Why Files If You Have a DBMS?

Lam-Duy Nguyen
Technische Universität München
lamduy.nguyen@tum.de

Viktor Leis
Technische Universität München
leis@in.tum.de

***Abstract*—Most Database Management Systems (DBMSs) support arbitrary-sized objects through the Binary Large OBjects (BLOBs) data type. Nevertheless, application developers usually store large objects in file systems and only manage the metadata and file paths through the DBMS. This combined approach has major downsides, including a lack of transactional and indexing capabilities. Two factors contribute to the rare use of database BLOBs: the inefficiency of DBMSs in such workloads and the interoperability difficulties when interacting with external programs that expect files. To address the former, we present a new BLOB allocation and logging design that exhibits lower write amplification, reduces WAL checkpointing frequency, and consumes less storage than the conventional strategies. Our approach flushes each BLOB only *once* and features only a *single* indirection layer. Moreover, using the Filesystem in Userspace framework, BLOBs can be exposed as read-only files, allowing unmodified applications to directly access database BLOBs. The experimental results show that our design outperforms both file systems and DBMSs in handling large objects.**

***Index Terms*—Database management systems, Operating systems, File systems, User interfaces, Large objects, Strings**

I. Introduction

Data management systems are ubiquitous. DBMSs are commonly used for addressing a wide and heterogeneous range of real-world data management problems, offering valuable features such as ACID transactions and declarative queries. Their performance has been significantly optimized through decades of dedicated research and engineering, making them the default choice for diverse data management needs.

Large objects are stored as files. One major exception to the dominance of DBMSs is *large binary objects*. Although most DBMSs support the Binary Large Object (BLOB) and arbitrary-length string (CLOB, VARCHAR, TEXT) data types, proprietary or specialized data such as audio, image, video, and document objects are usually stored as files rather than inside the database system. Imagine an application that manages medical X-ray images: most developers would probably store all structured application data (including the patient data, image metadata, and file paths) in a DBMS, but the image data in a file system.

Downsides of files. Storing large objects in a file system separately from the application data (which is maintained in a DBMS) has several downsides:

- Durability: File systems and DBMSs have separate and independent durability regimes (fsync vs. commit). Imagine a situation where a crash occurs during the insertion of a new X-ray image and its record: depending on whether fsync or commit is executed first, one may end up either with an X-ray scan without a patient record, or a patient record without its associated X-ray image.
- Transactions: File systems do not support transactions, making it difficult to perform multi-file operations correctly. Consider an administrator updating a web application, which modifies multiple configuration and resource files. Without atomic multi-file operations (i.e., transaction), incomplete updates may occur, e.g. configurations in config files may reference deprecated resource files, leading to software inconsistencies and instability.
- Indexing: File systems lack support for indexing file content or metadata, which is beneficial in many situations. For instance, indexing facilitates tasks such as deduplicating files based on file content or organizing the files in ascending order by their last modification date.
- Performance: As we will show in Section V, accessing files can be slow due to system call overheads [1, 2].

Downsides of BLOBs. Given these problems, one may wonder why storing BLOBs in database systems is uncommon. We attribute this to two primary factors. First, database systems are not optimized for handling BLOBs[1], with file systems often proving more efficient. Second, BLOBs are often accessed not just by the storage systems, but also by external programs that require the input data in files. Consider, for example, a computer vision tool for classifying images, or a web server serving image data for a content management system. In both cases, the images will be expected as files. Having to copy BLOBs to the file system for interoperability with external programs would exacerbate many drawbacks of storing large objects within the DBMS.

Contributions. As pointed out in a CIDR keynote by Hannes Mühleisen [26], there is little research on managing large binary objects and strings in DBMSs. This work aims to close this gap. First, we present techniques for efficiently managing BLOBs in database systems, showing that a DBMS can outperform state-of-the-art file systems. To achieve this, we propose a new BLOB physical storage format and logging scheme for DBMSs. We write every BLOB only *once* to the storage while ensuring its crash consistency, and use a single-layer indirection called *Blob State* to obtain the on-storage location of every BLOB. This differs from existing approaches that write every object *twice* [11, 14, 10, 9, 24, 23, 15, 19] and use multi-level indirection layers to store BLOB [20, 11, 15, 6,

[1]One notable exception is SQLite, which is specifically advertised for use cases that would normally rely on file systems [2, 25].

TABLE I
LARGE OBJECT IMPLEMENTATIONS IN THE EXT4 FILE SYSTEM AND SEVERAL WIDELY-USED DBMSS

System	Physical storage format	Max size	Read cost	Indexing - Prefix limit	Duplicated copies
Ext4 file system	Multi-level extent tree [3]	16TB [4]	High[2]	Not supported	Journal [5][4]
PostgreSQL	TOAST relation [6]	4TB [7]	Medium[1]	8191 bytes [8]	WAL [9, 10]
SQLite	Linked-list of pages [11]	2GB [12]	High[2]	Arbitrary size [13]	WAL [11, 14] & Index [13]
SQL Server	Tree-like structure of pages [15]	2GB [16]	High[2]	Not supported [17, 18]	WAL [19]
MySQL/InnoDB	Linked-list of pages [20]	4GB [21]	High[2]	767 bytes [22]	DWB[3] & Redo [23, 24]
Our design	Extent sequence	10PB[5]	Low	Arbitrary size	None[6]

[1] Multiple lookup/scan to read a BLOB [2] Many indirection layers, I/O and computation interleave [3] Double-Write Buffer
[4] Mount with data=journal [5] Theoretically 5.76×10^{17} YB with 127 extents and 4KB page, details in Section III [6] Except BLOB update

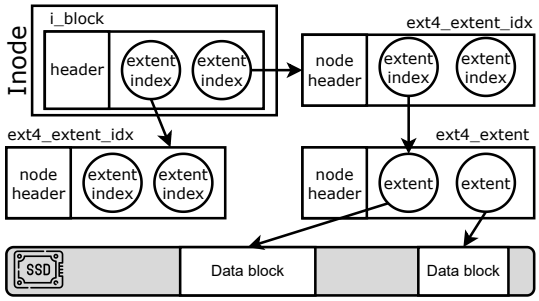
3, 5]. Second, we solve the interoperability issue with external programs using Filesystem in Userspace (FUSE) interface. We expose large objects as *read-only files*, allowing external software to *directly* access DBMS-managed BLOBs without code modifications. Overall, with our approach, applications managing BLOBs gain all functional benefits of DBMSs (e.g., transactions and indexing) without sacrificing performance or complicating interoperability with external programs.

II. BACKGROUND

A. Limitations of Existing Approaches

In the following, we describe how PostgreSQL, SQLite, Microsoft SQL Server, and MySQL/InnoDB manage BLOBs, and contrast them with Ext4, the default file system of Linux.

Ext4: Hierarchical extent tree. The standard file system in Linux, Ext4, maintains files in a multi-layer structure [3], as the following figure shows:



Ext4 builds an *extent tree* structure for every file larger than 512MB. The extent tree helps translate the file logical address to the corresponding physical blocks [3, 27]. It is complex (for good reasons, which we will discuss in Section III). However, extent tree also has some limitations, one particular issue is the tree traversal overhead. That is, accessing data in Ext4 requires navigating through several layers of the extent tree, mixing I/O and computation, which may reduce the performance.

Inefficient storage format in DBMSs. The surveyed DBMSs utilize either an auxiliary structure [20, 11, 15] or a relation to manage the BLOB chunks [6]. In the first approach used in MySQL, SQLite, and SQL Server, the DBMSs store every BLOB in multiple overflow pages, which are chained together using a linked list or a tree. Consequently, queries will access the overflow pages sequentially one after another [20, 11, 15], resulting in I/O interleaved with computation and thus higher query latency. The second method, The Oversized-Attribute Storage Technique (TOAST) implemented by PostgreSQL [6], does not force I/O and computation to interleave. It organizes the BLOB chunks (and metadata) in a separate "TOAST" table. Consequently, every BLOB read involves *two* relation lookups (the main relation and the TOAST table) in addition to one scan to read all chunks. Because every TOAST page contains only *four* chunks by default [6], read operations must scan through multiple database pages to retrieve the BLOB content. In summary, these indirections significantly contribute to the explanation of why accessing BLOBs is not always efficient.

Excessive BLOB writes. To ensure BLOB integrity, DBMSs write every entry at least *twice* to the storage, both to the database and log [11, 14, 10, 9, 24, 23, 15, 19]. This design has two consequences. First, it increases the log size and thus triggers WAL checkpointing more frequently, which slows down the database operations [28, 29]. Second, it increases write amplification excessively, reducing the longevity and performance of the storage device if the DBMS runs on top of an NVMe SSD [30, 31, 32]. When mounted with `data=journal` option, the Ext4 file system behaves similarly with the content of the new file also being written to the journal [3, 5].

Unnecessary BLOB copies. All systems maintain at least *two* copies of every BLOB, one in the database and one in the log. SQLite is the worst in terms of storage consumption because it includes whole BLOBs in `WITHOUT-ROWID` index, and it also logs the BLOB content from *both* the database and index [14]. In total, SQLite creates at least *four* copies per BLOB if both WAL and `WITHOUT-ROWID` index are enabled.

BLOB indexing limitations. Amongst the surveyed systems, only SQLite supports full BLOB indexing. However, SQLite doubles the content of those objects, storing them in both the main relation and the BLOB index (`WITHOUT-ROWID` index [13]), and thus is not recommended if the object size is huge [13]. PostgreSQL and MySQL/InnoDB only index BLOB prefixes [22, 8], while SQL Server disallows indexing BLOB data altogether [17, 18].

Summary. Table I summarizes the existing approaches and contrasts them with our solution. We ensure BLOB durability without writing it more than *once*. Additionally, our single-layer BLOB storage format is lightweight, simplifying BLOB operations. Finally, we support BLOB indexing like SQLite but require no BLOB copy, saving storage consumption.

