Are You Sure You Want to Use MMAP in Your Database Management System?

Andrew Crotty
Carnegie Mellon University
andrewcr@cs.cmu.edu

Viktor Leis
University of Erlangen-Nuremberg
viktor.leis@fau.de

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

ABSTRACT

Memory-mapped (mmap) file I/O is an OS-provided feature that maps the contents of a file on secondary storage into a program's address space. The program then accesses pages via pointers as if the file resided entirely in memory. The OS transparently loads pages only when the program references them and automatically evicts pages if memory fills up.

mmap's perceived ease of use has seduced database management system (DBMS) developers for decades as a viable alternative to implementing a buffer pool. There are, however, severe correctness and performance issues with mmap that are not immediately apparent. Such problems make it difficult, if not impossible, to use mmap correctly and efficiently in a modern DBMS. In fact, several popular DBMSs initially used mmap to support larger-than-memory databases but soon encountered these hidden perils, forcing them to switch to managing file I/O themselves after significant engineering costs. In this way, mmap and DBMSs are like coffee and spicy food: an unfortunate combination that becomes obvious after the fact.

Since developers keep trying to use mmap in new DBMSs, we wrote this paper to provide a warning to others that mmap is not a suitable replacement for a traditional buffer pool. We discuss the main shortcomings of mmap in detail, and our experimental analysis demonstrates clear performance limitations. Based on these findings, we conclude with a prescription for when DBMS developers *might* consider using mmap for file I/O.

1 INTRODUCTION

An important feature of disk-based DBMSs is their ability to support databases that are larger than the available physical memory. This functionality allows a user to query a database as if it resides entirely in memory, even if it does not fit all at once. DBMSs achieve this illusion by reading pages of data from secondary storage (e.g., HDD, SSD) into memory on demand. If there is not enough memory for a new page, the DBMS will evict an existing page that is no longer needed in order to make room.

Traditionally, DBMSs implement the movement of pages between secondary storage and memory in a buffer pool, which interacts with secondary storage using system calls like `read` and `write`. These file I/O mechanisms copy data to and from a buffer in user space, with the DBMS maintaining complete control over how and when it transfers pages.

Alternatively, the DBMS can relinquish the responsibility of data movement to the OS, which maintains its own file mapping and page cache. The POSIX mmap system call maps a file on secondary storage into the virtual address space of the caller (i.e., the DBMS), and the OS will then load pages lazily when the DBMS accesses them. To the DBMS, the database appears to reside fully in memory, but the OS handles all necessary paging behind the scenes rather than the DBMS's buffer pool.

On the surface, mmap seems like an attractive implementation option for managing file I/O in a DBMS. The most notable benefits are ease of use and low engineering cost. The DBMS no longer needs to track which pages are in memory, nor does it need to track how often pages are accessed or which pages are dirty. Instead, the DBMS can simply access disk-resident data via pointers as if it were accessing data in memory while leaving all low-level page management to the OS. If the available memory fills up, then the OS will free space for new pages by transparently evicting (ideally unneeded) pages from the page cache.

From a performance perspective, mmap should also have much lower overhead than a traditional buffer pool. Specifically, mmap does not incur the cost of explicit system calls (i.e., `read`/`write`) and avoids redundant copying to a buffer in user space because the DBMS can access pages directly from the OS page cache.

Since the early 1980s, these supposed benefits have enticed DBMS developers to forgo implementing a buffer pool and instead rely on the OS to manage file I/O [36]. In fact, the developers of several well-known DBMSs (see Section 2.3) have gone down this path, with some even touting mmap as a key factor in achieving good performance [20].

Unfortunately, mmap has a hidden dark side with many sordid problems that make it undesirable for file I/O in a DBMS. As we describe in this paper, these problems involve both data safety and system performance concerns. We contend that the engineering steps required to overcome them negate the purported simplicity of working with mmap. For these reasons, we believe that mmap adds too much complexity with no commensurate performance benefit and strongly urge DBMS developers to avoid using mmap as a replacement for a traditional buffer pool.

The remainder of this paper is organized as follows. We begin with a short background on mmap (Section 2), followed by a discussion of its main problems (Section 3) and our experimental analysis (Section 4). We then discuss related work (Section 5) and conclude with a summary of our guidance for when you *might* consider using mmap in your DBMS (Section 6).

2 BACKGROUND

This section provides the relevant background on mmap. We begin with a high-level overview of memory-mapped file I/O and the POSIX mmap API. Then, we discuss real-world implementations of mmap-based systems.

 12th Annual Conference on Innovative Data Systems Research (CIDR '22). January 9-12, 2022, Chaminade, USA.

