021. Efficient External Sorting in DuckDB. In *Proceedings of the The British International Conference on Databases 2021, London, United Kingdom, March 28, 2022 (CEUR Workshop Proceedings, Vol. 3163)*, Holger Pirk and Thomas Heinis (Eds.). CEUR-WS.org, 40–45. https://ceur-ws.org/Vol-3163/BICOD21_paper_9.pdf

[25] Bohyun Lee, Mijin An, and Sang-Won Lee. 2023. LRU-C: Parallelizing Database I/Os for Flash SSDs. *Proc. VLDB Endow.* 16, 9 (May 2023), 2364–2376. https://doi.org/10.14778/3598581.3598605

[26] Viktor Leis. 2024. LeanStore: A High-Performance Storage Engine for NVMe SSDs. *Proc. VLDB Endow.* 17, 12 (2024), 4536–4545. https://doi.org/10.14778/3685800.3685915

[27] Alberto Lerner and Philippe Bonnet. 2025. *Principles of Database and Solid-State Drive Co-Design*. Springer Nature.

[28] Simon A.F. Lund. 2025. Accelerator-integrated Storage I/O. Appeared at Open Compute Project Global Summit 2025, available at https://xnvme.io/_static/ocp25.pdf.

[29] Simon A.F. Lund. 2025. Feeding the Beast: Bridging NVMe Storage and GPUs while Preserving File Semantics. Appeared at FMS: the Future of Memory and Storage 2025, available at https://xnvme.io/_static/fms25.pdf.

[30] Simon Andreas Frimann Lund, Philippe Bonnet, Klaus B. A. Jensen, and Javier Gonzalez. 2022. I/O interface independence with xNVMe. In *SYSTOR '22: The 15th ACM International Systems and Storage Conference, Haifa, Israel, June 13 - 15, 2022*, Michal Malka, Hillel Kolodner, Frank Bellosa, and Moshe Gabel (Eds.). ACM, 108–119. https://doi.org/10.1145/3534056.3534936

[31] Jigao Luo, Nils Boeschen, Tobias Ziegler, and Carsten Binnig. 2025. An Evaluation of NVMe-over-Fabrics for Disaggregated Databases over Fast Networks. In *Datenbanksysteme für Business, Technologie und Web (BTW 2025), 21. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 03.-07, März 2025, Bamberg, Germany, Proceedings*. 113–125.

[32] Jonas Markussen, Lars Bjørlykke Kristiansen, Pål Halvorsen, Halvor Kielland-Gyrud, Håkon Kvale Stensland, and Carsten Griwodz. 2021. SmartIO: Zero-overhead Device Sharing through PCIe Networking. *ACM Trans. Comput. Syst.* 38, 1–2, Article 2 (July 2021), 78 pages. https://doi.org/10.1145/3462545

[33] Bill Martin and Jasson Molgaard. 2025. NVM Express® (NVMe®) Technology Support for CXL®: Unleashing Computational Workloads. https://nvmexpress.org/nvm-express-nvme-technology-support-for-cxl-unleashing-computational-workloads/.

[34] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*.

[35] NVMe. [n. d.]. Non-Volatile Memory Express. https://nvmexpress.org/.

[36] Marius Ottosen, Magnus Keinicke Parlo, and Philippe Bonnet. 2025. DuckDB on xNVMe. arXiv:2512.01490 [cs.DB] https://arxiv.org/abs/2512.01490

[37] Tarikul Islam Papon and Manos Athanassoulis. 2023. ACEing the Bufferpool Management Paradigm for Modern Storage Devices. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 1326–1339. https://doi.org/10.1109/ICDE55515.2023.00106

[38] Ivan Luiz Picoli, Philippe Bonnet, and Pinar Tözün. 2019. LSM Management on Computational Storage. In *Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN 2019, Amsterdam, The Netherlands, 1 July 2019*, Thomas Neumann and Ken Salem (Eds.). ACM, 17:1–17:3. https://doi.org/10.1145/3329785.3329927

[39] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, C. J. Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wen-mei Hwu. 2023. GPU-Initiated On-Demand High-Throughput Storage Access in the BaM System Architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Vancouver, BC, Canada) *(ASPLOS '23, Vol. 2)*. Association for Computing Machinery, New York, NY, USA, 325–339. https://doi.org/10.1145/3575693.3575748

[40] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

[41] Karl B. Torp, Simon A. F. Lund, and Pinar Tözün. 2025. Path to GPU-Initiated I/O for Data-Intensive Systems. In *Proceedings of the 21st International Workshop on Data Management on New Hardware, DaMoN 2025, Berlin, Germany, June 22-27, 2025*. ACM, 3:1–3:9. https://doi.org/10.1145/3736227.3736232

[42] Transaction Processing Performance Council (TPC). 2022. TPC Benchmark™ H. http://www.tpc.org/tpch/

[43] Lukas Vogel, Daniel Ritter, Danica Porobic, Pinar Tözün, Tianzheng Wang, and Alberto Lerner. 2023. Data Pipes: Declarative Control over Data Movement. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p55-vogel.pdf

[44] xNVMe. 2025. xNVMe API. https://xnvme.io/api/index.html#sec-api.

[45] xNVMe. [n. d.]. Cross-platform libraries and tools for NVMe devices. https://xnvme.io/.

[46] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. 2017. SPDK: A Development Kit to Build High Performance Storage Applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 154–161. https://doi.org/10.1109/CloudCom.2017.14 ISSN: 2330-2186.

[47] Xiangqun Zhang, Janki Bhimani, Shuyi Pei, Eunji Lee, Sungjin Lee, Yoon Jae Seong, Eui Jin Kim, Changho Choi, Eyee Hyun Nam, Jongmoo Choi, and Bryan S. Kim. 2025. Storage Abstractions for SSDs: The Past, Present, and Future. *ACM Trans. Storage* 21, 1 (2025), 2:1–2:44.