III. Large Object Life-cycle

In this work, we assume that the DBMS runs on an NVMe SSD with a buffer cache that supports fixed-size pages. In most DBMSs, the page size is usually 4-64 KB. The buffer manager heavily relies on page translation, which maps a page identifier (PID) to an in-memory pointer that refers to the page content. We refer to those in-memory pointers as *buffer frames*.

A. Extent Management

Existing approaches in DBMSs are ineffective. Current DBMSs implement either an auxiliary index structure to manage overflow pages or use a system relation to manage BLOB chunks. Despite being simple, these approaches have many limitations as described in Section II. This leads to an intriguing question: what if we implement the extent tree in DBMSs specifically for BLOB management?

Why extent tree? There are several reasons behind the extent tree. First, file systems should be efficient even in obscure scenarios, including the hole-punching operation that deletes middle extents and reclaim their space. Second, file systems use a *best-effort* approach to allocate new extents by seeking the largest free space available. Altogether, file systems store a file as an arbitrary number of extents of arbitrary size, which requires the extent tree to manage them effectively.

Are those requirements avoidable in DBMSs? We believe the answer is yes because typical applications either generate static objects, store multiple versions of objects, or replace object completely [19]. The operations in such scenarios – *create/replace*, *read*, and *delete* – do not interact with middle extents. Analogously, Amazon S3, a widely-used object storage system, also restricts user interactions to entire BLOBs, disallowing partial updates and removals [33].

Extent sequence. We suggest storing BLOBs as a *flat* list of extents (an extent is a contiguous range of physical pages), termed *extent sequence*. By enforcing exponential growth on this list – ensuring subsequent extents are always larger than previous ones – we can limit the size of this list while supporting huge BLOBs. In other words, the list of extent is small but still represent any arbitrarily sized BLOB.

Reducing BLOB metadata. The metadata necessary for BLOBs comprises the extent offset (the PID of the head page) and the extent size (number of pages). We propose replacing the extent size metadata with a table, which determines the extent size using the *static* extent position, thus halving the size of BLOB metadata. We call this table *extent tier*.

Extent tier: Constraint & goals. A good tier table is crucial to how the system manages large objects. That is, it affects maximum BLOB size, simplicity and efficiency of BLOB operations, amount of BLOB metadata, and storage utilization. Existing formulas such as Power-of-Two and Fibonacci are not suitable because of their high space consumption [34], i.e., 50% wasted space for Power-of-two and 38.2% for Fibonacci, hence a new formula is required.

Extent tier: Proposed formula. Instead, we propose a new formula that utilizes storage more effectively. First, we logically split the tiers into multiple levels, and each level has the same number of tiers, e.g., if the system has 10 levels and each level comprises 10 tiers, then the number of tiers is 100. Any tier after this has the same size as the largest tier. For any arbitrary tier, given its *level* and its *position* within that level (both counters start at 0), the size of that tier is:

$$(level+1)^{no\_tiers\_per\_level\ -\ position} \times (level+2)^{position}$$

With 10 tiers per level, the first two levels (20 tiers) are:

Level 0	1 32	2 64	4 128	8 256	16 512
Level 1	1k 7.8k	1.5k 11.7k	2.3k 17.5k	3.5k 26.2k	5.2k 39.4k

Assuming a 4KB page size, an extent sequence of 127 extents following this config can store a BLOB up to 10PB.

Balancing storage utilization and max size. This formula improves the storage utilization compared to Power-of-Two and Fibonacci. For instance, given a 4KB page size and five tiers per level, the wasted space for a 20MB BLOB is 25%. This number decreases as the BLOB size increases, dropping to 7.3% when the BLOB is 51GB. However, an 127-extent sequence only supports a BLOB up to 246GB with this setting. Increasing tier count per level allows larger BLOBs with a trade-off of lower storage utilization. With 30 tiers per level, the first level already support a 4TB BLOB, and the storage utilization of a BLOB fitting 120 extents is 20%, which is still better than both Power-of-Two and Fibonacci.

Tail extent: Arbitrarily-sized extent. For static BLOBs, the last extent may contain unused space, resulting in internal fragmentation. To prevent that, we allocate *exactly one* arbitrarily-sized extent, termed *tail extent*, to replace the last extent. For instance, in the example illustrated in Figure 1, the normal strategy (Figure 1(a)) allocates three extents, and the last one has one empty page. With tail extent, as Figure 1(b) depicts, the DBMS allocates only two extents to store *"Foo and Bar"* and stores the rest in three consecutive pages.

B. Blob State

Format. We bundle all BLOB metadata into a single structure named *Blob State*. Every Blob State refers to only one BLOB. Specifically, Blob State comprises the following properties.

- **Size**: Size of the referred BLOB.
- **SHA-256**: The computed SHA-256 of the BLOB, is used for BLOB durability & indexing.
- **SHA-256 intermediate digest**: The 32-byte intermediate SHA-256 hashed signature (i.e., before the last 512 bits of the BLOB and padding), used for BLOB growth operations.

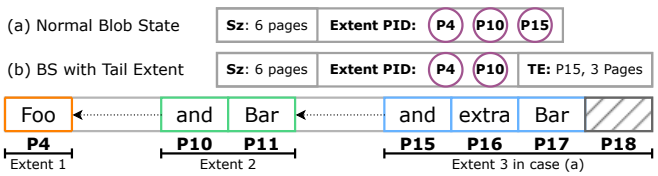


Fig. 1. A BLOB of 6 pages and two possible Blob States

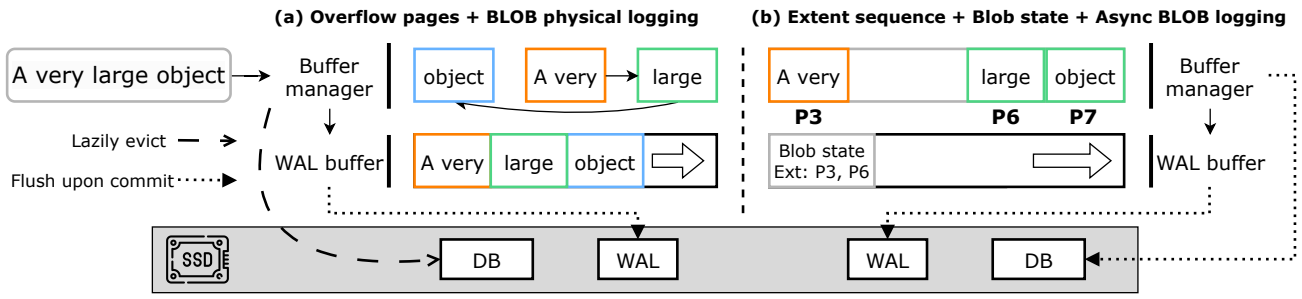


Fig. 2. Traditional design in popular DBMSs (a) vs. our proposed approach (b)

- **Prefix**: First 32 bytes of the BLOB. We will explain the usage and motivations behind **Prefix** and **SHA-256** for BLOB indexing in Section III-F.
- **Tail Extent**: A pair of a Page ID and the number of pages. Will only be populated if the BLOB has a tail extent.
- **Number of Extents**: The number of extents (excluding tail extent) used to store the content of the BLOB.
- **An Array of Head Page PID**: A dynamic array of Page IDs, all of which refer to the head page (first page) of all extents. By combining this array with the extent tier, the system can determine the physical address of all extents.

Physical BLOB size. As explained earlier, with *extent sequence* and *extent tier*, a small number of extents can represent a huge BLOB. Therefore, the flexible array of the head page PID is not necessarily long, allowing the Blob State to be small in size while still referencing an arbitrarily-sized BLOB. For example, with the number of tiers per level is 8, a Blob State of 801 bytes can refer to a BLOB of more than 16TB – the maximum file size that Ext4 supports [4].

Example. Figure 1 illustrates a Blob State for a 6-page BLOB. If the DBMS allocates the BLOB normally (Figure 1(a)), the Blob State will contain three extents: P4, P[10..11], P[15..18]. If the BLOB contains a tail extent (Figure 1(b)), the Blob State will only have two extents: P4 and P[10..11], and the tail extent starts at P15, spanning three pages.

Where to store Blob State. The DBMS should physically store the Blob State with the tuple for the BLOB column. Consider a sample relation of an Integer primary key and a BLOB column. Every row of this relation should store Blob State for its associated BLOB column. All BLOB accesses will first query this relation for the Blob State and then load all the extents using the retrieved Blob State.

C. Durability

Redundant BLOB writes in conventional logging. Figure 2(a) depicts the BLOB allocation and logging approach deployed in major DBMSs. In this approach, the DBMSs break each BLOB into multiple chunks and store BLOB chunks on random pages. After the allocation, all the BLOB parts are copied to the WAL buffer, which is flushed to the non-volatile storage later. All these BLOB chunks will also be written out to storage later during the buffer eviction process. That means, every BLOB is written to storage at least *twice*.

Asynchronous BLOB logging. Our approach neither appends BLOB content to the WAL nor writes the BLOB during buffer eviction. Instead, we write all BLOB chunks during the transaction commit. Figure 2(b) shows our design. First, the DBMS reserves the smallest extent sequence to store the new BLOB, i.e., two extents for a three-page BLOB in the example. After that, the system creates a Blob State, stores this Blob State in the corresponding relation, and then appends it to the WAL buffer. Upon transaction commit, the DBMS triggers multiple asynchronous I/O requests to flush the WAL buffer (which contains the Blob State) and the extent sequence. Note that the DBMS only writes the dirty pages to storage, e.g., only P[15..17] of the 3rd extent in Figure 1.

