# Understanding the Performance Implications of the Design Principles in Storage-Disaggregated Databases

XI PANG, Purdue University, USA
JIANGUO WANG, Purdue University, USA

Storage-compute disaggregation has recently emerged as a novel architecture in modern data centers, particularly in the cloud. By decoupling compute from storage, this new architecture enables independent and elastic scaling of compute and storage resources, potentially increasing resource utilization and reducing overall costs. To best leverage the disaggregated architecture, a new breed of database systems termed *storage-disaggregated databases* has recently been developed, such as Amazon Aurora, Microsoft Socrates, Google AlloyDB, Alibaba PolarDB, and Huawei Taurus. However, little is known about the effectiveness of the design principles in these databases since they are typically developed by industry giants, and only the overall performance results are presented without detailing the impact of individual design principles. As a result, many critical research questions remain unclear, such as the performance impact of storage-disaggregation, the log-as-the-database design, shared-storage, and various log-replay methods.

In this paper, we investigate the performance implications of the design principles that are widely adopted in storage-disaggregated databases for the first time. As these databases were usually not open-sourced, we have made a significant effort to implement a storage-disaggregated database prototype based on PostgreSQL v13.0. By fully controlling and instrumenting the codebase, we are able to selectively enable and disable individual optimizations and techniques to evaluate their impact on performance in various scenarios. Furthermore, we open-source our storage-disaggregated database prototype for use by the broader database research community, fostering collaboration and innovation in this field.

CCS Concepts: • **Information systems → DBMS engine architectures**; **Relational parallel and distributed DBMSs**.

Additional Key Words and Phrases: Disaggregated Databases, Cloud-Native Databases

**ACM Reference Format:**
Xi Pang and Jianguo Wang. 2024. Understanding the Performance Implications of the Design Principles in Storage-Disaggregated Databases. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 180 (June 2024), 26 pages. https://doi.org/10.1145/3654983

## 1 INTRODUCTION

Recently, storage-compute disaggregation has risen as a widely adopted architecture in cloud data centers such as Amazon AWS and Microsoft Azure [11, 28]. In contrast to traditional data centers composed of monolithic "converged" servers where compute and storage are tightly coupled in the same physical servers, in the new architecture of storage-compute disaggregation, compute and storage are separated and connected via networking. This design enables many advantages, such as independent and elastic scaling of compute and storage, increased resource utilization, reduced cost, and fast crash recovery [1, 11, 17, 28, 31].

Authors' addresses: Xi Pang, pang65@purdue.edu, Purdue University, West Lafayette, Indiana, USA; Jianguo Wang, csjgwang@purdue.edu, Purdue University, West Lafayette, Indiana, USA.


ACM 2836-6573/2024/6-ART180
https://doi.org/10.1145/3654983

Storage-compute disaggregation poses fundamental challenges to traditional database systems, as they were originally designed for monolithic servers that tightly integrate compute and storage. As a result, a new breed of databases, known as *storage-disaggregated databases*, has been designed specifically for this disaggregated architecture. Notable examples include Amazon Aurora [28], Microsoft Socrates [11], Google AlloyDB [1], Alibaba PolarDB [5, 14], and Huawei Taurus [17].

Storage-disaggregated databases generally embrace three innovative design principles that we summarized below:

**Design Principle P1: Software-level Disaggregation.** They perform *software-level disaggregation* that decouples the storage engine (e.g., logging and storage) from the compute engine (e.g., SQL layer, buffering, and transactions), where the storage engine runs on the storage node and the compute engine operates on the compute node [1, 11, 17, 28]. This design is important for storage-disaggregated databases to introduce new optimizations, such as the log-as-the-database mentioned below, to mitigate the network I/O overhead. However, the performance slowdown caused by disaggregation is not well understood.

**Design Principle P2: Log-as-the-Database.** To reduce the network I/O overhead between the storage engine and compute engine, besides using buffering, storage-disaggregated databases generally embrace the *log-as-the-database* design, which sends only write-ahead logs to the storage side upon transaction commit [1, 11, 17, 28], instead of sending the actual data pages as in traditional databases. This can reduce data movement over the network. The actual data pages are then asynchronously materialized from the storage side. However, the actual effectiveness of the log-as-the-database design remains unclear. For instance, [28] shows the size difference between logs and pages but overlooks that, even without the log-as-the-database approach, traditional databases do not immediately flush pages to storage upon transaction commit, because only dirty pages are flushed asynchronously.

**Design Principle P3: Shared-Storage Design.** Another crucial design principle in storage-disaggregated databases is the *shared-storage* (rather than conventional shared-nothing [23, 26]) design between multiple compute nodes [1, 11, 17, 28]. The shared-storage design enhances elasticity because it eliminates the need to copy or move data for newly added compute nodes as they can share the same data with existing compute nodes. However, this design presents additional challenges for the storage engine. Specifically, due to replication lag between the primary and secondary compute nodes, the storage engine must support *multi-version pages* to accommodate scenario where secondary compute nodes require access to older versions of a page. However, the performance implications of this design remain unknown.

**Motivation.** As we can see, there is a lack of study to investigate the effectiveness of the design principles in storage-disaggregated databases. This can be understandable as those systems are developed by industry. Their corresponding papers [1, 5, 11, 17, 28] often focus on presenting the *final* overall performance results, neglecting the impact of individual design principles.

**Research Questions.** As a result, many crucial research questions remain unanswered:

- **Q1** How much performance overhead is introduced by storage disaggregation? (Relevant to **P1**)
- **Q2** To what extent can buffering help in mitigating the performance degradation? (Relevant to **P1**)
- **Q3** How significant is the performance improvement (for both reads and writes) due to the log-as-the-database design? (Relevant to **P2**)
- **Q4** What is the performance impact caused by supporting multi-version pages in the shared-storage design? (Relevant to **P3**)
- **Q5** How does multi-version storage affect checkpointing? (Relevant to **P3**)

**Q6** How effective are various log-replay methods within the multi-version storage? (Relevant to **P3**)

**Overview.** In this paper, our goal is to understand the performance implications of the design principles within storage-disaggregated databases. In particular, we aim to address the above six research questions (Q1 to Q6) that remain unexplored.

Given that these databases, such as Amazon Aurora, Microsoft Socrates, Google AlloyDB, Alibaba PolarDB, and Huawei Taurus, are products of industry and usually closed-source,[1] we decided to spend a significant amount of time on implementing our own storage-disaggregated database prototype based on PostgreSQL v13.0. This hands-on implementation approach allows us to fully understand and investigate specific optimizations and techniques, providing insights into their effects on performance in different situations. This study leads to a collection of non-trivial findings that are important for enhancing storage-disaggregated databases.

**Contributions.** This paper makes the following contributions:

- We conduct the first in-depth experimental study that investigates the effectiveness of the design principles in storage-disaggregated databases by implementing and evaluating six different architectures to answer the above six research questions (Q1 to Q6). In particular, we focus on OLTP databases. We provide a series of insights that can guide the development of storage-disaggregated databases in the future. Overall, we believe this work significantly advances the understanding of storage-disaggregated databases.
- We open-source the disaggregated database platform that we built based on PostgreSQL v13.0 over the last few years.
  The platform incorporates six architectures (see §3) covering the key optimizations of major storage-disaggregated databases, including Amazon Aurora [28], Microsoft Socrates [11], Google AlloyDB [1], PolarDB [5, 14], Huawei Taurus [17], and Neon [4]. We believe this open-source platform is **particularly valuable in the field of disaggregated databases**, as these databases are often complicated and closed-source. By making it open-source, we can significantly lower the entry barrier for research in storage-disaggregated databases.

**Open-source.** The code is open-sourced at https://github.com/purduedb/OpenAurora/.

## 2 EXPERIMENTAL METHODOLOGY

In this section, we present the experimental methodology to conduct experiments.

First, we investigate the design principles sequentially in the order of P1 (Disaggregation), P2 (Log-as-the-Database), and P3 (Shared-Storage). This approach mirrors that of most storage-disaggregated databases, including Amazon Aurora [28], Microsoft Socrates [11], Google AlloyDB [1], Huawei Taurus [17], and Neon [4], which have applied these design principles in the same sequence. In particular, when examining P2, we assume that P1 has been implemented, as it would not make sense to study log-as-the-database without disaggregation. Similarly, our evaluation of P3 assumes that both P1 and P2 have been implemented. This sequence ensures that our experimental evaluation is not only more focused but also aligns closely with those industrial storage-disaggregated databases mentioned above.

Second, we evaluate the performance implications of the three design principles (P1, P2, and P3) through answering the six important research questions (Q1 to Q6), given their strong correlation

[1]The only open-source storage-disaggregated databases we are aware of are PolarDB [5, 14] and Neon [4]. However, we chose to implement our database prototype rather than using them for two reasons. First, both PolarDB and Neon became open-sourced in 2022 but our project started in 2021. Second, PolarDB is not based on the log-as-the-database principle, making it unsuitable for studying this principle.

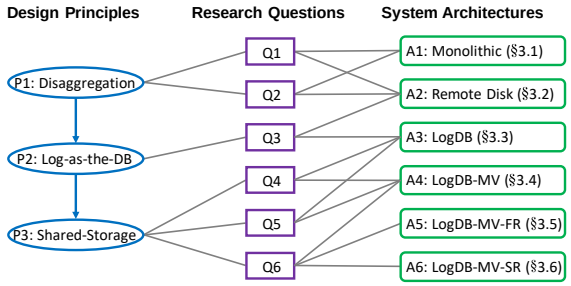



**Fig. 1.** Relationship of Design Principles, Research Questions, and System Architectures

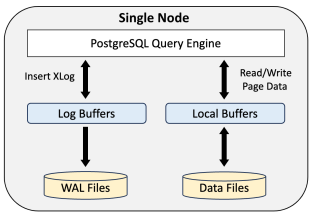



**Fig. 2.** Monolithic Architecture

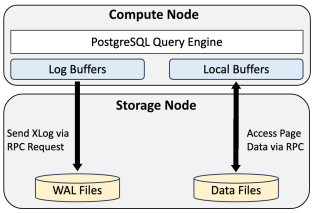



**Fig. 3.** Remote Disk Architecture

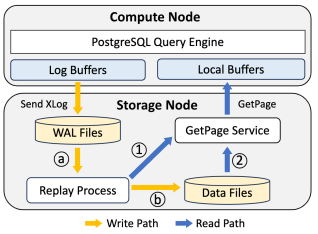



**Fig. 4.** Log-as-the-DB (LogDB)

with the principles, see Figure 1. Specifically, Q1 and Q2 are related to P1; Q3 is related to P2; and Q4 through Q6 are related to P3.

