Lakebase: Serverless Postgres over Open Lake Storage

Jasraj Dange, Andrei Dragus, Ali Ghodsi, Haoyu Huang, Yihe Huang, Stas Kelvich, Heikki Linnakangas, Hans Norheim, Ippokratis Pandis, Nikita Shamgunov, Em Sharnoff, John Spray, Zhou Sun, Reynold Xin, Matei Zaharia
lakebase@databricks.com
Databricks

ABSTRACT

Modern cloud applications and AI-driven workloads demand elastic compute, instant environment isolation, and open data interoperability. However, production databases remain largely based on monolithic architectures designed in the 1980s, tightly coupling storage and compute and creating operational fragility, cost inefficiencies, and vendor lock-in. This paper presents Lakebase, a new open database architecture that decouples transactional compute from storage while placing database data directly in low-cost cloud object stores in open format. Unlike second-generation cloud databases, such as Amazon Aurora, Microsoft Socrates and Google AlloyDB, which separate storage internally but retain proprietary formats and single-engine access, Lakebase exposes storage as an open, durable lake layer while running elastic, serverless PostgreSQL transactional engines independently on top. We describe the system design principles behind Lakebase, including open-format object storage persistence, compute elasticity down to zero, instant branching and cloning via copy-on-write semantics, and unified transactional-analytical access over shared lake storage, as deployed in the Databricks Lakebase service. We analyze how this architecture reduces operational complexity, enables Git-like database workflows, improves failure recovery semantics, and mitigates structural vendor lock-in. We further discuss production deployment characteristics, performance trade-offs, durability guarantees, and support for multicloud high availability. We evaluate Lakebase on standard OLTP and analytical benchmarks, alongside production telemetry on branching and elasticity. Lakebase represents a third generation of database architecture: transactional systems rebuilt natively for cloud object storage, elastic compute, and AI-era development workflows.

PVLDB Reference Format:
Jasraj Dange, Andrei Dragus, Ali Ghodsi, Haoyu Huang, Yihe Huang, Stas Kelvich, Heikki Linnakangas, Hans Norheim, Ippokratis Pandis, Nikita Shamgunov, Em Sharnoff, John Spray, Zhou Sun, Reynold Xin, Matei Zaharia. Lakebase: Serverless Postgres over Open Lake Storage. PVLDB, 19(12): 4385 - 4398, 2026.
doi:10.14778/3827998.3828040

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828040

1 INTRODUCTION

Transactional databases form the operational core of modern software systems. Every application—whether consumer-facing services, enterprise systems, financial infrastructure, or AI-driven platforms—ultimately relies on a transactional database to persist state and enforce correctness. Despite the radical evolution of application architectures over the past four decades, production database systems remain largely rooted in design assumptions from the 1980s. These systems were built for an era in which storage was scarce, networks were slow, and cloud elasticity did not exist. As a result, the architecture of most deployed databases is increasingly misaligned with modern infrastructure and development practices.

This misalignment manifests in three persistent challenges. First, tightly coupling compute and storage within a single provisioned unit creates operational fragility and economic inefficiency. Capacity must be provisioned for peak demand, leaving expensive resources idle during normal operation. When load exceeds provisioned capacity, performance degrades abruptly. Routine tasks such as snapshotting, backfills, or compliance-related cleanup queries compete directly with production traffic and can threaten availability. In short, the monolithic database remains one of the most delicate components in modern infrastructure.

Second, traditional databases are poorly aligned with contemporary development workflows. Modern software engineering is built around rapid iteration and isolation. Version control systems allow developers to branch entire codebases instantaneously. However, provisioning a realistic database environment often requires minutes or hours and substantial operational overhead. As development velocity accelerates, the database increasingly becomes the bottleneck rather than the enabler.

Third, existing architectures impose structural vendor lock-in. Storage formats are tightly coupled to a single execution engine, and data is accessible only through that engine. Migrations are costly and risky, analytical integration requires data duplication through ETL pipelines, and cross-cloud portability is limited. Once deployed at scale, organizations become deeply dependent on the vendor's proprietary storage layer.

The historical evolution of database systems helps explain how this situation emerged. Most of the database systems used today, such as MySQL [38], PostgreSQL [4], and Oracle [3], inherited their design from the first-generation database systems such as System R [13] and Ingres [26], and tightly bound compute and storage within a single machine. This design was rational in the pre-cloud era, when local disk access dramatically outperformed network storage and vertical scaling was the primary path to performance. As cloud infrastructure matured, second-generation systems such as Amazon Aurora [46], Socrates [8] and AlloyDB [1], introduced

physical separation between compute and storage, moving persistence into distributed backend layers. These systems improved elasticity and availability, but the separation remained internal to the vendor's architecture. The storage formats remained proprietary, and the data continued to be accessible only through the primary database engine. The result is a transitional architecture that improves performance characteristics without fundamentally addressing portability, openness, or multi-engine access.

Meanwhile, new workloads are placing unprecedented pressure on transactional systems. The rise of AI-driven development and autonomous agents introduces a fundamentally different interaction model with databases. Agents generate code, execute migrations, evaluate hypotheses, and perform experiments at machine speed. These workflows require the ability to create isolated, high-fidelity database environments on demand, run expensive exploratory queries without impacting production traffic, and scale compute elastically—often for short-lived tasks. Traditional always-on, tightly coupled database instances are poorly suited to such ephemeral and bursty workloads.

In this paper, we present Lakebase, a third-generation transactional database architecture. Lakebase fully separates compute from storage and persists database data directly in cloud object storage using open formats. Transactional engine serverless PostgreSQL operates independently from the storage layer and can scale elastically according to workload demand. As the storage is externalized and open, databases can be branched and cloned using copy-on-write techniques, enabling near-instant creation of isolated environments even at large scale. Operational tasks can run on separate compute instances without interfering with production traffic. Furthermore, sharing a common lake-backed storage layer enables unified transactional and analytical processing without costly data duplication.

Lakebase redefines the database as elastic transactional compute over open, durable storage. By aligning with modern cloud infrastructure and AI-native workflows, it addresses long-standing challenges in operational complexity, development velocity, and vendor lock-in. We argue that open, lake-backed transactional systems are the natural next generation of database architecture.

In this paper we make the following contributions. We present *Lakebase*, a lake-backed transactional architecture with full compute–storage separation, in Section 3. Lakebase builds on an open, MVCC-compatible storage layer that supports scalable copy-on-write branching, presented in Section 4, together with elastic, serverless compute, presented in Section 5. Lakebase natively integrates with the Lakehouse to provide a unified data solution, presented in Section 6. Finally, we empirically evaluate the branching, elasticity, OLTP performance, and Lakehouse integration of Lakebase in Section 7, and present related work in Section 8.

2 AGENTS AND DATABASE SYSTEMS

Before presenting Lakebase's design and evaluation, we examine how AI agents are reshaping database workloads.

At the time of writing, Lakebase creates and manages more than 10 million databases daily, with AI agents now creating roughly 4× more databases than human users. Large language models (LLMs) now enable agentic frameworks that generate code, run tests, and iteratively refine applications with minimal human involvement. The architectural primitives of third-generation databases—elastic compute, storage/compute separation, branching, and open formats—predate this trend, but agentic workloads amplify their importance and stress earlier OLTP architectures along three dimensions that directly shape database design.