BLOB Recoverability. To ensure recoverability, the DBMS must guarantee that the Blob State is durable before writing the extents. This is because if the BLOB is flushed before the Blob State is durable and then the DBMS crashes, the extents are lost and unusable, leaving unusable holes within the DBMS. Therefore, we write and call `fsync()` to persist the WAL buffer (which contains Blob State) before writing the extents. When a crash happens between the two events, during *Analysis phase* of the recovery process, we can use the SHA-256 checksum to validate the BLOB content. If BLOB content is faulty, the transaction committing that BLOB is considered *failed* and added to the UNDO transaction list.

BLOB eviction. Conventional methods additionally write all BLOBs during eviction because all pages storing BLOB content are marked *dirty* after the allocation. In contrast, we flush all BLOBs at transaction commit, thus all BLOB extents[2] are *clean*, eliminating the extra write. However, before completing the flush, concurrent transactions may evict one of the extents, causing the buffer pool to drop or replace the corresponding frame(s) with other page(s). We prevent that using an atomic `prevent_evict` flag per extent, set to *true* post allocation and reset to *false* upon the extent flush is complete. Buffer manager does not evict extents with `prevent_evict=true`, avoiding undefined behaviors.

D. Operations

BLOB read. To load a BLOB, DBMS first looks up the BLOB relation to obtain the Blob State. Using this Blob State, DBMS

[2]Our solution evicts/synchronizes BLOB accesses on extent granularity. We will discuss more on this later in Section III-G.

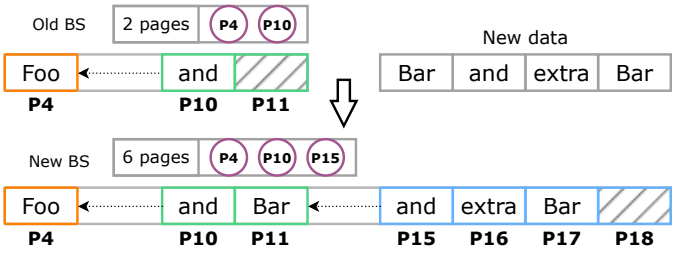


Fig. 3. Append new content to an existing BLOB

determines which extents are not in the buffer pool, assuming these extents consist of N pages. Then, the DBMS allocates N buffer frames for all those extents and reads the extents using a *single* asynchronous IO system call.

BLOB deletion and extent reusability. The extent tier design helps us reuse the deleted extents efficiently. Because tiers are static, it is sufficient to manage a list of free extents per tier. During BLOB removal, the start PID of all extents is added to a temporary list. At transaction commit, the DBMS moves the free extents from the temporary list to the free lists according to the extent tier. Subsequent transactions can either pick a free extent in these free lists or allocate a fresh one.

Growing a BLOB. In the example shown in Figure 3, we append a four-page chunk (*Bar and extra Bar*) to a 2-page BLOB. Because the last extent lacks space to store new content, DBMS allocates more extents (one in the example), and then `memcpy()` new data to the available space. Afterward, the DBMS adds the two dirty extents to the to-flush list but only writes the dirty pages (P11 and P[15..17] in the example). Then, DBMS re-calculates the SHA-256 signature by resuming previous SHA calculation (based on the stored intermediate SHA digest) with new appended data, i.e., preceding BLOB data is not loaded into the buffer pool. Finally, the DBMS updates the Blob State to reflect the latest details of the BLOB. For a BLOB with a tail extent, the DBMS can grow that object by cloning the tail into a new normal extent and following a similar procedure.

Updating a BLOB. To update a BLOB, the DBMS determines the extent(s) that should be modified. After that, for each extent, DBMS either (1) creates a *delta* log which contains the difference between the old and the new data, appends the log record to the WAL buffer, and then in-place updates the extent, or (2) allocates a clone extent of the same tier, then updates this clone and the corresponding metadata in the Blob State. These two schemes are better in different situations, i.e., in the first scheme, new data is written *twice*, while the second writes old data one more time. Evaluating the cost of both schemes and selecting the better approach at runtime is straightforward. Nevertheless, because most applications primarily interact with entire BLOBs, we argue that writing data twice (either old or new data) in this scenario is acceptable.

E. Interoperability With File Systems

External apps mainly use files. One limitation of storing large objects in DBMSs is that such systems do not provide file APIs, which is the primary method for external programs to access BLOBs [35]. For example, computer vision libraries such as Tesseract OCR [36, 37] or OpenCV [38] work with image files instead of raw binary image data. One workaround is to copy the BLOBs into the file systems, which may be expensive and potentially hide the efficiency of our design. Another way is to rely on an IO wrapper over binary data similar to [39], yet it incurs extra complexity on external programs. Such a wrapper is also not ubiquitous across programming languages and thus can not be deployed everywhere.

Filesystem in Userspace. One solution is to integrate with Filesystem in Userspace (FUSE) [40, 41] to provide the file system interface with DBMSs. FUSE is the most popular framework in Unix OS that allows non-privileged users to implement their file system in user space [41] without necessitating kernel code modifications. By integrating with FUSE, we can facilitate seamless interoperability between the DBMS and file systems, and applications can access their BLOBs in DBMSs without modifying the source code.

Relation as a directory. Consider a scenario where users want to store images within the DBMS, and now they want to expose those images as read-only files. All the images can be managed within the following relation:

```
CREATE TABLE image (
  filename VARCHAR PRIMARY KEY, content BLOB)
```

With FUSE, assuming the mount point of the sample DBMS is `/foo/bar`, users can access all images in `/foo/bar/image` directory. At the same time, users can also store BLOBs in other relations which appear in different directories. For example, documents can be stored in `document` relation, and users can interact with those BLOBs as files in `/foo/bar/document` directory.

```
1  int FUSE_open(char *path) { // open() system call
2    db->StartTransaction();
3    return 0;
4  }
5  int FUSE_flush(char *path) { // close() system call
6    db->CommitTransaction();
7    return 0;
8  }
9  // pread() system call
10 int FUSE_read(char *path, u8 *buf, u64 size, i64
       offset) {
11   // 1. Check whether file exists or not
12   auto &[relation, filename] =
         ExtractRelationAndFileName(path);
13   BlobState state = db->LookUp(relation, filename);
14   if (state == nullptr) { return -ENOENT; }
15   // 2. Path exists, read the BLOB
16   assert(state->size > offset);
17   size = std::min(size, state->size - offset);
18   db->ReadBlob(state, [&](std::span<u8> blob) {
19     std::memcpy(buf, blob.data() + offset, size);
20   });
21   return size;
22 }
```

Listing 1: FUSE integration: Pseudo code for read operation

Expose BLOBs as read-only files. Listing 1 shows how to implement `read` operation in FUSE integration. To ensure subsequent reads on the same BLOB are consistent, we wrap all `read` inside a transaction. This is achieved by implementing `open` and `flush` FUSE operations (which are triggered by `open()` and `close()` system calls, respectively) to start and commit a transaction (lines 1 to 8). For `read` operation, the DBMS first looks up the relation (e.g., table `image`) to check if the file exists (lines 12 to 14). If the file exists, we use the Blob State to load the BLOB content and copy that to the user buffer (lines 17 to 20). Other read-only operations such as `getattr` are implemented similarly to `read`, i.e., a point query to obtain the Blob State to satisfy those operations.

F. Indexing

Problems of current approaches. Popular DBMSs either index only BLOB prefixes that misses many records, or store full BLOBs in all tree nodes of secondary indexes including inner nodes, increasing tree height significantly and reducing performance. One solution is to use TOAST storage [6] and index the BLOB IDs based on their content. However, this requires tight coordination between relation APIs (e.g., tree scan) and indexing, which is hard to implement correctly.

Blob State index. Instead, by implementing a comparator for Blob State, index structures can store the Blob States in sorted order according to their BLOB content. This mirrors the above approach proposed for TOAST, which uses BLOB IDs as the indexed key. The difference is that the Blob State index avoids working with other relations, and it also accesses BLOB data directly, thus being cheaper and less complicated. Note that the indexing structure is untouched, and DBMSs can use any data structure like B-Tree or ART [42].

SHA-256 and BLOB prefix. For point queries, comparing entire BLOBs per every comparison is inherently expensive. Instead, we suggest using SHA-256 for more efficient BLOB equality checks[3]. Analogously, for range queries, a complete BLOB comparison may be unnecessary. A cheaper option is to store the BLOB prefix inside the Blob State, allowing the comparator to skip BLOB dereferencing in some situations.

Incremental comparator for Blob State. Because the comparator will be extensively used during the index operations, comparing the full BLOB content of two Blob States in a comparator is costly and possibly unnecessary. Instead, we propose to compare Blob States *incrementally*. Assuming that the two Blob States do not contain a tail extent. For point queries, the comparator evaluates the SHA-256 values embedded in the two Blob States and returns the result. Range queries involve additional steps: after the equality check, we use the embedded prefix for a cheap range check. If the two BLOBs have the same prefix, then we compare all the extents of the two BLOBs *incrementally*. Finally, if those extents are identical, then one of the two BLOBs is the prefix of the other, so we compare the size of the two objects and return the result.

Semantic index for BLOB. There are situations where indexing the semantic meaning of BLOBs is more appropriate than the raw binary data. One way to implement that is to support index based on a function or scalar expression computed from the BLOB attribute of the relation, similar to Expression Index in PostgreSQL [45]. With Blob Tuple, the DBMS can dynamically compute the derived data for the Blob Tuple comparator during query execution. Below is one example regarding how users interact with the semantic index:

```
CREATE UDF classify(blob) -> TEXT;
CREATE INDEX foo image(classify(content));
SELECT * FROM image WHERE classify(content)='cat';
```

In this example, the Blob Tuples are sorted according to the `classify()` UDF. During `SELECT`, the DBMS scans through all Blob Tuples classified as *cat* and returns those data records to the user.