Third, to answer the research questions (Q1 to Q6), we experiment with six different storage-disaggregated database architectures (A1 to A6 shown in Figure 1). Except for A1, which is a monolithic PostgreSQL database, all the other architectures (A2 through A6) are implemented by us following existing storage-disaggregated databases.

Specifically, A2 is the remote disk architecture, which separates the PostgreSQL's compute engine from its storage engine, with the compute engine running on the compute node and the storage engine on the storage node. By comparing A1 and A2, we can answer questions Q1 and Q2 to understand the overhead of disaggregation and the effect of buffering. In A3, we add the optimization of log-as-the-database, which only sends logs to the storage node, while in A2, both logs and data pages are sent. By comparing A2 and A3, we can answer Q3 to know the effectiveness of the log-as-the-database design. In A4, we add the implementation of multi-version pages to support shared-storage among multiple compute nodes. By comparing A3 and A4, we can answer questions Q4 and Q5 to understand the effect of multi-version pages and the impact on checkpointing. A5 and A6 are based on A4, but they implement different strategies for replaying logs. By comparing A4, A5, and A6, we are able to answer Q6 to show the effectiveness of different log-replay methods within multi-version storage.

Figure 1 shows the relationship between the design principles, research questions, and system architectures.

## 3 SYSTEM ARCHITECTURES FOR STORAGE-DISAGGREGATED DATABASES

### 3.1 Monolithic Architecture

Figure 2 illustrates the architecture of PostgreSQL (v13.0), which represents the traditional monolithic database. It is a disk-based database running on a single node. It serves as a reference database when comparing with storage-disaggregated databases (detailed from §3.2 through §3.6) because

these databases are built based on PostgreSQL. For instance, in §3.2, we decouple PostgreSQL's storage engine from its query engine.

### 3.2 Remote Disk

The traditional monolithic database architecture is not well-suited for the disaggregated architecture. Thus, the first step in storage-disaggregated databases is to decouple the storage engine and query engine and run them on two different nodes connected through networking. Specifically, the storage engine operates on the storage node while the query engine runs on the compute node. This is the Remote Disk architecture in Figure 3.

The transaction execution logic in the Remote Disk architecture is the same as that of the monolithic architecture. However, the key difference is that all local disk reads and writes are routed to the remote disk via RPCs rather than to a local disk.

The advantages of this software-level disaggregation are twofold. First, it enables independent and elastic scaling of compute and storage. Second, it paves the way for introducing more optimizations, such as the log-as-the-database design discussed in §3.3.

### 3.3 Log-as-the-Database (LogDB)

The Remote Disk architecture incurs performance degradation due to networking overhead. To improve performance, in addition to employing buffering in the compute node, storage-disaggregated databases typically adopt the log-as-the-database design principle [1, 11, 28]. This is the LogDB architecture illustrated in Figure 4.

In this architecture, when a transaction is committed, it only sends the write-ahead logs (called xlogs[2] in PostgreSQL) to the storage node without sending the full data pages. The actual data pages are generated by replaying the logs at the storage node asynchronously (see steps ⓐ and ⓑ in Figure 4).

For the read path, the compute node will first check the local buffer. If there is a cache miss, it will fetch the missed pages from the storage node. If the requested pages have not been replayed yet, the storage node will replay the required logs on the fly (step ① in Figure 4) and return the requested pages upon completion (step ② in Figure 4).

Compared to the Remote Disk architecture, LogDB is expected to enhance write performance as it only sends the logs. However, its effect is non-trivial because the subtlety is that even in the Remote Disk architecture, the transmission of data pages occurs asynchronously because the dirty pages are not immediately flushed when a transaction is committed. They are only flushed to the remote disk when the local buffer (in the compute node) is full. Thus, it remains unclear whether (and when) the log-as-the-database design principle actually improves performance.

### 3.4 Log-as-the-Database with Multi-Version Storage (LogDB-MV)

Another important design principle in storage-disaggregated databases is the shared-storage design [1, 11, 17, 28] where multiple compute nodes share the same underlying storage. This is the LogDB-MV architecture as illustrated in Figure 5. In this architecture, there is a single primary (compute) node and multiple secondary (compute) nodes. Only the primary compute node supports read and write transactions, whereas all secondary nodes can only handle read-only transactions. Compared with the traditional shared-nothing architecture [23, 26], the shared-storage design enhances elasticity because when new compute nodes are added, there is no need to copy or move data, as all the compute nodes will share the same data.

[2]In this paper, the terms "logs" and "xlogs" are used interchangeably.

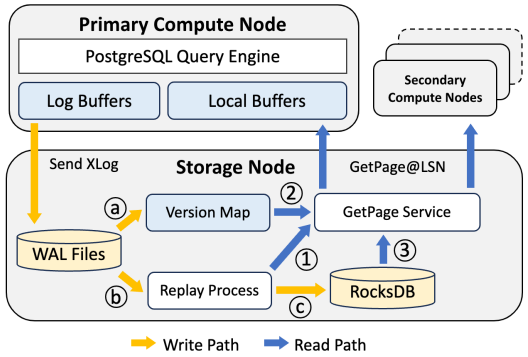



**Fig. 5.** Log-as-the-Database with Multi-Version Storage (LogDB-MV)

The shared-storage design introduces additional challenges for the storage engine. Specifically, the primary compute node sends xlogs to all secondary compute nodes for new database updates. Nonetheless, replication lag occurs due to networking overhead. Thus, secondary compute nodes might need to access older page versions by specifying a particular LSN. This operation is known as **GetPage@LSN** [11]. Consequently, the storage engine needs to support *multi-version pages* in storage-disaggregated databases.

In terms of the implementation, the storage node continuously replays the xlogs asynchronously and stores multiple versions for each page in RocksDB. We choose RocksDB as it is a mature and modern storage engine. While other storage-disaggregated databases have built their proprietary storage engines, they share similar ideas with our implementation.

In particular, after the storage node receives an xlog, it asynchronously checks which page modifications are included in this xlog. Afterward, the storage node appends this xlog's LSN and page id's mapping record to the "Version Map", which is a hashmap using the page id as its key and the version LSN list as its value (as depicted in step ⓐ in Figure 5). This "Version Map" is utilized by the GetPage@LSN request. Following this, the storage node replay process divides the xlog into several mini-xlogs, with each mini-xlog containing modifications for only one page. Subsequently, it replays multiple mini-xlogs to obtain multiple updated pages (step ⓑ in Figure 5). Finally, the storage node inserts the newly generated pages into RocksDB, using the page id combined with LSN as its key and the page content as its value (step ⓒ in Figure 5).

For the GetPage@LSN request, the storage node will first await the replay process until it has replayed xlogs that exceed the parameter LSN (step ① in Figure 5). Then, it will fetch the version LSN list from the Version Map using the request's parameter page id. By selecting from the list, the request will use the highest LSN that is smaller than or equal to the parameter LSN as its target version (step ② in Figure 5). Finally, the storage node will combine the page id and version LSN as its key to retrieve the target page from RocksDB (step ③ in Figure 5).

However, the performance implications of the shared-storage design principle remain unknown and complicated (that will be addressed in §5.3). This is because, on the one hand, supporting multiple versions introduces additional performance overhead because of the higher xlog replay overhead (as each xlog can now represent multiple pages) and the higher overhead when inserting pages into RocksDB (as the total number of pages increases considerably due to multi-versioning). On the other hand, multi-versioning can address the "torn page write" problem (as explained below), which will in turn improve I/O performance. Specifically, the "torn page write" problem arises due to the disparity in page sizes between databases and operating systems. For example, the page size

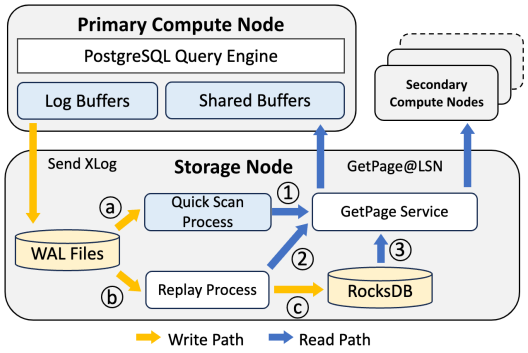



**Fig. 6.** Log-as-the-Database with Multi-Version Storage and Filtered Replay (LogDB-MV-FR)

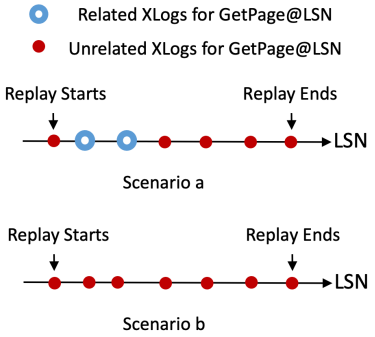



**Fig. 7.** XLog Replay Situations

in PostgreSQL is 8KB while the (atomic) page size in Linux is 4KB [6]. As a result, PostgreSQL introduces "full page write" [3] to solve the problem, where the entire page is recorded in the xlog if it is the first time being recorded after the latest checkpoint. This will unavoidably introduce extra overhead to the database. However, the "torn page write" problem is eliminated with the multi-version storage engine because it only appends new pages to the database instead of updating the pages in place [1, 28].

### 3.5 Log-as-the-Database with Multi-Version Storage and Filtered Replay (LogDB-MV-FR)

As mentioned in §3.4, replaying logs takes considerable time, especially in the multi-version storage setting. To improve the performance of replaying logs, various optimizations have been developed in existing storage-disaggregated databases. We refer to these optimizations as "Filtered Replay" (FR) [1, 28] and "Smart Replay" (SR) [4, 5], and we will explain them in §3.5 and §3.6, respectively. We will evaluate their effectiveness in §5.4.