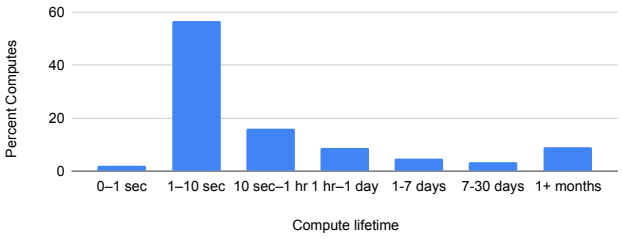

Figure 1: Compute lifetime observed in production.

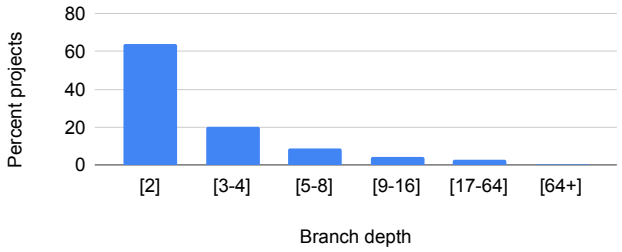

Figure 2: Branch depth observed in production.

Cost Sensitivity and Proliferation of Applications: Agentic development sharply lowers the cost of creating applications, yielding a proliferation of short-lived, experimental services that are mostly idle, frequently discarded, and individually low-value. As Figure 1 shows, taken from the Databricks Lakebase service, many production database instances stay active for under 10 seconds.

This poorly matches traditional OLTP deployments, which assume long-lived instances with continuously provisioned compute. Agentic workloads instead require fine-grained elasticity, near-zero idle cost, and automated lifecycle management—scaling down not only compute but operational complexity, so per-application cost stays negligible.

Evolutionary Workflows: At Databricks, we have observed that well-specified applications can now be generated from scratch in minutes, and features that once took quarters of coordinated human effort can be completed in days with agents. By compressing iteration cycles, agentic workflows let developers explore many candidate implementations in parallel rather than a single planned path.

This extends from source code to database state: each iteration branches off both the code and the database of the previous one [14]. Figure 2 shows the database branch depth we observe in the Databricks Lakebase service. Many production databases have deep branching hierarchies, some exceeding 500 branches. Database

checkpointing and branching thus become first-class primitives, providing a "git checkout -b" experience for databases.

Open Ecosystems: Agentic systems work best with widely used open ecosystems well represented in public training corpora, which strongly favors open-source interfaces and formats—particularly PostgreSQL compatibility.

This extends beyond SQL to the storage layer: open formats let external tools, agents, and analytical engines inspect and process data without proprietary infrastructure, motivating architectures that retain standard PostgreSQL storage semantics while placing data in shared object storage.

3 LAKEBASE ARCHITECTURE

Lakebase is a third-generation transactional database that separates compute from storage and persists all data in open formats on cloud object storage. It delivers enterprise OLTP across multiple clouds, serving both traditional operational workloads and the agentic patterns of the previous section. This section presents the design goals, then an overview of the architecture and its components.

3.1 Design Goals

3.1.1 Leveraging highly durable object storage. The most critical contract of an OLTP database is durability. Cloud object storage offers high durability and availability at a fraction of the cost of operating disks—e.g., AWS S3 guarantees 11 9s of durability, replicates across availability zones, and automatically tiers data by access pattern, lowering cost for cold data while keeping hot data fast. It is also versioned and supports restores, guarding against software errors. Lakebase persists all data, including write-ahead log (WAL) records, in object storage.

Storing data in open formats in the object storage lets Lakebase integrate natively with the Lakehouse ecosystem [9], unifying transactional and analytical access. Spark [49] and Lakehouse//RT [31] can read object storage directly for real-time analytics, isolated from transactional workloads; reverse-ETL writes Lakehouse tables into object storage for PostgreSQL to consume; and forward-ETL streams PostgreSQL changes to Lakehouse tables natively.

3.1.2 Leveraging cloud infra for high availability and performance. An OLTP database must also be highly available and performant, typically answering queries in a few milliseconds, which object storage alone cannot meet. Cloud providers offer services from bare-metal machines to purpose-built offerings at different performance, availability, durability, and price points, enabling a large design space. As Figure 3 shows, Lakebase employs a storage hierarchy: it runs managed PostgreSQL on bare-metal machines with local SSDs, caching frequently accessed data on compute SSDs for high performance. On cache misses, Pageservers serve pages they continuously materialize and cache on-disk; rarely accessed keys are fetched from object storage. A transaction is durable once its WAL is replicated across multiple Safekeeper disks.

3.1.3 Fast compute start up. Compute startup is on the critical path for agentic workloads. The agent platform starts a new compute in response to each user prompt. Managed OSS PostgreSQL services commonly take seconds to minutes to spin up, dominated by VM provisioning and crash recovery. Lakebase drastically shortens this to sub-second using a warm pool of ready computes, and with faster crash recovery because WAL redo is offloaded to distributed storage. We observe Lakebase compute startup typically under a second.

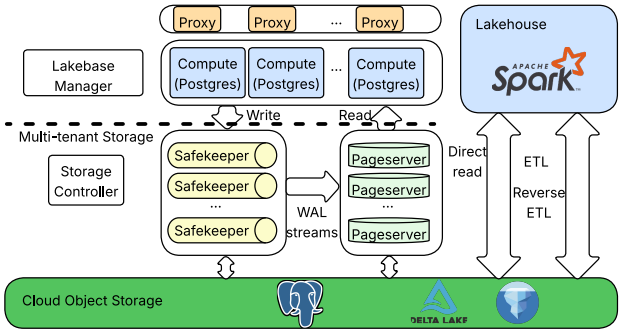

Figure 3: Lakebase architecture.

3.1.4 Zero-copy branching operation. Developers branch code to work on new features, but the database is still shared across code branches, and agents may create many branches per prompt. For example, an agent building an inventory web app may later be asked to track order history, updating both code and database schema and data. Copying data per prompt would add high latency. Lakebase instead offers zero-copy branching: creating a branch is an O(1) metadata operation, so dev-loop iterations branch data as well as code, shortening prompt response time.

3.1.5 Scaling compute and scale to zero. Workloads vary widely: a retail OLTP database may be diurnal, peaking during business hours, while a social-media app is bursty and event-driven, surging on major events. Provisioning for peak is not cost-effective. Lakebase scales compute with load, monitoring compute resource usage and working-set size to scale up in time for performance and down to save cost, and to zero when idle for a period.

3.1.6 Ease of operations. Traditional OLTP databases impose significant operational burden: administrators must plan capacity, manage backups, configure replicas, orchestrate failovers, and tune scaling parameters. Lakebase makes these automatic—autoscaling and scale-to-zero remove most capacity planning, branching and point-in-time restore simplify backup and recovery, and automated cross-AZ failover maintains availability—reducing operational complexity for both human operators and agents.

