Reducing Tail Latency in Storage-Disaggregated Database Systems

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

Storage-disaggregated databases have become the standard in the cloud due to many benefits, including improved resource utilization, reduced resource fragmentation, and the ability to independently and elastically scale compute and storage, ultimately leading to cost savings. This work focuses on OLTP databases. Examples include Amazon Aurora, Microsoft Socrates, and Neon. However, a significant limitation we have identified in storage-disaggregated databases is the long tail latency. This issue arises from the unique architecture of these databases, specifically the log-as-the-database design principle. Under this design, when a transaction is committed, only the logs are sent to the storage engine over the network to minimize data movement, while the actual pages are replayed on the storage side. Thus, certain page requests may encounter a lengthy log replay chain, which lead to long latency.

In this paper, we introduce a novel technique called *Replay-as-a-Service* (RaaS) to address the high tail latency issue in storage-disaggregated databases. The main idea behind RaaS is to decouple the log replay logic from the storage engine and make it as an independent service. This approach provides the flexibility to utilize idle servers or even dedicated servers within the cluster to efficiently execute the log replay. To enable this, we introduce a suite of techniques and optimizations to address key technical challenges. We have implemented the RaaS technique within OpenAurora, an open-source storage-disaggregated database based on PostgreSQL. Experiments on SysBench show that RaaS reduces P95 tail latency by 40.1% and improves the overall throughput by 75.9%.

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

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

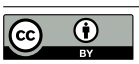
ACM Reference Format:
Xi Pang and Jianguo Wang. 2026. Reducing Tail Latency in Storage-Disaggregated Database Systems. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 74 (February 2026), 26 pages. https://doi.org/10.1145/3786688

1 Introduction

Storage-disaggregated databases, exemplified by Amazon Aurora [36], Microsoft Socrates [9], Google AlloyDB [1], Huawei Taurus [15], and Neon [3], are widely adopted in cloud-based database architectures. By decoupling compute and storage, these databases enable independent and elastic scaling of each component to meet varying workload needs. Furthermore, they can increase resource utilization and reduce resource fragmentation by pooling resources, resulting in lower costs. As a result, storage-disaggregated databases have become the top choice for customers who migrate their data to the cloud. For instance, Amazon Aurora has been used by tens of thousands of customers, making it the fastest-growing service in AWS history [2, 34].

Although storage-disaggregated databases are popular, we observe that their performance is unstable, and they face a significant tail latency issue. For instance, we conducted an experiment

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



ACM 2836-6573/2026/2-ART74
https://doi.org/10.1145/3786688

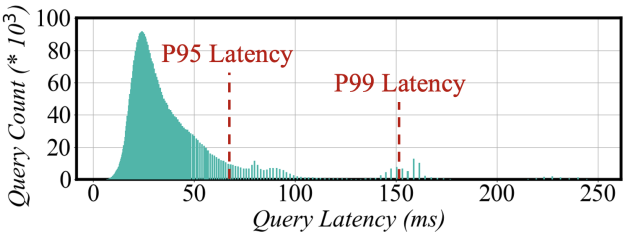


Fig. 1. Query Latency Distribution in Amazon Aurora

using Amazon Aurora (PostgreSQL v16.6) on a `db.r6g.large` instance (2 vCPUs, 16GB DRAM). We loaded 10GB data from the SysBench benchmark [6] and measured Aurora's performance. Specifically, we executed SysBench workload for 20 minutes and plotted the query latency distribution in Figure 1. The results show that Aurora's overall query latency is low with an average of 33.2 ms and a median latency of 26.7 ms. However, its tail latency is high with a P95 latency of 69.3 ms, which is 2.6X higher than the median latency, and a P99 latency of 153.1 ms, which is 5.7X higher than the median latency.

This work aims to reduce tail latency in storage-disaggregated database systems, particularly OLTP databases. Reducing tail latency is critical for many latency-sensitive applications [14, 16, 24], such as online gaming, financial trading platforms, real-time communication services, and web applications that require fast and predictable response times. Moreover, as storage-disaggregated databases become increasingly important for supporting agentic AI workloads [7, 45], tail latency will become an even more critical factor in ensuring stable responses for AI agents..

To address the tail latency issue, we observe that this issue arises from the unique design of storage-disaggregated databases, particularly the *log-as-the-database* [30, 36] design principle. Specifically, when a transaction is committed (in the compute node), instead of sending the actual data pages as in traditional databases, only write-ahead logs are sent to the storage node in order to minimize data movement over the network. The actual data pages are then asynchronously materialized from the storage node. As a result, different queries may experience different waiting times for the pages to be materialized by replaying a different number of logs. Though there is a background replay thread in the storage node, which can shorten the logs to be replayed to some extent, different pages may still have log replay chains of varying lengths. We will provide more analysis in Section 3.

A straightforward solution would be to allocate more computing resources to the storage node, which could enable faster log replay. However, this contradicts the principle of storage-compute disaggregation, where storage nodes typically have limited computing power, while compute nodes are equipped with ample computing capacity. In case if high computing power is provisioned for the storage nodes, it would result in significant resource waste (and high costs) when foreground workloads are light, where the log replay tasks in the storage nodes are small.

Main Idea. In this paper, we introduce a new technique called *Replay-as-a-Service* (RaaS) to tackle the tail latency issue in storage-disaggregated databases (particularly OLTP databases). The core idea behind RaaS is to decouple the background log replay logic from the storage engine and offload it to idle instances or even dedicated instances within the cluster. Our observation is that the background log replay process consumes a significant amount of computing resources and is often triggered by heavy foreground write workloads, which, in turn, leads to increased competition for the limited computing resources in the storage nodes. By pulling out the log replay logic, the background replay process will no longer compete for resources with foreground query processing.

Also, the background log replay can be executed more frequently and efficiently by utilizing idle resources in the cluster. This will shorten the log chain, which not only accelerates foreground query processing but also reduces tail latency – typically caused by accessing pages with a large accumulation of logs. Furthermore, this approach does not necessarily require additional resources, as there are typically ample resources available across other nodes in the cluster at any given time due to workload variations (e.g., bursty workloads), as mentioned in many prior works such as [17, 19, 33, 35, 39].

Challenges and Technical Overview. However, building RaaS is not trivial and requires addressing several challenges.

First, the background log replay logic is tightly coupled with the logic for handling foreground page requests (i.e., GetPage@LSN) in the storage engine, making it difficult to offload the replay task to idle instances. To address this, we identify a suitable point at which to decouple the resource-intensive page materialization stage from the storage node and make the log replay as a stateless service (Section 5.1).

Second, since the log replay task's execution relies on massive data and is offloaded from the resource-constrained storage node to idle instances, it is essential to minimize CPU overhead on the storage node during data transfer, as its CPU resources are limited. To achieve this, we introduce a new mechanism that leverages cloud object storage for efficient data transfers, significantly reducing the CPU burden on the storage node during task offloading (Section 5.2).

Third, since decoupling the log replay task from the storage node enables access to ample resources on idle instances, it is essential to fully utilize those resources. To this end, we redesign the original single-threaded log replay logic – which was tailored for the storage node's resource-constrained environment – to support parallel log replay by addressing key technical challenges. This new design significantly improves log replay performance (Section 5.3).

Finally, scheduling log replay tasks to appropriate instances is non-trivial, as it requires selecting instances with sufficient available resources to meet each task's demand while also respecting task priorities to prevent urgent tasks from being delayed by less critical ones. To address this, we introduce a new control coordinator that manages task scheduling and assignment by taking both task priority and the resource utilization of instances into consideration (Section 5.4).

Implementation. We have implemented RaaS in OpenAurora [4, 30], an open-source storage-disaggregated database built on PostgreSQL. The resulting system is named RaaS-OpenAurora. It is worth noting that RaaS can be integrated into other storage-disaggregated databases (e.g., Amazon Aurora or Microsoft Socrates), as they share the same design principles as OpenAurora.

Experimental Overview. We conducted experiments on an 84 GB SysBench dataset in Section 7. The results show that RaaS reduces P95 tail latency by **40.1%** (from 68.28 ms to 40.9 ms) and improves the overall database throughput by **75.9%**.

Contributions. This work makes the following contributions.

- This paper is the first to identify the important tail latency issue in storage-disaggregated databases (particularly OLTP databases) and provides an analysis of its root causes (Section 3).
- It proposes a new technique called RaaS, which decouples the background log replay tasks from the storage engine and executes them on idle instances, thereby reducing tail latency without introducing extra resources (Section 4 and Section 5).
- It implements RaaS in OpenAurora, a real storage-disaggregated database based on PostgreSQL. This demonstrates the compatibility of RaaS with existing databases, facilitating practical integration and deployment (Section 6).

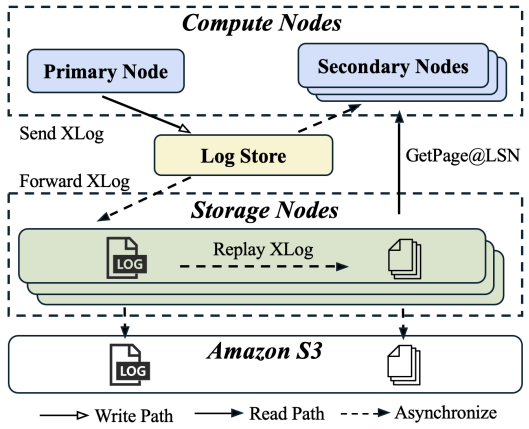


Fig. 2. Architecture of Storage-Disaggregated Databases

Open-source. The code is available at https://github.com/purduedb/OpenAurora/tree/RaaS.

2 Background

2.1 Storage-Compute Disaggregation

Storage-compute disaggregation has become a popular architecture in modern cloud databases [1, 3, 9, 15, 30, 36]. In traditional monolithic databases, the compute engine and storage engine are tightly integrated into a single server instance that handles both query processing and data persistence. In contrast, storage-disaggregated databases separate these components into different layers – connected via the network – to enable independent scaling.

