CloudJump II: Optimizing Cloud Databases for Shared Storage

Zongzhi Chen
Alibaba Cloud
Hangzhou, China
zongzhi.czz@alibaba-inc.com

Xinjun Yang
Alibaba Cloud
San Francisco, USA
xinjun.y@alibaba-inc.com

Mo Sha
Alibaba Cloud
Singapore
shamo.sm@alibaba-inc.com

Feifei Li
Alibaba Cloud
Hangzhou, China
lifeifei@alibaba-inc.com

Kang Wang
Alibaba Cloud
Beijing, China
gary.wk@alibaba-inc.com

Zheyu Miao
Alibaba Cloud
Hangzhou, China
zheyu.mzy@alibaba-inc.com

Jie Xu
Alibaba Cloud
Beijing, China
bayan.xj@alibaba-inc.com

Jianfeng Wang
Alibaba Cloud
Beijing, China
wangjianfeng.wjf@alibaba-inc.com

Sheng Wang
Alibaba Cloud
Singapore
sh.wang@alibaba-inc.com

Abstract

The CloudJump framework [14] aims to shed light on how compute-storage disaggregation can effectively enhance cloud databases' scalability, resilience, and efficiency by only relying on standard cloud storage services. To explore the evolving landscape of cloud-native databases, this study navigates through the design and implementation intricacies of the shared-storage architecture in the CloudJump framework. By focusing on the shared storage model integral to cloud database systems with a leader-follower pattern, we address the challenges of ensuring data consistency across multiple compute nodes—a cornerstone for achieving scalability in cloud environments. At its core, this study proposes Multi-Version Data (MVD), a technique designed to improve node consistency in shared storage environments. MVD enables fine-grained version management and precise reconstruction of data pages by compute nodes, facilitating streamlined recovery and robust data persistence without requiring a custom storage layer. This design simplifies operations, reduces costs, and presents a pragmatic response to the limitations of existing cloud storage approaches. The deployment of MVD in PolarDB demonstrates its practical benefits, including enhanced node decoupling and accelerated failover. The performance and reliability of the enhanced system, CloudJump II, are substantiated through comprehensive experimental evaluation and production usage.

CCS Concepts

• **Information systems** → **Cloud based storage**; **DBMS engine architectures**; • **Software and its engineering** → *Consistency*.

Keywords

cloud database, shared storage architecture, data consistency

ACM Reference Format:
Zongzhi Chen, Xinjun Yang, Mo Sha, Feifei Li, Kang Wang, Zheyu Miao, Jie Xu, Jianfeng Wang, and Sheng Wang. 2025. CloudJump II: Optimizing Cloud Databases for Shared Storage. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3722212.3724431

1 Introduction

The transition to cloud-native architectures represents a pivotal moment in database technology, heralding an era that has reshaped the synergy between computing and storage [5, 37, 38, 54]. Central to this evolution is the principle of compute-storage disaggregation [7, 19, 26, 64, 66], a shift that redefines the core architecture of cloud databases. This disaggregation delineates the database system into distinct compute (responsible for query and transaction processing) and storage (managing log and data page persistence) layers, both of which can scale independently. Complementing this shift is the adoption of shared storage, a foundational component of cloud-native databases that embodies the ***shared-everything*** paradigm [11, 17, 40]. In contrast to the ***shared-nothing*** systems exemplified by Google Spanner [15], where each compute node manages isolated storage, shared storage consolidates storage into a unified, interconnected service accessible by all database nodes [1, 23, 58]. Figure 1 illustrates this architecture, depicting a shared storage subsystem integrated with a leader-follower compute layer [2, 19], including one read-write (**RW**) node and multiple read-only (**RO**) nodes, all accessing a shared network of storage nodes. By eliminating cross-node duplication and transmission of data pages, shared-storage architectures meet the growing demand for systems that offer scalability, elasticity, and resource efficiency. Its capability to optimize resource use and support dynamic workloads highlights its significance in the cloud-native database landscape.

To construct an industrial cloud-native database system following the shared-storage architecture depicted in Figure 1, frameworks exemplified by Amazon Aurora [55] have further tailored the storage layer into dedicated page services that allow redo logs to be applied to data pages accessed by compute nodes in a centralized manner. However, this approach hinders the storage layer from

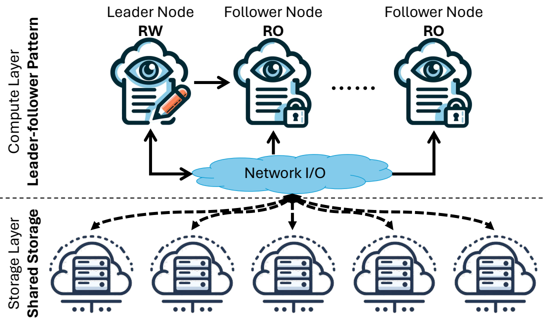

Figure 1: Cloud-native databases based on shared storage.

exploiting those standardized and continuously improved cloud storage services across various cloud vendors, including IBM Spectrum Scale (GPFS) [49], Amazon AWS EFS [4], Microsoft Azure Files [8], Google Cloud Filestore [23], and CephFS [60]. As such, we propose an exploration of constructing a shared storage layer for cloud databases using standardized cloud storage, based on the CloudJump [14] framework. This framework enhances the flexibility and scalability of storage solutions, better meeting the dynamic demands of cloud-native applications. By relying on standard components, CloudJump facilitates the creation of a high-quality cloud-native database service across various cloud platforms, thereby circumventing vendor lock-in [44]. It has become a cornerstone for a myriad of applications in orchestrating cloud resources.

The use of shared storage across multiple database nodes introduces significant challenges, foremost among them being data consistency—a central concern in cloud-native databases [35, 57]. When compute nodes depend on a single shared storage system, update propagation delays become inevitable. Writes by the **RW** node take immediate effect, requiring subsequent synchronization with **RO** nodes. This setup can result in inconsistencies, such as an **RO** node referencing a modified, yet unbuffered, data page via its stale in-memory index. Conventional mitigation strategies bind **RW** and **RO** nodes closely—either by limiting **RW** operations to allow timely **RO** synchronization or suspending **RO** services until consistency is assured—forcing a trade-off between performance and availability.

In this paper, we present the evolution of CloudJump's storage layer as a practical industrial solution to the complexity of cloud storage. Our design favors utilitarian simplicity by leveraging ubiquitous, cost-effective standard cloud storage services. This demonstrates that cloud databases can combine large-scale capacity, high reliability, cost efficiency, and scalability. A central challenge in CloudJump is ensuring data consistency across distributed compute nodes, which demands careful storage subsystem design. A key difficulty is that standard cloud storage services (*e.g.*, POSIX-compliant shared file systems [6, 11, 27]) do not support coordination of concurrent data streams. Consequently, leader nodes may commit updates before followers, creating inconsistencies with their in-memory state. Addressing this requires the storage system to expose multiple valid data versions concurrently, allowing lagging nodes to retrieve data aligned with their local view—a feature not natively supported by standard cloud storage.

To achieve this goal, we propose Multi-Version Data (MVD) technology within a leader-follower architecture, offering a robust mechanism for maintaining node consistency in shared storage environments. MVD enables storage engines to manage versioning with precision, supporting compute nodes in constructing accurate data pages. This approach aligns with Multi-Version Concurrency Control [63] (MVCC) practices, enhancing recovery efficiency and optimizing data persistence. By reducing node interdependencies and operational overhead, MVD provides a practical solution to node consistency issues. It serves as a foundational model for compute nodes, balancing structural consistency with access to data versions. Furthermore, this technology decouples node architecture, enabling faster recovery and more effective data persistence strategies without reliance on specialized intermediary layers.

CloudJump is a production-grade subsystem of the PolarDB service [10, 12], focused on decoupled and standardized storage. Embedded within the PolarDB storage engine, it supports over one million CPU database instances across 20,000 clusters and 10,000 users in 80 global availability zones, underscoring its maturity beyond prototype research. Its deployment has been extensively validated through testing and real-world operation. MVD builds a shared storage layer over standard shared storage instances, enabling page-level, multi-version access for database nodes in a leader-follower configuration. This design aligns with the physical architecture of mainstream database engines, reinforcing its generality [42]. Full compatibility with InnoDB further demonstrates its potential to reshape shared storage practices in cloud-native databases. The shift to shared storage is not only a technical progression but a strategic response to demands for high availability, strict QoS, elasticity, and cost efficiency in the digital economy.

Building on our earlier work, "CloudJump I" [14], which examined integrating cloud storage into database engines and highlighted limitations of conventional SSD-based designs, CloudJump II advances the exploration of shared storage-based architectures. In production, multi-version data (MVD) significantly reduced memory consumption in **RO** nodes, also improved failure recovery and minimized downtime, enhancing availability and reducing compensation costs. These outcomes reflect our continued efforts to improve efficiency through a unified shared storage model, departing from compute-storage binding in traditional architectures.

Our main contributions are summarized as follows:

- We address a fundamental challenge in cloud-native databases: constructing a shared storage layer on standardized cloud storage. Our approach integrates cloud storage with shared-storage architectures, achieving a balance of performance, availability, and consistency, and enhancing scalability and interoperability in cloud-native systems.
- We introduce MVD, a new technique for maintaining data consistency across distributed compute nodes. MVD enables fine-grained version management directly on standardized cloud storage, supporting precise page reconstruction and efficient persistence, and offering a strong foundation for reconciling performance and consistency in shared storage systems.
- MVD has been deployed in PolarDB, a commercial database system by Alibaba Cloud. Deployment results show a 40% reduction in out-of-memory errors for **RO** nodes, along with improved node decoupling and faster failover. Extensive testing and real-world usage validate the theoretical and practical impact of MVD on building high-performance, reliable cloud-native databases based on shared storage.