3.2 Overview

Figure 3 shows the Lakebase architecture. Lakebase separates compute from storage. It decouples a compute node's lifetime from its storage so the two scale independently and non-PostgreSQL computes can operate directly on shared storage. The Lakebase Manager manages a fleet of PostgreSQL Compute nodes; the Storage Controller manages the multi-tenant storage of Pageservers (PSs) and Safekeepers (SKs). All data is durably persisted in object storage. A write is durable once replicated to a quorum of SKs, and PSs continuously materialize pages for fast reads.

3.3 PostgreSQL Compute

A Lakebase instance has one PostgreSQL primary and optionally read replicas. The primary replicates data across 3 Safekeepers and is initially assigned one Pageserver; the database shards across more Pageservers as it grows.

A compute node runs PostgreSQL with a customized storage manager implementing the PostgreSQL SMGR APIs, which talk to the Pageservers and update the local SSD cache. An SMGR read is served from the local cache, or on a miss fetched from the Pageserver and cached. An SMGR write only updates the local cache, since PostgreSQL needs only the persisted WAL to recover.

A compute also runs a WAL proposer that continuously broadcasts the WAL records to all Safekeepers, advancing once a write is durable on a quorum.

A primary starts up by requesting a basebackup—the metadata files needed to start—from the Pageserver, avoiding a checkpoint; the recovery point is the last WAL record the Pageserver ingested rather than the last checkpoint.

Read performance scales horizontally by adding read replicas, which share the primary's Safekeepers and continuously stream WAL to update their local caches. Lakebase also supports static (read) replicas pinned at a given LSN for time-travel queries.

3.4 Non-PostgreSQL Compute

Beyond PostgreSQL compute, external engines can operate directly on the shared storage, giving a unified view across Lakebase and Lakehouse data. Lakehouse tables are synced directly into Lakebase storage, bypassing PostgreSQL compute. Distributed executors, such as Spark, generate PostgreSQL-compliant page files and write them to object storage, which PostgreSQL then consumes as regular tables.

For analytics, query engines, like Lakehouse//RT, read immutable PostgreSQL page files directly from object storage: queries bind to a committed snapshot and run over a consistent view while bypassing PostgreSQL compute. This gives zero-copy access to live data without interfering with transactional workloads, enabling horizontally scalable, vectorized processing. Such multi-engine access over shared storage distinguishes third-generation databases; second-generation systems expose data only through the primary engine. By persisting data in open PostgreSQL page format on object storage, Lakebase decouples data access from any single engine.

3.5 Storage Components

The storage controller creates one storage *tenant* per project, containing multiple *timelines*; each compute belongs to one tenant and timeline. A project starts with a root timeline, and branching or point-in-time restore (PITR) creates a new timeline under the same tenant. A tenant's timelines are co-located on the same Pageservers so a branch can efficiently read its copy-on-write ancestor's data, though they may replicate across different Safekeepers.

3.5.1 SafeKeeper. A Safekeeper (SK) maintains a list of isolated timelines, each storing WAL segments. It keeps the most recent WAL on local disk and proactively uploads quorum-committed segments to object storage. An SK timeline accepts writes from the PostgreSQL primary and streams WAL to replicas and Pageservers.

A timeline replicates WAL across SKs using a Raft [37]-like protocol: the PostgreSQL primary is the WAL proposer and the SKs are acceptors. A WAL record is durable once a quorum has written it to disk, and the system pushes WAL from the most advanced SK replica to lagging ones for redundancy.

To prevent split-brain (where multiple computes claiming to be writers) a PostgreSQL compute must elect itself leader before writing, and a SK quorum acknowledges only one leader. A SK maintains a monotonically increasing term number. A new compute primary increments the term and proposes itself to a SK quorum. SKs accept only the highest term they have seen. A WAL record is acknowledged only after a quorum of SKs has flushed it under the current term. Once the compute detects a term change, it must re-elect itself as the leader.

3.5.2 Pageserver. A Pageserver (PS) streams WAL from SKs, it maintains tenants and their timelines, and exposes two APIs to computes: `GetPage@LSN` and `GetBaseBackup`. Each timeline continuously reads WAL from the SK and materializes pages, organizing delta WAL records and materialized full-page images in an LSM-tree-like structure.

During a `GetPage@LSN` call, the PS finds the last full-page image at an earlier LSN, replays the delta WAL records up to the requested LSN onto it, and returns the page. The request blocks until the PS has ingested WAL up to that LSN.

A PS uploads a snapshot of the tree and metadata to object storage for fast crash recovery. Each timeline keeps a hot standby that downloads the snapshot and hot layer files, so when the primary fails it takes over rapidly and serves `GetPage` requests from already-cached layers.

3.5.3 Storage Controller. The storage controller (SC) manages all SKs and PSs. It assigns each new timeline to the most idle SKs and PSs with AZ affinity for performance and fault tolerance, and re-optimizes placement in the background as the workload evolves.

SC monitors SK and PS health via periodic liveness pings. On a SK failure it quickly migrates the affected timelines to a new replica to restore redundancy; on a PS failure it signals the hot standby to take over.

When a database grows large, SC shards its timeline across PSs transparently and with zero downtime, increasing the `GetPage` throughput. Each shard ingests its relevant WAL records from a SK replica.

4 STORAGE COMPUTE SEPARATION

This section details the mechanisms underlying Lakebase's storage-compute separation: how data is persisted, versioned, and served across the system's components.

4.1 Database Log (WAL) and LSNs

WAL storage is critical for performance, durability, and correctness. The compute primary generates the WAL byte stream, durably persisted in object storage through Safekeepers (SKs). Pageservers (PSs) receive WAL records from SKs and redo them to materialize

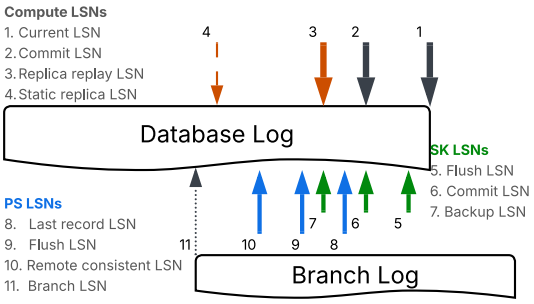

Figure 4: WAL and LSNs maintained by components.

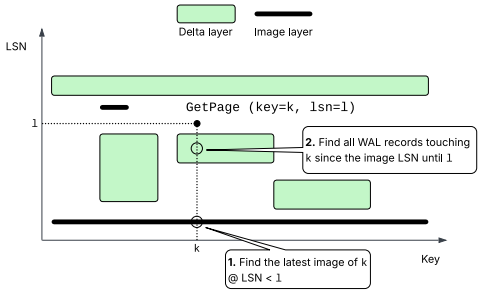

Figure 5: Pageserver storage organization.

page images at specific LSNs, returned to compute on request. Compute, SKs, and PSs each track various LSNs (byte offsets into the WAL stream) to provide durability, transactional consistency, and fast crash recovery.

As Figure 4 shows, a compute primary tracks the current LSN ① and commit LSN ②. It appends WAL to the current LSN, replicates each record to all SKs, and advances the commit LSN once the record is durable on a quorum; on crash recovery it restores data up to the commit LSN.