We begin by discussing the motivation behind "Filtered Replay" (FR), which is used in Aurora [28] and AlloyDB [1]. Recall that in LogDB-MV (Figure 5), when the GetPage@LSN request reaches the storage node, it replays all the logs up to the requested LSN before returning the page. However, this process can result in many wasted xlog replays because some xlogs may not be relevant to the requested page. Figure 7a illustrates a situation where the relevant xlogs for the requested page are placed at the beginning of the xlog pipeline. In this scenario, only replaying the first three xlogs is

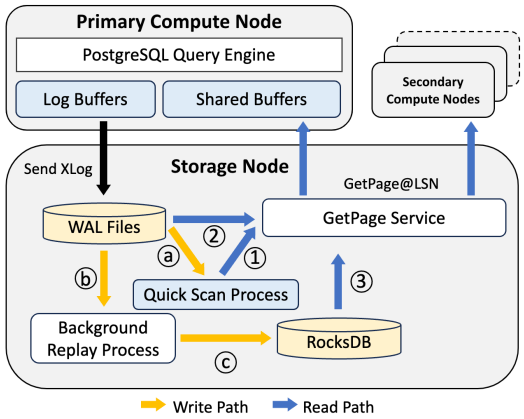



**Fig. 8.** Log-as-the-Database with Multi-Version Storage and Smart Replay (LogDB-MV-SR)

sufficient to ensure that GetPage@LSN obtains the correct page version. Figure 7b shows another case where all the xlogs to be replayed are irrelevant to the requested page, resulting in wasted time. In both scenarios, performance can be enhanced by filtering out non-relevant xlogs.

Figure 6 illustrates the system architecture (LogDB-MV-FR) with filtered replay. We have introduced a "Quick Scan" process in the storage node to accelerate "Version Map" update, and therefore filter unnecessary xlog replaying in GetPage@LSN request. For example, for Figure 7a, "Quick Scan" can quickly identify that storage node can get target page's first version by replaying the second xlog and get second version by replaying the third xlog. Without necessary to replay xlog, "Quick Scan" process can rapidly check the page ids related to each xlog, and update Version Map (step ⓐ in Figure 6). And the "Replay" process sequentially replays xlogs in the background (step ⓑ in Figure 6) and inserts the newly generated page versions into RocksDB (step ⓒ in Figure 6).

For the read path, when a GetPage@LSN request arrives at the storage node, it first awaits the "Quick Scan" process to sequentially check xlogs with LSNs exceeding the parameter LSN. Since the "Quick Scan" process advances LSN quickly, the waiting time is significantly shorter than in LogDB-MV. Afterward, the storage node retrieves this page's version list from the "Quick Scan" process's Version Map and determines which version it should adopt (step ① in Figure 6). Next, GetPage@LSN only waits for the "Replay" process to reach that version's LSN (step ② in Figure 6). Finally, the GetPage@LSN request retrieves the target page version from RocksDB using the combined page id and version LSN as its key (step ③ in Figure 6).

For example, if there is a request to retrieve page $A$ with LSN 200 and the "Quick Scan" process determines that the last related xlog for page $A$ is LSN 150, then the GetPage@200 will only wait for the replay process to replay up to LSN 150 (rather than 200) before returning the page.

However, it remains unclear on the effectiveness of the Filtered Replay approach. This will be addressed in §5.4.

### 3.6 Log-as-the-Database with Multi-Version Storage and Smart Replay (LogDB-MV-SR)

Although the LogDB-MV-FR architecture can skip many unnecessary xlogs to improve performance, there are still some unrelated xlogs that are replayed to serve GetPage@LSN. For example, in Figure 7a, to get target page's second version, the replay process needs to replay the first xlog which is unrelated with target page. As a result, there is another log replay approach (used in

PolarDB [5, 14] and Neon [4]) that we refer to as "Smart Replay". This is the LogDB-MV-SR architecture shown in Figure 8. It only replays the necessary xlogs related to the requested page. It does not replay any unnecessary xlogs. An additional advantage of LogDB-MV-SR is its capability for parallel replay, because different pages' version lists can be replayed by different processes in parallel instead of replaying all xlogs in sequence by one replay process.

When the storage node receives an xlog, the "Quick Scan" process asynchronously stores this xlog's mapping information in the Version Map, similar to LogDB-MV-FR (step ⓐ in Figure 8). Following this, several background replay processes are initiated. Each of these processes extracts an LSN list for a specific page id from the Version Map and executes the xlog replay operations in parallel (step ⓑ in Figure 8). These background replay processes insert the newly generated pages into RocksDB (step ⓒ in Figure 8). Additionally, they inform the Version Map of their progresses in replaying the LSN list corresponding to each page id.

For the read path, when a GetPage@LSN request arrives at the storage node, it will first check and wait for the "Quick Scan" process to examine xlogs until it exceeds its request parameter LSN and get its target version LSN. Then, it checks whether the target version has already been generated and saved in RocksDB (step ① in Figure 8). If the target version has not been generated, the GetPage@LSN request will collect xlogs from the disk and replay the related xlogs by itself (step ② in Figure 8), subsequently inserting the newly generated page versions into RocksDB. If the target page version has already been generated, GetPage@LSN will directly retrieve its target page version from RocksDB (step ③ in Figure 8).

However, prior studies have not evaluated the effectiveness of the Smart Replay approach. This will be addressed in §5.4.

## 4 EXPERIMENT SETUP

### 4.1 Experimental Platform

By default, we use a four-node setup that includes one compute node and three storage nodes. This configuration aligns with disaggregated databases like Aurora, Socrates, AlloyDB, PolarDB, and Neon, which only have one writer. We chose a 3-node storage setup because Socrates, AlloyDB, PolarDB, and Neon use 3-way replication. However, we also conducted experiments on a 6-node storage cluster (in §5.5.5), considering that Aurora uses 6-way replication. Each storage node in our set up is equipped with an Intel Xeon Platinum 8368 CPU (2.4 GHz), 188GB of DRAM, and a 1.5TB NVMe SSD. The compute node includes an Intel Xeon Gold 6330 CPU (2.0 GHz), 250GB of DRAM, and a 1.5TB NVMe SSD.

In some experiments (e.g., Figure 16), we expanded to using up to 15 read-only compute nodes, in line with the maximum support of 15 replicas by Aurora, Socrates, AlloyDB, PolarDB, and Neon. That is, the total number of compute nodes is 16. Each replica in these experiments is equipped with an Intel Xeon Silver CPU (2.3 GHz), 64GB DRAM, and a 900GB NVMe SSD.

All these nodes run Linux Ubuntu 22.04, and they are connected through a 10Gb TCP/IP network. Note that our experimental platform only includes SSDs (instead of HDDs) because SSDs are mature enough to be widely used in modern high-performance databases [11, 20, 37] and data centers [2, 11].

### 4.2 Datasets and Benchmarks

In this experimental study, we use two widely used OLTP benchmarks, namely SysBench [7] and TPC-C [8] to understand and investigate the performance implications of the design principles in storage-disaggregated databases. Due to space constraints, we mainly focus on the results on SysBench and only present results on TPC-C in §5.5.1.

**Table 1.** Buffer Size and Hit Ratio in SysBench

| **Buffer Size** | 1MB | 700MB | 8GB | 64GB |
|---|---|---|---|---|
| **Hit Ratio** | 0.1% | 40% | 80% | 99.5% |

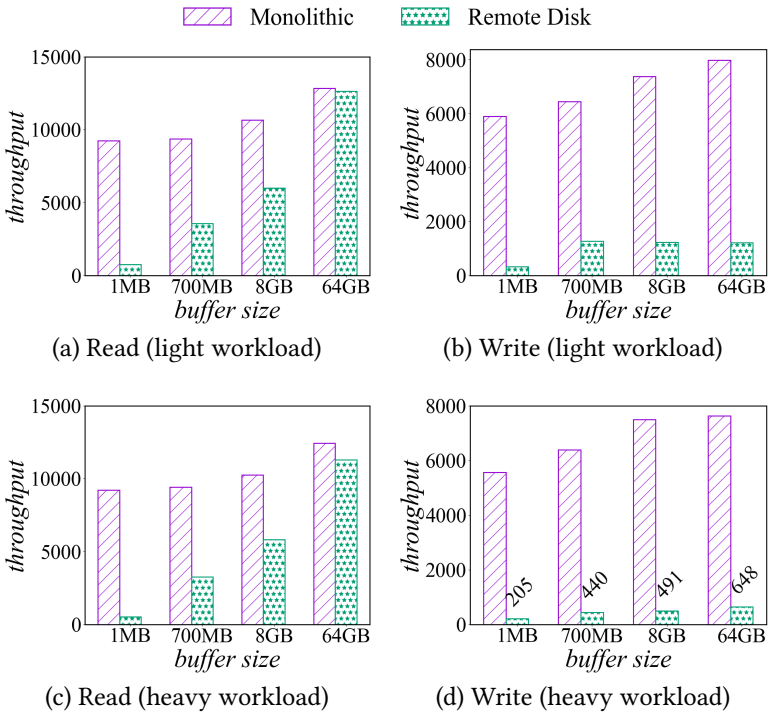



**Fig. 9.** Monolithic vs. Remote Disk on SSD

For SysBench, we prepare 2,000 tables, and each table contains 200,000 tuples. The total size of the SysBench dataset is 96GB. We use 16 threads to run the benchmark. For the buffer size in the compute node, we utilize buffer sizes of 1MB, 700MB, 8GB, and 64GB in our experiments. These correspond to buffer hit ratios of 0.1%, 40%, 80%, and 99.5%, respectively, as shown in Table 1.

## 5 EXPERIMENTAL RESULTS

### 5.1 Addressing Research Questions Q1 and Q2

**Overall Results on Reads and Writes (Figure 9).** Figure 9 compares the performance of the Monolithic and Remote Disk architectures with varying buffer sizes. We run SysBench under two scenarios: light workload and heavy workload. For the light workload, there is a 30-second idle time between each SysBench execution. In contrast, the heavy workload has no idle time between consecutive SysBench runs. There are several interesting observations from the figure.

First, storage disaggregation leads to a significant performance reduction for both reads (**16.4X**) and writes (**17.9X**). This is due to the networking overhead, as networking is slower than local SSDs.