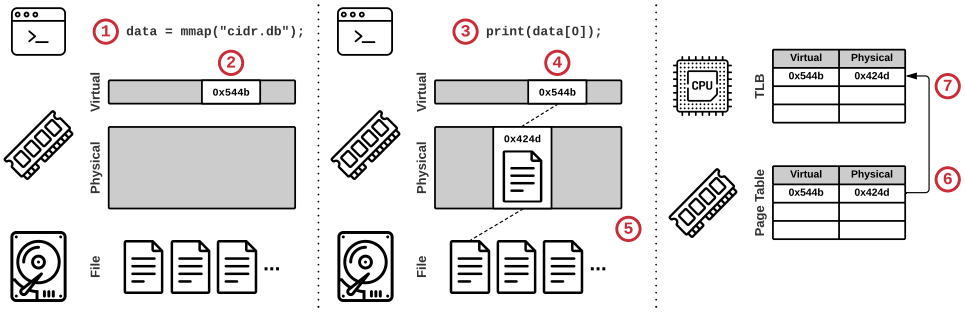


Figure 1: Step-by-step illustration of how a program accesses a file using `mmap`.

DBMS	MMAP Use	Details
MonetDB	2002–	[12, 21]
MongoDB	2009–2019	[14, 3]
LevelDB	2011–	[5]
LMDB	2011–	[20]
SQLite	2013–	[7]
SingleStore	2013–2015	[32]
QuestDB	2014–	[34]
RavenDB	2014–	[4]
InfluxDB	2015–2020	[8, 1]
WiredTiger	2020–	[17]

Table 1: Modern `mmap`-based DBMSs.

2.1 MMAP Overview

Figure 1 shows a step-by-step overview of how to access a file ("`cidr.db`") with `mmap`. ① A program calls `mmap` and receives a pointer to the memory-mapped file contents. ② The OS reserves part of the program's virtual address space but does not load any part of the file. ③ The program accesses the file's contents using the pointer. ④ The OS attempts to retrieve the page. ⑤ Since no valid mapping exists for the specified virtual address, the OS triggers a page fault to load the referenced part of the file from secondary storage into a physical memory page. ⑥ The OS adds an entry to the page table that maps the virtual address to the new physical address. ⑦ The initiating CPU core also caches this entry in its local translation lookaside buffer (TLB) to accelerate future accesses.

As the program accesses other pages, the OS will load them into memory, evicting pages as needed if the page cache fills up. When evicting pages, the OS also removes their mappings from both the page table and each CPU core's TLB. Flushing the local TLB of the initiating core is straightforward, but the OS must ensure that no stale entries remain in the TLBs of remote cores. Since current CPUs do not provide coherence for remote TLBs, the OS has to issue an expensive inter-processor interrupt to flush them, which is called a TLB shootdown [11]. As our experiments show (Section 4), TLB shootdowns can have a significant performance impact.

2.2 POSIX API

We now review the most important POSIX system calls for memory-mapped file I/O and describe how a DBMS can use them in place of a traditional buffer pool.

`mmap`: As previously explained, this call causes the OS to map a file into the DBMS's virtual address space. The DBMS can then read or write file contents using ordinary memory operations. The OS caches pages in memory and, when using the `MAP_SHARED` flag, will (eventually) write any changes back to the underlying file. Alternatively, the `MAP_PRIVATE` flag will create a copy-on-write mapping that is only accessible to the caller (i.e., changes are not persisted to the backing file).

`madvise`: This call allows the DBMS to provide hints to the OS about expected data access patterns, either at the granularity of the entire file or for specific page ranges. We focus on three common hints: `MADV_NORMAL`, `MADV_RANDOM`, and `MADV_SEQUENTIAL`. When a page fault occurs in Linux with the default `MADV_NORMAL` hint, the OS will fetch the accessed page, as well as the next 16 and previous 15 pages. With 4 KB pages, `MADV_NORMAL` causes the OS to read 128 KB from secondary storage, even though the caller only requested a single page. Depending on the workload, this prefetching might help or hurt the DBMS's performance. For example, the `MADV_RANDOM` mode, which reads only the necessary page, is a better choice for larger-than-memory OLTP workloads, whereas `MADV_SEQUENTIAL` is better for OLAP workloads with sequential scans.

`mlock`: This call allows the DBMS to pin pages in memory, ensuring that the OS will never evict them. However, according to the POSIX standard (and Linux's implementation), the OS is permitted to flush dirty pages to the backing file at any time, even if the page is pinned. Therefore, the DBMS cannot use `mlock` to ensure that dirty pages will never be written to secondary storage, which has serious implications for transactional safety.