A compute replica tracks its last replay LSN ③, below which data is visible to it. It streams WAL from a SK and applies records to update its local cache, advancing the last replay LSN after each. A static replica is instead pinned at a particular LSN ④ and reads only up to it.

A SK maintains flush, commit, and backup LSNs. It continuously flushes received WAL to disk, advancing the flush LSN ⑤; the primary notifies all SKs of its commit LSN ⑥, and a SK streams only data below it to Pageservers. For durability, a SK uploads all WAL segments up to the commit LSN to object storage, tracked by the backup LSN ⑦, and removes on-disk segments below it.

A PS ingests WAL from a SK into an optimized storage format (subsection 4.3) balancing ingestion speed and efficient `GetPage` requests. It advances the last record LSN ⑧ on in-memory ingestion, then batches and flushes records to disk, advancing the flush LSN ⑨ (from which it rehydrates on crash). It also uploads flushed files to remote storage, advancing the remote consistent LSN ⑩; if no valid local flush LSN exists (e.g., disk failure or first-time serving), it recovers from there.

4.2 Branching

Lakebase's branching forks a database in $O(1)$ without any expensive physical data copy. When the database history is forked, a new branch is created, and the original history becomes the *parent* branch. Parent and child branches share history (and indeed the same physical storage) up to the branch LSN ⑪ and diverge afterwards. The branch LSN must be chosen in such a way that it is possible to reconstruct every single page on the parent branch at this LSN, at the time of the branch creation. For simplicity, we restrict branch LSN selection to be an LSN within the parent branch's PITR window. As shown in Figure 4, after the child branch is created, new history/WAL records of the child branch are stored independently from those of the parent branch. Any transaction that has not committed at branch LSN is treated identically to an in-flight transaction during crash recovery. They are aborted and pending data modifications are not visible at the child branch.

When the child branch is first created, it simply contains a reference to the parent branch's timeline ID and the branch LSN. A `GetPage` request on the child branch recurses into the parent branch for any history or page data not directly available on the child branch. The child branch is modeled as a new timeline in Lakebase storage. Timeline-level operations, such as compaction and garbage collection, therefore operate largely independently on the child and parent timelines.

Lakebase supports detaching a child branch from its parent. Detaching a branch is instantaneous and a metadata operation. PS tracks the ancestry and asynchronously copies the ancestors' data to the detached branch. The detached branch may re-parent under one of its ancestors. Lakebase currently does not support merging two branches. Reconciling divergent physical data is not well-defined and often requires application to resolve conflicts.

4.3 Page Server Storage Format

All persistent data is stored in standard PostgreSQL page format on cloud object storage, readable by engines that consume PostgreSQL pages without the Lakebase compute layer. The PS ingests WAL records from SKs and materializes them into an LSM-tree-inspired structure, which sustains high WAL ingestion rate while supporting efficient page-level access for `GetPage` requests.

The PS storage format is write-optimized. An incoming WAL record is appended to an in-memory buffer that, once large enough or old enough, is flushed to the PS's local disk and uploaded to remote storage as a *delta layer*—a file of raw WAL records covering a range of keys and LSNs, shown in Figure 5. Delta-layer ingestion involves no WAL redo or transformation, letting the PS keep up with write-heavy workloads.

Figure 5 illustrates how Lakebase's storage format organizes layer files between the LSN and key dimensions, plus how the PS resolves a page image to satisfy a `GetPage@LSN` request through this process. For efficient reads at any LSN within the PITR window, the PS organizes layer files along both the LSN and key dimensions and runs background LSM-tree-style compaction, which replays delta-layer WAL records onto existing page images to produce new *image*

layers (a range of keys at one LSN). A `GetPage` request, specifying a page key and LSN, looks up this two-dimensional tree to find a base image (from an image layer) and any delta WAL records to replay, performs the redos, and returns the page. Compaction is tuned to balance write amplification from image rewrites against the read latency of replaying long WAL chains. When the number of delta layers above the latest available image layer to materialize a key at the latest LSN exceeds a threshold, PS triggers an image layer compaction to materialize data at the latest LSN into new image layers.

PS uses the cloud object storage as the source of truth for persisted layer files. It caches them on ephemeral local SSDs for fast access. PS maintains an *index file* in remote storage that lists all layer files in a timeline. When the local disk runs low, PS evicts the LRU local layer files. A future read of an evicted layer file will download it from remote storage.

The format supports branching: if the PS cannot satisfy a read from the timeline's own layer files, it recursively searches the parent timeline's layers, whose metadata is in the timeline index file. Deep branch hierarchies therefore hurt read performance, since a read spans more timelines. After an agent finalizes on the database, a production database benefits from detaching the branch and promoting it to the root.

Background garbage collection periodically removes delta and image layers whose historical data is outside the PITR window and unneeded for future replays.

4.3.1 Amplification. We characterize the cost of the Pageserver storage format along the standard LSM dimensions. Let N be the live dataset size (in pages) at the latest LSN, w the WAL (write) ingest rate, and P the point-in-time-recovery (PITR) window, beyond which history becomes reclaimable. Two knobs dominate the behavior. The *image threshold* I is the maximum number of delta layers allowed to stack over a key range before a fresh *image* (a full snapshot of that range) is written; it bounds the per-read reconstruction depth. The *GC-compaction triggering ratio* R fires the garbage-collecting compaction once newly accumulated history reaches R times the size of the previously-compacted *base*, collapsing all history below the GC horizon into a new base (a materialized snapshot at the horizon).

Write amplification is constant, $O(1)$: each byte is written once on delta ingestion, plus an $O(1/I)$ term for image creation (a fresh image is amortized over the I deltas that triggered it) and an $O(1/R)$ term for GC-compaction (each collapse writes one base of size $\approx N$ but is amortized over the $\approx R \cdot N$ of history accumulated since the last one). One major difference with a traditional LSM tree is that our storage format uses a constant number of levels. We do not claim our storage format to be the state-of-the-art, but rather tuned it for predictability.

Reads: the PS supports only the `GetPage@LSN` API and has no range-scan primitive. `GetPage@LSN` cost $O(I)$ layer visits, since the image threshold bounds the delta chain that must be replayed to reconstruct a page. Note that the compute-side cache typically attains >98.5% hit rate in our evaluations, hiding most of the latency of `GetPage@LSN` operations.

We define space amplification in Lakebase as (total Pageserver storage size)/(logical size of the user's database at the latest LSN).

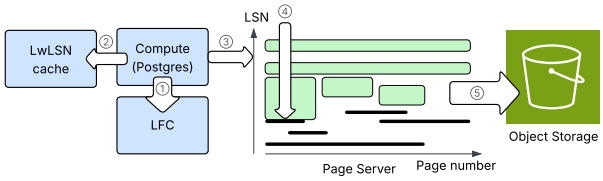

Figure 6: Read path – `GetPage@LSN`.

This is inherently an overestimate because we are obligated to retain PITR history and we bill the user for PITR. In Lakebase the *space amplification* is $O\left(1 + R + \frac{w \cdot P}{N}\right)$, which is one materialization of the live state (1), the un-reclaimed GC slack (R), and the retained in-window history $w \cdot P$, which is usually the dominant term. Across our production fleet, we observe ≈7x space amplification. We expect to further improve the efficiency of the storage format, but the values we observe in production already prove sufficient.