Second, using a buffer in the compute node enhances the read performance for the Remote Disk architecture, as depicted in Figure 9a. The larger the buffer size, the better the performance.

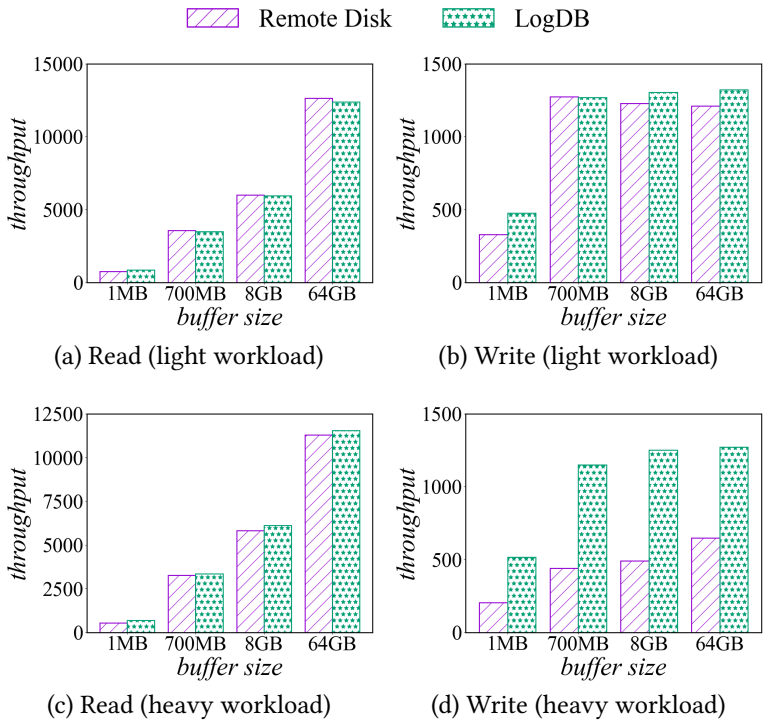



**Fig. 10.** Remote Disk vs. LogDB

However, it is interesting that the write performance does not improve much when the buffer size exceeds 700MB, as shown in Figure 9b. This happens because, with a light workload, the dirty pages generated are fewer than 700MB in this experiment. As a result, increasing the buffer size does not improve the write performance since all the dirty pages are cached. We then conducted another experiment with a heavy workload, as presented in Figure 9d, where more dirty pages are involved. The results indicate that a larger buffer size corresponds to better performance.

However, irrespective of the workload (whether light or heavy), Figure 9b and Figure 9d demonstrate a significant write performance gap between the Monolithic and Remote Disk architectures. For instance, even with a 64GB local buffer (resulting in a 99.5% cache hit ratio), the write performance in the Remote Disk decreases by **6.6X** and **11.8X** compared to the Monolithic architecture. This drop in performance is because the compute node always has to transfer xlogs to the storage node via networking for committed transactions, regardless of the size of the compute node's local buffer. Consequently, the speed disparity between the local SSD and remote storage is the primary factor causing the low write performance in the Remote Disk architecture.

### 5.2 Addressing Research Question Q3

In this experiment, we investigate the impact of log-as-the-database design principle.

**Overall Results on Reads and Writes (Figure 10).** Figure 10 presents the performance comparison between the Remote Disk and LogDB architectures.

Figure 10 shows that the read performance between the two architectures is comparable. However, LogDB significantly enhances write performance, especially under heavy workloads, compared to

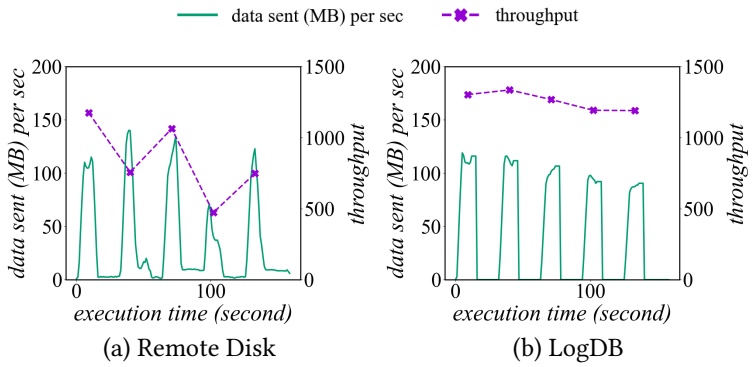



**Fig. 11.** Network Bandwidth Utilization for Light Workload in Figure 10b

the Remote Disk architecture. For example, the write performance is improved by **2.5X** when the buffer size is 8GB (Figure 10d).

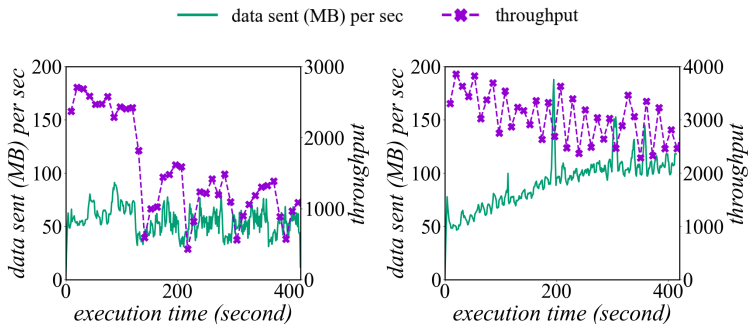
**Analyzing Read Performance (Figure 10a and 10c).** The comparable read performance between the Remote Disk and LogDB architectures is understandable given that the workload is read-only without any write operations. As a result, the read path for both architectures is identical, with all pages being materialized. In Figure 13, we will evaluate the performance of read-after-write.

**Analyzing Write Performance on Light Workload (Figure 10b).** For the light workload, Figure 10b shows that the write performance on the Remote Disk and LogDB architectures is quite similar. This means that LogDB does not improve much write performance in this scenario, which is **counter-intuitive** as log-as-the-database is supposed to improve write performance as indicated in [1, 11, 28].

However, under the light workload, we observe that the Remote Disk architecture does not require flushing dirty pages in the critical path during transaction commits. This is because the architecture has sufficient idle time to clean and flush the dirty pages in the background when the workload is light. As a result, when new transactions arrive, it is unnecessary to flush dirty pages on the fly due to the availability of clean buffer slots. This leads to comparable write performance between the two architectures.

To confirm the above analysis, Figure 11 illustrates the data transmitted over the network during a light workload where SysBench is executed five times. We can observe that the Remote Disk architecture has a higher peak data transfer rate, given its need to flush dirty pages to the storage node, a step not required by LogDB. Moreover, even in the absence of active transactions, the Remote Disk might still need to flush dirty pages during idle periods if the buffer contains dirty pages from previous transactions. If it manages to flush all the dirty pages during these idle periods (as seen in the 1st and 3rd run), it can achieve write performance comparable to LogDB. Otherwise (as seen in the 2nd, 4th, and 5th run), the write performance of Remote Disk may decline since the flushing process can interfere with ongoing transactions. Hence, in the light workload with ample idle time, the Remote Disk architecture can flush its dirty pages, resulting in write performance on par with LogDB.

**Analyzing Write Performance on Heavy Workload (Figure 10d).** Figure 10d shows that LogDB significantly improves write performance compared to the Remote Disk architecture. This is because, under the heavy workload, the local buffer in the Remote Disk architecture tends to become saturated with dirty pages, leaving inadequate time to flush these pages in the background.




(a) Remote Disk (b) LogDB

**Fig. 12.** Network Bandwidth Utilization for Heavy Workload in Figure 10d

As a result, when new transactions arrive, the Remote Disk architecture has to send both data (due to flushing dirty pages) and logs over the network. But LogDB only needs to send logs, leading to higher write performance.

To validate the above analysis, Figure 12 shows the data transfer over the network under the heavy workload, where there is no idle time between runs. Initially, the write performance of both the Remote Disk and LogDB architectures is similar, given that the buffer pool has a sufficient number of clean pages. Yet, after around 150 seconds, the Remote Disk's buffer becomes saturated with dirty pages, causing ongoing transactions to wait for the flushing of these pages before fetching new ones from the storage node, which results in a performance drop. In contrast, LogDB consistently maintains a higher write performance, even with occasional fluctuations, because only logs are transferred, eliminating the need to flush dirty pages.

**Overall Results on Read-After-Write (Figure 13).** In this experiment, we examine the read performance immediately after heavy writes to understand the overhead of replaying logs. This is because under heavy writes, data pages may not be immediately materialized in the LogDB architecture. As a result, subsequent read operations might be delayed due to the necessity of log replay.

Figure 13 illustrates the read performance following bulk insertion with varying durations of bulk insertion time. The figure shows that as the bulk insertion duration increases, LogDB's performance experiences a notable drop, while the performance of the Remote Disk architecture remains relatively stable. Specifically, when the bulk insertion lasts for 5 minutes, LogDB lags behind Remote Disk by 20.3%. Moreover, the gap widens when considering multi-version pages (as discussed in §5.3). For instance, LogDB-MV loses Remote Disk by 66.2% when the insertion duration is 5 minutes. This gap is attributed to the overhead of replaying logs.

### 5.3 Addressing Research Questions Q4 and Q5

In this experiment, we study the impact of multi-version pages that are required to support the shared-storage design. In particular, we answer the research questions Q4 and Q5.