`msync`: Lastly, this call explicitly flushes the specified memory range to secondary storage. Without `msync`, the DBMS has no other way to guarantee that updates are persisted to the backing file.

2.3 MMAP Gone Wrong

The allure of the OS-managed buffer pool for DBMSs has existed for decades [36], with QuickStore [40] and Dalí [22] being early examples of `mmap`-based systems from the 1990s. Today, several DBMSs continue to use `mmap` for file I/O, as shown in Table 1. For instance, MonetDB stores individual columns as memory-mapped files [12, 21], and SQLite provides an option to use `mmap` rather than the default `read`/`write` system calls [7]. LMDB relies on `mmap` exclusively, and the developers even cite it as a main contributing factor to the system's performance [20]. Other systems with `mmap`-based storage engines include QuestDB [34] and RavenDB [4].

Despite these apparent success stories, many other DBMSs have tried—and failed—to replace a traditional buffer pool with `mmap`-based file I/O. In the following, we recount some cautionary tales to illustrate how using `mmap` in your DBMS can go horribly wrong.

MongoDB is arguably the most well-known DBMS to have used `mmap` for file I/O. Our understanding from the developers is that they chose to base the original storage engine (MMAPv1) on `mmap` out of expediency as an early-stage startup. However, the design had a number of drawbacks, including an overly complex copying scheme to ensure correctness and the inability to perform any compression for data on secondary storage. For the latter, since the OS

managed the file mapping, the in-memory data layout needed to match the physical representation on secondary storage, leading to wasted space and reduced I/O throughput. With the introduction of WiredTiger as the default storage engine in 2015, MongoDB deprecated MMAPv1 and then completely removed it in 2019 [3]. In 2020, though, MongoDB reintroduced `mmap` as an option in WiredTiger, but it is used in a limited fashion to avoid boundary-crossing penalties between user space and the OS [17].

InfluxDB is a time series DBMS that used `mmap` for file I/O in earlier releases [8]. However, the developers replaced `mmap` after observing I/O spikes for writes when a database grew larger than a few GB in size, likely due to the overheads associated with page eviction (Section 3.4). They also faced other issues when running in containerized environments or on machines without direct-attached storage (e.g., cloud deployments), which further precluded the use of `mmap` in their new IOx storage engine [1].

SingleStore removed `mmap`-based file I/O after encountering poor performance on simple sequential scan queries [32]. The DBMS's calls to `mmap` were taking 10–20 ms per query, which accounted for nearly half of the overall query runtime. Upon further investigation, the developers identified the source of the problem as contention on a shared `mmap` write lock. By switching to `read` system calls, the queries became fully CPU-bound.

A number of other systems ruled out `mmap` early in their development. For example, Facebook created RocksDB as a fork of Google's LevelDB partly due to performance bottlenecks for reads caused by the latter's use of `mmap` [5]. TileDB found that `mmap` was more expensive than `read` system calls for SSDs [27], which we also observed in our experimental analysis (Section 4). Scylla, a distributed NoSQL DBMS, evaluated several alternatives for file I/O and rejected `mmap` due to loss of fine-grained control, both in terms of the page eviction strategy and I/O operation scheduling [23]. The time series DBMS VictoriaMetrics identified problems with `mmap`'s blocking I/O for page faults [37]. RDF-3X abandoned its original `mmap`-based engine due to incompatibility between the Windows and POSIX implementations of memory-mapped file I/O [26].

3 PROBLEMS WITH MMAP

Ostensibly, `mmap` seems like a great idea—the DBMS no longer needs to manage its own buffer pool, as it cedes this responsibility to the OS. By removing the components that deal with explicit file I/O, DBMS developers are free to focus on other aspects of the system. However, transparent paging actually introduces several serious problems for DBMSs, which we discuss in the following.

3.1 Problem #1: Transactional Safety

The challenges inherent with guaranteeing transactional safety of modified pages in `mmap`-based DBMSs are well-known [22, 18]. The core issue is that, due to transparent paging, the OS can flush a dirty page to secondary storage at any time, irrespective of whether the writing transaction has committed. The DBMS cannot prevent these flushes and receives no warning when they occur.

`mmap`-based DBMSs must therefore employ complicated protocols to ensure that transparent paging does not violate transactional safety guarantees. We classify methods for handling updates into three categories: (1) OS copy-on-write, (2) user space copy-on-write, and (3) shadow paging. To simplify our explanations, we assume that the DBMS stores the database in a single file.