Figure 2 illustrates the system architecture of a typical storage-disaggregated database (e.g., Microsoft Socrates [9] and Neon [3]), which consists of four layers: the compute layer, storage layer, log store, and object store. Compute nodes – with ample computing power – are responsible for compute-intensive tasks such as query parsing, optimization, execution, and transaction processing. Storage nodes – with vast storage capacity – are used to persist and store data. The log store is introduced to accelerate transaction commits. The object store layer (e.g., Amazon S3) is used to reduce the cost of data persistence by storing cold data. Note that some storage-disaggregated databases (e.g., Amazon Aurora) integrate the log store into the storage nodes.

To minimize communication overhead between compute and storage nodes, storage-disaggregated databases often adopt a design principle known as *log-as-the-database* [1, 3, 9, 15, 30, 36], where compute nodes send only logs to storage nodes during transaction commits, and storage nodes asynchronously replay the logs to reconstruct data pages.

2.2 Write and Read Path

For the write path, the primary compute node generates logs during transaction execution, as shown in Figure 2. These logs are flushed to the log store and persisted to ensure transaction durability. After that, the log store sends an acknowledgment back to the primary compute node to commit the transaction. The log store also asynchronously forwards the logs to the secondary compute nodes and the storage nodes. Upon receiving the logs, each secondary compute node checks whether they pertain to any pages currently cached in its buffer. If so, it replays the relevant logs to update its buffer pages; otherwise, it discards the logs, since any missing updates can be retrieved from the storage nodes later. Meanwhile, the storage nodes temporarily cache the incoming logs in

local storage. When the volume of accumulated logs exceeds a threshold, a background replay process is triggered to generate the new pages. To reduce storage costs, another background process periodically flushes cold pages and their unapplied logs to remote object storage, such as Amazon S3.

For the read path, when a compute node accesses a page not in its local buffer, it sends a request to the storage node. The storage node checks whether the target page is cached locally or in remote object storage. If cached, it is fetched and returned to the compute node. If not, the storage node retrieves the page's stale version and relevant logs from local disk and remote storage, replays the logs to reconstruct the page, caches it, and returns it to the compute node.

2.3 GetPage@LSN

As shown in Figure 2, the primary compute node and multiple secondary compute nodes share the same storage nodes. As described in Section 2.2, secondary compute nodes receive logs asynchronously. This means there is a delay before they receive and apply logs to update their local state, causing each secondary compute node to be at a different replication state. Thus, different compute nodes may request the same page but expect different versions based on their replication states. To support this, the storage engine must maintain multiple versions of each page.

With multi-version storage support, each compute node uses the Log Sequence Number (LSN) of the latest received log to indicate its current replication state. When a compute node needs to access a page, it issues a GetPage request along with its current LSN. This operation is known as GetPage@LSN [9, 36]. Upon receiving such a request, the storage node checks whether there are any unapplied logs related to the target page and replays them in LSN order to reconstruct the page version that matches the specified LSN. The reconstructed page is then returned to the compute node. This mechanism allows compute and storage nodes to scale independently while avoiding concurrency issues caused by replication lag across different compute nodes.

2.4 Background Log Replaying

With the log-as-the-database design, storage nodes may need to replay logs on demand to serve GetPage@LSN requests. This on-the-fly log replay introduces latency and can degrade overall database performance. To mitigate this, storage nodes run a background log replay process that proactively applies logs in the background. By reducing the number of unapplied logs at the time of a GetPage@LSN request, this approach helps reduce request latency and improve overall system performance.

To further reduce the cost of log replay, some databases, e.g., Amazon Aurora [36] and Neon [3], optimize GetPage@LSN by replaying only a small set of logs directly related to the target page, instead of scanning all logs in LSN order. Specifically, the storage node maintains a mapping between each page and its corresponding logs, forming a per-page log chain ordered by LSN. This design allows storage nodes to quickly retrieve only the minimal set of relevant logs for each GetPage@LSN request, thereby reducing its latency. The same mechanism is also fully utilized by the background log replay process.

Moreover, in some storage-disaggregated databases (e.g., Neon [3]), the background replay process is not continuously active; it is triggered periodically. Upon activation, it examines the log chain length for each page. If many pages have log chains that exceed a predefined threshold, the system begins replaying logs to reduce those chains to acceptable lengths. This mechanism helps ensure that most pages maintain short log chains, which stabilizes performance and reduces tail latency.

2.5 Log Store Design

As mentioned in Section 2.1, the log store is introduced to accelerate transaction commits. This is because committing a transaction requires compute nodes to flush logs to storage nodes to ensure durability. Therefore, accelerating log persistence is important. A widely adopted optimization – used in systems like Socrates [9] and Neon [3] – is to introduce a dedicated log layer between the compute and storage layers. This log layer is typically built using high-performance, low-latency devices such as NVMe SSDs, and employs distributed replication to ensure durability. To minimize cost, the log store retains logs only temporarily. As shown in Figure 2, it quickly persists incoming logs from compute nodes to enable transaction commits, then asynchronously forwards the logs to storage nodes for long-term retention, and eventually deletes them from its own local storage. In this way, the log store acts as a transient layer that accelerates transaction commits without increasing long-term storage costs.

2.6 Object Storage Layer

To further reduce costs, storage nodes often rely on low-cost storage devices for long-term data retention and ultra-high availability. A common approach is to organize data based on access frequency. Due to data locality, most data remains cold and is rarely accessed, while only a small portion is hot and accessed frequently. Therefore, cold data can be offloaded to slower, cheaper storage such as HDDs or cloud object stores. For example, in Neon [3], storage nodes store only hot data on local SSDs and offload cold data to remote object storage systems such as Amazon S3, as illustrated in Figure 2. This tiered storage strategy significantly reduces overall storage costs without sacrificing performance for frequently accessed data.

3 Performance Characterization

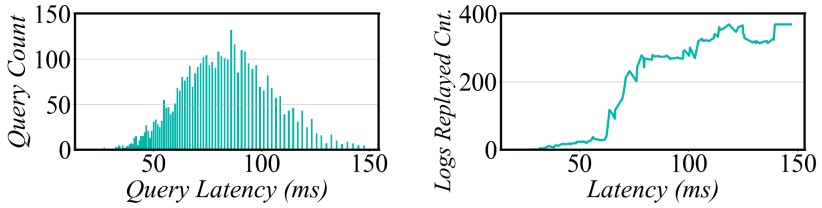
In this section, we provide a performance characterization to understand the root cause of the tail latency issue in storage-disaggregated databases. Specifically, we aim to validate two conjectures:

(1) The latency of foreground GetPage@LSN requests (at the storage node) is positively correlated with the number of logs being replayed.
(2) The background log replay process (also running on the storage node) competes for resources with the foreground GetPage@LSN requests, which increases request latency.

Open-source Platform. To verify these conjectures, the first question is which database to use. Amazon Aurora [36] is not an option because it is a closed-source system. We require an open-source disaggregated database, as this type of analysis often requires access to internal metrics for tail-latency requests – information that is typically not exposed through standard APIs in Aurora-like closed-source systems. By instrumenting the codebase of the open-source disaggregated database, we can precisely know when the background replay is triggered and how many logs are replayed, enabling us to effectively validate the conjectures.

As a result, we choose OpenAurora [4, 30] as our testbed because it is an open-source, storage-disaggregated database built on PostgreSQL. OpenAurora's compute node is built on PostgreSQL. Its storage node reuses PostgreSQL's log replication implementation to ingest updates from compute nodes and adopts Neon's LSM-based layered storage implementation [3] to organize and persist the replicated data, enabling cold data offloading to object storage for cost savings.

To verify the conjectures, we conducted two experiments to analyze the high tail-latency issue. We deployed OpenAurora [4, 30] in a disaggregated configuration (following Figure 2), consisting of a compute node, a log store, and a storage node connected via a 10 Gbps TCP/IP network. The compute node was provisioned with 8 CPU cores and 16 GB of memory, while the log store and



(a) Query Latency Distribution (b) Avg. Num. of Logs Replayed

Fig. 3. Relationship Between Query Latency and Number of Logs Replayed

the storage node were each equipped with 4 CPU cores, 8 GB of memory, and a 1.5 TB NVMe SSD. We used a 24 GB SysBench dataset [6] and configured the benchmark to run with 16 threads.

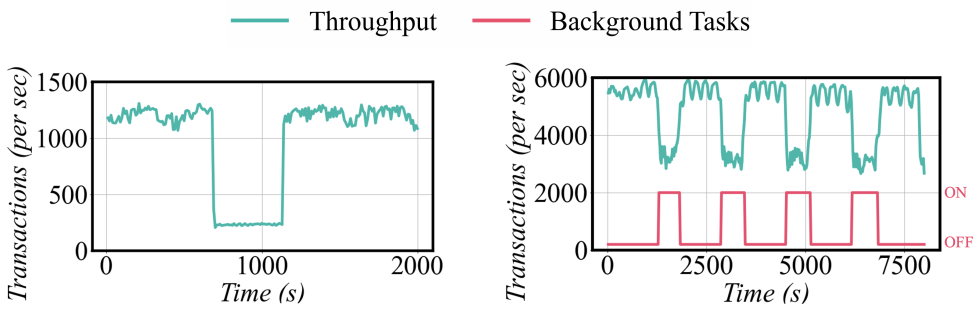
Experiment 1. The first experiment aims to verify the first conjecture – whether there is a strong correlation between the latency of GetPage@LSN requests and the number of logs replayed. This hypothesis is motivated by the observation that each GetPage@LSN request requires the storage node to replay the accumulated logs in the target page's log chain on-the-fly, as described in Section 2.4. Since log chain lengths can vary significantly, and log replay involves substantial computation and I/O, we suspect that GetPage@LSN requests with high latency tend to involve a larger number of logs to replay.