G. Extent Synchronization And Eviction

Synchronization: Coarse-grained vs. fine-grained. To access/evict an extent from the buffer pool, we can either use fine-grained synchronization (one latch per page) or use coarse-grained latching (synchronize on the first page of the extent). The former design is more complex and may introduce overheads. For example, when N threads attempt to read the same extent of N pages from storage, all workers contend for N latches, each wins one and then calls `pread()` to fetch one page. Contrarily, with coarse-grained latching, only one worker will call `pread()`, allowing the remaining workers to work on other tasks. Therefore, we opted for coarse-grained latching for extent synchronization/eviction.

Fair extent eviction. In the coarse-grained extent synchronization design, the eviction probability of an extent may be similar to that of a normal page. However, we argue that an N-page extent should have an eviction probability N times higher than a single page. To do that, we adjust the eviction probability of every page and extent according to its size:

```
if (rand(MAX_EXT_SIZE) ≤ extent_size[pid]) Evict();
```

H. Discussion

Tail extent vs. Extent tier formula. Tail extent completely resolves the storage utilization issue compared to the tier formula, but it slows down BLOB growth operations. That is, appending new data to a BLOB with a tail extent is more expensive than that for a normal BLOB because of the extent clone operation, which includes one extent allocation and `memcpy()` data from the tail extent to the new extent. Generally, the tail extent should be used if the workloads do not involve growth operations. We summarize the differences in the table below:

	internal frag.	growth op.
tail extent	minimal	slow
extent tier formula	low	fast

[3]We acknowledge that SHA-256 may not be theoretically foolproof for equality checks. However, the fundamental reliance of Bitcoin on SHA-256 to resist collision attacks [43, 44] implies its practical suitability for critical applications, including DBMSs, in ensuring reliable uniqueness checks.

Concurrency control for BLOBs. The primary focus of this work is orthogonal to BLOB concurrency control. However, let us mention one possible design: to use a Single-Version Concurrency Control protocol such as 2PL [46], OCC [47], or Silo [48] on the Blob State relation. For example, with 2PL, when transaction A wants to update a BLOB, it acquires an exclusive lock on the record that contains the required Blob State and then modifies the BLOB content. Now, transaction B concurrently accesses the same Blob State and then finds out that the required tuple is locked by transaction A. Consequently, transaction B aborts or waits according to any conflict resolution scheme [49, 50, 51, 52, 53, 54].

IV. Virtual-Memory Assisted Operations

As Section II discussed, popular DBMSs implement complex and inefficient mechanisms to store large objects, primarily due to their reliance on buffer management designs that only support fixed-size pages. Recent work on buffer management relies on virtual memory to implement a buffer manager that allows variable-sized pages. The proposed methods, vmcache and exmap [55], simplify the implementation and enhance the performance of BLOB operations compared to the previous buffer pool designs. This section details how our design benefits from these new buffer management techniques.

A. Virtual-Memory Assisted Buffer Manager

Problems of fixed-size pages. With fixed-size pages, DBMSs arrange extents and pages as arbitrary disjointed buffer frames, i.e., most BLOBs are not represented as contiguous memory. As a result, either external libraries must work explicitly with BLOB chunks, e.g., for regex search, external libraries apply regex matching on every BLOB chunk, or the DBMS must allocate a big memory chunk and then `memcpy()` BLOB content to this memory block before processing, consuming memory bandwidth extensively.

vmcache. A recent work, vmcache [55], exploits the virtual memory to implement a simple yet effective buffer manager. This technique helps manage BLOBs for two reasons. First, vmcache presents an extent as contiguous memory and needs only *one* page translation per extent to retrieve the buffer frame(s). Contrarily, previous buffer pool designs (e.g., hash table or pointer swizzling [56, 57, 58]) trigger exactly N page translations for the same task. Second, assuming we can adjust the mapping of virtual to physical memory *in user space* during runtime, vmcache can present *a list of disjointed memory blocks* (i.e., extent sequences) as contiguous memory.

Virtual memory remapping. One potential method that allows virtual memory remapping is Rewired User-space Memory Access (RUMA) [59]. However, RUMA slows vmcache down significantly in out-of-memory workloads due to its memory management method[4]. Instead, based on exmap which provides performant and scalable page table manipulation primitives [55], we propose *virtual memory aliasing*, a technique that copies the physical addresses of an extent sequence and maps that to a free virtual memory space, presenting disjointed extents as contiguous memory.

[4]RUMA uses an in-memory file and SHARED mmap() to manage OS page table. In this design, page eviction requires `fallocate(PUNCH_HOLE)` to free the physical memory of the memory file, which is very slow.

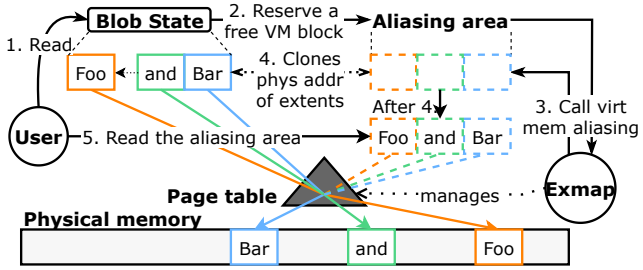


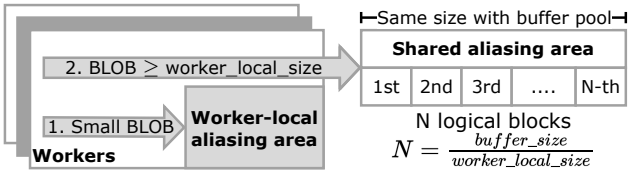
Fig. 4. Virtual memory aliasing operation. An aliasing area is a contiguous range of virtual memory addresses

B. Virtual Memory Aliasing

Operations. We depict virtual memory aliasing in Figure 4. First, when a transaction reads a BLOB of multiple disjointed extents, it retrieves the Blob State and loads all the extents into the buffer manager. After that, the transaction requests a free contiguous range of virtual addresses (termed *aliasing area*), and then calls memory aliasing operation on this aliasing area. Consequently, *exmap* updates the page table to map the physical addresses of the aliasing area to that of all the extents. Finally, users access the aliasing area which depicts the BLOB content as a single contiguous memory block.

Aliasing area: Constraints. It is reasonable to bound the required number of virtual addresses. Because the largest object is limited by the maximum buffer pool size, the aliasing area is unnecessary to be bigger than that. One may wonder whether to use N separated aliasing areas for N workers so concurrent workers do not need to synchronize. However, this approach consumes an excessive number of virtual addresses, i.e., ten workers with a database size of 160GB requires approximately 420M virtual addresses.

Aliasing area: Proposed design. The following figure illustrates the design of the aliasing area:



Every worker has one exclusive *worker-local aliasing area*. In the first case, when BLOBs are smaller than size of the worker-local area (`worker_local_size`), the worker uses its local area without contention with other workers. Otherwise (case 2), the worker requests free contiguous virtual addresses from a shared pool (*shared aliasing area*). The shared pool is split into N *logical blocks*, with each block is similar in size to the worker-local area. During reservation, the worker exclusively uses a range of contiguous logical blocks sufficiently large to alias the BLOB. The DBMS uses a range lock to synchronize concurrent accesses to the shared area.

Lightweight synchronization on shared area. An intriguing finding is that, with an appropriate worker-local size, we can limit the number of logical blocks to a small amount while also capping the number of virtual memory addresses. Assuming the buffer pool is 160GB, i.e., shared area is also 160GB. If the worker count is 10 and the size of a worker-local area is 1GB, then the number of logical blocks is 160 and the total size of the aliasing areas is 170GB, which is only 6.25% bigger than the buffer manager. And because the number of blocks is small, we can use a simple range lock using a bitmap and *compare-and-swap*, i.e., in the above example, the bitmap only has 160 bits which corresponds to 3 `uint64_t`.

Overhead: TLB shootdown. One issue is that the worker must invalidate the mapping between the virtual addresses of the aliasing area and the physical memory pages. This involves clearing the corresponding page table entries and invalidating the TLB cache (i.e., TLB shootdown), which interrupts all CPU cores and clears the TLB of all CPUs. Although the overheads can be nonnegligible [60, 55], we argue that memory aliasing substantially simplifies BLOB operations and is cheaper than the `malloc()` and `memcpy()` combination. We will explain this later in Section V-E.

Size of worker-local area. The worker-local area needs not to be big because the BLOB size constraint is small in practice [61, 62, 63]. With a proper configuration like 1 GB, the shared area will be rarely used. Even if the worker uses the shared aliasing area, i.e., BLOB is bigger than the local area, the contention on the shared area is insignificant to other operations. Further elaboration on this will be provided in Section V-F.

V. Evaluation

In this section, we empirically show that our approach depicts superior performance to file systems in BLOB management – although we disable `fsync()` for all competitor DBMSs and file systems – while still offering qualitative benefits like transactional semantics and durability.

A. Experiment Setup

Implementation. We integrate our proposed techniques, denoted as `Our`, into LeanStore [57], an open source storage engine. In this version of LeanStore, we implement vmcache and exmap [55] as the buffer manager. The default size of the buffer pool is 32GB. Our implementation uses distributed per-thread write-ahead logging with page-level dependency tracking [28, 64], combined with group commit [65, 66].

Hardware & OS. We ran all experiments on a single-socket machine with an Intel Core i7-13700K (16 cores, 32 hardware threads), 64GB DRAM, and a Samsung SSD 980 Pro M.2 as the storage. For OS, we use Linux 6.2 with exmap installed.

Competitors: DBMSs. We compare our implementation against three popular DBMSs: PostgreSQL [67], MySQL/InnoDB [68], and SQLite [69]. We do not evaluate DuckDB [70] because there exists a comparison between SQLite and DuckDB in BLOB workloads [2], and also because DuckDB was not designed for managing large objects.

DBMS config. For PostgreSQL and MySQL, we configure to connect to the server using a Unix socket. We use a 32GB buffer pool for MySQL and SQLite, and a 16GB shared buffer for PostgreSQL as recommended [71]. Since this work focuses on BLOB buffer and storage management which is orthogonal from transactional aspects, we run all DBMSs in the lowest transactional isolation level offered. To ensure fair comparisons with file systems, we disable both BLOB compression and `fsync()` for all competitor DBMSs.