OS Copy-On-Write: The idea behind this approach is to create two copies of the database file with `mmap`, both of which will initially point to the same physical pages. The first serves as the primary copy, while the second acts as a private workspace where transactions can stage updates. Importantly, the DBMS creates the private workspace using `mmap`'s `MAP_PRIVATE` flag to enable the OS's copy-on-write feature for pages. To the best of our knowledge, only MongoDB's MMAPv1 storage engine used this method.

To perform an update, the DBMS modifies the affected pages in the private workspace. The OS will transparently copy the contents to new physical pages, remap the virtual memory addresses to these copies, and then apply the changes. The primary copy does not see these changes, and the OS will not persist them to the database file. Therefore, to provide durability, the DBMS must use a write-ahead log (WAL) to record changes. When a transaction commits, the DBMS flushes the corresponding WAL records to secondary storage and uses a separate background thread to apply the committed changes to the primary copy.

Maintaining separate copies of updated pages causes two main problems. First, the DBMS must ensure that the latest updates from committed transactions have propagated to the primary copy before allowing conflicting transactions to run, which requires additional bookkeeping to track pages with pending updates. Second, the private workspace will continue to grow as more updates occur, and the DBMS may eventually end up with two full copies of the database in memory. To solve this second problem, the DBMS can periodically shrink the private workspace using the `mremap` system call. However, the DBMS must again ensure that all pending updates have propagated to the primary copy before destroying the private workspace. Moreover, to avoid losing updates during `mremap`, the DBMS needs to block pending changes until the OS completes the compaction of the private workspace.

User Space Copy-On-Write: Unlike OS copy-on-write, this approach involves manually copying the affected pages from `mmap`-backed memory to a separately maintained buffer in user space. SQLite, MonetDB, and RavenDB all use some variant of this method.

To perform updates, the DBMS applies the changes only to the copies and creates the corresponding WAL records. The DBMS can commit these changes by writing the WAL to secondary storage, at which point it can then safely copy the modified pages back to the `mmap`-backed memory. Since copying an entire page is wasteful for small changes, some DBMSs support applying WAL records directly to the `mmap`-backed memory.

Shadow Paging: LMDB is the most prominent proponent of this approach, which is based on System R's shadow paging design [13]. With shadow paging, the DBMS maintains separate primary and shadow copies of the database, both of which are backed by `mmap`. To perform an update, the DBMS first copies the affected pages from the primary to the shadow copy, where it then applies the necessary changes. Committing the changes involves flushing the modified shadow pages to secondary storage with `msync`, followed by updating a pointer to install the shadow copy as the new primary. The original primary then serves as the new shadow copy.

Although this approach is seemingly uncomplicated to implement, the DBMS must ensure that transactions do not conflict or see partial updates. For example, LMDB solves this problem by allowing only a single writer.

3.2 Problem #2: I/O Stalls

With a traditional buffer pool, a DBMS can use asynchronous I/O (e.g., `libaio`, `io_uring`) to avoid blocking threads during query execution. For instance, consider a common access pattern like a leaf node scan in a B+tree. The DBMS could asynchronously issue the read requests for these potentially non-contiguous pages to mask latency, but `mmap` does not support asynchronous reads.

Furthermore, since the OS can transparently evict pages to secondary storage, read-only queries can unknowingly trigger blocking page faults if they attempt to access evicted pages. In other words, accessing *any* page could result in an unexpected I/O stall because the DBMS cannot know whether the page is in memory.

To avoid these problems, the DBMS developers could implement a workaround using the system calls described in Section 2.2. The most obvious option is to use `mlock` to pin pages that the DBMS expects to access again in the near future. Unfortunately, the OS usually restricts the amount of memory that an individual process can lock, since pinning too many pages can cause problems for concurrently running processes and even the OS itself. The DBMS would also need to carefully track and unpin pages that are no longer being used so that the OS can evict them.

Another potential solution is to use `madvise` to provide hints to the OS about the expected access patterns of queries. For example, a DBMS that needs to execute a sequential scan could supply the `MADV_SEQUENTIAL` flag to `madvise`, which tells the OS to evict pages after they have been read and prefetch subsequent contiguous pages that will be accessed next. This approach is much less involved than using `mlock`, but it also offers significantly less control, since the flags are merely hints that the OS is free to ignore. Additionally, providing the OS with the wrong hint (e.g., `MADV_SEQUENTIAL` when the access pattern is random) can have dire implications for performance, as we show in our experiments (Section 4).

Yet another possibility is to spawn additional threads to prefetch (i.e., attempt to access) pages so that they will block in the event of a page fault rather than the main thread. However, although these solutions may (partially) resolve some issues, they all introduce significant additional complexity, which defeats the purpose of using `mmap` in the first place.