4.4 Reads

OLTP applications demand fast reads. Figure 6 shows the read path of Lakebase. Where OSS PostgreSQL reads a page from a local file, Lakebase caches the working set in a compute-side local file cache (LFC) for comparable performance.

A cache hit returns the page immediately ①; a miss issues a `GetPage@LSN` to a Pageserver ③. The request must carry the correct LSN for consistent reads. Issuing `GetPage` at the current LSN is slow, as the Pageserver must stall until it has ingested up to that LSN, since the page may have changed between its last record LSN and the current LSN. To avoid this, a request carries two LSNs: the *request LSN* specifies the page version to fetch, and the *not-modified-since LSN* promises the page was unchanged between it and the request LSN, so the Pageserver needs to only ingest up to the not-modified-since LSN.

A compute keeps a fixed-size last-written-LSN cache (LwLSN) mapping each page to its not-modified-since LSN ②, updated to a page's LSN when a dirty page is evicted. `GetPage` uses the LwLSN entry on a hit, or the global last-written LSN on a miss.

On an LFC miss, the PS handles the `GetPage@LSN` as in subsection 4.3, possibly fetching layer files from remote storage (④ and ⑤). The compute writes the returned page image to its LFC and updates the LwLSN cache. Prefetch reads follow the same path.

4.5 Writes

Figure 7 shows the Lakebase write path. The Lakebase compute primary replicates the PostgreSQL WAL using a Raft-style protocol, acting as the leader that appends WAL to the Safekeepers via a WAL proposer implemented as a PostgreSQL extension. At startup the proposer elects itself leader by proposing a new term; once a SK quorum acknowledges the higher term, it begins appending WAL ①.

The proposer tracks each SK's flush LSN and sets the commit LSN to the quorum flush LSN. A SK appends each WAL record to a

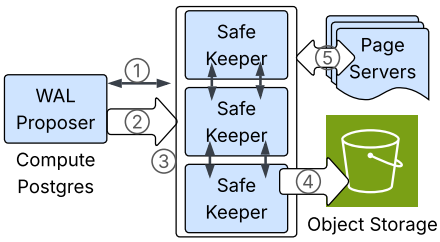

Figure 7: Write path.

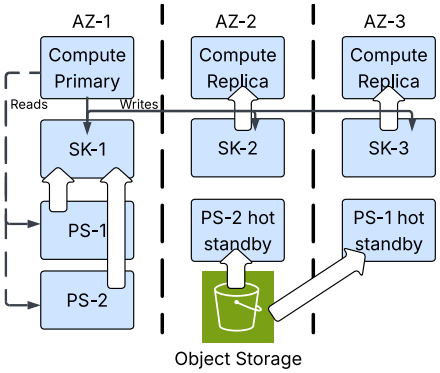

Figure 8: High availability.

local segment file, periodically flushing to disk to advance its flush LSN, and returns its flush and known commit LSNs ②.

The proposer advances the commit LSN once a quorum has flushed to disk, so lagging SK replicas can reduce redundancy; the proposer detects a lagging SK replica and backfills its missing WAL from the most advanced replica.

SKs eagerly upload WAL segments to object storage for durability ④. The 3 SKs gossip their commit LSNs ③ and deterministically elect one upload leader, load-balancing across timelines; once uploaded, local segments below the backup LSN may be removed.

When writes outpace PS ingestion, the proposer applies backpressure: the PS reports three LSNs—last ingested, flushed to local disk, and uploaded to the object storage—via SKs, and the proposer throttles writes if any lags the commit LSN beyond a threshold. These thresholds trade write throughput against read latency and recovery time.

The Pageserver ingests WAL from SKs ⑤. A SK decodes each record's impacted page ids and routes it only to the relevant Pageservers; metadata (e.g., relation size) is tracked solely by the special Shard-0, so all metadata updates go there. A Pageserver prefers ingesting from the SK replica with the highest commit LSN in its own AZ.

The system provides strong durability guarantees in the presence of AZ+1 outages. First, PS maintains additional copies of the most recent data as it closely tails the WAL stream with minimal lag. Second, in practice, recovering a failed SK requires replicating only a few 16 MB WAL segments that have not yet been backed up to remote storage.

4.6 Availability

Lakebase compute and storage components are highly available, see Figure 8. A compute and its replicas span different AZs, and a failed primary or replica is repaired with a new node in under 500 ms thanks to warm pools and storage decoupling. Customers may configure compute HA groups with hot standbys ready to take over on failure. Compute writes are replicated across 3 SKs in 3 AZs, tolerating one node or AZ failure.

A PS prefers streaming WAL from the SK replica in its AZ. Each PS shard keeps a hot standby in a different AZ that proactively downloads the most recently used layer files from the primary's snapshots, so on a primary failure it takes over immediately and serves `GetPage` requests from already-cached hot data. Failovers are automatic, requiring no operator intervention.

4.7 Compute Read Replicas

Lakebase scales read performance horizontally with read replicas. A replica is initialized from a slim basebackup and connects to the same Safekeepers and Pageserver as the primary. Lakebase supports two kinds of replicas: a regular replica that follows the primary, and a static replica pinned at a particular LSN for time-travel and ad hoc queries.

4.7.1 Read Replica. A read replica streams WAL from a SK and continuously applies records to keep its cache current, skipping records for pages absent from its cache. This is safe, because it can always fetch a page from a Pageserver.

Primary and replica availability are decoupled. A replica remains available and keeps ingesting through primary failures, and switches to the next available SK if it loses its current one.

A Lakebase read replica can be created at any LSN and opens for queries immediately, seeing the latest data. This is faster than OSS PostgreSQL, which must redo from the last checkpoint and observe a running-transactions WAL record before accepting connections; Lakebase instead initializes from the PS's last record LSN and builds the running-transactions list from the PostgreSQL commit log.

4.7.2 Static Replica. A static replica is pinned at an LSN within the PITR window. It uses the same lease mechanism to keep the Pageserver from garbage-collecting versions it needs, but unlike a read replica it does not stream WAL from SKs, simply caching pages for lower latency.

5 COMPUTE AUTOSCALING

Lakebase computes are serverless: they scale up under load, down as it decreases, and to zero when idle, removing manual capacity planning and making compute cost proportional to usage. Popular container orchestrators such as Kubernetes are ill-suited to provisioning Lakebase computes: prior work [41] shows the weak host–container isolation is a major source of security vulnerabilities, and provisioning new resources in cloud-managed Kubernetes can take tens to hundreds of seconds for some VMs [7].

To overcome this, Lakebase's compute is built on *NeonVM*, an API for running virtual machines in Kubernetes with in-place vertical scaling and live migration; each compute runs as its own

QEMU/KVM VM. Inside, Lakebase adds a custom resizable cache to complement PostgreSQL's statically-sized shared buffers.

Above NeonVM, we keep a pool of warm computes so that provisioning is off the hot path when creating a database or scaling one from zero, letting Lakebase wake a database in a median of under 500 ms.