2 Preliminaries and Motivation

2.1 Cloud-native Databases

The advancement of cloud infrastructure has markedly accelerated the utilization of cloud resources, such as Elastic Compute Service (ECS) and cloud storage, revolutionizing data system construction. This shift has enabled databases to overcome single-machine limitations, facilitating a more balanced distribution of service traffic and risk mitigation. Central to this evolution is the cloud-native shared storage architecture, offering substantial benefits in performance, elasticity, scalability, and reliability.

The integration of a compute-storage disaggregation architecture in cloud databases marks a significant advancement in data management technology [59, 65], delivering diverse benefits suited to modern enterprise needs. It enables elastic scalability [61], allowing independent scaling of compute and storage resources to match business fluctuations [2], thereby optimizing resource allocation and enhancing cost efficiency. This architecture bolsters availability and supports disaster recovery via data redundancy and rapid compute node replacement [11], guaranteeing continuous operations and data integrity. Performance benefits from the separation of processing and storage duties, effectively managing large data volumes and high concurrency. Furthermore, it facilitates flexible data management, encompassing backup, restoration, and migration without hindering compute activities [17, 20], crucial for managing extensive datasets. Simplified maintenance and upgrades reduce system upkeep complexity and risks, enabling independent management of compute and storage layers to minimize service disruptions. Altogether, compute-storage disaggregation transforms cloud database management, setting a new paradigm for cloud-based data processing aligned with the strategic growth and innovation goals of data-driven enterprises.

2.2 Shared Storage Framework for the Leader-follower Configuration

Integrating a shared storage framework within distributed database systems to support the concept of "log-centric" [39, 56] represents a significant evolution toward more efficient, scalable, and resilient data management practices. This framework enables the centralized storage of logs, facilitating unified access and processing within a distributed system environment. By considering logs as a primary data source, the framework leverages their traditional roles in transaction management and system recovery, while also utilizing them for real-time data analytics, event sourcing, and stream processing, thus maximizing the utility of logs. Without loss of generality, consider the implementation of Alibaba Cloud's PolarDB within a shared storage architecture, as an example shown in Figure 2. PolarDB supports a leader-follower model based on a singular shared dataset, comprising one leader node **RW**, and multiple follower nodes **RO**. Upon write operations, the **RW** node generates write-ahead logging [41] (WAL) files to the shared storage, and every log is identified by a log sequence number (LSN), corresponding to a particular version of the database. The persistence of a modification is signified by the success of log persistence, after which the corresponding updated data page will be written back from the **RW**'s local buffer to the shared storage as well. **RO** nodes should replay the logs to catch up with recent updates. A critical aspect

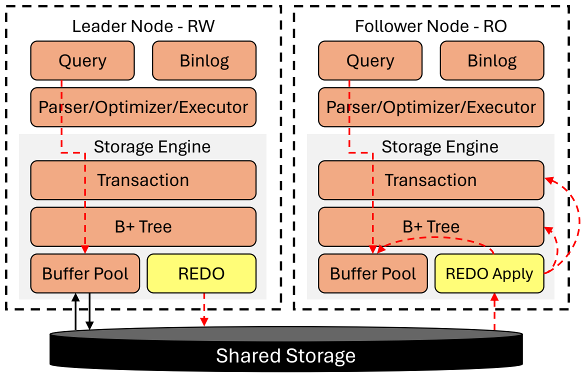

Figure 2: Legacy system architecture on shared storage.

is that each compute node possesses an in-memory *Buffer Pool* to cache page data. When a query is executed, the required data pages already buffered in local memory are directly used; if not, these pages are loaded from the distributed storage layer into the buffer. The essence of shared storage is that all compute nodes access a unified dataset (modified solely by the singular **RW**). Specifically, all in-memory data on **RO** nodes, including pages within the *Buffer Pool*, transaction states, and various memory cache structures, rely on synchronization through the redo logs written to shared storage by the **RW** node. In contrast to traditional local database architectures that utilize binlog for leader-follower logic replication, PolarDB adopts a synchronization method through physical logging (redo), which unfolds in four primary steps:

(1) The **RW** node exclusively accepts write operations, writing the generated redo logs into the shared storage.
(2) The **RW** node periodically broadcasts the latest location of redo logs (LSN) on the shared storage to the **RO** nodes via network.
(3) **RO** nodes, in turn, continuously load and parse redo logs from shared storage based on received LSN.
(4) Redo logs are then applied to pages within the *Buffer Pool*, with transaction states and various caches in memory accordingly.

Thus, **RO** nodes perceive page updates through two mechanisms:

- **Active log update chasing**: Upon receiving periodic broadcasts of the redo log interval awaiting application in shared storage from the **RW**, **RO** nodes fetch and parse these logs. If a redo log corresponds to a data page existing in the local *Buffer Pool*, this log is applied to synchronize the page update within the buffer.
- **Passive on-demand access**: Conversely, if a data page impacted by a redo log is not locally buffered, no action is taken. Should a node subsequently require this data page for a query, it will be fetched from shared storage, already updated by the **RW** node.

2.3 Core Concern—Data Consistency

Despite the adoption of shared storage in the aforementioned multi-node database architecture, the operational memory of each node remains independent. On the one hand, the cost of constructing a unified distributed memory pool across multiple nodes is significantly higher compared to file storage [30, 51]. On the other hand, the leader-follower deployment approach for databases generally assumes that for **RO** nodes, data states that are slightly outdated or delayed in updates are acceptable and by design, thereby sacrificing **RW**-**RO** synchronization to foster decoupling and achieve

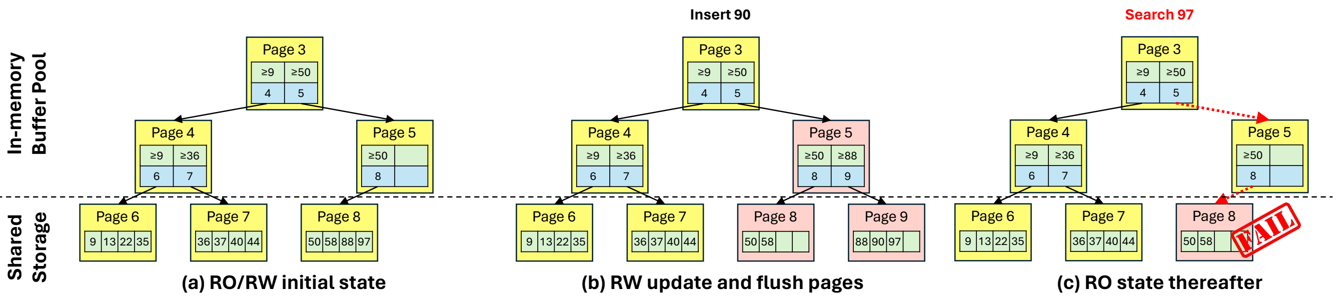

Figure 3: An example of data inconsistency due to the shared stroage architecture.

better scalability. Consequently, it is evident that the update of in-memory data on **RO** nodes depends on asynchronously catching up with the redo log, while the update of external data pages relies on the write-back from the *Buffer Pool* of **RW** nodes, which cannot inherently maintain consistency. This leads to the core technical concern discussed in this paper: the data consistency of **RO** nodes.

To further illustrate the situation, consider a B+-tree as an example shown in Figure 3, where it is assumed that the index's non-leaf nodes corresponding to data pages are in the local *Buffer Pool*, whereas the leaf nodes are not cached. Figure 3(a) represents an initial state of a B+-tree index on a **RO** that has synchronized with **RW**. At this moment, as shown in Figure 3(b), **RW** accepts an operation `Insert-90`, which triggers node splitting, updates `Page-5`, modifies `Page-8`, and creates `Page-9` on the shared storage. Although such insertion would result in **RW** writing WAL in shared storage and broadcasting the latest LSN to all **RO** nodes, one **RO** may not have had the chance to process these corresponding redo logs before receiving a query, such as `Search-97` shown in Figure 3(c). This triggers a catastrophic inconsistency, where `Page-5` in **RO**'s local *Buffer Pool* remains in its old version before the split caused by `Insert-90`. Thus, **RO** attempts to traverse `Page-8`, but since `Page-8` is not in its *Buffer Pool*, **RO** node fetches this data page from shared storage. However, by this time, `Page-8` has been modified and written back by **RW**, clearly making it impossible to find the expected record, as the index viewed by **RO** is in an illegitimate and confused state. It is crucial to note that in such inconsistent states, **RO**'s response to `Search-97`, which is "not found" is not an acceptable "lag" but an error. This is distinct from the situation where, after **RW** inserts 90, **RO** regards `Key-90` as nonexistent for a period. Therefore, such inconsistencies must be avoided completely.

2.4 Current Technical Challenges

In database systems on cloud environments, representative indexing structures such as LSM-trees and B+-trees present distinct advantages and challenges, particularly in the context of maintaining data consistency and version management across multiple nodes. LSM-trees, by virtue of their intrinsic mechanism for version management and their capacity to limit the scope of compaction, provide an advantageous framework for shared storage nodes to access required data versions. This stands in contrast to B+-trees, which inherently lack support for multiple versions since their design permits the overwriting of existing data with new versions, thereby necessitating the adoption of alternative strategies to ensure consistency across nodes. To mitigate the issues related to data consistency, as delineated in Section 2.3, CloudJump formerly imposed a dual physical replication constraint as shown in Figure 4.

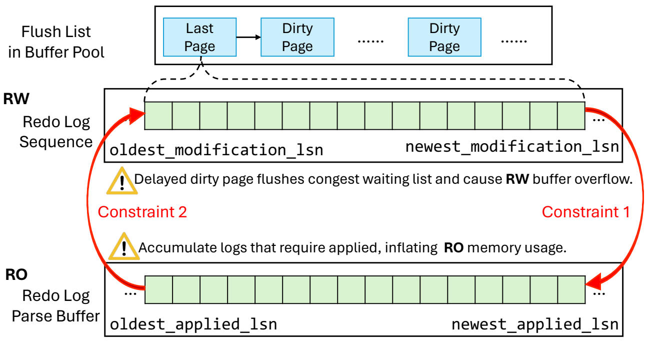

Figure 4: Technical challenges to force data consistency.