In this experiment, we instrument the codebase of OpenAurora [4, 30] and augment each GetPage@LSN request with additional metrics to record both the number of replayed logs and the latency observed by the compute node. We warm up OpenAurora using a 24 GB SysBench dataset [6] with an update-only workload. After a 120-second warm-up phase, we run an additional 10-second update-only workload to collect log metrics for analysis.

Figure 3a presents the query latency distribution. As shown, most queries exhibit latencies around 80 ms. However, some queries experience significantly higher latencies – approximately 150 ms – which are classified as tail latencies. For each group of queries with the same latency, we then calculate the average number of replayed logs, as shown in Figure 3b. **The results reveal a strong positive correlation between request latency and the number of logs replayed**. For example, queries with a latency of 86.5 ms replayed an average of 272 logs, while those with a latency of 145 ms replayed an average of 380 logs. In contrast, queries with low latencies (<50 ms) replayed fewer than 25 logs on average. These results demonstrate that GetPage@LSN queries requiring more logs to be replayed tend to experience longer replaying times, ultimately leading to higher request latencies. Therefore, this experiment validates our first conjecture.

Experiment 2. The second experiment aims to validate our second conjecture – that background log replay processes compete for limited resources with foreground GetPage@LSN requests, thereby increasing their latency. This hypothesis is inspired by observations from the prior Amazon Aurora experiment in Figure 1.

To further analyze the high tail latency issue observed in Figure 1, we reorganized the experimental results by grouping requests based on their start times into 10-second intervals and computing the average throughput for each interval. Plotting these intervals, as shown in Figure 4a, reveals a noticeable drop in throughput over short periods, which aligns with the expected execution pattern of background replay processes. However, since Amazon Aurora is closed-source and does not expose background replay metrics through its APIs, we cannot directly analyze the root cause of the throughput drop.



(a) Amazon Aurora Throughput (b) OpenAurora Throughput

Fig. 4. Database Throughput and Background Replay Tasks Execution

To address this, we analyze the same issue in OpenAurora [4, 30], an open-source alternative. We run a continuous update-only SysBench workload and monitor throughput changes. As shown in Figure 4b, OpenAurora also exhibits periodic throughput drops, similar to those observed in Aurora. We then instrument OpenAurora to log the start and end times of background replay tasks. By correlating these drops with the replay task logs in Figure 4b, we find that each throughput drop coincides with the launch of background replay tasks. Once these tasks complete, throughput returns to normal. These findings reveal a strong correlation between background replay execution and throughput degradation.

Next, we analyze the underlying reasons why background replay processes reduce database throughput. To do so, we use *perf* and *flame graphs* to analyze CPU usage on the storage node and investigate the relationship between throughput drops and background replay task execution. Specifically, we record CPU usage over two 10-second periods: one without background replay tasks and one with background replay. We classify the CPU usage into three categories: *GetPage@LSN* (foreground request processing), *Background Replay Tasks* (timers and execution logic), and *Others* (miscellaneous functions). As shown in Figure 5, when no background replay tasks are running, GetPage@LSN processing consumes 90.6% of the CPU in the storage node. And there are only 3.7% is used by the background replay process's periodic timeout trigger for threshold checking. However, during background replay execution, these tasks consume 43.0% of the CPU, reducing the CPU share available to GetPage@LSN to 49.4% – a 45.5% drop. This resource contention explains the observed throughput degradation.

Although background replay tasks are essential to reduce accumulated logs and lower future request latency, they are resource-intensive, involving heavy log processing and I/O. As a result, they limit the CPU resources available to foreground GetPage@LSN requests, increasing their latency. Consequently, all requests issued during background replay experience higher latency, contributing to the tail latency problem in disaggregated databases. **These findings reveal that background replay processes increase GetPage@LSN request latency by contending for resources, thereby supporting our second conjecture.**

4 System Overview

In this section, we present the main idea of RaaS (Replay-as-a-Service) in Section 4.1 and the overall system architecture that integrates RaaS into a storage-disaggregated database system in Section 4.2.

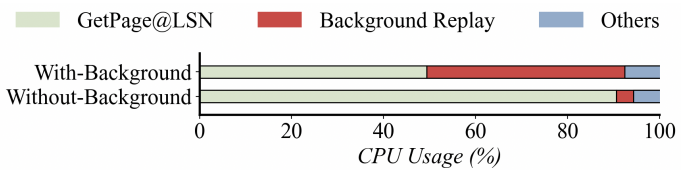


Fig. 5. CPU Usage on Storage Node

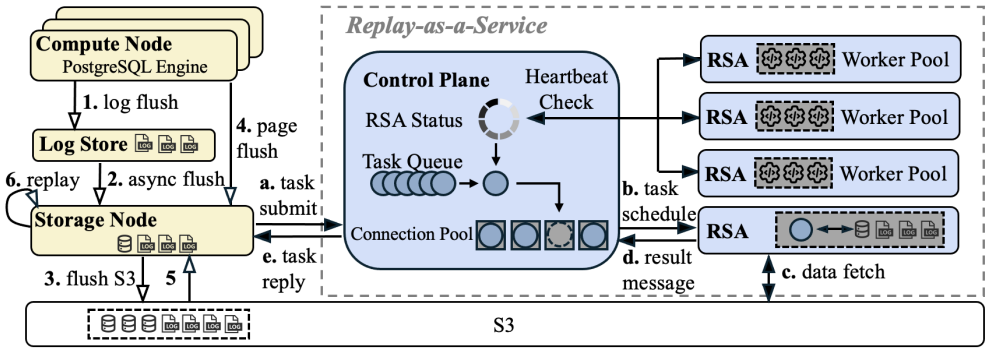


Fig. 6. System Architecture Overview

4.1 Main Idea

As shown in Section 3, tail latency in storage-disaggregated databases arises because different pages may require replaying varying numbers of logs. However, addressing this issue is challenging. On the one hand, the background log replay process should run aggressively on the storage node to shorten log chain lengths as much as possible. On the other hand, doing so consumes significant compute resources on the storage node, which can negatively impact foreground page accesses (on the storage node). Simply adding more compute resources to the storage node is also not an ideal solution, as it contradicts the principle of storage-compute disaggregation and may result in wasted resources when the workload is light.

In this paper, we propose a new technique called RaaS (Replay-as-a-Service). The main idea of RaaS is to decouple the background log replay logic from the storage engine and execute it using idle instances or even dedicated instances within the cluster. RaaS addresses the tail latency issue from two aspects: (1) RaaS can offload background log replay tasks to other servers. By moving these compute-intensive tasks away from the storage node, foreground GetPage@LSN requests no longer compete for resources with background replay, thus avoiding the high tail latency that occurs during background task execution. (2) By enabling background tasks to be executed on other servers, RaaS allows them to run more frequently and aggressively without concern for resource constraints on the storage node. More frequent background replay reduces log accumulation, shortens the log replay chain, and further reduces the latency of GetPage@LSN requests, thereby reducing overall tail latency.

Note that RaaS remains effective even when the storage node has ample compute resources. In such cases, replay tasks can still be executed locally by the storage node. This is because the storage node checks its available resources before offloading tasks to RaaS and will choose to execute them locally when resources are adequate. Thus, RaaS provides the flexibility in executing log replay tasks by decoupling them from the storage node.

However, building RaaS is non-trivial, as its design and implementation present several challenges:

- **C1:** How to address the code coupling between the background log replay logic and the foreground page access logic in the storage engine?
- **C2:** How to offload log replay tasks while minimizing CPU overhead on the resource-constrained storage node?
- **C3:** How to fully utilize the ample resources of idle instances to accelerate the log replay tasks?
- **C4:** How to design a scheduling policy that assigns log replay tasks to appropriate instances?

4.2 Overall Architecture

Figure 6 shows the architecture of RaaS (Replay-as-a-Service) integrated with OpenAurora. Note that RaaS is also compatible with other storage-disaggregated databases, such as AWS Aurora [36], Microsoft Socrates [9], Google AlloyDB [1], and Huawei Taurus [15]. In this paper, we use OpenAurora as an example to illustrate RaaS, and we refer to the system as OpenAurora-RaaS.

OpenAurora-RaaS retains all the core components of OpenAurora. RaaS only modifies OpenAurora's background replay mechanism and does not affect any other database functionalities, such as the read and write paths (as described in Section 2.2). In particular, RaaS introduces two new components: *replay service agent (RSA)* and *control coordinator*.

The RSA is a service designed to receive offloaded log replay tasks and execute them as a stateless function. It can be quickly deployed on any instance. When the host instance is idle, the RSA accepts replay tasks from the coordinator and executes them to utilize the idle resources. To execute a task, in addition to receiving data from the control coordinator, the RSA also fetches the necessary database files from AWS S3. In Figure 6, three RSAs are deployed, each capable of independently receiving and executing background replay tasks.

The control coordinator is a stateless service responsible for receiving background replay tasks from storage nodes and forwarding them to appropriate RSAs. Its stateless nature allows for quick recovery after crashes, and it can be deployed as either a single node or as part of a cluster. Since tasks may vary in computational cost, the coordinator selects an appropriate RSA based on available resources. To support this, it performs periodic heartbeat checks to collect each RSA's latest resource status. Task scheduling decisions are then made based on this information and predefined policies, which will be detailed in Section 5.4.

With RaaS, the background replay execution flow proceeds as follows. When background tasks are triggered, the storage node first checks whether it has sufficient local resources to execute the task. If so, it continues with its original local background replay workflow. Otherwise, the storage node offloads the background task to RaaS through the following steps: **(a)** The storage node collects and encapsulates the necessary metadata for the replay task – such as page-to-log mapping and log location indexes –and sends it to the RaaS control coordinator. **(b)** The coordinator receives the task and selects a suitable RSA based on its scheduling policies. **(c)** The coordinator forwards the task to the selected RSA. **(d)** Upon receiving the task, the RSA fetches any additional required database files from AWS S3, performs the replay, and flushes the resulting data files back to AWS S3. **(e)** The RSA reports the task status and the locations of the result files to the coordinator. **(f)** The coordinator forwards this information to the storage node, which then retrieves the result files from AWS S3.