3.3 Problem #3: Error Handling

One core responsibility of a DBMS is to ensure data integrity, so error handling is of paramount importance. For example, some DBMSs (e.g., SQL Server [6]) maintain page-level checksums to detect data corruption during file I/O. When reading a page from secondary storage, the DBMS validates the page contents against the checksum stored in the header. However, with `mmap`, the DBMS would need to validate the checksum on every page access, as the OS may have transparently evicted the page at some point since the previous access.

Similarly, many DBMSs (including several mentioned in Section 2.3) are written in memory-unsafe languages, which means that pointer errors might corrupt pages in memory. A defensively coded buffer pool implementation could check these pages for errors before writing them to secondary storage, but `mmap` will silently persist corrupted pages to the backing file.

Lastly, gracefully handling I/O errors becomes much more difficult when working with `mmap`. Whereas a traditional buffer pool would allow developers to contain I/O error handling within a single module, any code that interacts with `mmap`-backed memory can now produce a `SIGBUS` that the DBMS must deal with via cumbersome signal handlers.

3.4 Problem #4: Performance Issues

The largest and most significant drawback of `mmap`'s transparent paging relates to performance. Although DBMS developers could conceivably overcome the other issues through careful implementation, we believe that `mmap` has serious bottlenecks that cannot be avoided without an OS-level redesign.

The conventional wisdom [28, 23, 29, 16, 17, 30] holds that `mmap` should outperform traditional file I/O because it avoids two major sources of overhead. First, `mmap` circumvents the cost of explicit `read`/`write` system calls because the OS handles file mappings and page faults behind the scenes. Second, `mmap` can return pointers to pages stored in the OS page cache, thereby avoiding an extra copy into a buffer allocated in user space. As an added bonus, `mmap`-based file I/O also results in lower total memory consumption, as the data is not unnecessarily duplicated in user space.

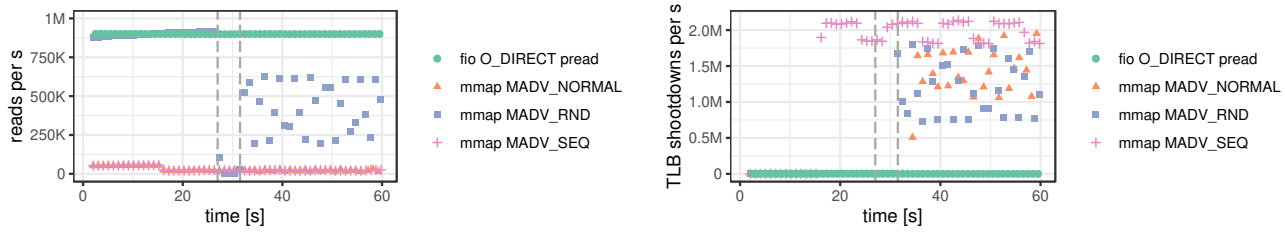
Given these advantages, one would expect that the performance gap between `mmap` and traditional file I/O methods should continue to widen with the emerging availability of better flash storage (e.g., PCIe 5.0 NVMe) that will provide bandwidth comparable to memory [19]. Surprisingly, we have found that the OS's page eviction mechanisms cannot scale beyond a few threads for larger-than-memory DBMS workloads on high-bandwidth secondary storage devices. We believe that one of the main reasons these performance issues have gone largely unnoticed is due to historically limited file I/O bandwidth.

Specifically, we have identified three key bottlenecks that plague `mmap`-based file I/O: (1) page table contention, (2) single-threaded page eviction, and (3) TLB shootdowns. Relatively straightforward adjustments to the OS could partially mitigate the first two issues, but TLB shootdowns present a much trickier problem.

Recall from Section 2.1 that TLB shootdowns occur during page eviction when a core needs to invalidate mappings in a remote TLB. Whereas flushing the local TLB is inexpensive, issuing interprocessor interrupts to synchronize remote TLBs can take thousands of cycles [39]. Workarounds to this problem involve either proposed microarchitectural changes [39] or extensive modification of OS internals [15, 9, 10].

4 EXPERIMENTAL ANALYSIS

As the previous section explained, some of `mmap`'s problems can be overcome through careful implementation, but we argue that its inherent performance limitations cannot be resolved without significant OS-level rewrites. In this section, we present our experimental analysis that empirically demonstrates these issues.