Lakebase handles millions of scaling operations daily, so some strategic choices limit the maintenance burden. Backing the primary compute cache by disk is one example: using the operating system's page cache rather than normal allocations—even when the cache fits in memory—eases pressure on the kernel allocator during memory scaling, which normal allocations would otherwise make more fragile.

5.1 Scaling Mechanics

NeonVM supports in-place vertical scaling of CPU, memory, and disk without VM restart. CPU scaling enables and disables individual CPUs within the guest OS via a guest agent. Memory scaling uses virtio-mem, balancing the ratio of movable and kernel memory; we limit the scaling range to avoid fragmentation-induced downscaling failures. Disk scaling combines cooperative reclamation—filesystems issue discard operations that translate to hole punches on host backing files—with oversubscription for swap and temporary storage. The disk-backed compute cache is synchronously resized during scaling operations.

5.2 Scaling Policies

Lakebase's autoscaling policy is enacted per host by the *autoscaler-agent*, which collects metrics, seeks scheduler approval, and sets the desired NeonVM size. It fetches metrics from each compute every 5s, keeping only the most recent. The scaling algorithm is effectively stateless; the state needed to minimize volatility lives in the compute, exposed as measures of load over a longer time window.

Scaling is linear and uniform, in increments of a scaling unit of 1 GiB of RAM and the proportionate fraction of a CPU. After each collection, the desired units for CPU, memory, and cache are estimated independently, and the VM size is set to their maximum.

5.2.1 CPU Scaling Policy. The CPU target blends the 1- and 5-minute load averages from the guest kernel: the 1-minute average is used when it differs substantially from the 5-minute one (for reactivity), and the 5-minute average otherwise (to reduce volatility).

5.2.2 Cache Scaling Policy. OLTP performance typically depends on whether the working set fits in memory, and with separated storage and compute this matters even more—yet users struggle to size their working set, let alone their compute to fit it. Lakebase estimates the working set automatically: our custom compute cache maintains a modified HyperLogLog that tracks the number of unique pages accessed over time.

Each bit in the HyperLogLog's buckets is replaced by the most recent timestamp of a hash with a non-zero value for that bit, giving a cheap approximation of the unique pages accessed since any past point. The autoscaler-agent then estimates the working set with a plateau-detection algorithm over the unique pages accessed in the past 1–60 minutes. To avoid instability from binary cutoffs, the scaling value is a weighted average across all cutoff points with growth rates projected to one hour; earlier points get lower weights for larger workloads, reducing volatility when eviction costs outweigh compute savings.

5.2.3 Memory Scaling Policy. The desired memory size accommodates two needs: The common case, that the compute cache should use the vast majority of memory; and the uncommon case, that allocation-heavy operations may trigger scaling up, but we need to reserve equivalently larger amounts of memory for the cache.

For the common case, the memory scaling policy sets the desired size such that the amount of allocated memory, plus the amount of page cache memory *within the desired cache size*, is less than 90% of the total memory provided to the virtual machine. For the allocation-heavy case, we also ensure that allocated memory alone is less than 75% of the total memory.

5.2.4 Fast Path under Memory Pressure. In addition to periodic metrics collection, we also maintain a fast path for "immediate" scale-up under memory pressure. A guest agent polls memory usage every 100ms and initiates an outbound request to the autoscaler-agent for additional memory, regardless of the most recent metrics.

5.3 Scale to Zero and Quick Provisioning

Scaling to zero—the automatic suspension of idle compute resources—addresses both cost and operational concerns. It removes the cost floor associated with always-on provisioning, which is particularly relevant for the ephemeral workloads described in Section 2, and frees users from manually managing the lifecycle of temporary resources. Quickly provisioning or scaling *from* zero keeps this transparent to the user. Lakebase achieves less than 500ms median cold start time, using a pool of warm virtual machines to keep provisioning off the hot path.

5.3.1 Terminating a Compute. Scaling to zero is handled by a periodic job that checks the time since the last query was issued to each compute. If that duration is longer than the user-configured timeout (by default, 5 minutes), we shut down the compute. No state is preserved across compute restarts except for what is persisted to storage. Full virtual machine termination and cleanup are then handled asynchronously.

5.3.2 Waking a Compute. Users can connect to a database that has previously scaled down to zero, and it will be automatically started for them. On inbound PostgreSQL connections, the Lakebase proxy holds the connection and asks for a compute to be assigned from the pool. During the assignment, the compute downloads the basebackup from the Pageserver and starts compute, after which proxy may start forwarding connections to the compute. The virtual machine is then asynchronously resized to the user-configured minimum size.

5.3.3 Warm Pooling. Speeding up virtual machine provisioning time is technically challenging. Instead, we keep the provisioning off the hot path by pre-provisioning virtual machines into a *warm pool*, so that the user-visible cold start time only includes communication with storage and setup of PostgreSQL inside the compute.

Virtual machines in this pool are uniformly sized, and in-place vertical scaling allows them to be resized on assignment, rather

than maintaining independent pools for different user-configured scaling ranges. Inside the compute, binaries are pre-warmed to avoid startup latency from lazy memory allocation or disk IO.

Lakebase production observes seasonal changes in start compute requests and sudden surges. The pool is sized to fit the predicted needs using a machine learning model that takes into account seasonality and periodic spikes of user activity.

6 INTEGRATION WITH LAKEHOUSE

Lakebase persists all data in open PostgreSQL page format on cloud object storage and exposes a consistent, LSN-addressable page view through the Pageserver. This allows external engines to interact with Lakebase storage directly without routing through PostgreSQL compute. This section describes how Lakebase leverages this property to integrate with the Lakehouse ecosystem along three axes: unified governance through Unity Catalog, efficient data ingestion from Lakehouse to Lakebase, and analytical access to Lakebase data from distributed query engines.

6.1 Unified Governance and Identity

Lakebase centralizes security by integrating with Unity Catalog (UC) for unified authentication and authorization. Each caller's Databricks effective identity is propagated to Lakebase via OAuth and mapped to a PostgreSQL role on first use, so downstream services such as Spark pipelines or Model Serving connect under their own identity without managing separate database credentials.

We modified Unity Catalog to delegate access control and enforcement to Lakebase to unify direct PostgreSQL connection and access through catalog. Administrators can issue standard grants (e.g., `GRANT SELECT`) through Unity Catalog, which are translated to native PostgreSQL queries and applied synchronously. Unity Catalog's catalog metadata is also served live from PostgreSQL system tables, so the view in UC always reflects the current state of the database, eliminating ACL drift between the governance layer and the operational database.

6.2 Serving Lakehouse Data

Databricks customers have extremely valuable gold datasets in Lakehouse, produced by complex analytics pipelines that join and aggregate many disparate data sources. These curated datasets are expensive to compute and refresh. Customers want a low-latency and high-throughput serving layer to bridge the gap between analytical pipelines and production workloads.

We make this possible through *Synced Tables*, a simple abstraction that enables efficient syncing of data from Lakehouse to Lakebase. To reconcile the competing demands of high-throughput ingestion and near-real-time freshness, we architected two distinct write paths. The standard path creates a Spark streaming DataFrame that reads the Delta change feed and updates the destination table in PostgreSQL.