**Overall Results on Reads and Writes (Figure 14).** Figure 14 presents the read and write comparisons between LogDB and LogDB-MV. Notably, given that the multi-version storage engine can optimize (actually bypass) the "torn page write" issue [3] explained in §3.4, we first show the write performance without optimization (see Figure 14b) to highlight the overhead introduced by supporting multi-version pages. Subsequently, we show the optimized results that bypass the "torn

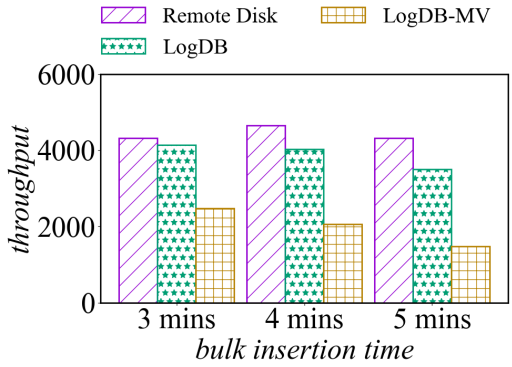



**Fig. 13.** Read After Bulk Insertion

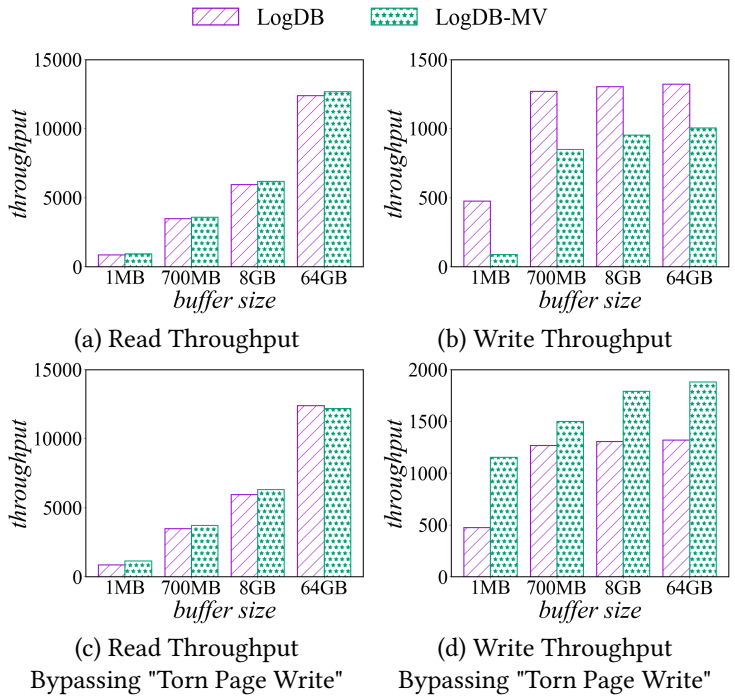



**Fig. 14.** LogDB vs. LogDB-MV

page write" issue using multi-version pages (Figure 14d) to clearly understand the performance implications. Note that the results in Figure 14c and Figure 14d are not relevant for systems that do not have the torn page issue. However, since both PostgreSQL and MySQL have such issue, studying it remains valuable.

Figure 14 shows that the multi-version storage engine mainly affects write performance and not read performance, which is understandable since there is no need to replay logs for read-only workloads. Thus, we will mainly focus on the write performance next. Figure 14b shows that, with the traditional "torn page write", LogDB-MV has lower write performance than LogDB. For example,

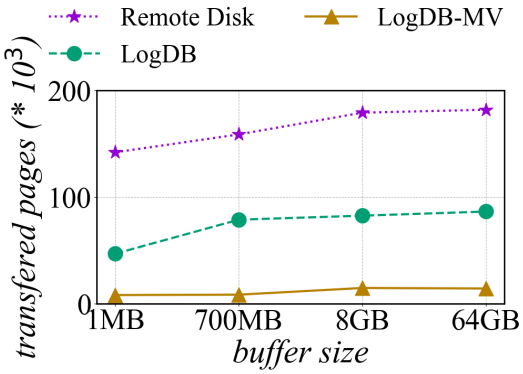



**Fig. 15.** Transferred Pages in Figure 14d

when the buffer size is 8GB, the performance slowdown is **27%**. However, LogDB-MV achieves higher write performance than LogDB if we optimize (actually bypass) the "torn page write" issue using multi-version pages. For example, after optimization, LogDB-MV improves write performance by **37%**. **This is an interesting finding that has not been discovered in previous research**.

**Analyzing Write Performance with "Torn Page Write" (Figure 14b).** Next, we delve into the factors contributing to the performance slowdown. This is due to the fact that supporting multiple versions leads to increased performance overhead, both from the elevated xlog replay overhead (since a single xlog can now correspond to multiple pages) and the added overhead during page insertion into RocksDB (given the significant increase in the number of pages as a result of multi-versioning).

**Analyzing Write Performance Bypassing "Torn Page Write" (Figure 14d).** Figure 14d shows that the write performance in LogDB-MV outperforms that in LogDB, meaning that the multi-version storage engine enhances write performance. This enhancement stems from the fact that page updates, due to multi-version pages, do not occur in place. As a result, they naturally address the "torn page write" issue, leading to reduced data transfer over the network.

To have a better understanding, Figure 15 shows the network data transfer during a 10-second SysBench write workload. In the Remote Disk architecture, the compute node must flush both dirty pages and xlogs to the storage node. In the LogDB architecture, due to the torn-page write issue, some xlogs do not merely contain tuple modification information; they also store the entire data pages (8KB per data page) within the xlogs. In contrast, the LogDB-MV architecture only requires the transfer of xlogs without the need to flush dirty pages. Moreover, with the support of multi-version pages, its xlogs solely capture tuple modifications. As shown in Figure 15, the data transfer for LogDB-MV is considerably less than that of the other two architectures.

**Supporting Multi-Compute Nodes (Figure 16).** The multi-version storage engine also allows multiple compute nodes to share the same storage. Figure 16 shows that the read performance scales as the number of compute nodes increases up 16 nodes. Note that there is usually a single writer in storage-disaggregated databases such as Amazon Aurora [28], Microsoft Socrates [11], Google AlloyDB [1], and Alibaba PolarDB [5, 14].

**Impact on of Multi-versioning on Checkpointing (Figure 17).** In this experiment, we examine the effect of multi-version pages on checkpointing, a topic overlooked in previous research. By supporting multi-version pages in the storage engine, we can address the checkpointing challenge.

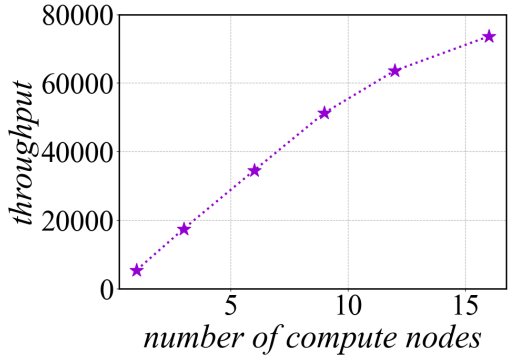



**Fig. 16.** Results on Multiple Compute Nodes (Up to 16)

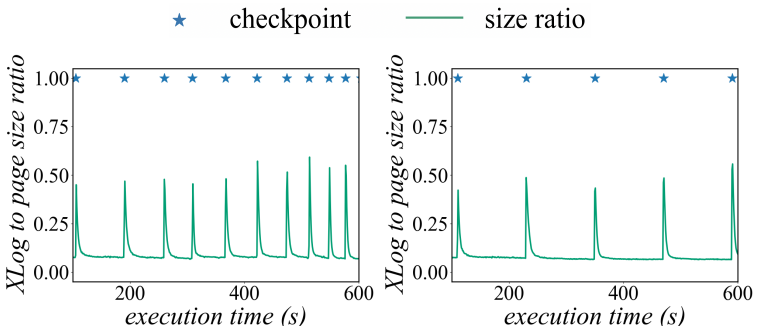
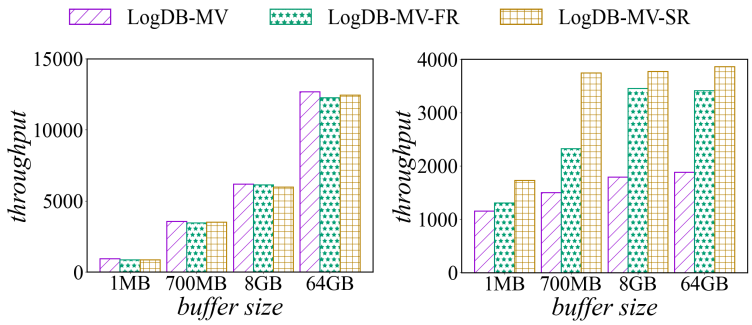



**Fig. 17.** Checkpoints Influence XLog Size

Checkpointing is crucial in databases since it offers a consistent database state to users in the event of a system crash [25]. However, checkpointing incurs significant I/Os as it needs to flush all dirty pages to disk. While the log-as-the-database design can alleviate this by only flushing logs, the log size remains significant due to the "torn page write" problem, where PostgreSQL logs the entire page in xlog if it is the page's first xlog entry after each checkpoint. Figure 17 shows the xlog size with different checkpoint frequencies in LogDB. The results indicate that after each checkpoint, there is a noticeable increase in xlog size. However, the checkpointing issue can be addressed very well in LogDB-MV because LogDB-MV fully bypasses "torn page write" problem and enable xlogs only record the modifications. Consequently, the spikes in xlog size that occurred after each checkpoint are eliminated. As a result, checkpointing becomes a "free" operation without introducing additional overhead.

### 5.4 Addressing Research Question Q6

In this experiment, we will evaluate the effectiveness of various log replay methods by comparing the performance between LogDB-MV, LogDB-MV-FR, and LogDB-MV-SR based on storage nodes cluster setting. We will first use the standard SysBench read/write workload from previous experiments, with the results shown in Figure 18. After that, we will introduce another workload, which involves a bulk insertion followed by index creation, to best showcase the effectiveness of these log replay methods (in Figure 19).




(a) Read Performance (b) Write Performance

**Fig. 18.** LogDB-MV vs. LogDB-MV-FR vs. LogDB-MV-SR

**Table 2. XLog Replay Waiting Time of GetPage@LSN**

| | Num. of GetPage Requests | Waited Replayed LSN |
|---|---|---|
| **LogDB-MV** | 100000 | 89002096 |
| **LogDB-MV-FR** | 100000 | 24370816 |
| **LogDB-MV-SR** | 100000 | 13053248 |

**Overall Results on Reads and Writes (Figure 18).** Figure 18 shows the performance comparison among LogDB-MV, LogDB-MV-FR, and LogDB-MV-SR. As expected, different log replay methods affect write performance (not read performance) because there is no need to replay logs for the read-only workload. Next, we mainly focus on writes. Figure 18 indicates that LogDB-MV-SR outperforms the others, with LogDB-MV-FR coming next, and LogDB-MV being the slowest.