(a) Bandwidth **(b)** TLB Shootdowns

Figure 2: Random Reads – 1 SSD (100 threads)

We ran all experiments on a single-socket machine with an AMD EPYC 7713 processor (64 cores, 128 hardware threads) and 512 GB RAM, of which 100 GB was available to Linux (v5.11) for its page cache. For persistent storage, the machine had 10 × 3.8 TB Samsung PM1733 SSDs (rated with 7000 MB/s read and 3800 MB/s write throughput). We accessed the SSDs as block devices to avoid potential file system overhead [19].

As a baseline, we used the `fio` [2] storage benchmarking tool (v3.25) with direct I/O (`O_DIRECT`) to bypass the OS page cache. Our analysis focused exclusively on read-only workloads, which represent the best-case scenario for `mmap`-based DBMSs; otherwise, they would need to implement complex update protections (Section 3.1) that incur substantial additional overhead [30]. Specifically, we evaluated two common access patterns: (1) random reads and (2) sequential scan.

4.1 Random Reads

For the first experiment, we used a random access pattern over a 2 TB SSD range to simulate a larger-than-memory OLTP workload. Since the page cache had only 100 GB of memory, 95% of all accesses resulted in page faults (i.e., the workload was I/O-bound).

Figure 2a shows the number of random reads per second with 100 threads. Our `fio` baseline exhibited stable performance and achieved close to 900K reads per second, which is in line with the expected performance from 100 outstanding I/O operations and NVMe latency of roughly 100 μs. In other words, this result demonstrates that `fio` can fully saturate the NVMe SSD.

`mmap`, on the other hand, performed significantly worse, even when using a hint that matched the workload's access pattern. We observed three distinct phases for `MADV_RANDOM` in our experiment. `mmap` was initially similar to `fio` during the first 27 seconds, then dropped to nearly zero for about five seconds, and finally recovered to approximately half of `fio`'s performance. This sudden drop in performance occurred when the page cache filled up, forcing the OS to begin evicting pages from memory. Unsurprisingly, the other access pattern hints exhibited much worse performance.

In Section 3.4, we enumerated three key sources of page eviction overhead. The first issue is TLB shootdowns, which we measured using `/proc/interrupt` and show in Figure 2b. As mentioned, TLB shootdowns are expensive (i.e., thousands of cycles [39]), as they involve sending an inter-processor interrupt to flush the TLB of every core. Second, the OS uses only a single process (`kswapd`) for page eviction, which was CPU-bound in our experiments. Finally, the OS must synchronize the page table, which becomes highly contended with many concurrent threads.

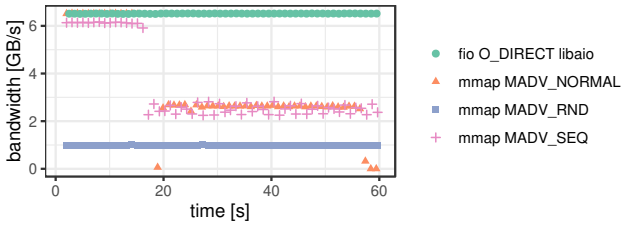


Figure 3: Sequential Scan – 1 SSD (`mmap`: 20 threads; `fio`: `libaio`, 1 thread, `iodepth` 256)

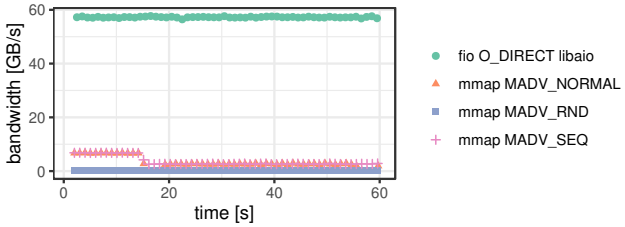


Figure 4: Sequential Scan – 10 SSDs (`mmap`: 20 threads; `fio`: `libaio`, 4 threads, `iodepth` 256)

4.2 Sequential Scan

Sequential scans are another common access pattern for DBMSs, particularly in OLAP workloads. Therefore, we also compared the scan performance of `fio` and `mmap` over a 2 TB SSD range. We first ran our experiments using only a single SSD, and then we reran the same workload using 10 SSDs with software RAID 0 (`md`).

The results in Figure 3 show that `fio` can utilize the full bandwidth of one SSD while maintaining stable performance. Like in the previous experiment, `mmap`'s performance was initially similar to `fio`, but we again observed a precipitous drop in performance once the page cache filled up after about 17 seconds. Additionally, as expected for this workload, the `MADV_NORMAL` and `MADV_SEQUENTIAL` flags performed better than `MADV_RANDOM`.