5 System Design

In this section, we present the design choices in RaaS and explain how they address the challenges discussed in Section 4.1.

5.1 Decoupling Log Replay Logic

Background replay tasks were originally designed to run locally on storage nodes, with their logic tightly coupled to the handling of foreground GetPage@LSN requests. To enable offloading of these tasks, we must address two key issues: *how to make the tasks executable on RSAs* and *how to execute them efficiently*.

To make background replay tasks executable on RSAs, the RSA must reuse the database's replay logic, which includes parsing logs and applying them to update database files. However, executing these tasks also requires metadata, such as page-log mapping (i.e., which logs belong to which pages) and location metadata (i.e., where pages and logs are stored). This metadata is shared between background replay tasks and foreground GetPage@LSN requests, resulting in tightly coupled code. To decouple them, the storage node must extract the minimal metadata required for each task and send it to RaaS to guide remote execution. Since the RSA and the storage node now maintain separate copies of the metadata, a final metadata synchronization step is needed after each remote task to ensure consistency. This approach enables the decoupling of the background replay logic, making it executable on remote RSAs.

To improve offloading efficiency, we need to identify the appropriate point in the background replay task at which offloading should begin. This requires splitting the task into multiple steps and determining which parts should remain local to avoid unnecessary overhead. In storage-disaggregated databases (e.g., Neon), background replay usually involves two main steps: (1) a timeout-triggered metadata check that scans metadata to decide whether replay is needed, and (2) the actual log replay execution. We find that only the log replay step is compute-intensive, while metadata checks are lightweight but occur much more frequently. In many cases, the storage node offloads a task due to a timeout, but the RSA cancels it after the metadata check reveals that there are insufficient logs to justify a replay. These frequent cancellations waste resources on task packaging and network transmission.

To avoid this, we offload only the log replay step to RaaS, while keeping metadata checks on the storage node. This eliminates unnecessary network overhead from frequent task cancellations and improves offloading efficiency.

With background replay tasks made executable on remote RSA instances and a suitable offloading point identified, the code coupling issue described in Challenge **C1** is effectively addressed.

5.2 Minimizing Offloading Overhead

Simply offloading background tasks from storage nodes to remote RSAs may not lead to significant performance gains. This is because background replay tasks depend on large volumes of data, and preparing the data files (e.g., serialization and transmission) consumes substantial resources on storage nodes. Since storage nodes are already resource-constrained – otherwise, they would have executed the background replay themselves – it is essential to minimize the cost of data preparation and transfer during task offloading.

To minimize offloading overhead, we introduce a new mechanism that leverages cloud object storage (e.g., AWS S3) for efficient data transfers. In storage-disaggregated databases, in addition to the storage nodes, there is usually an additional layer of cloud storage for ultra-high availability. For example, Amazon Aurora and Neon use AWS S3, while Azure Socrates uses XStore. The files required for background replay tasks are usually stored in these cloud storage systems. Therefore,

instead of transferring large files directly, the storage node can send only the metadata to the RSA, indicating the locations of the required files in cloud storage. The RSA can then fetch these files directly. After completing the replay tasks, the RSA can also write the updated files back to cloud storage, allowing the storage node to retrieve them on demand. This design significantly reduces the overhead on the storage node during task offloading.

A straightforward solution is to flush all unflushed data from the storage node to cloud storage to ensure cloud storage has a complete set of data before each replay task. However, this introduces excessive data transfer and can also degrade the performance of foreground requests. We observe that each replay task typically requires only a small subset of the unflushed files. Based on this insight, we allow the storage node to selectively flush only the necessary unflushed files to cloud storage. This targeted synchronization is lightweight and preserves resources for foreground requests.

With the above optimizations, the storage node effectively minimizes the offloading overhead with RSAs during task offloading, as highlighted in Challenge **C2**.

5.3 Parallel Replay

Since background log replay consumes significant resources, it is typically executed in a single-threaded manner in current storage-disaggregated databases (e.g., Neon) due to the limited resources in storage nodes. With RaaS, these replay tasks can be offloaded to resource-rich instances, eliminating such constraints. To fully leverage the available resources and enhance execution efficiency, the log replay process must be redesigned to support efficient parallel execution.

There are two main steps involved in log replay for a particular page p. The first step is to collect the logs related to p. Since logs are initially stored in LSN order and partitioned into fixed-size files, logs for a single page may be scattered across multiple files, increasing access overhead. To address this, we group logs by page and place them in the same file. OpenAurora also implements this operation, but in a single-threaded manner. In contrast, we adopt a parallel MergeSort-based approach. Multiple threads read and locally sort logs by page. After all threads complete, a global merge and repartition is performed. This ensures that logs for the same page are colocated, reducing file access during GetPage@LSN.

The second step is to apply these logs to the base page of p. Since applying logs to a base page makes intermediate versions unavailable (considering that each page has multiple versions to support multiple compute nodes), the system must first verify that these versions are no longer needed. OpenAurora implements this operation using a single-threaded background task that scans base pages and applies logs sequentially. This approach is slow and underutilizes the ample resources available on RSAs once tasks are offloaded. To address this, we adopt a multi-threaded design, where each thread independently applies logs to different pages, ensuring parallelism without data conflicts.

To summarize, RaaS enables background log replay tasks to be executed in parallel, addressing Challenge **C3**. This approach fully utilizes the idle resources of remote instances, significantly improves the execution efficiency of replay tasks, and consequently reduces the latency of GetPage@LSN requests.

5.4 Control Coordinator

The control coordinator is a stateless service that manages background replay tasks offloaded from compute nodes and distributes them to appropriate RSAs. As shown in Figure 7, it consists of three key components: **Monitor**, **Scheduler**, and **Dispatcher**. The Monitor tracks available resources on each RSA instance. The Scheduler receives incoming tasks, adds them to the pending queue,

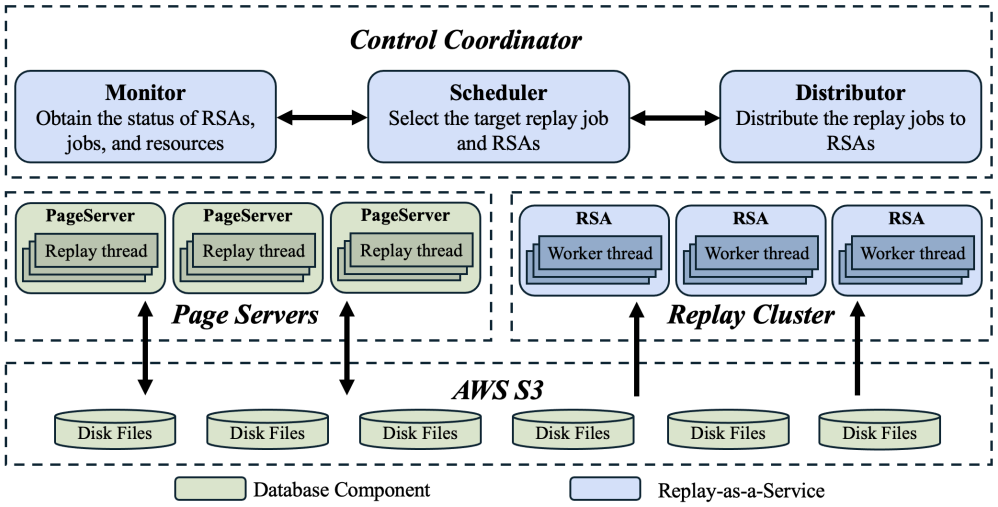


Fig. 7. Control Coordinator Design

and selects high-priority tasks for execution. The Dispatcher then sends the selected tasks to a suitable RSA.

5.4.1 Monitor. The control coordinator aims to assign each task to a suitable RSA based on its available resources. Since RSAs run as daemon services on shared instances alongside other workloads, assigning compute-intensive tasks to resource-constrained RSAs may degrade the performance of co-located services. To prevent this, the Monitor maintains a list of RSA IP addresses and periodically polls them for resource usage. Each RSA responds with its current resource availability, which is then collected by the Monitor.

The Monitor stores this information in an ordered list of entries in the format (available CPU and memory resources, RSA IP), sorted by available resources. This list enables the Dispatcher to filter out RSAs that lack sufficient capacity and to select the one with the most available resources. When a task is assigned, the estimated resource demand is subtracted from the selected RSA's entry in the list to reflect its updated state.

5.4.2 Scheduler. The Scheduler is responsible for receiving background replay tasks from storage nodes and selecting the task with highest priority for execution. To enable task offloading to RaaS, all storage nodes register with the control coordinator and maintain a persistent socket connection to the Scheduler. Replay tasks are submitted through this connection.

When a new task arrives, the Scheduler's behavior depends on the current queue state. If there are no pending tasks, the Scheduler immediately forwards the new task to the Dispatcher for execution. If multiple tasks are queued, it calculates their priorities and selects the highest-priority task to forward to the Dispatcher. Task priority is determined based on the following rules:

First, tasks with heavier workloads are assigned higher priority. Replay tasks are triggered when the average log chain length exceeds a threshold, indicating potential performance degradation. A heavier task implies a longer log chain and higher priority. The Scheduler estimates workload by analyzing task metadata and the number of associated disk files.

Second, to avoid starvation, task priority increases with waiting time. Lighter tasks may initially be deferred in favor of heavier ones, but this rule ensures they are eventually executed.

Third, extremely heavy tasks may be temporarily postponed. If no RSA has sufficient resources to execute a task, the Scheduler delays its dispatch. If the task's waiting time exceeds a predefined timeout, it is rejected to prevent indefinite queuing.

5.4.3 Dispatcher. The Dispatcher is responsible for selecting an appropriate RSA for the task chosen by Scheduler. It uses the ordered list of RSAs, maintained by the Monitor, which ranks instances by available resources. Upon receiving a task from the Scheduler, the Dispatcher first estimates the task's resource demand based on its metadata. Then, it selects a suitable RSA by the following heuristic strategy:

First, it filters out RSAs with resource utilization above 75%, as assigning background tasks to heavily loaded instances may cause contention and degrade the performance of co-located services.