For a dirty page residing in the **RW** *Buffer Pool* awaiting flush, the difference between the dirty page and the persisted version that can be seen by all **RO** nodes in shared storage is represented by a series of redo logs. This can be denoted as an interval of log sequences [`oldest_modification_lsn`, `newest_modification_lsn`].

Constraint 1 (Restricting **RW** flush dirty pages): When **RW** flushes dirty pages from its *Buffer Pool* into the shared storage, it must ensure `newest_modification_lsn` of the page to be flushed is not greater than the applied LSN among all **RO** nodes, in order to prevent any **RO** from fetching a future and overly forward page.

Constraint 2 (Augmenting **RO** memory): Upon reading a page, **RO** must process all related redo logs within its log parse buffer to apply the corresponding modifications to the page retrieved from shared storage, ensuring accurate updates to reflect its latest state.

The aforementioned dual constraint can effectively address the issue of inconsistencies between **RO** in-memory data and data pages read from shared storage. However, this approach is not optimal, as these constraints introduce deficits in performance and flexibility:

- **Stringent RW-RO Coupling**: The reliance of **RO** on the **RW**'s data engenders a bottleneck, significantly constraining the **RW**'s capacity for write operations. This necessitates synchronization across all **RO** nodes to preserve data integrity, adversely affecting write throughput and complicating the enhancement of scalability and redundancy mechanisms. Consequently, the **RW**'s performance is directly influenced by the efficacy of the slowest **RO** node.
- **Constrained RO Latency**: The close interdependence limits the permissible lag of **RO** behind **RW**, since excessive delays in **RO** could result in rapidly expanding and accumulating redo logs that must be managed and applied.

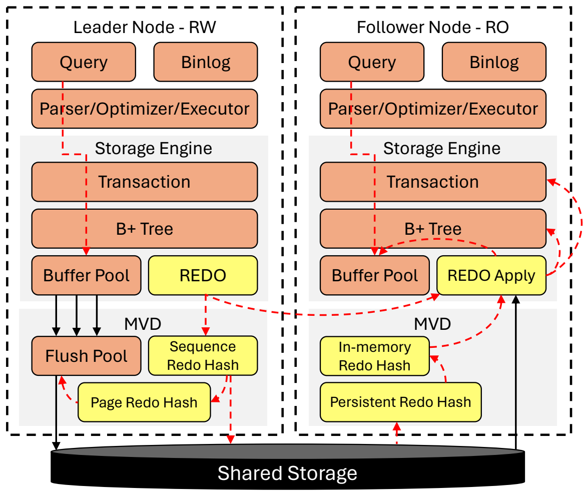

Figure 5: System architecture of CloudJump II with MVD.

These conditions could precipitate a detrimental loop, rendering the **RO** remain perpetually unsynchronized, thus leading to data inconsistencies and negatively affecting the system's capacity to deliver timely information. Concurrently, the immobilized **RO** hinders the timely flushing of dirty pages by **RW**, culminating in buffer overflow. To overcome these obstacles, a reevaluation of the storage engine design for shared storage architectures is essential. This reevaluation should focus on decoupling the dependencies between **RW** and **RO** nodes, enabling independent scaling and improving system resilience. Furthermore, integrating a multi-version engine would allow for more efficient version management, reducing the overhead associated with maintaining data consistency across multiple nodes. Such an engine would facilitate the handling of historical data versions without the need for complex synchronization mechanisms, ultimately enhancing performance and providing greater flexibility in the design of upper-layer application logic.

3 CloudJump II—The MVD Approach

The aforementioned challenges highlight a dilemma: the immediate exposure of pages written by **RW** to all ROs may lead to inconsistencies due to **RO** accessing overly advanced data pages. To maintain data consistency, a cautious update approach is required in shared storage, delaying page permanence until the slowest **RO** catches up, and ensuring data from shared storage is appropriately synchronized. This method, however, hinders **RW** writes and forces **RO** to chase excess redo logs for data freshness. The underlying problem is the forced overwrite of existing data during dirty page flushes, as multi-versioning is not ubiquitous in cloud storage or under the POSIX protocol. CloudJump addresses this by integrating MVD, enabling multiple valid version writes and reads within compute nodes, thus overcoming the single-version overwrite limitation.

3.1 System Overview

In CloudJump II, the integration of the MVD module is situated between the storage engine within database compute nodes and the storage layer, as depicted in Figure 5. This configuration endows the compute nodes with the capability to recognize and utilize page files with multiple versions in shared storage. The MVD is predicated on a multi-node architecture adhering to leader-follower patterns, in which the components housed within the leader (**RW**) and follower (**RO**) nodes differ. Specifically, within the **RW**, a *Sequence Redo Hash* and a *Page Redo Hash* are employed to consolidate redo logs by page in memory. The *Flush Pool* caches dirty page writing back and maintains valid versions, thereby enabling versioned writes (associated with the corresponding LSN) to the shared storage by the **RW**, rather than overwrites. For the **RO** node, a *Persistent Redo Hash* is maintained in memory to facilitate indexing all relevant redo logs by page. This enables **RO** to retrieve pages of particular versions, acquiring the most appropriate page file that has been persisted, along with the collection of redo logs that need be applied to achieve the desired precise page.

Hence, MVD guidance engine offers the capability to access a page from any LSN within the range from the garbage collection (GC) version to the latest version. This feature significantly supports the architecture with multiple database nodes, catering to the diverse requirements for page versions. Within the MySQL-InnoDB framework, the system adopts a WAL strategy that transforms random database access requests into sequential redo log IO operations. Utilizing the *Buffer Pool*, the system aims to consolidate and postpone data persistence operations, thus enhancing efficiency and reducing IO overhead. In the event of failure, the system is capable of recovering unflushed modifications by replaying the redo logs, ensuring data integrity and resilience. Importantly, the redo logs are designed with a page-centric approach, where each log pertains exclusively to a single page. This simplifies the recovery process by focusing solely on the relevant page, enhancing both efficiency and precision. The redo logs ensure completeness and locality, *i.e.*, the logs contain all necessary information for comprehensive database recovery and are designed to be page-oriented, allowing the recovery mechanism to focus exclusively on the page in question. By enabling access to redo logs based on page ID, the design allows for the on-demand retrieval of specific page versions, ensuring operational flexibility and reliability.

3.2 Key Design Ideas

Within the CloudJump framework, the development of MVD technology has led to the integration of key technical features aimed at refining cloud database performance for shared storage. These innovations are central to overcoming the challenges related to data consistency, synchronization delays, and scalable allocation of computing resources in cloud-native databases:

Efficient version management and access control. Central to our approach is the advanced management of database page versions through precise control by the storage engine. This mechanism supports a leader-follower architecture, allowing multiple nodes to access required data versions with minimal latency. This flexibility in version management significantly boosts concurrency processing and data access efficiency, mitigating data inconsistencies and promoting seamless operational continuity.

Optimized log management and IO efficiency. Our strategy employs WAL with a page-oriented redo log design, streamlining random access into sequential log IO operations. By leveraging the *Buffer Pool*, we enhance disk IO efficiency by consolidating data modifications, thereby reducing disk load and supporting high transaction volumes without compromising performance.

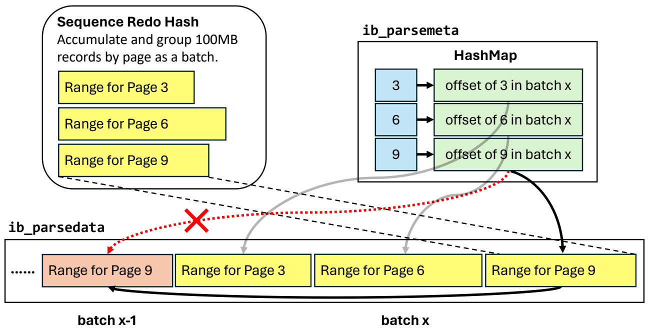

Figure 6: An example of log index generation.

Comprehensive and efficient data recovery mechanisms. The integrity of redo logs ensures database recovery to its pre-failure state via log replay, preserving data consistency. MVD's support for single-page recovery allows for the swift, on-demand restoration of any page version online, bypassing full log replay. This capability greatly enhances recovery speed and accuracy, shortening system recovery times and ensuring uninterrupted business operations and data accessibility in cloud-native settings.

These traits within the proposed shared storage framework highlight our commitment to advancing cloud-native database technologies. By addressing shared storage challenges, *i.e.*, data consistency, optimized IO operations, and swift recovery processes, we position CloudJump as a robust, scalable solution for cloud-based databases.

3.3 Essential Data Structure—Log Index

In the architectural design presented in Section 3.1, there exists a pivotal process that involves retrieving all absent redo logs for a specific page upon request. Hence, a mechanism designed to categorize redo logs by page is necessary, denominated as *log index*.

3.3.1 Log Index Generation. In the leader-follower pattern, MVD adopts **RW** to facilitate the generation of log indexes. Within contemporary database architectures, such as MySQL 8.0, there have been considerable advancements in the methodology of log generation, through the use of multi-threading and lock-free updates, enhancing the efficiency of redo log creation. The process of flushing redo logs to persistent storage is crucial for confirming transactions and flushing dirty pages, significantly impacting database performance. Moreover, the creation of redo logs via mini-transactions [22] (mtr) results in a dispersed record distribution across pages, which can lead to performance degradation and increased metadata volume due to the necessity of maintaining a log index.