**Analyzing Write Performance (Table 2).** In this experiment, we execute the write workload on LogDB-MV, LogDB-MV-FR, and LogDB-MV-SR with a 700MB buffer size. We collect the last 100,000 GetPage@LSN requests to analyze how many xlogs replayed were waited for.

Table 2 shows the results. For the LogDB-MV's GetPage@LSN request, as described in §3.4, it needs to wait for the storage node to advance xlog replaying until it exceeds its parameter LSN. According to Table 2, LogDB-MV's GetPage@LSN requests have waited for 89002096 LSNs of xlog replaying. As mentioned in §3.5, the LogDB-MV-FR's GetPage@LSN will truncate its xlog replaying list to a shorter one. Table 2 shows that LogDB-MV-FR's GetPage@LSN requests only waited for 24370816 LSNs of xlog replaying, which saves about 72.6% of waiting xlog replaying time compared with LogDB-MV. Therefore, in Figure 18, LogDB-MV-FR's write throughput is about **1.5X** that of LogDB-MV.

As mentioned in §3.6, LogDB-MV-SR's GetPage@LSN request will only selectively replay the xlogs that are directly related to its target page. In our experiment, LogDB-MV-SR replayed only 13053248 LSNs of xlog replaying, which is only 53.6% of the LSNs waited by LogDB-MV-FR. Meanwhile, LogDB-MV-SR can replay xlogs in parallel, which can further enhance its write performance. In Figure 18, LogDB-MV-SR's write throughput is about **1.6X** that of LogDB-MV-FR.

**Results on Bulk Insertion and Create Index Workload (Figure 19).** For a better understanding of the performance implications of LogDB-MV-FR and LogDB-MV-SR, we crafted a workload that includes a bulk insertion followed by index creation. In this experiment (Figure 19), we employed SysBench's database preparation phase, which consists of two steps: (1) bulk insertion of tuples into tables, totaling 9GB; and (2) the creation of a secondary index for the newly prepared tables.

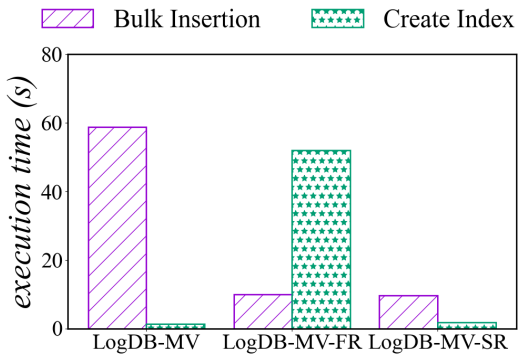



**Fig. 19.** Evaluating LogDB-MV vs. LogDB-MV-FR vs. LogDB-MV-SR on "Bulk Insertion" and "Create Index" Workload"

For LogDB-MV, the two steps take approximately 60 seconds. During the bulk insertion phase, the compute node consistently sends GetPage@LSN requests to the storage node. Each request results in a notable waiting time for the log replay. As a result, during the secondary index creation phase, LogDB-MV only has to replay a minimal amount of xlogs associated with index creation, leading to a significantly reduced index creation time.

In contrast, for LogDB-MV-FR, the bulk insertion is completed in around 10 seconds, while the index creation takes around 52 seconds. This is because, during the bulk insertion phase, most GetPage@LSN requests either do not require xlog replay or only need to replay a very small number of xlogs. As a result, a significant amount of xlogs remain to be replayed in the index creation phase.

However, LogDB-MV-SR takes about 10 seconds to complete the bulk insertion, which is comparable to LogDB-MV-FR. This similarity arises because there is minimal xlog replaying during the bulk insertion phase. However, when executing the index creation workload, LogDB-MV-SR needs to read metadata pages which require replaying a large number of xlogs. Yet, LogDB-MV-SR selectively replays only the xlogs relevant to the desired metadata pages, deferring the replaying of other xlogs until the database is idle. As a result, LogDB-MV-SR takes only about 13 seconds to complete the entire preparation workload.

**Impact on the Idle Time Interval Between Bulk Insertion and Create Index (Figure 20).** To better understand the performance characteristics of LogDB-MV-FR, in this experiment, we aim to analyze the scenarios in which LogDB-MV-FR's optimizations are effective. As depicted in Figure 19, even though LogDB-MV-FR takes much less time during the bulk insertion phase, it takes considerably longer in the immediately subsequent secondary index creation phase. In Figure 20, we introduce intervals between the bulk insertion and index creation phases and test the "Create Index" time for both LogDB-MV and LogDB-MV-FR. As shown in Figure 20, LogDB-MV's execution time remains consistent regardless of the interval duration. In contrast, LogDB-MV-FR's index creation time decreases as the interval becomes longer. When the gap between bulk insertion and index creation reaches 60 seconds, the index creation latency for LogDB-MV-FR aligns with that of LogDB-MV. This suggests that LogDB-MV-FR takes approximately 60 seconds to finish replaying the accumulated xlogs in the background. Consequently, during the index creation phase, it only needs to replay the same number of xlogs as LogDB-MV.

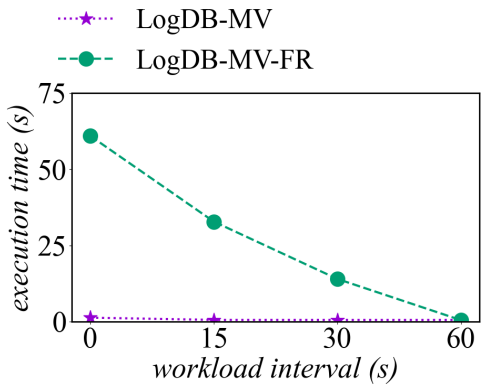



**Fig. 20.** LogDB-MV vs. LogDB-MV-FR on "Create Index" Workload

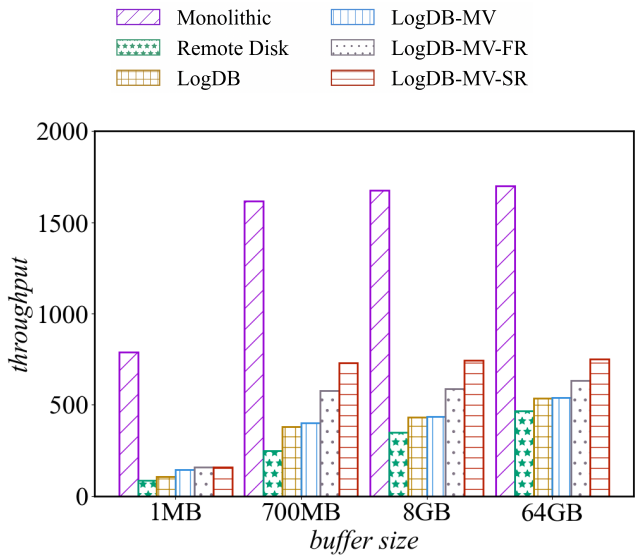



**Fig. 21.** Results on TPC-C

### 5.5 Additional Experiments

*5.5.1 Results on TPC-C.* In this experiment, we evaluate the six architectures mentioned above on the TPC-C benchmark. Note that the TPC-C differs from SysBench as it executes mixed reads and writes. We prepare a database with 30 warehouses, and inside each warehouse, there are 30 table sets. The overall database size is 94GB. We use 16 threads to execute the TPC-C benchmark.

Figure 21 shows the results. Overall, the results are similar to those of SysBench, though there are some minor differences. For instance, the performance of LogDB-MV is similar to that of LogDB. However, in SysBench, LogDB-MV clearly outperforms LogDB. This is because LogDB-MV has better write performance than LogDB, as shown in Figure 14d, but LogDB-MV's read performance is worse than that of LogDB, as depicted in Figure 13. As a result, under a mixed read and write workload, LogDB and LogDB-MV exhibit comparable performance in the TPC-C experiment.

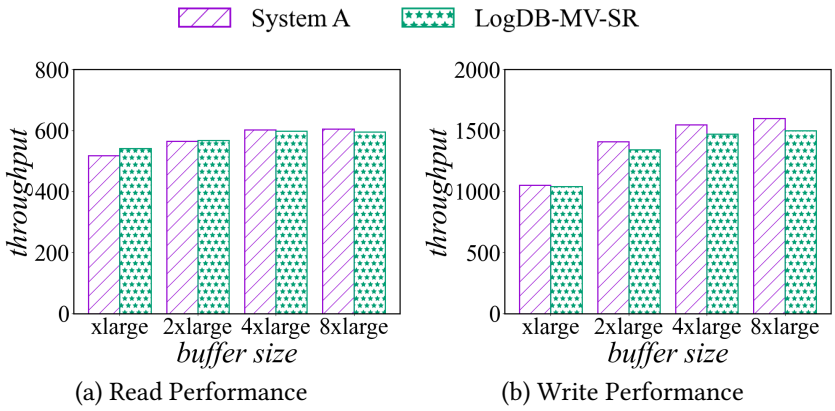



**Fig. 22.** System A vs. LogDB-MV-SR

*5.5.2 Comparison with System A.* In this experiment, we compare our testbed (LogDB-MV-SR) with an industrial storage-disaggregated database, referred to as System A for confidentiality. This comparison aims to demonstrate the high-performance implementation of our testbed and enhance the reliability of our findings.

Since System A is not open-sourced, we purchased System A's cloud version with four instances: instance-1 (4 cores and 32GB memory), instance-2 (8 cores and 64GB memory), instance-3 (16 cores and 128GB memory), and instance-4 (32 cores and 256GB memory), each equipped with a 10Gb network.

We made our best effort to make the comparison fair with details omitted. Figure 22 shows the results. For read performance, System A and our testbed (LogDB-MV-SR) perform similarly because they share the same read path. For write performance, System A is slightly better than our testbed (but only up to 6.6%) across different instances. This means that our implementation is comparable with System A. Thus, we are confident that the results and findings in this paper are reliable.

*5.5.3 Results on 30Gb Network.* In this experiment, we evaluate the impact of upgrading to a 30Gb network. In this new setting, both the compute and storage nodes are equipped with Intel Xeon Platinum CPUs @ 2.70GHz, 2.95TB of DRAM, and 8.7TB NVMe SSDs. The round trip latency ranges from 0.13ms to 0.22ms, compared to 0.29ms to 0.33ms on the 10Gb network.