Second, it excludes RSAs whose available resources are insufficient to meet the current task's estimated resource demand, which is estimated by the number of files involved in this task.

Third, if the RSA that previously served the same storage node is still eligible after the first two filters, the Dispatcher gives it preference. This RSA may still cache disk files – such as previously generated result files – that can be reused by the new task, reducing network overhead (e.g., fetching data from Amazon S3).

Fourth, if the previous RSA is not eligible, the Dispatcher selects the RSA with the most available resources among the remaining candidates.

To summarize, the control coordinator uses the Scheduler to accept and prioritize urgent tasks, and leverages the combination of the Monitor and Dispatcher to distribute tasks to suitable RSAs for execution. This design effectively addresses the task assignment challenge outlined in Challenge **C4**.

5.5 Fault Tolerance

As the RaaS is a new module introduced to storage-disaggregated databases, we now discuss how to handle RaaS-related failures. We classify possible failures into three levels: individual task failures on an RSA, RSA crash, and control coordinator crash.

First, task failures on an RSA may occur in two scenarios: corrupted data or network errors. The RSA automatically retries network-related failures until they succeed. For data corruption, the RSA notifies the storage node to retry the task offloading.

Second, an RSA may fail independently. The control coordinator detects such failures through periodic heartbeat messages and removes the corresponding RSA from the pool. If the failed RSA has an unfinished task, the control coordinator notifies the storage node to retry the task offloading.

Third, we address the crash of the control coordinator. The coordinator is designed as a stateless service, allowing it to restart quickly and recover once it receives resource reports from all RSAs. Also, it can be deployed redundantly to eliminate single points of failure and ensure high availability. However, if all coordinator instances fail simultaneously, the storage nodes will temporarily revert to operating without RaaS until the coordinator recovers.

Note that the fault tolerance described in this section is the *new types* of failures introduced by our RaaS component. Moreover, using a better configuration cannot solve the fault-tolerance issue in RaaS because it cannot prevent task failures on an RSA, RSA crashes, or control-coordinator crashes, which are specific to our RaaS component.

6 Integrating RaaS into OpenAurora

In this section, we present how we implement RaaS in OpenAurora. RaaS can also be implemented in other storage-disaggregated databases. The integration involves two key steps: (1) *migrating the storage node's background task execution code to RaaS*, and (2) *enabling the storage node to remotely invoke task execution via RaaS.*

First, migrating the storage node's background task execution code to RSA enables offloaded tasks to be correctly parsed and executed. To achieve this, we decouple OpenAurora's background log replay function, *compact_tiered*(), and deploy it on RSA as a stateless service called *remote_compact_tiered*(). This function becomes executable once it receives the necessary metadata from the storage node. The metadata includes four key components: (1) *KeySpace*, which specifies the range of page IDs involved in the replay task – typically the most frequently modified pages; (2) *LsnRange*, representing the time span with the most accumulation of logs; (3) A mapping index that links each page ID to its corresponding log IDs; and (4) A location index for locating the data files stored in AWS S3. With this metadata, RSA fetches the required data files from S3 and begins executing *remote_compact_tiered*(). Upon completion, the function generates two types of output files: *delta_layer* and *page_layer*, which contain the resulting logs and pages, respectively. RSA then flushes these files back to S3 and notifies the control coordinator of their locations as part of the response.

Second, we enable the storage node to invoke the stateless service on RaaS. In OpenAurora, background replay tasks are originally triggered by the *compaction_execution*() function, which serves as the entry point for log replay. As described in Section 5.1, this function includes two steps: (1) determining whether the accumulated logs reach replay task's activation threshold and (2) executing the actual log replay. Only the second step needs to be offloaded to RaaS. In detail, the first step checks metadata to generate *KeySpace* and *LsnRange*, then estimates the number of involved files to decide whether there are enough accumulated logs. The second step, which performs log replay, invokes the *compact_tiered*() function. We replace this function call with a trigger to RaaS's *remote_compact_tiered*() via the socket connection between the storage node and RaaS's control coordinator. Once the task completes, the storage node receives a response indicating whether the execution was successful and the locations of the result files in AWS S3. The storage node then updates its index metadata with these file locations, allowing the foreground GetPage@LSN requests to access the newly generated data and thus reduce requests latency.

7 Experimental Evaluation

7.1 Experiment Setup

For database deployment, we launch 8 OpenAurora instances in disaggregated mode. Each instance includes a compute node, a log store, a storage node and uses AWS S3 to store its cold data (following Figure 2). These nodes run on servers equipped with Intel Xeon Gold 6330 CPUs (2.0 GHz), 250 GB DRAM, and 1.5 TB NVMe SSDs, connected via a 10 Gb TCP/IP network. Each compute node is allocated 8 CPU cores and 32 GB DRAM. Each log store and storage node is allocated 4 CPU cores and 16 GB DRAM.

For the RaaS deployment, we provision a coordinator node with 2 CPU cores and 8 GB DRAM. The coordinator node connects each storage node with 10 Gb TCP/IP network as well. Additionally, each storage node hosts a co-located RSA service to utilize its idle resources without introducing additional resources. This deployment assumes that, in real-world scenarios, most storage nodes are located near each other. By placing RSA on a nearby storage node, the network overhead of task offloading can be reduced. However, RSA can also be deployed on compute nodes or dedicated instances if needed.

All experiments are conducted using a 86 GB SysBench dataset to evaluate system performance. We execute read-write mixed workload with 32 threads.

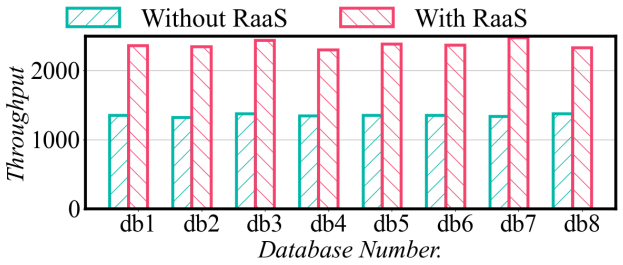


Fig. 8. Average Throughput Comparison

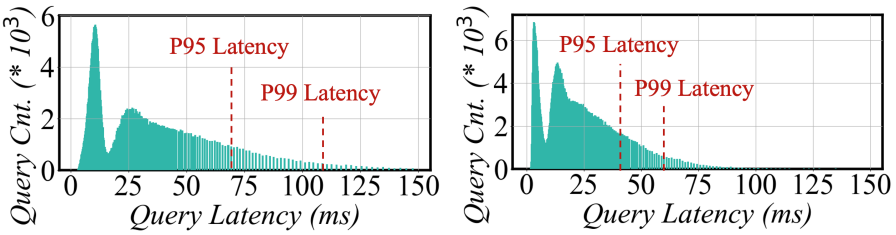


Fig. 9. Query Latency Distribution in Neon

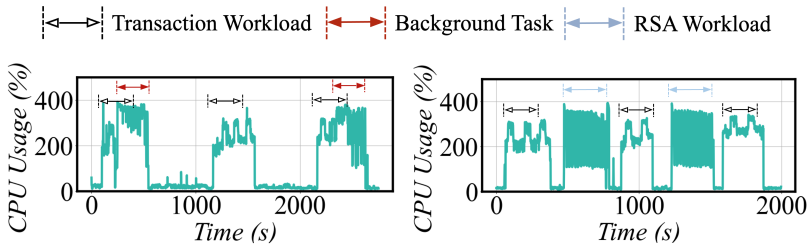
7.2 Overall Throughput

In this experiment, we evaluate the throughput improvement enabled by RaaS by comparing the performance of two database clusters – one with RaaS enabled and one without. Specifically, we deployed 8 OpenAurora database instances in disaggregated mode, each loaded with an 86 GB SysBench dataset. We first ran a 10-minute mixed read/write workload to warm up the databases. Then, we executed three rounds of 5-minute mixed read/write workloads on each instance, with a 10-minute idle interval between rounds to simulate real-world intermittent workloads. To model workload variation across users, the 8 databases did not receive workloads simultaneously; instead, db5 – 8 experienced an 8-minute delay compared to db1 – 4.

The first cluster consisted of the original 8 OpenAurora instances without RaaS integration. Each instance executed its workload independently. We measured the average transaction throughput at one-second intervals.

The second cluster used the same 8 OpenAurora instances, configured identically, but integrated with RaaS. In this setup, each storage node hosted a dedicated RSA. The same workload was applied to this cluster.

Figure 8 presents the results. Without RaaS, the average throughput across all instances was 1351 transactions per second. With RaaS enabled, the average throughput increased significantly to 2376 transactions per second. This represents a **75.9%** improvement, achieved solely by offloading compute-intensive background tasks to idle storage nodes via RaaS. By offloading log replay tasks, RaaS eliminates resource contention between foreground GetPage@LSN requests and background replay. Additionally, the frequent and efficient log replays ensure that each GetPage@LSN request requires only a small number of logs to be applied.



(a) CPU Usage without RaaS (b) CPU Usage with RaaS

Fig. 10. CPU Usage

7.3 Evaluating Tail Latency

In this experiment, we analyze the query latency drop brought by enabling RaaS. Using the same 8-instance setup as in the previous experiment, we compare two clusters: one with RaaS and one without. After executing three rounds of 5-minute mixed workloads, we collect and analyze the query latency histograms of db1 from each cluster.

Figure 9a shows the latency histogram of db1 without RaaS, while Figure 9b shows the result with RaaS. Without RaaS, the P95 latency is 68.28 ms; with RaaS, it drops to 40.9 ms—a **40.1%** reduction. For the P99 latency, it drops from 106.75 ms to 62.19 ms, resulting in a **41.7%** latency drop.

Each histogram in Figure 9 displays two noticeable peaks. The first peak corresponds to queries that involve no or very limited log replay, and their latency mainly comes from network delay. These queries access pages that either experienced few updates or had their accumulated logs already applied through prior log replay tasks. The second peak represents queries that require extensive log replay from multiple log files, and their latency is dominated by disk I/O and replay computation.