For massive snapshot loads, we introduce a Direct-to-Storage ingestion mode. Distributed Spark executors construct PostgreSQL-compliant page files and write them directly to the object storage, sending only metadata records through Lakebase compute. This mode achieves horizontal scalability, avoids the write throughput limitations that typically bottleneck OLTP databases, and provides isolation for live workloads, which are often latency sensitive. Snapshot loads are a powerful tool that can be used in many situations: initial loading of a dataset, periodic syncs for time-partitioned datasets, and batch ML feature computation (which can be merged with real-time data to provide features with long windows). In the future, direct-to-storage can be applied to optimize other bulk operations, such as distributed index building and physical schema alterations.

The Direct-to-Storage loading path provides atomicity at relation granularity. Spark first materializes the complete set of table pages and uploads them to object storage, after which the Lakebase compute performs a metadata-only relation swap in a transaction. Isolation is preserved by generating the new pages with frozen visibility metadata, allowing them to become immediately visible to any transaction whose snapshot observes the swap. Durability follows from persisting all pages to S3 before commit, while the relation-swap WAL record is replicated through Safekeepers using the standard WAL replication protocol. To preserve consistency, concurrent DML operations targeting a table undergoing a Direct-to-Storage snapshot load are rejected.

6.3 Analytics on Lakebase Data

Users frequently require analytical access to operational data, ranging from ad hoc queries in real time to complex data pipelines. Lakebase facilitates this through two complementary integration paths that are inherently cost-efficient and minimize the overhead on the operational database.

For data pipelines, we introduce *native sync to Lakehouse* using a logical decoding extension (`wal2delta`). This extension hooks into the PostgreSQL logical replication stream, decoding WAL records and appending them directly to a Delta table in SCD Type 2 format. Downstream Spark jobs can consume the history stream for complex transformations or compact it into snapshots as needed.

For real-time analysis, we introduce a *Direct Access mode* for analytics engines such as Spark [49] and Lakehouse//RT [31], Databricks' real-time analytics engine, enabling their compute to read PostgreSQL page files directly from the shared storage layer—object storage or Pageserver. This mode preserves the primary benefit of traditional federation—instant, zero-copy access to live data—while bypassing the PostgreSQL compute. By shifting execution to an analytics engine, we avoid OLTP resource contention and enable horizontal scalability, advanced distributed query operations, and vectorized execution. For example, Lakehouse//RT retrieves consistent PostgreSQL pages at target LSN, using SIMD parsing and a simplified MVCC checker. The resulting parsed Arrow batches are cached in-memory, keyed by relation and page range. When data changes, the Pageserver provides a *modified-blocks bitmap*—a roaring bitmap of pages changed between two LSNs—so only stale pages are refreshed. In Section 7.4, we demonstrate Direct Access is efficient for the majority of analytical workloads without requiring a separate persisted columnar copy.

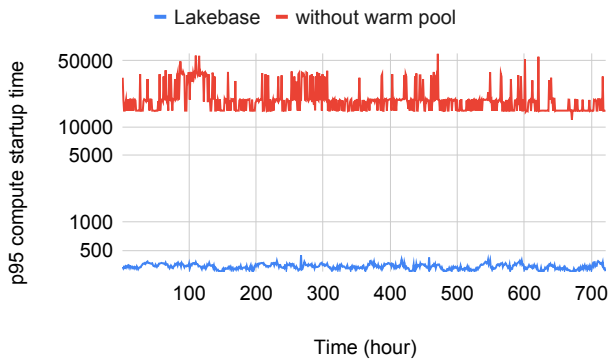

Figure 9: P95 Compute startup time (ms), y-axis is in log-scale.

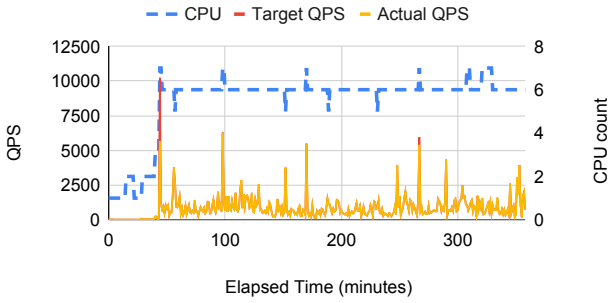

Figure 10: Compute autoscaling.

7 EVALUATION

We evaluate Lakebase across the core workload dimensions targeted by the system design. We first measure the performance of branching and elastic autoscaling, which are critical for agentic workloads. We then evaluate the core OLTP performance and the benefits of Lakehouse integration.

7.1 Branching Performance

Branching on Lakebase is almost instantaneous. It is a $O(1)$ metadata operation that writes a single remote file recording the new timeline identifiers, see details in Section 4.2. Branch restore simply attaches the compute primary to a previous timeline.

Compute startup performance is critical for agentic workloads. Lakebase consistently achieves sub-second startup times. We show the 95th percentile compute startup in milliseconds observed in production in Figure 9. The startup time is consistently lower than 500 ms. This is because the Lakebase manager maintains a warm pool of compute VMs that are readily available to spin up new computes. Also, compute startup does not perform crash recovery. PostgreSQL starts up with the redo LSN set to the last record LSN ingested on the Pageserver. It does not wait for PS to catch up to the latest of SK.

The compute startup time breaks down into four steps. Starting a compute first requests a basebackup from the Pageserver. The basebackup is a tarball that contains all the metadata required to start compute. This typically only takes 100 ms. Next, it syncs all Safekeepers to ensure SK replicas are up-to-date. This also claims the current compute as the sole primary to prevent split-brain scenarios. Syncing SKs only takes 5 ms since failures are rare and writes are replicated to all SK replicas in normal cases. Then, compute starts the compute instance that takes 150 ms. The last step is to configure compute which takes less than 5 ms.

In contrast, without a warm pool, starting a new compute takes up to several minutes, and the tail latency can reach hours. We have observed that the cloud provider may sometimes take a long time to provision a new VM due to various reasons.

7.2 Autoscaling Performance

To demonstrate the autoscaling dimensions, we extracted a workload from a stock trading application in production. The workload exhibits a diurnal pattern where the workload peaks during the day and stays low when the market closes. We use an open-world simulation that issues the target QPS to the database. The compute is configured with a minimum CPU of 1 and a maximum CPU of 7.

Initially, the database load is low and the compute uses 1 CPU. When the application drives a higher load to the system, the autoscaling agent on the compute detects an increase on CPU usage and assigns 3 more CPUs. Scaling operations do not interfere with the customer workload, as observed in the actual QPS in Figure 10. At 44 minutes, when the market opens up, the application sends a surge of queries to the database from 32 to 10K queries per second. The estimated working set size also exceeds the current LFC size. The autoscaling agent observes a high CPU usage and the LFC hit rate is less than 30%. It reacts promptly and directly increases the available CPU count to the max CPU count specified by the compute. The compute LFC also scales up with the CPU size.