Figure 4, which shows the results of repeating the sequential scan experiment with 10 SSDs, further accentuates the gap between what modern flash storage can theoretically provide versus what `mmap` can achieve. We observed a roughly 20× performance difference between `fio` and `mmap`, with `mmap` showing virtually no improvement over the results from using one SSD.

In summary, we found that `mmap` performs well only on a single SSD during the initial loading phase. Once page eviction begins or when using multiple SSDs, `mmap` is 2–20× worse than `fio`. With the imminent release of PCIe 5.0 NVMe, which is expected to double

bandwidth per SSD, our results show that mmap cannot match the performance of traditional file I/O for sequential scans.

5 RELATED WORK

To the best of our knowledge, there has been no thorough study of the issues related to using mmap-based file I/O in a modern DBMS. In the following, we describe some of the prior research work that has examined different aspects of mmap.

Given the problems with ensuring transactional safety when using mmap, one line of work introduced a new failure-atomic msync system call [31, 38]. Normally, if the system crashes during a call to msync, the DBMS has no way of knowing which pages were successfully written to secondary storage. Failure-atomic msync offers the same API as msync but ensures that all pages involved are written atomically. As a side effect of the implementation, failure-atomic msync disables the OS's ability to transparently evict pages, which eliminates the need for many of the safety mechanisms we described in Section 3.1.

Tucana [28] and Kreon [29] are experimental key-value DBMSs built around mmap-based file I/O. However, they both note several core issues with mmap (e.g., loss of fine-grained control over I/O scheduling), which prompted Kreon to implement its own custom system call (kmmap). These systems also had to incorporate complex copy-on-write schemes to ensure transactional consistency.

Other research projects have creatively used mmap in ways beyond a buffer pool replacement. For instance, one project leveraged the OS's virtual paging mechanism via mmap as a low-overhead way to migrate cold data to secondary storage [35]. RUMA utilized mmap for "rewiring" page mappings to perform various operations (e.g., sorting) without physically copying data [33].

Lastly, rather than relying on mmap to avoid page mapping overhead, several recent approaches [18, 24, 25] have instead advocated for pointer swizzling. As we have argued throughout this paper, we believe that these lightweight buffer management techniques are the right approach because they can offer similar performance to mmap with none of the downsides.

6 CONCLUSION

This paper made the case against the use of mmap for file I/O in a DBMS. Despite the apparent benefits, we have presented the main drawbacks of mmap, and our experimental analysis confirms our findings related to performance limitations. To conclude, we offer DBMS developers the following advice.

When you should **not** use mmap in your DBMS:

- You need to perform updates in a transactionally safe fashion.
- You want to handle page faults without blocking on slow I/O or need explicit control over what data is in memory.
- You care about error handling and need to return correct results.
- You require high throughput on fast persistent storage devices.

When you should **maybe** use mmap in your DBMS:

- Your working set (or the entire database) fits in memory and the workload is read-only.
- You need to rush a product to the market and do not care about data consistency or long-term engineering headaches.
- Otherwise, never.

ACKNOWLEDGMENTS

This paper is the culmination of an unhealthy, years-long obsession with the idea of developers incorrectly using mmap in their DBMSs. The authors would like to thank everyone who contributed and provided helpful feedback: Chenyao Lou (PKU), David "Greasy" Andersen (CMU), Michael Kaminsky (BrdgAI), Thomas Neumann (TUM), Christian Dietrich (TUHH), Todd Lipcon (lipcon.org), and Sasha Fedorova (UBC).

This work was supported (in part) by the NSF (IIS-1846158, III-1423210, DGE-1252522), research grants from Google and Snowflake, and the Alfred P. Sloan Research Fellowship program.