Competitors: File systems. We also evaluate our design with four file systems: Ext4 [3], XFS [72], BtrFS [73], and F2FS [74]. For the Ext4 file system, we mount it with two options: `data=journal` and `data=ordered`, and we refer to them as `Ext4.journal` and `Ext4.ordered`, respectively. With `Ext4.journal`, data is also written to the journal. On the other hand, `Ext4.ordered` only writes file metadata to the journal and only does so after the data is flushed to the secondary storage.

File system config. We disable readahead because it is orthogonal to the core operation. As stated earlier, we do not use `fsync()` for all file system benchmarks because it would become the dominant overhead in every file system benchmark if it was enabled. Moreover, our implementation uses group commit so the critical path usually does not involve I/O.

B. Evaluation of BLOB Logging

Experiment information. We evaluate our logging scheme using synthetic YCSB workloads with different payload sizes. Specifically, we use five configurations: 120 bytes, 100KB, 10MB, one workload with a random size between 4KB and 10MB, and 1GB. The working dataset of all experiments fits in memory. We run the following experiments in single-threaded mode with a read ratio of 50%. We use a simple `memcpy()` as the BLOB read operator. BtrFS is not shown because its performance is almost identical to Ext4 ordered.

Baselines. We implemented two baselines: `Our.ht` and `Our.physlog`. `Our.ht` uses a traditional hash table buffer pool instead of `vmcache+exmap`, thus not benefit from virtual memory aliasing. `Our.physlog` employs all techniques except async BLOB logging. Instead, it appends every large object to the write-ahead log. To accommodate BLOBs larger than the WAL buffer, we split every BLOB into small segments and append these segments to the WAL buffer.

120B payload. First, we evaluate all systems with normal YCSB and show the result in Figure 5. All file systems and SQLite provide higher throughput than PostgreSQL and

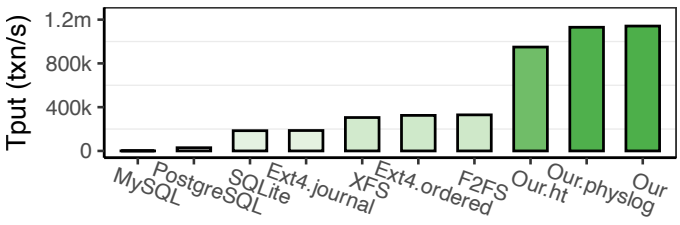
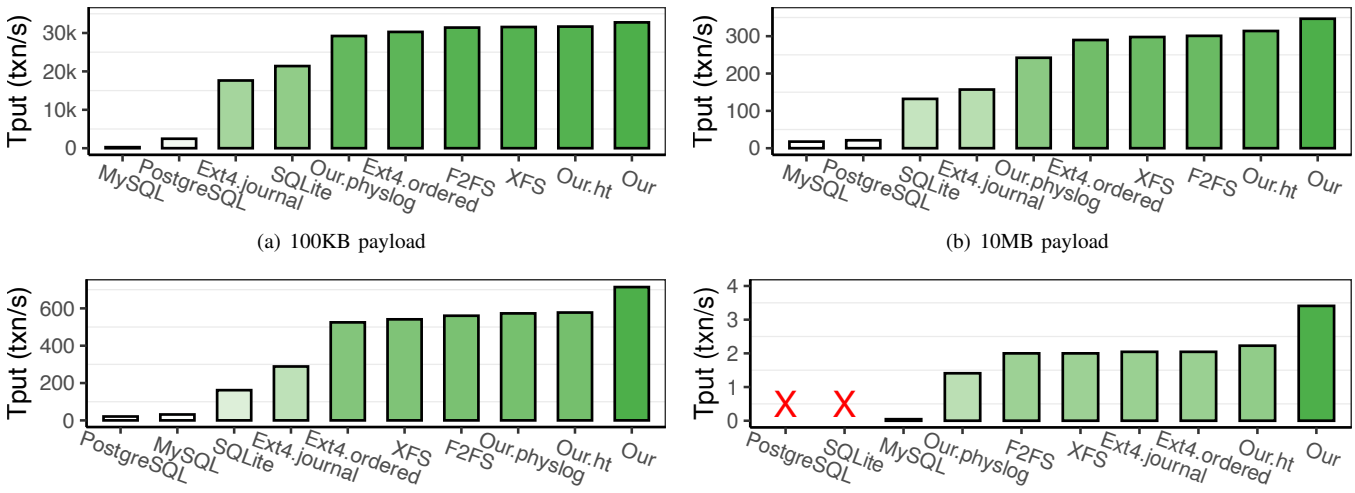


Fig. 5. YCSB benchmark with normal payload size (120B)



(a) 100KB payload

(b) 10MB payload

(c) Mixed (4KB - 10MB) payload

(d) 1GB payload. The PostgreSQL and SQLite benchmark scripts fail

Fig. 6. YCSB benchmark with BLOB payload. `fsync()` is turned off for all systems except `Our`, `Our.ht`, and `Our.physlog`

MySQL because these systems only operate in main-memory. In contrast, PostgreSQL and MySQL incur additional communication and (de)serialization overheads. Our DBMS provides at least 3.5× higher throughput compared to other systems.

100KB payload. As Figure 6(a) shows, MySQL and PostgreSQL provide poor throughput, also because of the network and serialization overheads of these DBMSs. All file systems have comparable throughput (including BtrFS), except `Ext4.journal`. Ext4 journal exhibits bad performance because includes I/O in the execution time while other file systems do not, and it also triggers journaling operations more excessively. One notable observation is that SQLite is faster `Ext4.journal`, i.e., it does not trigger I/O during transaction execution. All file systems are slower than `Our` and `Our.ht` because of system call overheads. `Our.physlog` is 11% slower than the `Our` because of the WAL operations.

10MB payload: All systems vs. Our. Figure 6(a) shows that PostgreSQL and MySQL still depict bad performance. For file systems, `Ext4.journal` remains the slowest due to the journaling. SQLite is slower than `Ext4.journal` in this experiment because it triggers WAL checkpointing aggressively (2.5 checkpoints per BLOB write [2]). Other file systems provide comparable throughput, all are at least 13% slower than `Our` because of one extra memory copy call. That is, file systems cause two memory copy, i.e., one from `pread()` system call and another from the BLOB read operator in the application. Contrarily, only one memory copy is required in `Our` because it replaces `pread()` with the lightweight *virtual memory aliasing*.

10MB payload: Our.physlog vs. Our. `Our.physlog` is slower than `Our`, provides 30% less throughput, mainly because the hot path includes time waiting for the group committer to flush the BLOBs. That is, because the BLOB size is as big as the configured WAL buffer, transactions must spend considerable time waiting for the group commit to finish. By increasing the size of the WAL buffer (e.g., from 10 MB to 50 MB), this overhead becomes smaller, but the overall throughput is still lower than that of `Our`.

Mixed 4KB-10MB payload. Popular DBMSs exhibit poor performance in this experiment, as depicted in Figure 6(c). `Ext4.journal` is the worst amongst all file systems, trailing `Ext4.ordered` by 45%. Surprisingly, the performance differences between `Our` and file systems are larger than in previous experiments because of OS file size modification overhead, which includes `ftruncate()` to resize files and new buffer allocation in the page cache. On the other hand, `Our` and `Our.ht` handle this workload without imposing extra overhead. This also explains why `Our.physlog` is faster than file systems in this experiment.

1GB payload. As illustrated in Figure 6(d), all enterprise DBMSs perform poorly. Specifically, the PostgreSQL client library returns *Statement parameter length overflow*, and SQLite gives *BLOB too big* error, leading to benchmark failure. Two baselines and file systems perform similarly, and they show at least 70% less throughput than `Our`. This experiment exhibits the benefits of our proposal compared to existing techniques.

Hash table buffer pool vs. vmcache+exmap. In the experiment shown in Figure 6, `Our.ht` surpasses all other systems, showing the advantages of the proposed BLOB designs. Still, `Our.ht` is not as performant as `Our` because (1) `vmcache+exmap` is lightweight and more performant than hash table [55] and (2) `Our.ht` does not benefit from virtual memory aliasing. We will demonstrate the differences further in Section V-E.

C. Evaluation of Metadata Operations

Description. In the experiment illustrated in Figure 7, we compare the efficiency of metadata operations between our approach and file systems. To do that, we either retrieve the Blob State of 10 consecutive BLOBs or call `fstat()` on ten

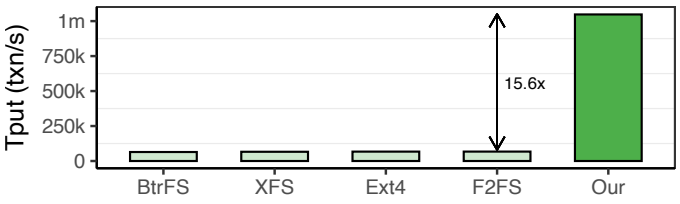


Fig. 7. Metadata operations in our approach vs. file systems

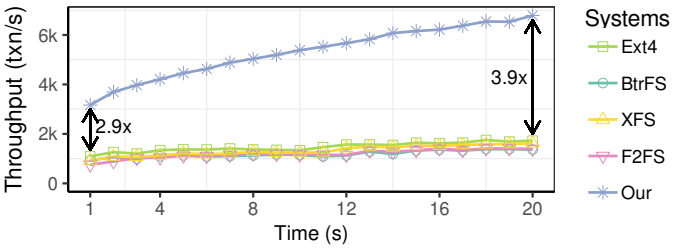


Fig. 9. BLOB Read evaluation (cold cache)

consecutive files in all file systems. We do not evaluate the competitor DBMSs because they are not performant enough as shown in previous experiments. The BLOB payload size in this experiment is 100KB.