By comparing the two histograms, we observe that in the cluster with RaaS, more queries fall into the first peak than in the cluster without RaaS, indicating reduced log accumulation. Specifically, with RaaS, 53.3% of queries complete within 10 ms, compared to only 32.5% without RaaS—an increase of 20.8%. As for the second peak, without RaaS, the query latency is more flat distributed. Specifically, there are 1.33% queries exceed 100 ms latency, while there are only 0.06% queries exceed 100 ms latency with RaaS enabled. This shows that RaaS enables more frequent and efficient log replay, which reduces log accumulation on frequently updated pages and significantly drops the tail latency.

7.4 CPU Utilization of Database Cluster

In this experiment, we use *perf* to analyze the CPU utilization of different components within the storage node during mixed read and write workloads. The goal is to understand how RaaS reduces tail latency by resolving the resource contention issue between foreground GetPage@LSN requests and background log replay tasks.

We record the CPU utilization of the storage node under two configurations: one without RaaS and one with RaaS. Each storage node is allocated 4 CPU cores, so 400% utilization represents full usage of all cores. The figures show how CPU usage changes over time under both setups.

Figure 10a shows the CPU utilization of a standalone OpenAurora instance without RaaS. In the figure, black arrows indicate the workload execution periods, and red arrows mark when background log replay tasks are triggered. Three workloads are executed during the experiment.

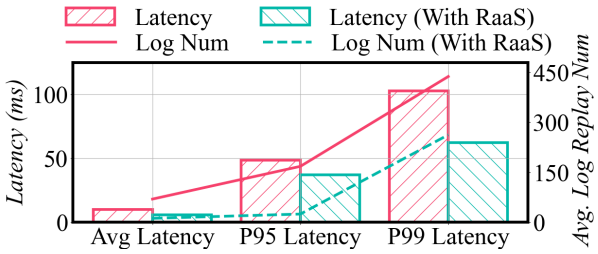


Fig. 11. Tail Latency and Log Replaying

In the first and third workload periods, the storage node initiates background replay because the xlog buffer exceeds its threshold due to continuous xlog inflow from the compute node. During the second workload, where no background tasks are triggered, CPU utilization remains between 200% and 320%, meaning each core operates at around 50% to 80% utilization. This level is sufficient for user workloads and avoids performance degradation due to excessive CPU saturation.

However, during the first and third workloads, after background tasks are triggered, CPU utilization reaches up to 400%. As shown in Figure 5b, only 49.4% of CPU resources are used for foreground queries, leaving around 197.6% available—far below the 320% required for optimal performance. Moreover, 100% utilization on each core increases kernel scheduling overhead, which further degrades performance. Notably, after the second workload finishes, CPU usage remains above 200% for around 50 seconds, even without background replay. This is because the xlog production rate at the compute node exceeds the xlog consumption rate at the storage node, leading to xlog accumulation in the storage node's memory. The storage node temporarily buffers these logs and gradually parses and persists them after the workload.

Figure 10b shows the CPU utilization of a storage node participating in the RaaS cluster. Again, three workloads are executed. And we use blue arrows to indicate RSAs executing offloaded tasks from other servers. During each transaction workload, CPU utilization ranges from 200% to 320%, similar to the second workload in Figure 10a, which was unaffected by background replay. Unlike the standalone setup, no background replay is visible during these periods, even though background tasks are triggered. This is because RaaS offloads replay tasks to other idle servers, avoiding contention for CPU resources and maintaining stable performance. During the idle intervals between workloads, the RSA receives and executes replay tasks from other nodes. As indicated by the blue arrows, these tasks consume around 360% CPU, but since no user transactions are active, they do not impact database performance.

In a standalone setup, the storage node must be provisioned to handle the combined peak demand of foreground workloads and background replay, since both can occur simultaneously. However, in practice, user workloads are typically bursty, leading to idle periods where background tasks either compete for resources or leave them unused. This coupling results in underutilized or overwhelmed resources, depending on the timing.

RaaS decouples these two resource demands. When under load, a storage node offloads its background tasks to other nodes; when idle, it accepts tasks from others. This dynamic task exchange allows the cluster to fully utilize existing resources without adding new hardware. In our experiment, the standalone storage node was idle for 48% of the first 2000 seconds, while the RaaS-enabled node was idle for only 18%. This demonstrates that RaaS improves system performance primarily by increasing resource utilization rather than by increasing provisioned capacity.

7.5 Log Replay Number Analysis

This experiment demonstrates how RaaS reduces tail latency by mitigating log accumulation.

We compare two configurations: a standalone database and a database connected to RaaS. Both are prewarmed using a 1200-second update-only workload. After prewarming, we run a 30-second update-only workload and measure the average latency, P95 latency, and P99 latency for each setup. Additionally, for every GetPage@LSN request, we record the number of logs replayed on the fly. For each latency bucket shown in Figure 11, we identify all requests with that latency and compute the average number of logs replayed to serve them.

First, we analyze the standalone database. The results show a strong correlation between request latency and the number of logs replayed. The average request latency is 9.95 ms, with an average of 69.4 logs replayed per request. The P95 latency is 48.34 ms — around 485% of the average latency — and corresponds to requests replaying 167.8 logs (241% of the average log count). The P99 latency reaches 102.69 ms (1031% of the average), and these requests replay 437.8 logs on average (630% of the average log count). This confirms that requests with more logs to replay tend to experience higher latency.

Next, we compare this with the RaaS-enabled database. On average, it replays only 11.5 logs per request—just 16.6% of the standalone case—and achieves a lower average latency of 5.88 ms (59.1% of the standalone latency). For tail latency, the P95 and P99 requests in the RaaS setup replay only 24.4 and 262.5 logs, which are 14.5% and 59.95% of the respective replay counts in the standalone database. As a result, the P95 latency drops by 26.91% (from 48.34 ms to 35.33 ms), and the P99 latency drops by 37.3% (from 102.69 ms to 64.35 ms).

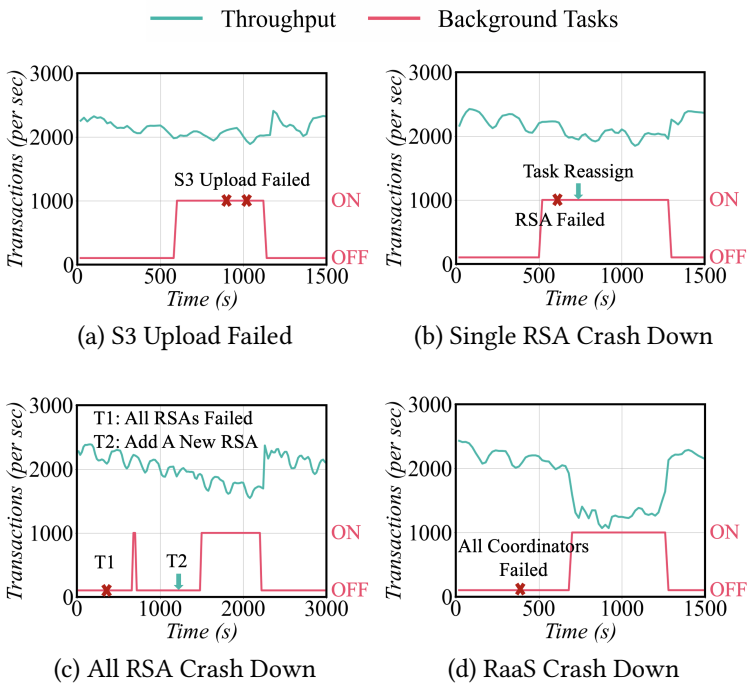
These results demonstrate that RaaS effectively executes background replay tasks in advance, reducing the number of logs that each GetPage@LSN request must replay. This reduction in replay effort significantly decreases tail latency.

7.6 Failure Processing

We present four failure recovery scenarios. First, we consider network failures occurring during RSA task execution. An RSA must download large chunks of data from S3 to execute replay jobs, and upload newly generated files back to S3. During these stages, network instability may cause download or upload failures. The RSA detects such failures and retries the corresponding operations after a short delay. As shown in Figure 12a, the RSA successfully generates new data files, but two upload failures occur when sending these files to S3. The RSA automatically retries the uploads without reporting the failure to the coordinator or the storage node. As a result, the storage node's background tasks proceed unaffected, and system performance remains stable.

Second, the RSA may crash due to reasons such as power loss or host machine failure, and it may not recover immediately. Figure 12b illustrates this scenario. After the storage node offloads a background task, the assigned RSA crashes during execution. The coordinator detects the disconnection, marks the failed RSA as unavailable, and reschedules the task to another healthy RSA, which completes it successfully. Throughout this process, the storage node remains unaware of the failure and does not reissue the task. The only observable effect is a delay in task completion—approximately 200 seconds longer than usual—which does not lead to noticeable performance degradation.

Third, we simulate a scenario where all RSAs are temporarily unavailable. To do so, we initially connect only one RSA to the coordinator. As shown in Figure 12c, we then shut down this RSA, effectively removing all RSAs from the system. When the storage node later issues a background task request, the coordinator rejects it due to the absence of available RSAs. As a result, the storage node abandons the request and waits for the next opportunity to trigger a background task. After that, we add a new RSA to the system. Upon the next replay trigger, the coordinator successfully schedules the task to the newly joined RSA. In this scenario, the storage node may temporarily miss a background replay opportunity, but this has no long-term impact, as complete RSA unavailability is rare and usually short-lived.



(a) S3 Upload Failed (b) Single RSA Crash Down

(c) All RSA Crash Down (d) RaaS Crash Down

Fig. 12. Evaluating Failure Recovery

Table 1. Networking Reduction by Utilizing S3

	Data Sending	Data Receiving
Without RaaS	38 GB	11 GB
With RaaS	3 GB	26 KB

Fourth, coordinator failure is simulated by shutting it down after several successful task offloads (Figure 12d). The storage node detects the unavailability and falls back to local replay. Performance temporarily reverts to that of a standalone node, but background replay continues, with no risk of data loss or storage node failure.