To address these challenges, an asynchronous log index generation method is implemented. This method retains the standard procedure for redo log writing and leverages the redo cache's ability to hold the latest segment of redo logs in memory temporarily. An asynchronous parse thread then reads these logs, parses them, and generates log indexes. When a substantial number of log index entries accumulates, they are flushed to persistent storage.

The primary challenge involves synchronizing the rate of log index generation with the rapid production of redo logs without overusing CPU and IO resources. Since redo logs are produced sequentially, maintaining a comprehensive log index is infeasible due to variable space demands for page-specific indexes, complicating space management and necessitating complex allocation and recycling strategies, which increase space and IO usage. A segmented sorting (batch) approach is thus employed for the batch generation of log index entries, as illustrated in Figure 6. The size of each segment is constrained by available memory for sorting and the maximum tolerable **RO** delay. Smaller segments reduce the efficiency of merging log index entries and their retrieval speed, whereas larger segments increase memory demand, delay log index creation, and escalate the IO required per operation.

There is considerable flexibility in balancing these elements. The *In-memory Redo Hash* in **RO** implies that newly generated log index entries are not immediately required, permitting a delay in index creation. Depending on operational needs, **RO** can tolerate deferred access to the log index by 500MB to 1GB. To manage memory usage and I/O overhead during log index generation in a multi-version system, a practical configuration is to use 100MB batches of redo records, each covering disjoint page ranges. An asynchronous parse thread reads and parses redo logs from the redo cache, organizing them into sorted page ranges in memory. Each range includes a header with page and redo log location metadata. Upon accumulating 100MB of redo logs or encountering a redo file change, data is written to `ib_parsedata`. Ranges are appended sequentially, with each updating its last position in `ib_parsemeta` and linking to the preceding range for the same page. `ib_parsemeta` is flushed only at the end of each file cycle, meaning the log index becomes accessible only after this metadata is persisted, resulting in a file-size lag.

3.3.2 Log Index Access. During operation, **RO** continuously scrutinizes redo logs from shared storage, parsing and updating existing pages in the *Buffer Pool* as well as various in-memory states. Throughout this process, a memory-based log index organized by page, referred to as the *In-memory Redo Hash* in Figure 5 is concurrently generated. This hash map associates page IDs with their corresponding redo operations. Previously, memory expansion in this segment could lead to self-inflicted system termination. The introduction of MVD facilitates the use of a more compact redo hash resident in memory. When space becomes scarce, **RO** can immediately expunge the oldest entries within the redo hash. If a user request necessitates accessing a new page and the LSN of the in-place page is assessed as outdated, it requires not only the *In-memory Redo Hash* but also logs from the *Persistent Redo Hash*. This process involves fetching the needed redo records from storage through the log index. By arranging redo records by page, the log applier can systematically apply these records to revert the page to its intended version. Thus, the implementation of the log index frees **RO** from the complications of memory expansion, effectively resolving Constraint 2, and ensures the continuous maintenance of an optimal apply LSN, indirectly resolving Constraint 1.

3.3.3 Crash Safety Guarantees. In databases, crash safety mechanisms are primarily ensured through WAL or constraints on the order of storage writes. The log index serves as an index for Redo operations, and to avoid redundancy and for simplicity, a disk write order constraint is utilized for multi-version concurrency control. The process involves two files, `ib_parsedata` and `ib_parsemeta`, during disk writes, adhering strictly to a sequence where file data precedes file headers, and parsedata is written before parsemeta. Modifications to `ib_parsemeta` data signify the actual completion of disk writes, facilitating the generation of log index checkpoints.

In the event of a crash before the `ib_parsemeta` content is durably written, the partially written data in `ib_parsedata` is deemed invalid and subsequently overwritten. The feasibility of this approach for ensuring crash recovery in multi-version systems is largely attributed to the append-only writing method of `ib_parsedata` and its comparatively simple file structure.

3.3.4 Resource Consumption and Performance. The generation and persistence of log index on **RW** are optimized for minimal impact on typical workloads, involving aspects of resource consumption:

- **CPU usage for parsing redo logs in the redo cache and organizing them by page**: This consumption is proportional to the volume of user-generated redo writes, with tests indicating an additional CPU overhead of approximately 3%-5%.
- **Memory allocation for batch sorting, `ib_parsedata`, and `ib_parsemeta`**: For a 100MB LSN batch, the estimated memory allocation is 200MB for sorting batch and `ib_parsedata`, with `ib_parsemeta` structures requiring around 50MB. Overall, this is relatively minimal and manageable.
- **IO resources consumed during the storage write operations of `ib_parsedata` and `ib_parsemeta`**: ib_parsedata adopts an append-only approach and achieves high IO efficiency with 100MB batch accumulation; for `ib_parsemeta`, only 50MB is needed per 1GB of redo logs. This method ensures that the generation speed of log index can keep pace with the redo production.

4 Design and Implementation Details

MVD's architecture adheres to the Copy-On-Write (CoW) principle to manage shared storage updates without in-place modifications, using a log index mechanism. This allows **RO** nodes to dynamically reconstruct pages using base pages and redo logs instead of replicating pages with each modification, thus maintaining logical versioning with conventional storages. Given that most database kernels operate at the page level, this design is widely applicable. Additionally, CloudJump is deeply rooted in commercial products, particularly focusing on MySQL's InnoDB engine, where significant engineering has improved the B+-trees. Upcoming subsections will explore these technical enhancements in detail, emphasizing optimized logical page assembly and management.

4.1 Write Elision

In WAL-based database architectures, page modifications produce compact redo records and mark the page as "dirty" in the *Buffer Pool*. Due to limited capacity, eviction strategies such as LRU are employed. If a dirty page is selected for eviction, it must be written to disk, incurring a page-sized write IO. When the data volume exceeds the *Buffer Pool*'s capacity, such evictions become frequent: each loaded and slightly modified page is promptly evicted and flushed, leading to excessive IO and potential performance bottlenecks in IO-bound scenarios. MVD mitigates this with Write Elision, omitting the flush of dirty pages upon eviction and thereby avoiding page-sized IO. Upon later access, the page is treated as "corrupt," and required log records are retrieved via page indexing to reconstruct changes. This significantly reduces IO and improves performance under IO-bound conditions. The procedure is depicted in Figure 7 (a zoom-in view of Figure 5's bottom left corner), encompassing the following stages:

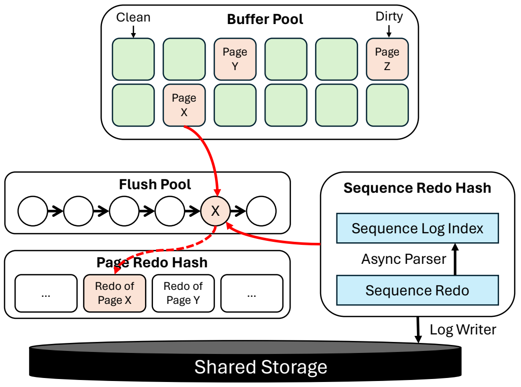

Figure 7: The schematic process of write elision.

(1) Operationally, a continuous *Sequence Redo Hash* is preserved in memory, derived from the redo log's flush activity and the log index generation's async parser buffer, facilitating recent redo log access by page.
(2) Pages designated for *Buffer Pool* flushing, influenced by policies such as LRU, undergo a selection process via a multi-version write elision strategy. This procedure evaluates various factors, including current user load, the degree of modifications on the dirty page, and existing memory utilization.
(3) Pages selected for write elision have their respective redo logs compiled from the *Sequence Redo Hash* by page ID into the *Page Redo Hash*, and are then incorporated into the *Flush Pool*, exempting them from the current flush cycle. Pages not chosen adhere to conventional flush protocols.
(4) Upon subsequent access post-IO completion, the corresponding *Page Redo Hash* is extracted from the *Flush Pool*, with the relevant redo logs applied.
(5) Pages within the *Flush Pool* are propelled to persisted storage upon fulfilling flush prerequisites, whether through dirty flushing or via periodic inspections by a write elision flusher. Thereafter, they are expunged from the *Page Redo Hash*.

Specifically, the *Sequence Redo Hash* manages two main types of data: conventional redo data and log index data, the latter generated via asynchronous parsing. MVD optimizes performance by minimizing redundant operations, directly transferring redo data from the Redo Buffer—flushed by the log writer—through a unified redo log architecture, thereby avoiding additional contention. Concurrently, log index data are derived from a buffered copy prior to flushing and incorporate information from the operational memory cache. Due to limited memory allocated for write elision, the *Sequence Redo Hash* must evict obsolete data, for which two strategies are considered. The first locates corresponding redo files via the log index, but this introduces extra IO, potentially offsetting the IO savings from write elision. The second flushes pages from the *Flush Pool* before eviction, which may compromise the benefits of deferred dirty page flushing. The write elision mechanism relies on aggregating multiple IO requests for the same page in the *Flush Pool*, while also managing cache pressure by preventing unnecessary retention of pages outside the pool. Write elision improves dirty page management by reducing redundant IO operations. It delays flushing based on defined thresholds—such as a per-page redo size limit of 255 bytes, tracked using mtr—ensuring efficiency even when IO is not the bottleneck. Key metrics, including the dirty page ratio in the *Buffer Pool*, guide adaptive behavior. Pages with outdated `oldest_modification_lsn` are prioritized, as they are likely to be flushed by background tasks if modifications continue to accumulate, which nullifies the benefit of delay. The mechanism also enforces two constraints: the `newest_modification_lsn` of a page must remain below the position of the asynchronous parser; the `oldest_modification_lsn` must stay above the lower bound of the *Sequence Redo Hash*. Resource limitations, such as memory allocation failures, are also accounted for. By adhering to these constraints, write elision maintains high performance across diverse and demanding database workloads.