Result. As being shown, all file systems have similar performance. Our DBMS provides 15.6× more throughput than all file systems. This is because our approach maintains all BLOB metadata in a B-Tree index that provides efficient lookup/scan queries, while metadata operations of file systems are very slow [1, 2], resulting in significant differences.

D. Evaluation of Proposed Extent Data Structure

Description. This experiment evaluates the proposed physical storage format using real read-only Wikipedia analytic datasets. First, we collect English Wikipedia analytic data, specifically the article size and their corresponding views, and build a database based on this distribution. The total size of all articles in this experiment is 23GB. During the initial phase, we insert random data according to the article sizes. In the benchmark phase, we pick a random article according to the article views and execute a `memcpy()` to simulate the article read. Similar to the previous experiment, we do not evaluate the widely-used DBMSs (i.e., PostgreSQL, MySQL, and SQLite). We run all benchmarks in two modes: when all data resides in memory (hot cache) and when all data is evicted. Because this is a read-only workload, the file system journal is unlikely to affect system performance, and thus, we do not evaluate Ext4 journal mode.

Hot cache experiment. In the experiment illustrated in Figure 8, we keep the page cache (or buffer manager) untouched after loading initial data. This figure shows that our DBMS outperforms all file systems by at least 40%. There are two reasons. First, the overheads of `fstat`, `open`, and `close` in file systems are significant, while our approach does not suffer from that as Section V-B shows. Second, `pread()` in file systems causes extra memory copy from kernel space to user space, while we use virtual memory aliasing which avoids one memory copy operation.

Cold cache experiment. Figure 9 depicts the performance of all systems when the page cache (or buffer pool) is empty. All file systems perform similarly, and Ext4 shows the highest performance among them. Our DBMS consistently outperforms all file systems, at least 2.9× at the start of the benchmark. This is because our proposed storage format is more simple than that of file systems, as described earlier in Section III. Therefore, our DBMS is better at utilizing the NVMe SSD compared to file systems, i.e., the upper bound read I/O of Ext4 is 59MB/s, while that of our DBMS is 174MB/s. And, when the buffer cache gets full quicker, our DBMS can serve more only-in-memory transactions, thus resulting in a 3.9× difference in throughput at the end of this experiment.

E. Benefits of vmcache & exmap

Description. `Our` constantly outperforms `Our.ht` throughout the experiments in Figure 6, proving the effectiveness of `vmcache+exmap` for BLOB operations. This is mainly because `vmcache+exmap` is better at read operations than the traditional hash table buffer cache. In this experiment, we further analyze the benefits of `vmcache+exmap` using a read-only in-memory YCSB workload. Similar to the logging experiment in Section V-B, we use a simple `memcpy()` as the BLOB read operator.

Result. As Figure 10 illustrates, both `Our` and `Our.ht` perform similarly for small BLOBs (100KB), and actually, the hash table approach is slightly faster than `vmcache+exmap`. This is because TLB flush is more expensive than `malloc()` & `memcpy()` when BLOBs are small. However, with bigger BLOBs, 1MB and 10MB, `Our` surpasses `Our.ht` signifi-

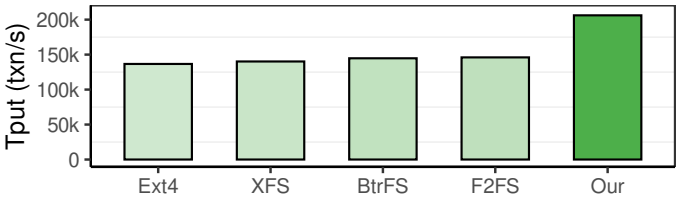


Fig. 8. BLOB Read evaluation (hot cache)

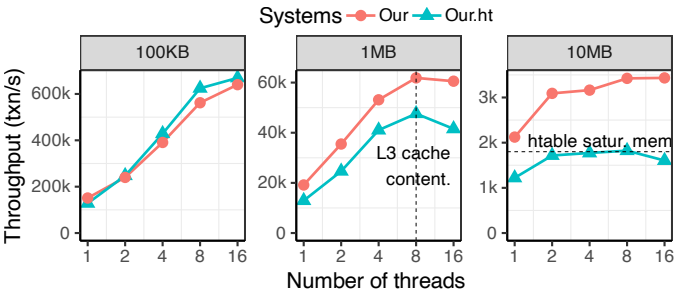


Fig. 10. vmcache+exmap vs. hash table-based buffer pool

cantly, up to 2.1× when the worker count is 16 and 10MB BLOB. There are two reasons; the first is that the cost of the `memcpy()` becomes considerable. Second, `malloc()` creates an anonymous memory block to be filled later by the `memcpy()`, causing page faults and allocation.

Key: memcpy() saturate memory hierarchy. Another observation is that the `Our.ht` can not scale to 16 workers when the BLOB size is either 1MB or 10MB. For the first case, i.e., when the BLOB size is 1MB and 16 workers, the combined size of the client-side buffer and the internal DBMS memory block for the BLOB exceeds L3 cache capacity (30MB in our machine), leading to contention at L3 cache. In the latter case, the 16-workers variant not only contends for the L3 cache, but it also saturates the available memory bandwidth because of two `memcpy()` calls.

F. Shared-Area Synchronization Overhead

Description. To evaluate the synchronization overhead, we run a YCSB read-only workload with 10MB BLOBs. We run the benchmark with 16 workers, and the maximum buffer size is 128GB. We use two worker-local sizes: 4MB and 16MB. With the 4MB setting, the local area is smaller than a BLOB, so transactions ask for free virtual addresses from the shared aliasing area, which incurs contention overhead. For the 16MB setting, because the worker-local area is bigger than the BLOB, no synchronization overhead occurs.

TABLE II
OVERHEAD OF SHARED-AREA SYNCHRONIZATION

Use shared area (wrk-local size)	txn/s	instruct.	cycles	kernel cycles	cache misses
Yes (4MB)	3,453	1,311k	14M	714k	14k
No (16MB)	3,477	1,321k	14M	703k	14k

Result. Table II depicts that the two variants perform similarly. All statistics such as the number of cycles, cache misses, instructions, and kernel cycles are all almost analogous. As a result, the throughput of both variants is similar. Therefore, the extra overhead caused by the synchronization on the shared area is trivial, while providing all necessary functionalities and also limiting the number of virtual addresses.

G. Extent Reusability

Description. In this experiment, we evaluate the free extent management design by constantly allocating and deleting objects. Specifically, we perform two operators: (1) allocate a BLOB of random size between 1MB and 10MB, and (2) delete a random BLOB. The allocation ratio is 80%, and the deletion ratio is 20%. Because allocation is 4× more frequent than deletion, the storage capacity will increase with time until the database/file system is full. We fix the database size (partition size in the case of file systems) to 32GB. For Ext4, the journal mode will reduce both the storage utilization and the throughput, hence we do not evaluate `Ext4.journal`, i.e., the Ext4 in this experiment is `Ext4.ordered`.

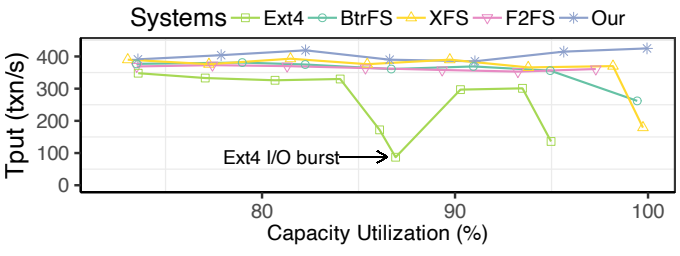


Fig. 11. Performance at different storage utilization. All systems eventually stop at full storage capacity. System performance is stable before the storage utilization reaches 80%.

Result. As Figure 11 shows, almost all file systems except F2FS drop in throughput when the storage nearly reaches its limit. This is because those file systems use complicated mechanisms to prevent fragmentation, which will not work well when the storage is almost full. Our extent recycling design, on the other hand, is lightweight and effective at reusing deallocated extents, thus can maintain system performance in different storage utilization states. This also proves that our design works reasonably well with mixed payload size, both in terms of performance and storage utilization.

H. BLOB Indexing

Description. To evaluate the Blob State index, we compare it with the 1K prefix index which presents the approach used in MySQL and PostgreSQL. The indexed data is the English Wikipedia [75], which contains many big articles. In this dataset, 43 percentile of the article is larger than 767 bytes, which is the indexing limit of MySQL [22]. For PostgreSQL limit, i.e., 8191 bytes [8], it is 95 percentile.

TABLE III
STATISTICS OF TWO INDEXING VARIANTS

Variant	miss (%)	build time (ms)	size (MB)	# leaf	throughput (lookup/s)
Blob State	0%	350	88	22k	443k
1K Prefix	17%	1,323	737	187k	438k

Result. As Table III shows, the Blob State index can store all articles in the index, i.e., `miss(%)=0`, while the prefix index can not serve 17% of all queries. This is because many documents have the same prefix, and the prefix index can only store one of them. Contrarily, the Blob State index can differentiate the articles using their full content and hence can index all articles. Furthermore, Blob State index creates significantly fewer leaf nodes (22k compared to 187k), resulting in faster construction time and lower storage consumption (3.8× and 8.4×, respectively). Besides, because we implement prefix compression which is preferable to prefix index [76], both indexes have the same tree height and thus provide similar lookup performance.

I. Real Write-Intensive Workload: Git Clone

Description. We contrast our approach with file systems using a simulated Git Clone benchmark. To do that, we collect the filesystem-level traces of the following git command:

$$git\ clone\ \text{--}depth\ 1\ git@github.com{:}torvalds/linux.git$$

We implement the simulated workload according to the traces and run the workload in single-threaded mode. The size of the experimental dataset is 1.28GB.