7.7 Networking Overhead Reduction

Table 1 evaluates how leveraging AWS S3 reduces the storage node's offloading overhead. In the baseline setup, the storage node must transfer large data files to RSAs for task preparation and later receive updated files after replay completion, incurring substantial network costs. Our results in Table 1 show that leveraging AWS S3 can reduce this networking overhead.

First, during task initialization, each storage node prepares about 33 GB of data (152 files, 256 MB each) for RSAs. With S3 integration, the node only synchronizes recently generated unflushed files averaging 3 GB (12 files, 256 MB each) and sends a small 18 KB metadata message to RaaS. Second, during task completion, the storage node would otherwise receive 11 GB of replayed data (44 files, 256 MB each) from RSAs. With S3, RSAs directly flush results to S3–the storage node's data repository–so the node only receives a 26 KB metadata message to synchronize results.

Overall, leveraging S3 in the offloading path reduces the storage node's networking overhead by 93.9% (from 49 GB to 3 GB).

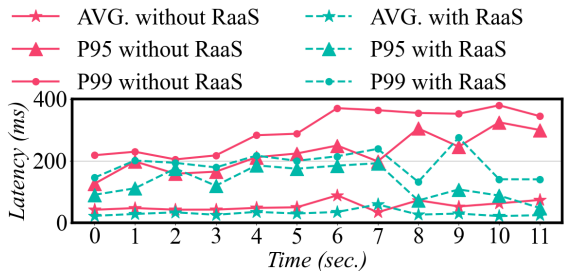


Fig. 13. Experiments on TPC-C

Table 2. Evaluating Read-Only Queries

	Core Cnt.	Avg.	P95	P99	P999
Without RaaS	2	15.58	28.67	35.58	44.97
With RaaS	2	15.70	29.19	36.89	45.79
Without RaaS	4	2.67	6.43	9.06	12.52
With RaaS	4	2.60	6.09	8.58	12.30
Without RaaS	8	2.28	4.65	7.29	11.45
With RaaS	8	2.29	4.67	7.29	11.87

7.8 Results on TPC-C

We conducted experiments to verify RaaS's performance on TPC-C. We prepared a 35 GB TPC-C dataset and fully warmed up the database. Then we evaluated the database with and without RaaS separately. The results are shown in Figure 13. The results show that RaaS still reduces tail latency for TPC-C. For example, with RaaS enabled, the P95 latency decreases by 42.76% (from 225.49 ms to 129.08 ms) and the P99 latency decreases by 36.81% (from 300.43 ms to 189.84 ms).

7.9 Read-only Workload

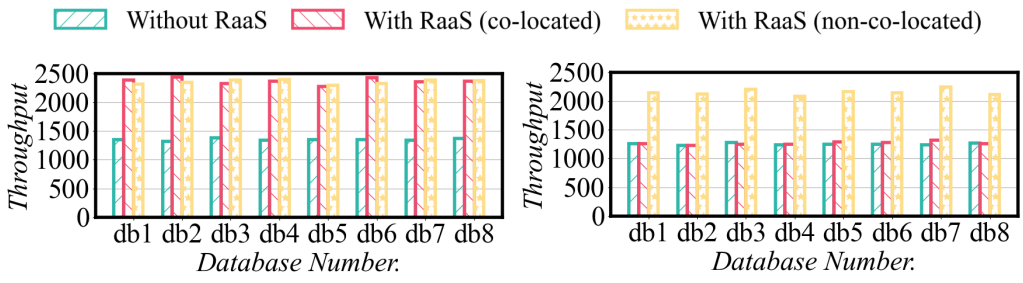
In this experiment, we analyzed RaaS performance on read-only workloads under three resource configurations consistent with Section 7.2. Before running the experiment, we warmed up the database and waited for all background tasks to complete, ensuring that accumulated logs were cleared. As shown in Table 2, the tail latency was significantly reduced. For example, compared with the write workload results in Figure 9a, the P95 latency dropped from 68.28 ms to 6.43 ms (90.6% reduction), and the P99 latency dropped from 106.75 ms to 9.06 ms (91.5% reduction) under the same configuration.

Moreover, RaaS exhibited similar performance to the case without RaaS across all three configurations. This is because, under a read-only workload, no logs are accumulated on storage nodes, and any replay tasks triggered by timers are canceled before reaching the RaaS execution stage.

7.10 Non-co-located RSAs

In this experiment, we evaluate RaaS under a non-co-located scenario, where RSAs are deployed on dedicated nodes separate from compute and storage nodes. We evaluate RaaS performance within the same cluster configuration described in Section 7.2 and compare it with both the co-located RaaS deployment and the setup without RaaS.

Base on the experiments, the non-co-located deployment achieved 2365 transactions per second (TPS), while the configuration without RaaS reached 1351 TPS, demonstrating that RaaS remains effective even when deployed on dedicated nodes. The co-located deployment achieved 2376 TPS,



(a) Throughput on Burst Workload

(b) Throughput on Non-Burst Workload

Fig. 14. Evaluating Non-co-located Deployment and Non-burst Workload

Table 3. RaaS vs. Local Parallel Replay

	Avg.	P99	P999
Local Single-thread Replay	23.69	106.75	164.45
Local Parallel Replay	19.83	147.61	308.84
RaaS Single-thread Replay	16.28	81.48	144.97
RaaS Parallel Replay	13.47	62.19	92.42

showing similar performance to the non-co-located setup. This is because, in the co-located setting, storage nodes offload replay tasks to other idle storage nodes, achieving comparable functionality to the non-co-located setting, which offloads to dedicated nodes.

7.11 Non-bursty Workload

In this experiment, we evaluate RaaS performance under non-bursty workload with all database's within the cluster are continuously experiencing heavy write workload. As shown by Figure 14b, under this non-bursty workload, the average performance of RaaS is 1267 and the average performance without RaaS is 1253, which means under non-bursty heavy workload, the RaaS would not introduce noticeable effect on database performance. This is expected because, under non-bursty workloads, all the storage nodes are fully occupied without idle resources to perform RSAs, especially in the co-located deployment. Thus, the RaaS with co-located deployment targets bursty workloads that are common in real-world scenarios, especially in the cloud.

However, Figure 14b also shows that if additional servers (different from storage or compute nodes) are allowed to be used for log replay, i.e., under the non-co-located deployment, our RaaS can still improve performance even for non-bursty workloads, e.g., the throughput improves to 2151 TPS with RaaS.

7.12 Evaluating Parallel Replay

In this experiment, we evaluate the effectiveness of parallel replay, as shown in Table 3. The results show that local parallel replay cannot mitigate the tail latency issue. For example, it even increases the P99 latency by 38.3% (from 106.75 ms to 147.61 ms) because parallel replay consumes significant system resources, leaving fewer resources for foreground `GetPage@LSN` requests. Note that in Section 4.1, we also explained that simply adding more compute resources to the storage node is not an ideal solution, as it contradicts the principle of storage-compute disaggregation and may result in wasted resources when the workload is light.

We further evaluate the effectiveness of parallel replay within our RaaS. It shows that by incorporating parallel replay into our RaaS, the average latency decreases by 17.3% (from 16.28 ms to

Table 4. Replay Frequency

Replay Interval	Avg.	P99	P999
5 mins	22.72	152.15	324.45
2 mins	19.83	147.61	308.84
1 min	19.86	147.19	295.29
0.5 min	19.82	148.52	303.80

13.47 ms), and the P95 latency drops by another 23.7% (from 81.48 ms to 62.19 ms), by comparing "RaaS Parallel Replay" and "RaaS Single-thread Replay" in Table 3.

7.13 Evaluating Replay Frequency

In this experiment, we analyze how replay frequency affects database performance. We deploy OpenAurora without RaaS and run a 10-minute update-only workload while varying the replay frequency from 30 seconds to 5 minutes. The results are shown in Table 4. The experiment shows that increasing replay frequency does not significantly reduce tail latency. For example, when the replay interval decreases from 5 minutes to 2 minutes (i.e., more frequent), the P99 latency improves by only 3.0%. The improvement is limited because, although more frequent replay reduces log accumulation, the replay process runs concurrently with the workload, causing resource contention that slows down foreground `GetPage@LSN` requests. Moreover, when the replay interval becomes even shorter (e.g., 1 minute or 30 seconds), the performance remains unchanged. This is because replay tasks check whether the accumulated logs exceed the execution threshold and cancel if not. Under aggressive replay intervals, most tasks are canceled due to insufficient log accumulation.

8 Related Work

Storage-Disaggregated Databases. This paper focuses on storage-disaggregated databases [40], in particular OLTP databases. Examples of databases in this category are Amazon Aurora [36], OpenAurora [30], Microsoft Socrates [9], Google AlloyDB [1], Huawei Taurus [15], Alibaba PolarDB [5, 11], and Neon [3]. However, none of them have addressed the tail latency issue that this paper focuses on. This paper is the first one that optimizes the tail latency problem in storage-disaggregated OLTP databases. In the literature, there are also disaggregated OLAP databases (e.g., Snowflake [13, 38], AnalyticDB [46], Polaris [8], Redshift [10, 29], and Dremel [27]), but this work focuses on disaggregated OLTP databases.

Reducing Tail Latency in Data Systems. Reducing tail latency has been studied in various data systems, but the storage-disaggregated OLTP context presented a unique challenge due to its log-as-the-database design, which prior work did not address. For example, Li et al. [25] conducted a systematic study of tail latency in high-throughput servers running on multi-core systems, identifying key system-level factors that contributed to latency spikes. Bonspiel proposed designing new concurrency control protocols to mitigate tail latency in geo-distributed databases [12]. Their insights were complementary to our RaaS approach, but this paper focused on reducing tail latency from the unique perspective of log replay specific to storage-disaggregated databases.