4.2 Instant Recovery

Fault recovery is a crucial feature of database systems, designed to restore committed data that was not yet persisted before interruptions, by using redo logs. This function is vital not only for recovering from system failures but also supports various administrative tasks over the product's lifespan, especially when significant configuration changes require a database restart. The speed of fault recovery is paramount because it affects how quickly users can access the database and their overall experience. For instance, in MySQL, the undo phase occurs quietly after service restart, avoiding any extension to the discussed startup time. The real factors contributing to downtime are setting up necessary data structures and allocating memory. However, the most time-intensive part involves performing and applying redo operations.

4.2.1 Primary Recovery Process. The recovery process generally follows these steps. Initially, the current checkpoint position is read from the `ib_checkpoint` file. Starting from this checkpoint, a sequential scan of redo logs is performed until the last complete mtr is found. The redo of the last incomplete mtr is then truncated, ensuring the atomicity of the mtr. Throughout this scan, all encountered redo logs are parsed and maintained in an in-memory hash map, sorted by page. The scan either concludes or triggers an exception of Page Apply if the hash map consumes excessive memory. This involves using the maintained redo records in the hash map to replay page contents, thus obtaining the latest page version. The process encompasses reading, parsing, and applying all active Redos. The unpredictability in timing mainly stems from:

- **Volume of redo**: The time to read, parse, and process transactions correlates with the active redo log volume. *Buffer Pool* aggregates pages, leading to multi-gigabyte checkpoints. Excessive load, IO blocking, or software bugs may exacerbate delays in checkpoint completion, thereby extending the time consumption.
- **Page IO amplification**: Pages are modified across different redo segments following the access sequence. Exceeding the *Buffer Pool*'s capacity necessitates frequent swap to shared storage, compounded by memory usage of extra hash maps during recovery, further exacerbating IO amplification.
- **Underutilization of storage characteristics**: IO operations, particularly with logs and page recycling, incur higher latency on distributed storage systems compared to local disks, requiring more concurrent reads to offset delays.

During practical deployment practices, instances have experienced prolonged redo phases, resulting in extended periods of unavailability. The capability of single-page recovery in MVD enables the deferment of the time-consuming redo phase until after service provision. By leveraging the page-oriented nature of redo logs and the high throughput of distributed shared storage, it is possible to significantly reduce downtime and potentially accelerate overall completion time in the background. The modified process involves:

(1) Starting the scan from the position where the log index is already generated, rather than from the checkpoint, as log index typically remains in a relatively closer position, unrelated to the length of active redo logs.
(2) Completing the log index during the scan without maintaining the previous *In-memory Redo Hash*.
(3) Scanning from `ib_parsemeta` to identify which pages are involved in post-checkpoint redo logs, marking these as "register pages". Instead of applying pages synchronously during startup, the application to pages is postponed until after the instance becomes operational.
(4) Subsequently, a background thread batch-processes the application recovery for pages.

Moreover, if a user accesses any page from the register pages beforehand, all necessary redo logs must be applied to this page before it is returned. Whether for foreground or background application, this process requires accessing all necessary redo logs for the page through log index, similar to the **RO**'s log index reading process previously mentioned, with the restored page being removed from register pages. This workflow demonstrates how instant recovery can significantly advance the timing of instance recovery and service provision from completing an extensive Redo Apply to just scanning a minimal segment of redo logs.

4.2.2 Segment-oriented Recovery. Instant recovery uses a background process to apply pages while mitigating IO amplification. However, this introduces extra redo log and index reads due to fine-grained access patterns, resulting in slow IO that can delay background recovery and degrade user performance. A typical solution is to cache redo logs and indexes in memory. Yet, since recovery may span a wide range of logs, distinguishing data "hotness" becomes difficult, reducing cache effectiveness. Multi-version instant recovery addresses this by avoiding immediate restoration of pages to their latest version. Instead, it adopts a segment-based strategy with caching: memory holds redo log segments and corresponding index entries in order, bounded by memory limits. Background threads then restore all pages within these segments iteratively until recovery complete. As a result, pages in the *Buffer Pool* may reflect intermediate versions, leading to three possible outcomes:

(1) Subsequent recovery segments continue the restoration process, applying redo logs in sequence based on the current state.
(2) User request access necessitates the preemptive application of all subsequent redo logs, accepting potential delays.
(3) Before further access, pages are flushed or evicted, after which they can be reloaded and accessed anew, either continuing with the recovery process or being accessed by users.

Table 1: Comparison of Recovery Stages

Stage	Instant Recovery	
	Without	With
Redo Scans	• All Active redo log • Single-thread	• Only after log index • Fetch Register Page from `ib_parsemeta`
Redo Parse	• All Active redo log • Single-thread	
Redo Apply	• Synchronous • In Redo sequence • Limited parallelism	• Asynchronous background • By segments • Intra-segment concurrency
When available?	After **Redo Apply**	After **Redo Scans**

In segment-oriented recovery, background application threads initiate the process by loading essential memory segments and redo logs from `ib_parsedata` into a *Shared Recv Cache*. This cache is then available to all threads that need log index entries for page recovery, covering both user and background threads. These threads prioritize retrieving data from the *Shared Recv Cache* for log access. This strategy significantly reduces the I/O requirements during the restoration of background pages, thus speeding up the recovery process. Additionally, as log segments are processed, the system's checkpoint progresses, addressing the problem of checkpoint stagnation during prolonged recovery operations. We detail in Table 1 the principal differences between the instant recovery facilitated by MVD, relevant to this section, and traditional recovery methods.

4.2.3 One-Pass Restore. In IO-bound scenarios, the restoration process, following the order of redo log access, can lead to the repetitive loading of the same page into the *Buffer Pool*. This process—applying a small redo segment, writing to disk, and swapping out—induces significant IO amplification and inefficiency, constrained by IO capacity. Mitigating this issue requires addressing IO amplification for individual pages effectively.

The capability offered by multi-versioning to obtain a specific version of a page on demand naturally solves this problem. The core idea behind One-Pass Restore is to perform restoration according to the sequence of Pages rather than the sequence of redo log access. As the name One-Pass implies, each page undergoes just one read IO and one write IO in the process, applying all the necessary redo logs. During the One-Pass process for a page, it is essential to access all Redo content for that page through multi-version log indexes. Given that redo logs for different pages are interwoven in continuous Redo files, and considering the segmented sorting characteristic of log indexes, it is crucial to avoid additional IO amplification from accessing redo logs or log indexes. To this end, a log merging strategy has been implemented, focusing on:

(1) **Log index merging:** By extending the log index format to support storing redo content directly, in addition to the location information of redo logs, this approach eliminates the overhead of random access of redo logs after obtaining the log index.
(2) **Intra-file segment merging:** As mentioned, an individual `ib_parsedata` may contain multiple segments, which are ordered by page within each segment but not connected between segments. The first step of `One-Pass Restore` is to merge these segments within a file to achieve overall order. This is accomplished through direct parsing of redo logs, which supports backup and restoration of historical instances, including those without multi-versioning.
(3) **Inter-file log index multi-way merging:** Before restoring pages, a multi-way merging of all log index files is conducted to sequentially obtain all redo logs for each page.

This comprehensive strategy effectively mitigates IO amplification issues in IO-bound scenarios, ensuring efficient and streamlined data recovery processes.

4.3 Elastic Scaling of RO Node

In high-stress scenarios, the expedited scaling of **RO** nodes is commonly pursued to enhance horizontal read scalability through rapid augmentation of **RO** instances. Traditionally, integrating a new **RO** node necessitated initiating a synchronization process from the data flush in **RW** node's *Buffer Pool*. To incorporate a new **RO** node while mitigating the risk of failure due to inadequate memory space of the redo cache, an enforced checkpoint to the most recent position was triggered upon the **RO** node's addition. This checkpoint primarily depended on the *Buffer Pool*'s flushing of the oldest dirty pages, a procedure limited by the positions of other **RO** nodes and thus subject to unpredictable durations.

Furthermore, under substantial write loads, this methodology could intensify the likelihood of sequential **RO** node failures. With multiple **RO** nodes within a cluster, the integration of a new **RO** node or the reboot of a previously failed one awaiting registration (during the checkpoint process) would restrict its applied LSN position. This restriction could precipitate *In-memory Redo Hash* expansion or even out-of-memory (OOM) failures in other **RO** nodes tracking the write LSN closely, engendering a cycle of recurrent crashes.

The deployment of a multi-version log index alters this paradigm. Upon connection to the **RW** node, a new **RO** node is no longer compelled to delay for a flush; rather, it can initiate a replication relationship from the position dictated by the log index. Any missing redo logs can be sourced from the log index as necessary. Through a solitary log index flush, the goal is to synchronize this position closely with the **RW** node's current write LSN, fostering the premise that multi-version **RO** nodes can begin replication from a position proximate to the **RW** node's latest write activity.