TABLE IV
GIT-CLONE BENCHMARK

System	time (ms)	instructions	kernel cycles
Our	906	65k	9k
Ext4.ordered	1,834	256k	81k
Ext4.journal	2,330	311k	108k
BtrFS	1,688	194k	66k
F2FS	2,112	236k	97k
XFS	1,464	188k	56k

Result. As illustrated in Table IV, file systems fail behind our DBMS significantly, largely because of the overheads of metadata operations: `fstat`, `close`, and especially `open`. Specifically, Ext4 ordered spends 36% of the execution time on `open` for file creation. This number for `fstat` and `close` are 4.8% and 1.6%, respectively. XFS performs the best because it only spends 36.6% of the execution time on system calls, the least compared to other file systems. Our approach, however, mitigates this overhead, i.e., replacing all three system calls with efficient B-Tree operations as demonstrated in Section V-C.

VI. Related Work

Ubiquitous BLOB storage: File systems. File systems have always been one of the core research areas of computer science, and they have adapted to manage objects of various sizes, including large objects. One benefit regarding large objects that file systems have over DBMSs is access simplicity and efficiency. That is, file systems support direct access to BLOB data, while DBMSs introduce additional overheads during BLOB operations such as transactional processing, logging, networking overhead, and many more.

Large object management in DBMSs. There is little work on BLOB management in DBMSs, which partly explains the prevalence of file systems. To our knowledge, there are only two academic works on this topic, both conducted before 2010, and these works primarily focused on the performance characteristics of DBMSs. In 2006, Sears et al. [19] argued that file systems are better than DBMSs for large objects, and provided some comparative BLOB benchmarks between SQL Server and NTFS file system. Subsequently, in 2008, another study delivered experimental results of different BLOB workloads of several DBMSs, discussing the performance bottlenecks of these systems [77]. SQLite is the only DBMS optimized for BLOB operations, and its team even suggest that SQLite can replace file systems for such tasks [78, 25]. Still, as discussed throughout this paper, there are many opportunities for improvement in SQLite, an observation that aligns with findings from numerous previous studies [79, 13, 2]. Nevertheless, with our proposed techniques, we challenge the conventional wisdom and prove that DBMSs can provide superior performance to file systems for BLOB management.

Networks. As Section V-B shows, networking is one primary overhead of MySQL and PostgreSQL, partially explaining why SQLite significantly outperforms the two client-server DBMSs. Existing works on improving the DBMS network stack fall into two categories: (1) avoid unnecessary computation in the network stack and (2) utilize modern hardware. Some notable techniques for the first category include a new data serialization method for large result sets [80], pushing DBMS logic to the kernel space to mitigate the networking overhead of DBMS proxy [81]. For the second category, some studies focused on utilizing RDMA for remote data accesses [82, 83, 84, 85] or leveraging NVMe over Fabrics [86, 87, 88]. One particular work that can placed in both categories: Fent et al. [89] proposes to replace conventional network protocols (e.g., TCP over Ethernet) with a novel communication library that provides unified APIs for adaptive selection of RDMA and Shared Memory. These techniques can enhance BLOB access over the network, an area we will explore in upcoming research.

DBMS-backed file systems. There is limited work on file systems backed by DBMSs, with the Oracle Database Filesystem [35] as a notable exception. Aligning with our approach, Oracle DBFS utilizes the FUSE library to provide POSIX-standard file system interfaces to connect to the DBMSs. It differs from our approach in that Oracle DBFS essentially acts as a translation layer from file APIs to DBMS interface (i.e., PL/SQL procedure calls), while our solution provides direct data accesses identical to file systems.

Aging and fragmentation. All systems supporting variable-sized objects suffer from the *aging* problem, i.e. performance can decline with time in some workloads because of the increasing fragmentation. For example, after the application allocates lots of small BLOBs and deletes most of them, the storage system may struggle to locate a suitable extent for a huge BLOB allocation. We note that system aging is an active research topic in file system community [90, 91, 92, 93, 94]. Despite that, file system aging is still not a solved problem [92, 93], and some recent works only tried to mitigate it [92, 94]. We believe that, in principle, out-of-place write policy can solve the aging problem. The core idea is to decouple logical PID from the on-storage physical address. Consequently, the DBMS can allocate every extent as new and map those PIDs with the available physical addresses in secondary storage. Because it is a significant topic, we plan to investigate it in the future.

VII. Summary

In this paper, we demonstrate that DBMSs can be more efficient than file systems in handling BLOBs. To achieve this, we introduce a comprehensive design for allocating and logging large objects in DBMSs. Our performance study shows that our proposed approach successfully outperforms many popular file systems while ensuring transactional consistency and durability for large objects. Moreover, FUSE integration allows external apps to access BLOBs similarly to file systems, paving the way toward a unified storage system for objects of arbitrary size. Our implementation is open source and available at https://github.com/leanstore/leanstore/tree/blob.

References

[1] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter, "Betrfs: A right-optimized write-optimized file system," in *FAST*. USENIX Association, 2015, pp. 301–315.

[2] K. P. Gaffney, M. Prammer, L. C. Brasfield, D. R. Hipp, D. R. Kennedy, and J. M. Patel, "Sqlite: Past, present, and future," *Proc. VLDB Endow.*, vol. 15, no. 12, pp. 3535–3547, 2022.

[3] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux symposium*, vol. 2. Citeseer, 2007, pp. 21–33.

[4] "Ext4 Howto," https://ext4.wiki.kernel.org/index.php/Ext4_Howto, 2019.

[5] "ext4(5) — Linux manual page," https://man7.org/linux/man-pages/man5/ext4.5.html, 2023.

[6] "PostgreSQL TOAST format," https://www.postgresql.org/docs/current/storage-toast.html, 2023.

[7] "PostgreSQL Release notes 9.3," https://www.postgresql.org/docs/9.3/release-9-3.html, 2023.

[8] "Postgresql index tuple size limit," https://github.com/postgres/postgres/blob/master/src/include/access/itup.h#L71, 2023.

[9] "PostgreSQL pg_largeobject," https://www.postgresql.org/docs/current/catalog-pg-largeobject.html, 2023.

[10] "Large object in PostgreSQL," https://pgpedia.info/l/large-object.html, 2023.

[11] "Sqlite file format," https://www.sqlite.org/fileformat2.html, 2004.

[12] "Limits In SQLite," https://www.sqlite.org/limits.html, 2023.

[13] "Sqlite clustered indexes and the without rowid optimization," https://www.sqlite.org/withoutrowid.html, 2023.

[14] "SQLite WAL mode," https://sqlite.org/wal.html, 2022.

[15] D. Korotkevitch, *Pro SQL Server Internals*. Apress, 2016.

[16] "SQL Server: binary and varbinary (Transact-SQL)," https://learn.microsoft.com/en-us/sql/t-sql/data-types/binary-and-varbinary-transact-sql, 2023.

[17] "Sqlserver error code 1919," http://www.sql-server-helper.com/error-messages/msg-1501-2000.aspx, 2023.

[18] "Sqlserver create index," https://learn.microsoft.com/en-us/sql/t-sql/statements/create-index-transact-sql, 2023.

[19] R. Sears, C. van Ingen, and J. Gray, "To BLOB or not to BLOB: large object storage in a database or a filesystem?" *CoRR*, vol. abs/cs/0701168, 2007.

[20] "Externally Stored Fields in MySQL/InnoDB," https://dev.mysql.com/doc/refman/8.0/en/innodb-row-format.html, 2023.

[21] "MySQL Data Type Storage Requirements," https://dev.mysql.com/doc/refman/8.0/en/storage-requirements.html, 2023.

[22] "Mysql index prefix limits," https://dev.mysql.com/doc/refman/8.0/en/column-indexes.html, 2023.

[23] "MySQL/InnoDB Redo Log for LOB," https://github.com/mysql/mysql-server/blob/8.0/storage/innobase/include/lob0zip.h#L78, 2023.

[24] "Large object in MySQL/InnoDB," https://www.percona.com/blog/how-innodb-handles-text-blob-columns/, 2023.

[25] "SQLite: 35% Faster Than The Filesystem," https://www.sqlite.org/fasterthanfs.html, 2022.

[26] H. Mühleisen, "Cidr keynote," https://www.youtube.com/watch?v=dv4A2LIFG80?t=1811, 2023.

[27] M. Cao, T. Y. Tso, B. Pulavarty, S. Bhattacharya, A. Dilger, and A. Tomas, "State of the art: Where we are with the ext3 filesystem," in *Proceedings of the Ottawa Linux Symposium (OLS)*. Citeseer, 2005, pp. 69–96.

[28] M. Haubenschild, C. Sauer, T. Neumann, and V. Leis, "Rethinking logging, checkpoints, and recovery for high-performance storage engines," in *SIGMOD Conference*. ACM, 2020, pp. 877–892.

[29] J. Park, G. Oh, and S. Lee, "SQL statement logging for making sqlite truly lite," *Proc. VLDB Endow.*, vol. 11, no. 4, pp. 513–525, 2017.

[30] G. Haas, M. Haubenschild, and V. Leis, "Exploiting directly-attached nvme arrays in DBMS," in *CIDR*. www.cidrdb.org, 2020.

[31] D. Kim, C. Park, S. Lee, and B. Nam, "Bolt: Barrier-optimized lsm-tree," in *Middleware*. ACM, 2020, pp. 119–133.

[32] M. Kang, S. Choi, G. Oh, and S. W. Lee, "2r: Efficiently isolating cold pages in flash storages," *Proc. VLDB Endow.*, vol. 13, no. 11, pp. 2004–2017, 2020.

[33] "AWS S3 Actions," https://docs.aws.amazon.com/AmazonS3/latest/API/API_Operations.html, 2023.

[34] D. S. Hirschberg, "A class of dynamic memory allocation algorithms," *Communications of the ACM*, vol. 16, no. 10, pp. 615–618, 1973.