We benchmark the Remote Disk, LogDB, and LogDB-MV-SR architectures under both 10Gb and 30Gb networks. The results (Figure 23) show that during reads, the 30Gb network achieves improved performance, especially with smaller cache sizes. However, when the cache size is increased to 64GB, the performance difference becomes negligible due to the high cache hit ratio of 99.4%, meaning that almost all necessary pages are cached locally.

Regarding write performance, we observe consistent improvements across all architectures with the 30Gb network. The main reason is that log flushing, a potential bottleneck in write performance, is inevitable in all architectures. With a smaller cache size, the 30Gb network speeds up page transferring and XLog flushing. With a larger cache size, it primarily accelerates XLog flushing.

*5.5.4 CPU and Memory Usage.* In this experiment, we examine the CPU and memory usage while running a 96GB SysBench workload with an 8GB buffer. The CPU usage is the total utilization across all cores. It often exceeds 100% because we use multiple CPU cores. As shown by Figure 24, we tracked CPU and memory usage separately for reads and writes due to their different characteristics in disaggregated databases.

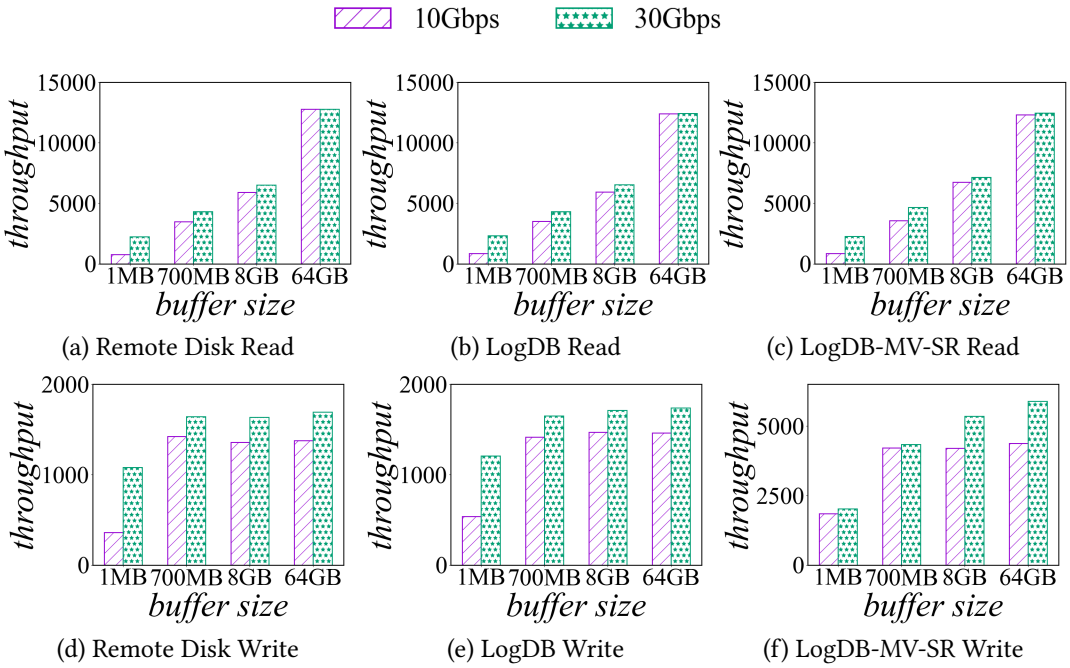
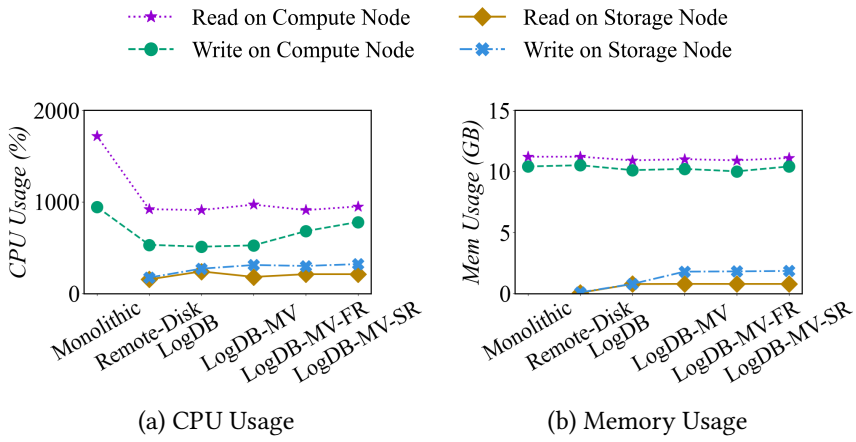



**Fig. 23.** Results on 10Gb vs. 30Gb Network

On the compute side, memory use stays consistent for both reads and writes across different architectures. This is because the compute node consistently uses an 8GB buffer pool and about 2GB of extra memory for processing under SysBench. For CPU usage, we found it is higher in the monolithic architecture during reads, because it achieves better performance than other five disaggregated architectures. During writes, the CPU usage drops in Remote Disk, then increases in LogDB-MV-FR and LogDB-MV-SR, aligning with the observed pattern of write performance observed.

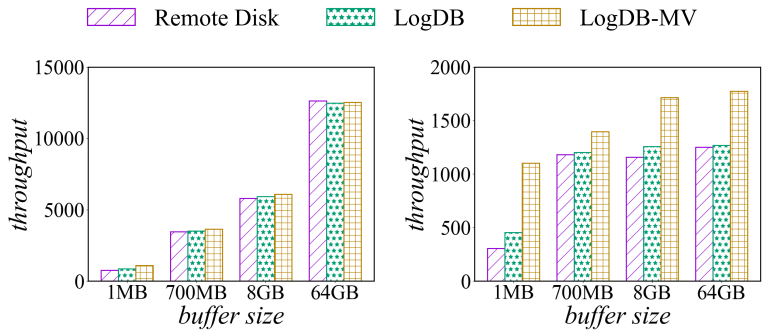
On the storage side, the log-based architectures consume more CPU resources compared to the Remote Disk architecture due to the added step of log replay. Regarding memory use, during reads, the log-based architectures need about 700MB more memory than the Remote Disk architecture. This is because we pre-allocate a 700MB buffer pool in all log-based architectures for xlog replaying. During writes, the LogDB-MV, LogDB-MV-FR, and LogDB-MV-SR use around 1GB more memory than LogDB. This increase is due to these architectures employing RocksDB for storage, where RocksDB can use up to 1GB of memory for layer merging and compaction.

*5.5.5 Results on 6 Storage Nodes.* In this experiment, we evaluate the impact of 6-way replication by using a storage cluster of 6 nodes. In particular, we run SysBench on three representative architectures: Remote Disk, LogDB, and LogDB-MV. Figure 25 shows the results. It shows that the read performance is similar to that of a 3-node storage cluster, as the read path is not changed. For writes, the performance drops slightly compared to the 3-node storage cluster, as described in §5, due to the tail latency of replicating data to all 6 nodes. However, the relative performance among the three architectures remains unchanged.




(a) CPU Usage (b) Memory Usage

**Fig. 24.** CPU and Memory Usage




(a) Read Throughput (b) Write Throughput

**Fig. 25.** Results on 6 Storage Nodes

### 5.6 Discussion

In this paper, we built our testbed using PostgreSQL due to its widespread popularity. Considering that some disaggregated databases, like Aurora and Taurus, also support MySQL, we will next discuss how our findings relate to MySQL.

We believe that our conclusions can be extended to MySQL. This is because MySQL and PostgreSQL share similar high-level design principles (for disaggregation), such as software-level disaggregation, log-as-the-database, and shared storage. Although there are some differences in details, e.g., MySQL uses double writes to deal with torn page writes, they are not expected to change the conclusions.

## 6 RELATED WORK

**Storage-Disaggregated OLTP Databases.** This paper focuses on storage-disaggregated OLTP databases. Examples of databases in this category are Amazon Aurora [28], Microsoft Socrates [11], Google AlloyDB [1], Huawei Taurus [17], Alibaba PolarDB [5, 14], and Neon [4]. As mentioned in §1, these databases typically embrace three key design principles: Software-level Disaggregation (P1), Log-as-the-Database (P2), and Shared-Storage Design (P3). Note that apart from PolarDB

which implemented P1 and P3, all other databases mentioned above have incorporated P1, P2, and P3. However, previous research does not evaluate the performance impact of these individual design principles, which is the goal of this paper.

**Storage-Disaggregated OLAP Databases.** While this paper focuses on storage-disaggregated OLTP databases, there is a category of storage-disaggregated databases for OLAP databases. Examples include Snowflake [16, 29], AnalyticDB [40], Polaris [9], Redshift [12, 24], Dremel [22], Eon Mode [27], and FlexPushdownDB [38]. However, two design principles (P2 and P3) discussed in this paper do not apply to OLAP databases. For instance, the log-as-the-database (P2) design is not employed in such OLAP databases due to their emphasis on read-intensive tasks. Furthermore, they do not support multi-version pages (P3) because the replication lag between compute nodes is not a concern, given that writes are not a primary focus. However, the design principle of software-level disaggregation (P1) is relevant to those OLAP databases.

**Memory-Disaggregated Databases.** In the literature, there is a line of research working on memory-disaggregated databases that decouple memory from compute. Examples include [21, 22, 32–35, 41, 42]. However, this paper focuses on storage-disaggregated instead of memory-disaggregated databases. The design considerations in memory-disaggregated databases differ from those in this paper. For instance, they all employ ultra-fast networking technologies like RDMA [32, 35, 43] or CXL [10, 19] to match the high performance of local DRAM, whereas storage-disaggregated databases often rely on conventional TCP/IP networking [11, 28]. Moreover, given the high-speed of their network, they typically transmit pages from the compute node directly to the memory node [15, 42], in contrast to storage-disaggregated databases which commonly send logs across the network [11, 28]. As a result, many optimizations designed for memory-disaggregated databases are not applicable to storage-disaggregated databases.