4.4 Rapid Backtrack

Point-in-time restoration initiates a new database instance. Despite optimizations like One-Pass Restore, this may still require up to several tens of minutes from start to finish. During this period, the new instance cannot handle user requests. Such restoration times may be acceptable for anticipated recovery needs, but in more urgent scenarios, such as a need for rapid data rollback due to user error, extended downtime is untenable and can halt user operations. Additionally, the presence of an additional instance incurs increased costs. Thus, we aim to offer backtrack capabilities to support rapid restoration of the current instance. Backtrack prioritizes service resumption, delaying the time-consuming page processing to the background and accepting a temporary performance degradation post-restoration. With backtrack enabled, users can issue a "Backtrack to Timestamp" command via the console when needed, reverting to a specified point in time. The instance will then restart,

Table 2: Evaluated Benchmarking Cases

Benchmark	Scenario	Dataset Size
SysBench	Sample OLTP	30GB/300GB
TPC-C	Standard OLTP	30GB/300GB
Hotspots	Live Commerce customer	100GB
Multi-Index Table	SaaS customer	100GB

and upon reboot, reflect the state at the designated timestamp. The backtrack process for MVD is as follows:

(1) During operations, the need to generate an older version of the page is assessed when flushing dirty pages to the `.ibd` file. If no older version is needed, the process proceeds as usual. Otherwise, previous versions of the page from the `.ibd` file are first loaded into a historical page management file before the regular flushing of the `.ibd` file continues.
(2) If a backtrack request is made, the request's scope is recorded in the backtrack history, and appropriate checkpoint locations and restart states are set before shutdown.
(3) Upon instance restart, the state determines if it initiates as a backtrack. Employing a methodology similar to instant recovery, it first resumes replication, then uses background threads to restore necessary pages, *i.e.*, reinstates pages modified from the checkpoint to the backtrack position.
(4) For user-driven and background repair reads, the operation first retrieves the target page from the `.ibd` file. It then traverses the page's version chain for an appropriate version if required. Next, relevant redo logs are applied to the selected page version for restoration, obviating immediate `.ibd` file version overwrites, which are instead replaced during future page flushes.
(5) As the configured backtrack window progresses, historical page versions no longer needed are gradually purged.

5 Evaluation

5.1 Experimental Setups

Our evaluations are conducted on an actively marketed PolarDB database cluster consisting of one leader node with read-write capabilities and two follower nodes for read-only operations. Each node is equipped with 16 Intel Xeon Platinum 8369B CPU cores and 64GB of DDR4 memory at 3200 MT/s. The database *Buffer Pool* allocates 75% of the total memory, with the remaining 25% reserved for operational execution. We note that such a configuration (1**RW**+2**RO**) reflects a typical online deployment scenario, accounting for over 90% of database instances. Additionally, as **RO** nodes operate independently, their inclusion theoretically does not impact the MVD. The primary source of performance disparity stems from increased read pressure on shared storage, which is not the main focus of this study. Currently, Alibaba Cloud's retail product offers a maximum cluster size of 16 nodes. The cluster utilizes shared storage through PolarFS@PolarStore [11], a POSIX-compliant distributed file system on a cloud block service, featuring 50 ChunkServers, each with a capacity of 10GB, and a provision of 108K IOPS. Stress testing conducts using an ECS equipped with 16 cores and 64GB memory.

We evaluate using SysBench [34] (write-only and read-write) and TPC-C [16], across medium (30GB) and large (300GB) data sizes. Additional tests cover workloads with hot data updates, reflecting typical online transactional patterns (*e.g.*, counters in e-commerce), and multi-index updates common in SaaS applications. All benchmarks are summarized in Table 2.

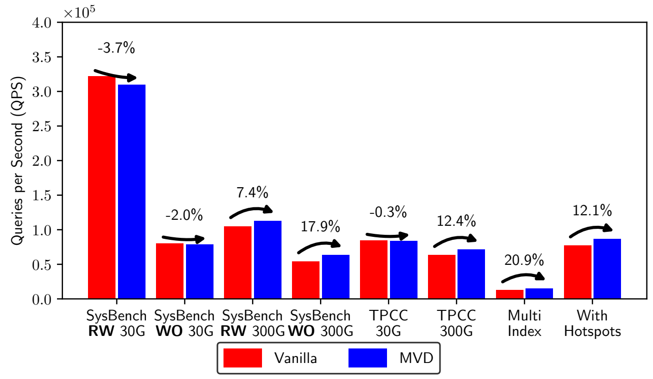

Figure 8: Results of query processing performance.

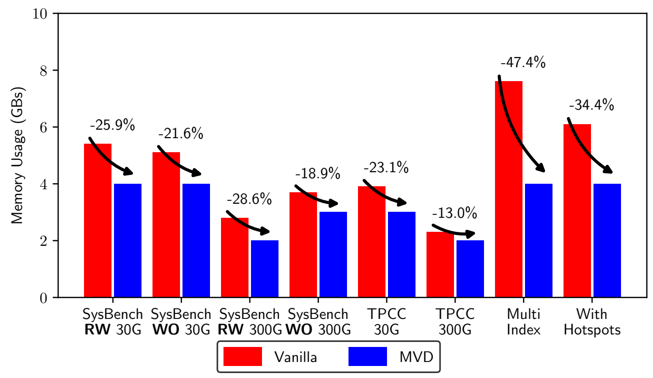

Figure 9: Results of memory usage.

5.2 Query Processing Performance

We first focus on the primary factor affecting database query response performance, which is measured in queries per second (QPS). The performance comparison between the CloudJump II framework integrated with MVD, and a vanilla setup across various workloads is illustrated in Figure 8. It is observed that under low workload conditions, the inclusion of MVD does not enhance performance and may even slightly degrade it. This phenomenon occurs because the low work pressure means that the performance issues caused by traditional strategies to ensure data consistency in shared storage are not significant. Consequently, the benefits do not offset the complexity introduced by MVD, such as the overhead associated with log index. However, beyond this, under higher loads, MVD consistently facilitates a clear improvement in QPS, ranging between 7%-20%. This enhancement stems from two primary sources. Firstly, the incorporation of MVD effectively disrupts the detrimental loop described in Figure 4, preventing delays in refreshing pages in persistent storage due to the slowest **RO** node. This allows for fewer catch-up actions on redo logs after **RO** acquire pages, reducing system resource consumption. Secondly, the MVD enables write elision detailed in Section 4.1, significantly reducing IO amplification for minor page modifications. This factor becomes even more pronounced in IO-bound scenarios, as exemplified in the multi-index scenario.

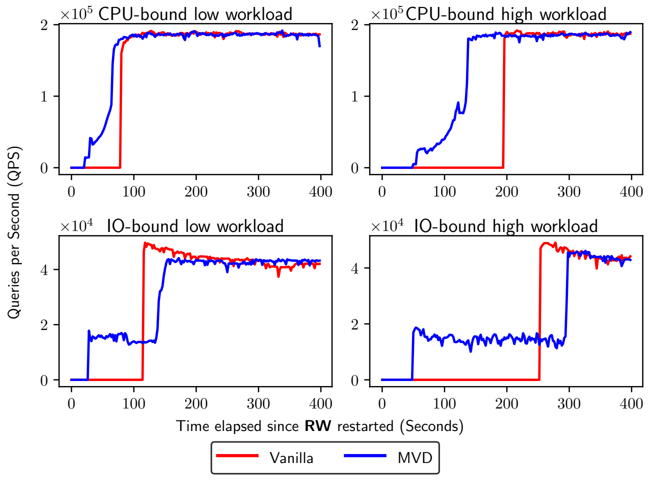

Figure 10: Results of RW recovery.

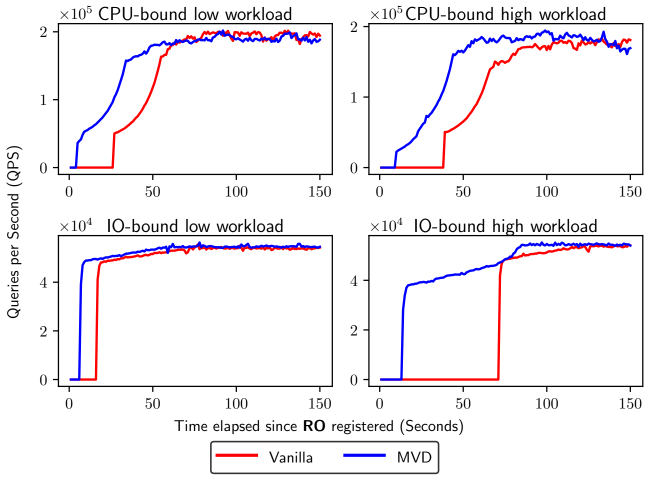

Figure 11: Results of RO online scaling.

5.3 Memory Usage

We next assess the situation regarding memory usage, as illustrated in Figure 9. It is evident that the introduction of MVD technology significantly reduces the consumption levels of operational memory across various types of workloads, with reductions ranging from 13% to 47%. This outcome aligns with expectations, as the introduction of multi-version capabilities effectively breaks the cycle of data updates required to maintain consistency in single-version, overwrite-based shared storage systems, thereby reducing the necessity to maintain an increased number of dirty pages and redo logs in the database compute nodes' memory. It is imperative to emphasize that, in the context of elastic resource allocation in cloud environments, memory usage is often the most critical factor for cost optimization due to its high cost and the relative inflexibility of dynamic allocation. The significant decrease in memory usage resulting from the adoption of MVD technology will directly impact the cost of corresponding cloud database deployments. This cost reduction is especially relevant in a "pay-as-you-go" or when deploying additional instances at a specific physical scale, substantially minimizing the likelihood of instance failure due to OOM under the premise of pre-allocated memory.

5.4 RW Disaster Recovery

We use SysBench to construct both CPU-bound and IO-bound scenarios under low and high load conditions, aiming to assess the cluster's cold start and the recovery of a **RW** following the leader node failure, as illustrated in Figure 10. We refer readers to Table 1 for the procedural distinctions between the vanilla method and the instant recovery offered by MVD, in the context of recovery scenarios. The vanilla method employs a full synchronous recovery, allowing services to commence only after all redo logs have been applied. Consequently, its performance across various scenarios exhibits a stable two-phase characteristic, with a leap to peak QPS performance following the completion of the Redo Apply phase. In contrast, the recovery process with MVD's instant recovery presents a more complex pattern. Notably, it introduces an intermediate stage between the commencement of service and reaching peak QPS, where Redo Apply operations are executed asynchronously in the background while queries are being responded to. For CPU-bound scenarios, this results in a gradual climb to peak QPS as more pages become updated and ready to serve during this process. However, in IO-bound scenarios, the performance demonstrates a square wave pattern due to the sharing of limited bandwidth between queries and reconstruction efforts. Critically, MVD enables service to resume earlier, becoming operational immediately after the Redo Scans stage. This reduces service downtime caused by **RW** failure to 20%-30% of that observed with the vanilla method, which requires the full execution of the Redo Apply phase. This improvement significantly enhances service availability in disaster scenarios, a key consideration for robust IT infrastructure design.