[35] K. Kunchithapadam, W. Zhang, A. Ganesh, and N. Mukherjee, "Oracle database filesystem," in *SIGMOD Conference*. ACM, 2011, pp. 1149–1160.

[36] "Tesseract ocr," https://github.com/tesseract-ocr/tesseract, 2023.

[37] R. Smith, "An overview of the tesseract ocr engine," in *Ninth international conference on document analysis and recognition (ICDAR 2007)*, vol. 2. IEEE, 2007, pp. 629–633.

[38] G. Bradski, "The opencv library." *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, vol. 25, no. 11, pp. 120–123, 2000.

[39] "io — Core tools for working with streams," https://docs.python.org/3.11/library/io.html, 2023.

[40] "Filesystem in Userspace," https://github.com/libfuse/libfuse, 2023.

[41] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To FUSE

or not to FUSE: performance of user-space file systems," in *FAST*. USENIX Association, 2017, pp. 59–72.

[42] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: Artful indexing for main-memory databases," in *ICDE*. IEEE Computer Society, 2013, pp. 38–49.

[43] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized business review*, p. 21260, 2008.

[44] D. Yaga, P. Mell, N. Roby, and K. Scarfone, "Blockchain technology overview," *CoRR*, vol. abs/1906.11078, 2019.

[45] "PostgreSQL: Indexes on Expressions," https://www.postgresql.org/docs/current/indexes-expressional.html, 2024.

[46] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[47] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, 1981.

[48] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *SOSP*. ACM, 2013, pp. 18–32.

[49] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil, "A critique of ANSI SQL isolation levels," in *SIGMOD Conference*. ACM Press, 1995, pp. 1–10.

[50] A. D. Fekete, E. J. O'Neil, and P. E. O'Neil, "A read-only transaction anomaly under snapshot isolation," *SIGMOD Rec.*, vol. 33, no. 3, pp. 12–14, 2004.

[51] R. Ramakrishnan and J. Gehrke, *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[52] Z. Guo, K. Wu, C. Yan, and X. Yu, "Releasing locks as early as you can: Reducing contention of hotspots by violating two-phase locking," in *SIGMOD Conference*. ACM, 2021, pp. 658–670.

[53] L.-D. Nguyen, S. W. Lee, and B. Nam, "In-page shadowing and two-version timestamp ordering for mobile dbmss," *Proc. VLDB Endow.*, vol. 15, no. 11, pp. 2402–2414, 2022.

[54] C. Ye, W. Hwang, K. Chen, and X. Yu, "Polaris: Enabling transaction priority in optimistic concurrency control," *Proc. ACM Manag. Data*, vol. 1, no. 1, pp. 44:1–44:24, 2023.

[55] V. Leis, A. Alhomssi, T. Ziegler, Y. Loeck, and C. Dietrich, "Virtual-memory assisted buffer management," *Proc. ACM Manag. Data*, vol. 1, no. 1, pp. 7:1–7:25, 2023.

[56] G. Graefe, H. Volos, H. Kimura, H. A. Kuno, J. Tucek, M. Lillibridge, and A. C. Veitch, "In-memory performance for big data," *PVLDB*, vol. 8, no. 1, pp. 37–48, 2014.

[57] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann, "Leanstore: In-memory data management beyond main memory," in *ICDE*. IEEE Computer Society, 2018, pp. 185–196.

[58] T. Neumann and M. J. Freitag, "Umbra: A disk-based system with in-memory performance," in *CIDR*. www.cidrdb.org, 2020.

[59] F. M. Schuhknecht, J. Dittrich, and A. Sharma, "RUMA has it: Rewired user-space memory access is possible!" *Proc. VLDB Endow.*, vol. 9, no. 10, pp. 768–779, 2016.

[60] A. Crotty, V. Leis, and A. Pavlo, "Are you sure you want to use MMAP in your database management system?" in *CIDR*. www.cidrdb.org, 2022.

[61] "Pinterest help center: Review ad specs," https://help.pinterest.com/en/business/article/pinterest-product-specs, 2023.

[62] "LinkedIn: Media file types," https://www.linkedin.com/help/linkedin/answer/a564109, 2023.

[63] "How to post photos or GIFs on Twitter," https://help.twitter.com/en/using-twitter/tweeting-gifs-and-pictures, 2023.

[64] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *Proc. VLDB Endow.*, vol. 7, no. 10, pp. 865–876, 2014.

[65] A. Alhomssi, M. Haubenschild, and V. Leis, "The evolution of leanstore," in *BTW*, ser. LNI, vol. P-331. Gesellschaft für Informatik e.V., 2023, pp. 259–281.

[66] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[67] "PostgreSQL source code," https://github.com/postgres/postgres/tree/REL_15_3, 2023.

[68] "MySQL source code," https://github.com/mysql/mysql-server/tree/mysql-cluster-8.0.33, 2023.

[69] "SQLite source code," https://github.com/sqlite/sqlite/tree/version-3.40.1, 2022.

[70] M. Raasveldt and H. Mühleisen, "Duckdb: an embeddable analytical database," in *SIGMOD Conference*. ACM, 2019, pp. 1981–1984.

[71] "PostgreSQL Server Configuration," https://www.postgresql.org/docs/15/runtime-config-resource.html, 2023.

[72] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS file system," in *USENIX Annual Technical Conference*. USENIX Association, 1996, pp. 1–14.

[73] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: the linux b-tree filesystem," *ACM Trans. Storage*, vol. 9, no. 3, p. 9, 2013.

[74] C. Lee, D. Sim, J. Y. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *FAST*. USENIX Association, 2015, pp. 273–286.

[75] "enwiki dump progress," https://dumps.wikimedia.org/enwiki/latest/, Jun. 2023.

[76] R. Bayer and K. Unterauer, "Prefix b-trees," *ACM Trans. Database Syst.*, vol. 2, no. 1, pp. 11–26, 1977.

[77] S. Stancu-Mara and P. Baumann, "A comparative benchmark of large objects in relational databases," in *IDEAS*, ser. ACM International Conference Proceeding Series, vol. 299. ACM, 2008, pp. 277–284.

[78] "What If OpenDocument Used SQLite?" https://www.sqlite.org/affcase1.html, 2023.

[79] "Internal Versus External BLOBs in SQLite," https://

www.sqlite.org/intern-v-extern-blob.html, 2011.

[80] M. Raasveldt and H. Mühleisen, "Don't hold my data hostage - A case for client protocol redesign," *Proc. VLDB Endow.*, vol. 10, no. 10, pp. 1022–1033, 2017.

[81] M. Butrovich, K. Ramanathan, J. Rollinson, W. S. Lim, W. Zhang, J. Sherry, and A. Pavlo, "Tigger: A database proxy that bounces with user-bypass," *Proc. VLDB Endow.*, vol. 16, no. 11, pp. 3335–3348, 2023.

[82] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian, "The end of slow networks: It's time for a redesign," *Proc. VLDB Endow.*, vol. 9, no. 7, pp. 528–539, 2016.

[83] F. Li, S. Das, M. Syamala, and V. R. Narasayya, "Accelerating relational databases by leveraging remote memory and RDMA," in *SIGMOD Conference*. ACM, 2016, pp. 355–370.

[84] T. Ziegler, J. Nelson-Slivon, V. Leis, and C. Binnig, "Design guidelines for correct, efficient, and scalable synchronization using one-sided RDMA," *Proc. ACM Manag. Data*, vol. 1, no. 2, pp. 131:1–131:26, 2023.

[85] T. Ziegler, V. Leis, and C. Binnig, "RDMA communciation patterns," *Datenbank-Spektrum*, vol. 20, no. 3, pp. 199–210, 2020.

[86] H. Li, S. Jiang, C. Chen, A. Raina, X. Zhu, C. Luo, and A. Cidon, "Rubbledb: Cpu-efficient replication with nvme-of," in *USENIX Annual Technical Conference*. USENIX Association, 2023, pp. 689–703.

[87] T. A. Nguyen, H. Jeon, D. Han, D. Bae, Y. Yu, K. Kim, S. Park, J. Jeong, and B. Nam, "Nvme-driven lazy cache coherence for immutable data with nvme over fabrics," in *CLOUD*. IEEE, 2023, pp. 394–400.

[88] D. Han and B. Nam, "Improving access to HDFS using nvmeof," in *CLUSTER*. IEEE, 2019, pp. 1–2.

[89] P. Fent, A. van Renen, A. Kipf, V. Leis, T. Neumann, and A. Kemper, "Low-latency communication for fast DBMS using RDMA and shared memory," in *ICDE*. IEEE, 2020, pp. 1477–1488.

[90] A. Conway, E. Knorr, Y. Jiao, M. A. Bender, W. Jannen, R. Johnson, D. E. Porter, and M. Farach-Colton, "Filesystem aging: It's more usage than fullness," in *HotStorage*. USENIX Association, 2019.

[91] A. Conway, A. Bakshi, Y. Jiao, W. Jannen, Y. Zhan, J. Yuan, M. A. Bender, R. Johnson, B. C. Kuszmaul, D. E. Porter, and M. Farach-Colton, "File systems fated for senescence? nonsense, says science!" in *FAST*. USENIX Association, 2017, pp. 45–58.

[92] R. Kadekodi, S. Kadekodi, S. Ponnapalli, H. Shirwadkar, G. R. Ganger, A. Kolli, and V. Chidambaram, "Winefs: a hugepage-aware file system for persistent memory that ages gracefully," in *SOSP*. ACM, 2021, pp. 804–818.

[93] S. Kadekodi, V. Nagarajan, and G. R. Ganger, "Geriatrix: Aging what you see and what you don't see. A file system aging approach for modern storage systems," in *USENIX Annual Technical Conference*. USENIX Association, 2018, pp. 691–704.

[94] J. Park and Y. I. Eom, "Filesystem fragmentation on modern storage systems," *ACM Transactions on Computer Systems*, 2023.