**Disaggregated NoSQL Databases.** There are other studies focusing on disaggregating NoSQL databases like key-value stores [13, 18, 39], vector databases [30], graph databases [36]. However, this paper focuses on (disaggregated) relational databases.

## 7 CONCLUSION

In this paper, we investigated the performance implications of the fundamental design principles, namely software-level disaggregation, log-as-the-database, and shared-storage in storage-disaggregated databases. We studied six research questions (Q1 to Q6) and summarized the main findings below.

**Answers to Question Q1**: *How much performance overhead is introduced by storage disaggregation?*

- Storage disaggregation results in a significant performance reduction for both reads (**16.4X**) and writes (**17.9X**) assuming the storage media are SSDs. This is because accessing storage remotely over the network is slower than accessing local storage.

**Answers to Question Q2**: *To what extent can buffering help in mitigating the performance degradation?*

- Utilizing a buffer in the compute node can enhance read performance. For instance, when the buffer size is 8GB (80% hit ratio), the read performance gap between disaggregated and non-disaggregated databases is **1.8X**.
- Nevertheless, **buffering does not significantly improve write performance**, even with a sufficiently large buffer size (e.g., reaching 99.5%). This is because the compute node must always send xlogs to the storage node via the network when transactions are committed, irrespective of the buffer size in the compute node.

**Answers to Question Q3**: *How significant is the performance improvement (for both reads and writes) due to the log-as-the-database design?*

- For writes, **under light workloads, the log-as-the-database design principle does not enhance performance**. This is due to the database having sufficient idle time to clean and flush dirty pages in the background. Therefore, it does not necessitate flushing dirty pages in the critical path for committed transactions. However, **under heavy workloads, the performance improvement of the log-as-the-database principle is significant** (by **2.58X** with 8GB buffer size). This is because the buffer pool becomes saturated with dirty pages for heavy workloads and then new transactions would have to flush these dirty pages on the fly without the log-as-the-database principle.
- For reads, we distinguish between two scenarios: read-only and read-after-write. For **read-only workloads, log-as-the-database design does not have much performance impact** as all the pages have already been materialized. However, for **read-after-write, it introduces performance overhead**, e.g., 18.9% when the buffer size is 8GB, due to the cost of relaying logs.

**Answers to Question Q4**: *What is the performance impact caused by supporting multi-version pages in the shared-storage design?*

- With traditional "torn page write", supporting multi-version pages **reduces the write performance by 27%** (with a buffer size of 8GB). This is because multi-versioning increases the overhead of xlog replaying and page insertion, which causes performance drop.
- However, we observe that the introduction of multi-version pages can optimize the "torn page write" issue. With such optimization, it **improves the write performance by 37%** (with a buffer size of 8GB).

**Answers to Question Q5**: *How does multi-version storage affect checkpointing?*

- Page-based multi-version storage engine **completely addresses the high I/O issue in conventional database checkpointing**. Multi-version pages eliminate the "torn page write" problem, resulting in a log size much smaller than that of LogDB. As a result, checkpointing becomes a "free" operation without introducing additional overhead in the multi-version storage engine.

**Answers to Question Q6**: *How effective are various log-replay methods within the multi-version storage?*

- Different log-replay approaches **significantly impact write performance**. Compared to the straightforward approach, Filtered Replay (FR) enhances write performance by **1.6X**. Smart Replay (SR) further improves upon FR by an additional **1.6X**, owing to its ability to prune unnecessary xlogs replayed. Most notably, SR can achieve even higher performance for the workload with bulk insertion followed by index creation, where SR improves FR by **7X**.

## ACKNOWLEDGEMENTS

Jianguo Wang acknowledges the support of the National Science Foundation under Grant Number 2337806. We also sincerely thank Dr. Yingjie He for his help in answering many questions regarding disaggregated databases.

## REFERENCES

[1] [n.d.]. AlloyDB for PostgreSQL, https://cloud.google.com/alloydb.
[2] [n.d.]. Choose between SSD and HDD Storage, https://cloud.google.com/sql/docs/mysql/choosing-ssd-hdd.

[3] [n.d.]. Full Page Writes in PostgreSQL, https://wiki.postgresql.org/wiki/Full_page_writes.
[4] [n.d.]. Neon, https://github.com/neondatabase/neon.
[5] [n.d.]. PolarDB for PostgreSQL, https://github.com/ApsaraDB/PolarDB-for-PostgreSQL.
[6] [n.d.]. PostgreSQL Database Page Layout, https://www.postgresql.org/docs/current/storage-page-layout.html.
[7] [n.d.]. SysBench Manual, https://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf.
[8] [n.d.]. TPC-C is an On-Line Transaction Processing Benchmark, https://www.tpc.org/tpcc/.
[9] Josep Aguilar-Saborit and Raghu Ramakrishnan. 2020. POLARIS: The Distributed SQL Engine in Azure Synapse. *PVLDB* 13, 12 (2020), 3204–3216.
[10] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna T. Malladi, and Yang-Seok Ki. 2022. Enabling CXL Memory Expansion for In-Memory Database Management Systems. In *DaMoN*. 8:1–8:5.
[11] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *SIGMOD*. 1743–1756.
[12] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *SIGMOD*. 2205–2217.
[13] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. 2020. Hailstorm: Disaggregated Compute and Storage for Distributed LSM-based Databases. In *ASPLOS*. 301–316.
[14] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: An Ultra-Low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *PVLDB* 11, 12 (2018), 1849–1862.
[15] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *SIGMOD*. 2477–2489.
[16] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*. 215–226.
[17] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, and Yongjun He. 2020. Taurus Database: How to be Fast, Available, and Frugal in the Cloud. In *SIGMOD*. 1463–1478.
[18] Siying Dong, Shiva Shankar P., Satadru Pan, Anand Ananthabhotla, Dhanabal Ekambaram, Abhinav Sharma, Shobhit Dayal, Nishant Vinaybhai Parikh, Yanqin Jin, Albert Kim, Sushil Patil, Jay Zhuang, Sam Dunster, Akanksha Mahajan, Anirudh Chelluri, Chaitanya Datye, Lucas Vasconcelos Santana, Nitin Garg, and Omkar Gawde. 2023. Disaggregating RocksDB: A Production Experience. *Proc. ACM Manag. Data* 1, 2 (2023), 192:1–192:24.
[19] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *USENIX ATC*. 287–294.
[20] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR*.
[21] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan S. Milojicic, and Gustavo Alonso. 2022. Farview: Disaggregated Memory with Operator Off-loading for Database Engines. In *CIDR*.
[22] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *PVLDB* 13, 12 (2020), 3461–3472.
[23] M. Tamer Özsu and Patrick Valduriez. 2020. *Principles of Distributed Database Systems, 4th Edition.* Springer.
[24] Ippokratis Pandis. 2021. The Evolution of Amazon Redshift. *PVLDB* 14, 12 (2021), 3162–3163.
[25] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. 2020. *Database System Concepts, Seventh Edition.* McGraw-Hill Book Company.
[26] Michael Stonebraker. 1986. The Case for Shared Nothing. *IEEE Data Engineering Bulletin* 9, 1 (1986), 4–9.
[27] Ben Vandiver, Shreya Prasad, Pratibha Rana, Eden Zik, Amin Saeidi, Pratyush Parimal, Styliani Pantela, and Jaimin Dave. 2018. Eon Mode: Bringing the Vertica Columnar Database to the Cloud. In *SIGMOD*. 797–809.

[28] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD*. 1041–1052.
[29] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *NSDI*. 449–462.
[30] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *SIGMOD*. 2614–2627.
[31] Jianguo Wang and Qizhen Zhang. 2023. Disaggregated Database Systems. In *SIGMOD*. 37–44.
[32] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *SIGMOD*. 1033–1048.
[33] Ruihong Wang, Chuqing Gao, Jianguo Wang, Prishita Kadam, M. Tamer Özsu, and Walid G. Aref. 2024. Optimizing LSM-based Indexes for Disaggregated Memory. *VLDB Journal* (2024).
[34] Ruihong Wang, Jianguo Wang, Stratos Idreos, M. Tamer Özsu, and Walid G. Aref. 2022. The Case for Distributed Shared-Memory Databases with RDMA-Enabled Memory Disaggregation. *PVLDB* 16, 1 (2022), 15–22.
[35] Ruihong Wang, Jianguo Wang, Prishita Kadam, M. Tamer Özsu, and Walid G. Aref. 2023. dLSM: An LSM-Based Index for Memory Disaggregation. In *ICDE*. 2835–2849.
[36] Sebastian Wong. 2019. Disaggregated Graph Database with Rich Read Semantics, https://atscaleconference.com/videos/scale-2019-disaggregated-graph-database-with-rich-read-semantics/.
[37] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. 2015. Performance Analysis of NVMe SSDs and Their Implication on Real World Databases. In *SYSTOR*. 6:1–6:11.
[38] Yifei Yang, Matt Youill, Matthew E. Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. FlexPushdownDB: Hybrid Pushdown and Caching in a Cloud DBMS. *PVLDB* 14, 11 (2021), 2101–2113.
[39] Qiaolin Yu, Chang Guo, Jay Zhuang, Viraj Thakkar, Jianguo Wang, and Zhichao Cao. 2024. CaaS-LSM: Compaction-as-a-Service for LSM-based Key-Value Stores in Storage Disaggregated Infrastructure. In *SIGMOD*.
[40] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, and Chengliang Chai. 2019. AnalyticDB: Real-time OLAP Database System at Alibaba Cloud. *PVLDB* 12, 12 (2019), 2059–2070.
[41] Qizhen Zhang, Yifan Cai, Sebastian Angel, Vincent Liu, Ang Chen, and Boon Thau Loo. 2020. Rethinking Data Management Systems for Disaggregated Data Centers. In *CIDR*.
[42] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Jimmy Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. 2021. Towards Cost-Effective and Elastic Cloud Database Deployment via Memory Disaggregation. *PVLDB* 14, 10 (2021), 1900–1912.
[43] Tobias Ziegler, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig. 2023. Design Guidelines for Correct, Efficient, and Scalable Synchronization using One-Sided RDMA. *Proc. ACM Manag. Data* 1, 2 (2023), 131:1–131:26.

Received October 2023; revised January 2024; accepted February 2024