REFERENCES

[1] Announcing InfluxDB IOx - The Future Core of InfluxDB Built with Rust and Arrow. https://www.influxdata.com/blog/announcing-influxdb-iox/.
[2] fio: Flexible I/O Tester. https://github.com/axboe/fio.
[3] MongoDB MMAPv1 Storage Engine. https://docs.mongodb.com/v4.0/core/mmapv1/.
[4] RavenDB Storage Engine. https://ravendb.net/docs/article-page/4.0/csharp/server/storage/storage-engine.
[5] RocksDB FAQ. https://rocksdb.org/docs/support/faq.html.
[6] SQL Server technical documentation. https://docs.microsoft.com/en-us/sql/relational-databases/policy-based-management/set-the-page-verify-database-option-to-checksum?view=sql-server-ver15.
[7] SQLite Memory-Mapped I/O. https://www.sqlite.org/mmap.html.
[8] The InfluxDB storage engine and the Time-Structured Merge Tree. https://docs.influxdata.com/influxdb/v1.8/concepts/storage_engine/.
[9] N. Amit. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *USENIX ATC*, pages 27–39, 2017.
[10] N. Amit, A. Tai, and M. Wei. Don't shoot down TLB shootdowns! In *EuroSys*, pages 35:1–35:14, 2020.
[11] D. L. Black, R. F. Rashid, D. B. Golub, C. R. Hill, and R. V. Baron. Translation Lookaside Buffer Consistency: A Software Approach. In *ASPLOS*, pages 113–122, 1989.
[12] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, 2008.
[13] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. Gray, W. F. K. III, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, G. R. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost. A History and Evaluation of System R. *Commun. ACM*, 24(10):632–646, 1981.
[14] K. Chodorow. How MongoDB's Journaling Works. https://www.mongodb.com/blog/post/how-mongodbs-journaling-works, 2012.
[15] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *EuroSys*, pages 211–224, 2013.
[16] A. Fedorova. Why mmap is faster than system calls. https://sasha-f.medium.com/why-mmap-is-faster-than-system-calls-24718e75ab37, 2019.
[17] A. Fedorova. Getting storage engines ready for fast storage devices. https://engineering.mongodb.com/post/getting-storage-engines-ready-for-fast-storage-devices, 2020.
[18] G. Graefe, H. Volos, H. Kimura, H. A. Kuno, J. Tucek, M. Lillibridge, and A. C. Veitch. In-Memory Performance for Big Data. *PVLDB*, 8(1):37–48, 2014.
[19] G. Haas, M. Haubenschild, and V. Leis. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR*, 2020.
[20] G. Henry. Howard Chu on Lightning Memory-Mapped Database. *IEEE Softw.*, 36(6):83–87, 2019.
[21] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.
[22] H. V. Jagadish, D. F. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dalí: A High Performance Main Memory Storage Manager. In *VLDB*, pages 48–59, 1994.
[23] A. Kivity. Different I/O Access Methods for Linux, What We Chose for Scylla, and Why. https://www.scylladb.com/2017/10/05/io-access-methods-scylla/, 2017.
[24] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. LeanStore: In-Memory Data Management beyond Main Memory. In *ICDE*, pages 185–196, 2018.
[25] T. Neumann and M. J. Freitag. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*, 2020.
[26] T. Neumann and G. Weikum. RDF-3X: a RISC-style Engine for RDF. *PVLDB*, 1(1):647–659, 2008.
[27] S. Papadopoulos, K. Datta, S. Madden, and T. G. Mattson. The TileDB Array Data Storage Manager. *PVLDB*, 10(4):349–360, 2016.

[28] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas. Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store. In *USENIX ATC*, pages 537–550, 2016.

[29] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas. An Efficient Memory-Mapped Key-Value Store for Flash Storage. In *SoCC*, pages 490–502, 2018.

[30] A. Papagiannis, G. Xanthakis, G. Saloustros, M. Marazakis, and A. Bilas. Optimizing Memory-mapped I/O for Fast Storage Devices. In *USENIX ATC*, pages 813–827, 2020.

[31] S. Park, T. Kelly, and K. Shen. Failure-Atomic msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data. In *EuroSys*, pages 225–238, 2013.

[32] A. Reece. Investigating Linux Performance with Off-CPU Flame Graphs. https://www.singlestore.com/blog/linux-off-cpu-investigation/, 2016.

[33] F. M. Schuhknecht, J. Dittrich, and A. Sharma. RUMA has it: Rewired User-space Memory Access is Possible! *PVLDB*, 9(10):768–779, 2016.

[34] D. G. Simmons. Re-examining our approach to memory mapping. https://questdb.io/blog/2020/08/19/memory-mapping-deep-dive/, 2020.

[35] R. Stoica and A. Ailamaki. Enabling Efficient OS Paging for Main-Memory OLTP Databases. In *DaMoN*, 2013.

[36] M. Stonebraker. Operating System Support for Database Management. *Commun. ACM*, 24(7):412–418, 1981.

[37] A. Valialkin. mmap may slow down your Go app. https://valyala.medium.com/mmap-in-go-considered-harmful-d92a25cb161d, 2018.

[38] R. Verma, A. A. Mendez, S. Park, S. S. Mannarswamy, T. Kelly, and C. B. M. III. Failure-Atomic Updates of Application Data in a Linux File System. In *FAST*, pages 203–211, 2015.

[39] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramírez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal. DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory. In *PACT*, pages 340–349, 2011.

[40] S. J. White and D. J. DeWitt. QuickStore: A High Performance Mapped Object Store. In *SIGMOD*, pages 395–406, 1994.