5.5 RO Online Scaling

We also adopt an evaluation similar to that in Section 5.4 to access the scalability of read performance in a cluster upon registering a new **RO** node, as shown in Figure 11. It is evident that, across all evaluation scenarios, MVD demonstrates shorter initialization times and quicker attainment of peak QPS when compared to the vanilla. This advantage stems from MVD's relaxation of the constraints identified in Figure 4, where any **RO** node joining the cluster must not lag behind the dirty page flushes (*i.e.*, `newest_modification_lsn`) on the shared storage, to avoid triggering catastrophic inconsistencies as depicted in Figure 3. Upon connecting to the **RW**, a new **RO** is no longer compelled to await flush completion; instead, it can initiate the replication process from any position specified by log index. Furthermore, we also observe that in CPU-bound scenarios, both methods exhibit a gradual increase in QPS, attributable to the warming up of the *Buffer Pool* within **RO** nodes.

5.6 Rapid Backtrack

Finally, we assessed MVD's efficacy in database instance backtrack, with findings in Figure 12. We craft an experiment for retrospective database navigation, using redo log data volume as measure of the temporal span for backtrack. Unlike vanilla methods lacking a forward traversal capability in persisted states, in which historical version recovery requires the presumption of a full backup that is in an earlier state compared to the target, followed by restoration and

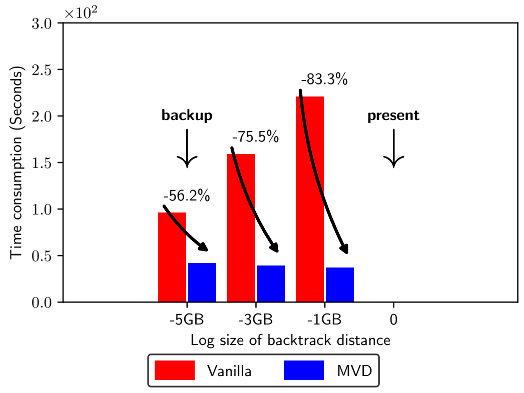

Figure 12: Results of backtrack performance.

backward redo log traversal. Therefore, we assume the existence of a full backup earlier than the "present" by 5GB of redo logs. For a -5GB backtrack, the vanilla method reconstructs the instance based on this backup; for -3GB, it applies 2GB of redo logs after restoring the backup. Conversely, MVD employs a rapid backtrack approach (Section 4.4), significantly reducing retrospection time to 44% of conventional full backup reconstruction for -5GB. For non-exact backup matches, as in -3GB and -1GB cases, redo log application increases overhead. MVD's backtrack time remains stable, leveraging early startup from instant recovery and One-Pass Restore to avoid inefficiencies from repeated page swapping during sequential log replay from the *Buffer Pool*. Additionally, multi-version page management allows direct retrieval of earlier page versions, eliminating the need for reconstruction.

5.7 Discussions

CloudJump II offers useful theoretical and architectural observations for cloud-native databases built on shared storage. The first concerns consistency in disaggregated environments. Traditional models, designed for monolithic systems, do not address the needs of distributed settings. MVD reframes consistency as a continuum of versioned states, rather than fixed snapshots. This shift allows the system to better handle node transitions and recoveries, introducing a multi-dimensional view that integrates both temporal and spatial aspects. Second, our experience reveals a trade-off between storage efficiency and computational autonomy in shared storage systems. Rather than requiring major architectural redesigns, our results suggest that incremental evolution is often a more practical and effective path to balance these goals. Third, IO access patterns optimized for monolithic setups lead to contention under shared storage. CloudJump II highlights the need to revise assumptions about access coordination and resource scheduling in disaggregated settings. While grounded in our system, these insights point to broader principles that may help guide the design of future cloud-native database architectures under storage disaggregation.

6 Related Work

Storage Disaggregation in Databases. Storage disaggregation marks a paradigm shift in database architecture from conventional, unified storage systems to a distributed model that enhances scalability, performance, and resource efficiency. This strategy separates storage from computational resources, enabling dynamic storage scaling without affecting computing capacity. Stemming from this transition, numerous successful and innovative database designs have emerged, *e.g.*, Aurora [55], Cornus [26], Socrates [5], Taurus [19]. Storage disaggregation promotes advanced data management policies, including tiering and automated lifecycle management [9, 25, 28, 29, 32, 50], and strengthens fault tolerance and disaster recovery through flexible replication and backup strategies [21, 24, 31, 62]. It addresses challenges such as increasing data volumes, the necessity for high availability, and the demand for cost-effective scalability, facilitating efficient management of diverse workloads through optimized data placement and access.

Shared Storage Architectures. In distributed systems, shared storage architecture underpins cloud computing by enabling scalable, efficient, and fault-tolerant storage for big data. Early exploration of database systems on shared storage dates back to RAID [45]. As data volumes and infrastructure advanced, techniques like fractional repetition coding [33, 46, 48, 53] enhanced system repairability and adaptability, driving a shift from local storage to distributed and parallel frameworks that address consistency, security, and fault tolerance. This evolution is reflected in systems such as GPFS [49], GFS [23], Ceph [1], Dynamo [18], and PolarFS [11]. CloudJump II, built on shared storage, presents a noteworthy contrast to Google Spanner [15]. While both systems decouple compute and storage, Spanner exploits spatial locality through the use of TrueTime and carefully orchestrated data placement. In contrast, CloudJump II adopts a fully disaggregated storage architecture, prioritizing consistent and efficient access across nodes regardless of data locality. This contrast illustrates a fundamental trade-off in distributed database design: locality-aware performance optimization versus the scalability and flexibility afforded by complete storage abstraction.

Data Consistency and Synchronization. Ensuring data consistency and synchronization in distributed systems, particularly within primary-secondary architectures, mandates the amalgamation of diverse methodologies, including multi-version concurrency control [36, 52, 67], replication protocols [3, 47], and consensus algorithms, notably Paxos [13] and Raft [43]. This body of research underscores the ongoing development in distributed data management aimed at optimizing consistency, availability, and resilience.

7 Conclusion

The journey of Alibaba Cloud's CloudJump framework through the intricate landscape of compute-storage disaggregation elucidates a path toward redefining cloud database scalability, resilience, and efficiency. By leveraging standard cloud storage services to foster a shared-storage architecture, this study underscores the pivotal role of Multi-Version Data (MVD) technology in addressing the paramount challenge of data consistency across compute nodes. MVD enhances operational simplicity, node consistency, recovery, and data persistence without a custom storage layer. The practical application and benefits of MVD technology, as demonstrated in the PolarDB product, signify a leap toward optimizing cloud-native database systems, ensuring their adaptability, high availability, and cost-efficiency in the face of evolving digital demands. This paper contributes to the ongoing discourse in the cloud database domain by offering a nuanced understanding of shared storage solutions and their impact on cloud-native databases.

References

[1] Abutalib Aghayev, Sage A. Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. 2019. File systems unfit as distributed storage backends: lessons from 10 years of Ceph evolution. In *SOSP*. ACM, 353–369.

[2] Divyakant Agrawal, Amr El Abbadi, Sudipto Das, and Aaron J. Elmore. 2011. Database Scalability, Elasticity, and Autonomy in the Cloud - (Extended Abstract). In *DASFAA (1) (Lecture Notes in Computer Science)*, Vol. 6587. Springer, 2–15.

[3] Raihan Al-Ekram and Richard C. Holt. 2010. Multi-consistency Data Replication. In *ICPADS*. IEEE Computer Society, 568–577.

[4] Amazon Web Services. 2024. *Amazon Elastic File System.* https://docs.aws.amazon.com/efs/ Accessed: 2024-12-01.

[5] 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 Conference*. ACM, 1743–1756.

[6] Alysson Neves Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Ferreira Neves, Miguel Correia, Marcelo Pasin, and Paulo Veríssimo. 2014. SCFS: A Shared Cloud-backed File System. In *USENIX Annual Technical Conference*. USENIX Association, 169–180.

[7] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. 2020. Hailstorm: Disaggregated Compute and Storage for Distributed LSM-based Databases. In *ASPLOS*. ACM, 301–316.

[8] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. 2011. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, Ted Wobber and Peter Druschel (Eds.). ACM, 143–157. https://doi.org/10.1145/2043556.2043571

[9] Ignacio Cano, Srinivas Aiyar, Varun Arora, Manosiz Bhattacharyya, Akhilesh Chaganti, Chern Cheah, Brent N. Chun, Karan Gupta, Vinayak Khot, and Arvind Krishnamurthy. 2017. Curator: Self-Managing Storage for Enterprise Clusters. In *NSDI*. USENIX Association, 51–66.

[10] Wei Cao, Feifei Li, Gui Huang, Jianghang Lou, Jianwei Zhao, Dengcheng He, Mengshi Sun, Yingqiang Zhang, Sheng Wang, Xueqiang Wu, Han Liao, Zilin Chen, Xiaojian Fang, Mo Chen, Chenghui Liang, Yanxin Luo, Huanming Wang, Songlei Wang, Zhanfeng Ma, Xinjun Yang, Xiang Peng, Yubin Ruan, Yuhui Wang, Jie Zhou, Jianying Wang, Qingda Hu, and Junbin Kang. 2022. PolarDB-X: An Elastic Distributed Relational Database for Cloud-Native Applications. In *ICDE*. IEEE, 2859–2872.

[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. *Proc. VLDB Endow.* 11, 12 (2018), 1849–1862.