Focusing on datacenter applications, Xu et al. [43] identified performance interference from co-located virtual machines – often referred to as "noisy neighbors" – as a major source of tail latency. They proposed Bobtail, a system that proactively detects and avoids such interference. But their approach focuses on isolating resource contention among multiple tenants. Our work complements Bobtail by focusing on eliminating resource contention within a single, resource-constrained VM through offloading background tasks to remote storage instances.

In information retrieval systems, tail latency is often driven by slow queries or heavy background tasks. To address this, Haque et al. [16] proposed dynamically accelerating slow queries by allocating additional resources and increasing parallelism based on the task's execution time. The longer a task runs, the more resources it receives, helping to shorten its remaining runtime. This adaptive parallelism mechanism can also be combined with RaaS to improve the efficiency of handling extremely heavy log replay tasks.

In addition to reactive strategies, several works [20, 21, 23] focus on predicting slow queries ahead of time. For example, Jeon et al. [20] proposed estimating each task's execution time and resource requirements, using dynamic correction techniques to handle prediction errors. Similarly, Jeon et al. [21] introduced methods for identifying long-running queries and selectively parallelizing them to reduce latency. While these techniques could be useful for identifying heavy background replay tasks, they do not address the tail latency issue in storage-disaggregated databases. Our RaaS approach directly tackles this challenge by offloading resource-intensive background tasks to remote, underutilized nodes.

Other studies [18, 37] propose leveraging idle resources on other servers to reduce tail latency. Jalaparti et al. [18] explored reissuing slow queries, returning partial results, and assigning additional resources to selected queries. Vulimiri et al. [37] proposed initiating redundant operations across idle servers and returning the first completed result. While these approaches share the high-level idea of exploiting idle resources, they are not directly applicable to storage-disaggregated databases.

Component Decoupling in Data Systems. Offloading compute-intensive components from resource-constrained databases to idle servers is a widely adopted strategy. This follows the recent trend of building composable data systems [31, 32], which aims to decouple individual modules of a complex data system to gain benefits such as customized optimizations, streamlined development, independent scaling, and elasticity. For example, CaaS-LSM decouples compaction in LSM-based systems as a service [44], Microsoft Oasis decouples the query optimizer as a service [22], Amazon Redshift decouples compilation as a service [10], and PostgreSQL-V decouples vector indexes from the PostgreSQL engine [26]. Our work is inspired by this line of research and decouples log replay from storage-disaggregated databases, which is a new context that prior work has not studied.

9 Conclusion

In this paper, we first diagnosed the root cause of high tail latency in storage-disaggregated databases, identifying two key factors: accumulated logs increase GetPage@LSN request latency, and background log replay tasks contend for resources with foreground queries. To address these issues, we proposed decoupling log replay as a stateless service. Specifically, we outlined the challenges and presented our solutions through the design and implementation of the Replay-as-a-Service (RaaS) framework, which enables offloading log replay tasks from resource-constrained databases to idle instances. Our evaluation shows that RaaS significantly improves throughput and reduces tail latency without requiring additional hardware resources.

As future work, an interesting direction is to investigate tail latency in storage-disaggregated databases under multi-tenant environments [28], where resource sharing and interference may introduce new sources of tail latency. Another direction is to study tail latency in memory-disaggregated databases [41, 42], in which RDMA-based or CXL-based remote memory access may introduce additional latency variability.

Acknowledgements

Jianguo Wang acknowledges the support of the National Science Foundation under Grant Number 2337806.

References

[1] [n. d.]. AlloyDB for PostgreSQL, https://cloud.google.com/alloydb.
[2] [n. d.]. Amazon Aurora Customers, https://aws.amazon.com/rds/aurora/customers/.
[3] [n. d.]. Neon, https://github.com/neondatabase/neon.
[4] [n. d.]. OpenAurora, https://github.com/purduedb/OpenAurora.
[5] [n. d.]. PolarDB for PostgreSQL, https://github.com/ApsaraDB/PolarDB-for-PostgreSQL.
[6] [n. d.]. SysBench Manual, https://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf.
[7] 2025. Databricks Agrees to Acquire Neon to Deliver Serverless Postgres for Developers + AI Agents, https://www.databricks.com/company/newsroom/press-releases/databricks-agrees-acquire-neon-help-developers-deliver-ai-systems.
[8] Josep Aguilar-Saborit and Raghu Ramakrishnan. 2020. POLARIS: The Distributed SQL Engine in Azure Synapse. *PVLDB* 13, 12 (2020), 3204–3216.
[9] 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.
[10] 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.
[11] 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.
[12] Fan Cui, Eric Lo, Srijan Srivastava, and Ziliang Lai. 2025. Bonspiel: Low Tail Latency Transactions in Geo-Distributed Databases. *PVLDB* 18, 11 (2025), 3840–3853.
[13] 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.
[14] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (2013), 74–80.
[15] 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.
[16] Md. Enamul Haque, Yong Hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. 2015. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *ASPLOS*. 161–175.
[17] Rubaba Hasan, Timothy Zhu, and Bhuvan Urgaonkar. 2024. AutoBurst: Autoscaling Burstable Instances for Cost-effective Latency SLOs. In *SoCC*. 243–258.
[18] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. 2013. Speeding Up Distributed Request-response Workflows. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 219–230.
[19] Deepal Jayasinghe, Simon Malkowski, Jack Li, Qingyang Wang, Zhikui Wang, and Calton Pu. 2014. Variations in Performance and Scalability: An Experimental Study in IaaS Clouds Using Multi-Tier Workloads. *IEEE Transactions on Services Computing* 7, 2 (2014), 293–306.
[20] Myeongjae Jeon, Yuxiong He, Hwanju Kim, Sameh Elnikety, Scott Rixner, and Alan L. Cox. 2016. TPC: Target-Driven Parallelism Combining Prediction and Correction to Reduce Tail Latency in Interactive Services. In *ASPLOS*. 129–141.
[21] Myeongjae Jeon, Saehoon Kim, Seung-won Hwang, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. 2014. Predictive Parallelization: Taming Tail Latencies in Web Search. In *SIGIR*. 253–262.
[22] Alekh Jindal and Jyoti Leeka. 2022. Query Optimizer as a Service: An Idea Whose Time Has Come! *SIGMOD Record* 51, 3 (2022), 49–55.
[23] Saehoon Kim, Yuxiong He, Seung-won Hwang, Sameh Elnikety, and Seungjin Choi. 2015. Delayed-Dynamic-Selective (DDS) Prediction for Reducing Extreme Tail Latency in Web Search. In *WSDM*. 7–16.
[24] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. 2020. Enabling Low Tail Latency on Multicore Key-Value Stores. *PVLDB* 13, 7 (2020), 1091–1104.
[25] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *SoCC*. 9:1–9:14.
[26] Jiayi Liu, Yunan Zhang, Chenzhe Jin, Aditya Gupta, Shige Liu, and Jianguo Wang. 2026. Fast Vector Search in PostgreSQL: A Decoupled Approach. In *CIDR*.

[27] 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.
[28] Vivek Narasayya and Surajit Chaudhuri. 2022. Multi-tenant Cloud Data Services: State-of-the-art, Challenges and Opportunities. In *SIGMOD*. 2465–2473.
[29] Ippokratis Pandis. 2021. The Evolution of Amazon Redshift. *PVLDB* 14, 12 (2021), 3162–3163.
[30] Xi Pang and Jianguo Wang. 2024. Understanding the Performance Implications of the Design Principles in Storage-Disaggregated Databases. *Proc. ACM Manag. Data* 2, 3 (2024), 180:1–26.
[31] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satyanarayana R. Valluri, Mohamed Zaït, and Jacques Nadeau. 2023. The Composable Data Management System Manifesto. *PVLDB* 16, 10 (2023), 2679–2685.
[32] Pedro Pedreira, Deepak Majeti, and Orri Erling. 2024. Composable Data Management: An Execution Overview. *PVLDB* 17, 12 (2024), 4249–4252.
[33] Xiaoting Qin, Minghua Ma, Yuheng Zhao, Jue Zhang, Chao Du, Yudong Liu, Anjaly Parayil, Chetan Bansal, Saravan Rajmohan, Íñigo Goiri, Eli Cortez, Si Qin, Qingwei Lin, and Dongmei Zhang. 2023. How Different are the Cloud Workloads? Characterizing Large-Scale Private and Public Cloud Workloads. In *IEEE/IFIP International Conference on Dependable Systems and Network (DSN)*. 522–530.
[34] Amazon Web Services. 2020. Why Tens Of Thousands Of Companies Rely On Amazon Aurora To Power Their Most Important And Demanding Applications, https://www.forbes.com/sites/amazonwebservices/2020/11/04/why-tens-of-thousands-of-companies-rely-on-amazon-aurora-to-power-their-most-important-and-demanding-applications/.
[35] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *OSDI*. 69–87.
[36] 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.
[37] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. 2013. Low Latency via Redundancy. In *CoNEXT*. 283–294.
[38] 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.
[39] Hui Wang. 2024. Burst Load Frequency Prediction Based on Google Cloud Platform Server. *IEEE Trans. Cloud Comput.* 12, 4 (2024), 1158–1171.
[40] Jianguo Wang and Qizhen Zhang. 2023. Disaggregated Database Systems. In *SIGMOD*. 37–44.
[41] Ruihong Wang, Jianguo Wang, and Walid G. Aref. 2025. Cache Coherence Over Disaggregated Memory. *PVLDB* 18, 9 (2025), 2978–2991.
[42] 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.
[43] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. 2013. Bobtail: Avoiding Long Tails in the Cloud. In *NSDI*. 329–341.
[44] 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. *Proc. ACM Manag. Data* 2, 3 (2024), 124: 1–28.
[45] Matei Zaharia. 2025. Bringing the Operational and Analytical Worlds Together with Lakebase. *PVLDB* 18, 12 (2025), 5539.
[46] 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.

Received July 2025; revised October 2025; accepted November 2025