[12] 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 Conference*. ACM, 2477–2489.

[13] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *PODC*. ACM, 398–407.

[14] Zongzhi Chen, Xinjun Yang, Feifei Li, Xuntao Cheng, Qingda Hu, Zheyu Miao, Rongbiao Xie, Xiaofei Wu, Kang Wang, Zhao Song, Haiqing Sun, Zechao Zhuang, Yuming Yang, Jie Xu, Liang Yin, Wenchao Zhou, and Sheng Wang. 2022. CloudJump: Optimizing Cloud Databases for Cloud Storages. *Proc. VLDB Endow.* 15, 12 (2022), 3432–3444.

[15] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, 251–264. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett

[16] Transaction Processing Performance Council. 2024. TPC-C Benchmark. Accessed: 2024-03-15, http://www.tpc.org/tpcc/.

[17] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration. *Proc. VLDB Endow.* 4, 8 (2011), 494–505.

[18] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *SOSP*. ACM, 205–220.

[19] 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 Conference*. ACM, 1463–1478.

[20] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD Conference*. ACM, 301–312.

[21] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter R. Pietzuch. 2013. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD Conference*. ACM, 725–736.

[22] Peter Frühwirt, Peter Kieseberg, Sebastian Schrittwieser, Markus Huber, and Edgar R. Weippl. 2013. InnoDB database forensics: Enhanced reconstruction of data manipulation queries from redo logs. *Inf. Secur. Tech. Rep.* 17, 4 (2013), 227–238.

[23] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *SOSP*. ACM, 29–43.

[24] Caixin Gong, Chengjin Tian, Zhengheng Wang, Sheng Wang, Xiyu Wang, Qiulei Fu, Wu Qin, Qian Long, Rui Chen, Jiang Qi, Ruo Wang, Guoyun Zhu, Chenghu Yang, Wei Zhang, and Feifei Li. 2022. Tair-PMem: a Fully Durable Non-Volatile Memory Database. *Proc. VLDB Endow.* 15, 12 (2022), 3346–3358.

[25] Jorge Guerra, Himabindu Pucha, Joseph S. Glider, Wendy Belluomini, and Raju Rangaswami. 2011. Cost Effective Storage using Extent Based Dynamic Tiering. In *FAST*. USENIX, 273–286.

[26] Zhihan Guo, Xinyu Zeng, Kan Wu, Wuh-Chwen Hwang, Ziwei Ren, Xiangyao Yu, Mahesh Balakrishnan, and Philip A. Bernstein. 2022. Cornus: Atomic Commit for a Cloud DBMS with Storage Disaggregation. *Proc. VLDB Endow.* 16, 2 (2022), 379–392.

[27] Jacob Gorm Hansen and Eric Jul. 2010. Lithium: virtual machine storage for the cloud. In *SoCC*. ACM, 15–26.

[28] Herodotos Herodotou and Elena Kakoulli. 2019. Automating Distributed Tiered Storage Management in Cluster Computing. *Proc. VLDB Endow.* 13, 1 (2019), 43–56.

[29] Wentao Huang, Mo Sha, Mian Lu, Yuqiang Chen, Bingsheng He, and Kian-Lee Tan. 2024. Bandwidth Expansion via CXL: A Pathway to Accelerating In-Memory Analytical Processing. In *VLDB Workshops*. VLDB.org.

[30] Peter J. Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. 1994. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *USENIX Winter*. USENIX Association, 115–132.

[31] Bettina Kemme and Gustavo Alonso. 2010. Database Replication: a Tale of Research across Communities. *Proc. VLDB Endow.* 3, 1 (2010), 5–12.

[32] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *OSDI*. USENIX Association, 427–444.

[33] Joseph C. Koo and John T. Gill III. 2011. Scalable constructions of fractional repetition codes in distributed storage systems. In *Allerton*. IEEE, 1366–1373.

[34] Alexey Kopytov. 2024. Sysbench: A system performance benchmark. Accessed: 2024-03-15, https://github.com/akopytov/sysbench.

[35] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. 2009. Consistency Rationing in the Cloud: Pay only when it matters. *Proc. VLDB Endow.* 2, 1 (2009), 253–264.

[36] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan D. Fekete. 2013. MDCC: multi-data center consistency. In *EuroSys*. ACM, 113–126.

[37] Feifei Li. 2019. Cloud native database systems at Alibaba: Opportunities and Challenges. *Proc. VLDB Endow.* 12, 12 (2019), 2263–2272.

[38] Guoliang Li, Haowen Dong, and Chao Zhang. 2022. Cloud Databases: New Techniques, Challenges, and Opportunities. *Proc. VLDB Endow.* 15, 12 (2022), 3758–3761.

[39] Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. 2016. Towards Accurate and Fast Evaluation of Multi-Stage Log-structured Designs. In *FAST*. USENIX Association, 149–166.

[40] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. 2015. On the Design and Scalability of Distributed Shared-Data Databases. In *SIGMOD Conference*. ACM, 663–676.

[41] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.* 17, 1 (1992), 94–162.

[42] Gihwan Oh, Chiyoung Seo, Ravi Mayuram, Yang-Suk Kee, and Sang-Won Lee. 2016. SHARE Interface in Flash Storage for Relational and NoSQL Databases. In *SIGMOD Conference*. ACM, 343–354.

[43] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference*. USENIX Association, 305–319.

[44] Justice Opara-Martins, Reza Sahandi, and Feng Tian. 2016. Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective. *J. Cloud Comput.* 5 (2016), 4.

[45] David A. Patterson, Garth A. Gibson, and Randy H. Katz. 1988. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD Conference*. ACM Press, 109–116.

[46] Sameer Pawar, Nima Noorshams, Salim Y. El Rouayheb, and Kannan Ramchandran. 2011. DRESS codes for the storage cloud: Simple randomized constructions. In *ISIT*. IEEE, 2338–2342.

[47] Sebastiano Peluso, Pedro Ruivo, Paolo Romano, Francesco Quaglia, and Luís E. T. Rodrigues. 2012. When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication. In *ICDCS*. IEEE Computer Society, 455–465.

[48] Salim El Rouayheb and Kannan Ramchandran. 2010. Fractional repetition codes for repair in distributed storage systems. In *Allerton*. IEEE, 1510–1517.

[49] Frank B. Schmuck and Roger L. Haskin. 2002. GPFS: A Shared-Disk File System for Large Computing Clusters. In *FAST*. USENIX, 231–244.

[50] Mo Sha, Yifan Cai, Sheng Wang, Linh Thi Xuan Phan, Feifei Li, and Kian-Lee Tan. 2024. Object-oriented Unified Encrypted Memory Management for Heterogeneous Memory Architectures. *Proc. ACM Manag. Data* 2, 3 (2024), 155.

[51] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed shared persistent memory. In *SoCC*. ACM, 323–337.

[52] Jie Shao, Boxue Yin, Bujiao Chen, Guangshu Wang, Lin Yang, Jianliang Yan, Jianying Wang, and Weidong Liu. 2016. Read Consistency in Distributed Database Based on DMVCC. In *HiPC*. IEEE Computer Society, 142–151.

[53] Natalia Silberstein and Tuvi Etzion. 2015. Optimal Fractional Repetition Codes Based on Graphs and Designs. *IEEE Trans. Inf. Theory* 61, 8 (2015), 4164–4180.

[54] Junjay Tan, Thanaa M. Ghanem, Matthew Perron, Xiangyao Yu, Michael Stonebraker, David J. DeWitt, Marco Serafini, Ashraf Aboulnaga, and Tim Kraska. 2019. Choosing A Cloud DBMS: Architectures and Tradeoffs. *Proc. VLDB Endow.* 12, 12 (2019), 2170–2182.

[55] 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 Conference*. ACM, 1041–1052.

[56] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. 2012. LogBase: A Scalable Log-structured Database System in the Cloud. *Proc. VLDB Endow.* 5, 10 (2012), 1004–1015.

[57] Hiroshi Wada, Alan D. Fekete, Liang Zhao, Kevin Lee, and Anna Liu. 2011. Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective. In *CIDR*. www.cidrdb.org, 134–143.

[58] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy H. Katz, and Ion Stoica. 2012. Cake: enabling high-level SLOs on shared storage systems. In *SoCC*. ACM, 14.

[59] Jianguo Wang and Qizhen Zhang. 2023. Disaggregated Database Systems. In *SIGMOD Conference Companion*. ACM, 37–44.

[60] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, Brian N. Bershad and Jeffrey C. Mogul (Eds.). USENIX Association, 307–320. http://www.usenix.org/events/osdi06/tech/weil.html

[61] Christian Winter, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2022. On-Demand State Separation for Cloud Data Warehousing. *Proc. VLDB Endow.* 15, 11 (2022), 2966–2979.

[62] Timothy Wood, H. Andrés Lagar-Cavilla, K. K. Ramakrishnan, Prashant J. Shenoy, and Jacobus E. van der Merwe. 2011. PipeCloud: using causality to overcome speed-of-light delays in cloud-based disaster recovery. In *SoCC*. ACM, 17.

[63] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (2017), 781–792.

[64] 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*. www.cidrdb.org.

[65] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Understanding the Effect of Data Center Resource Disaggregation on Production DBMSs. *Proc. VLDB Endow.* 13, 9 (2020), 1568–1581.

[66] Zhanhao Zhao, Hexiang Pan, Gang Chen, Xiaoyong Du, Wei Lu, and Beng Chin Ooi. 2023. VeriTxn: Verifiable Transactions for Cloud-Native Databases with Storage Disaggregation. *Proc. ACM Manag. Data* 1, 4 (2023), 270:1–270:27.

[67] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. 1997. Multiple View Consistency for Data Warehousing. In *ICDE*. IEEE Computer Society, 289–300.