During the day, the workload fluctuates, and the agent continuously adjusts the available CPU based on the load. The available CPU count stabilizes around 6 CPUs. It increases the available CPU count to 7 (5) as the workload temporarily increases (decreases) its load. Throughout the entire experiment, the actual QPS almost matches the target QPS. This demonstrates that the autoscaling agent scales compute size up and down promptly to sustain the database load. When the application is idle for a prolonged period of time (5 minutes by default), the compute is shut down.

We also perform the same experiment with a 7-CPU fixed-size compute. The query response time of the autoscaling compute is only slightly higher, as the fixed-size compute benefits from a larger LFC upfront and incurs fewer cache misses. Once the workload stabilizes, the latency becomes comparable. The autoscaling compute uses fewer CPU-hours overall: it runs at reduced capacity during low-load periods and shuts down entirely when idle, whereas the fixed-size compute consumes its full allocation continuously.

7.3 OLTP Performance

To evaluate Lakebase in OLTP workloads, we compare it against reference systems using the HammerDB TPROC-C benchmark. The

Table 1: Peak HammerDB TPROC-C NOPM (x1000).

Tier (vCPU)	Gen-1	Gen-2	Lakebase
1	—	9.0 (1.0×)	23.3 (2.6×)
4	92.6 (1.0×)	61.7 (0.7×)	86.8 (1.0×)
16	118.3 (1.0×)	265.7 (2.3×)	331.3 (2.8×)
24	96.8 (1.0×)	407.5 (4.2×)	515.6 (5.3×)

*Gen-1 has no 24 vCPU tier; the next-available 32 vCPU configuration is used.

first comparison is with a canonical second-generation, compute-storage-separated database (Gen-2). To make the comparison equitable on durability, we also compare with a first-generation that runs PostgreSQL on a multi-AZ replicated disk (Gen-1), which adds synchronous cross-AZ replication on commit. We focus on cached scenarios in which the entire TPROC-C working set fits in DRAM.

We provision four hardware tiers: 1, 4, 16, and 24 vCPUs. We use 16 warehouses per vCPU. Each virtual user accesses all warehouses. We sweep the virtual user count over powers of two from 1 to 512. We report the throughput as the number of NewOrder transactions per minute (NOPM) subject to a p95 NewOrder latency bound of 20 ms. Table 1 reports peak NOPM for each tier, subject to a p95 NewOrder latency bound of 20 ms. The throughput is normalized to Gen-1. The first table entry is omitted because the Gen-1 provider does not offer a 1 vCPU configuration. Similarly, the provider does not support a 24 vCPU tier for Gen-1, so we use the next available 32 vCPU configuration instead, with the same 384 warehouses as Gen-2 and Lakebase at 24 vCPU for comparability.

We report per-vCPU throughput because compute cost dominates overall service cost in typical deployments, making it a proxy for cost efficiency. For example, an AWS EC2 R8gd.4xlarge compute instance costs ~$1.18/hr on demand, whereas retaining 10 GB on AWS S3 Standard costs ~$0.0003/hr—over 3 orders of magnitude lower—and per-request (PUT/GET) and compaction-tiering fees remain small relative to compute even under heavy ingestion. We compare against representative first- and second-generation archetypes (Gen-1 and Gen-2) rather than each individual commercial offering.

Lakebase achieves throughput comparable to or higher than Gen-2 across all tiers. At 1 vCPU, Lakebase delivers 2.6× higher throughput than Gen-2. At this scale, Gen-2's storage fan-out consumes a substantial fraction of the available core, whereas Lakebase streams WAL as a single quorum-replicated write, leaving more CPU capacity available for transaction processing. At 24 vCPU, Lakebase sustains 26% higher throughput than Gen-2 while maintaining p95 latency below 20 ms. At the intermediate 4 and 16 vCPU tiers, Lakebase stays ahead of Gen-2 by 22–43%.

Compared to Gen-1, Lakebase achieves up to 5.3× higher throughput at 24 vCPU. At the smallest 4 vCPU tier the two are on par, as neither saturates the storage path at this scale; the gap widens with core count. Gen-1 does not scale to the throughput levels reached by Lakebase because additional cores contend for the same storage I/O path, rather than benefiting from the parallelism provided by Safekeepers and Pageservers. Furthermore, Gen-1 relies on full-page-image WAL records to ensure durability, whereas Lakebase offloads page materialization to S3-backed Pageservers outside the compute node. As a result, Lakebase generates substantially lower WAL volume.

Table 2: Heap-write time speedup over COPY.

Size	5 GB	10 GB	50 GB	100 GB	500 GB	1 TB
DTS	2.6×	3.4×	6.2×	7.9×	31.3×	73.3×

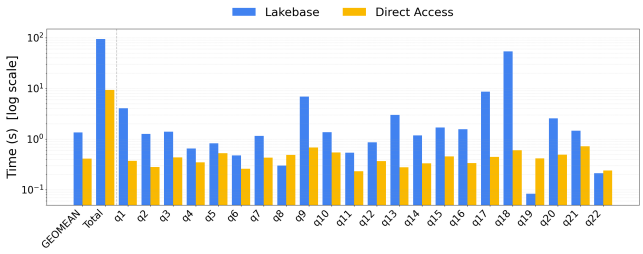

Figure 11: TPC-H of Lakebase vs Lakehouse//RT w/ Direct Access on Lakebase storage.

7.4 Performance from Lakehouse Integration

The conventional `COPY` path serializes ingestion through a single PostgreSQL writer. Direct-to-Storage (DTS) eliminates this bottleneck. Spark executors write PostgreSQL-compliant heap pages directly to object storage in parallel. Table 2 shows the speedup of DTS to write a 1-KB row table with mixed text and integer columns into compute. With horizontal scaling, DTS heap-write time yields speedups up to 73.3× at 1 TB.

We use TPC-H to evaluate Direct Access and compare it with Lakebase. The scale factor is 10. We are using Lakehouse//RT with Direct Access to the same Lakebase storage with the same hardware resources as Lakebase.

Figure 11 reports the performance of Direct Access on the TPC-H benchmark. Overall, Direct Access reduces total execution time by 10× and improves the geomean query latency by 3× relative to PostgreSQL. These improvements stem from several factors. First, the SIMD-accelerated parser and vectorized execution engine eliminate much of the overhead associated with PostgreSQL's volcano-style tuple-at-a-time processing. Second, columnar execution improves the efficiency of analytical operators, particularly joins and aggregations. Finally, Direct Access employs an analytical query optimizer that frequently generates more efficient execution plans.

At the query level, we observe substantial speedups for aggregation- and join-intensive workloads. Q1 achieves an 11× speedup, primarily due to more efficient parallel aggregation. PostgreSQL incurs additional overhead because parallel execution is implemented using multiple processes, whereas Direct Access uses threads. In general, queries with a larger number of joins and aggregations benefit more from vectorized columnar execution. For example, Q18 is nearly 90× faster under Direct Access.

Q17 illustrates the importance of query optimization. Direct Access achieves a 19× speedup by converting the subquery into a join-based plan. In contrast, PostgreSQL generates a suboptimal

plan that repeatedly evaluates the subplan, resulting in significantly higher execution costs.

Not all queries benefit from Direct Access. Q8 and Q19 are slower because the current implementation supports only sequential scans. Both queries benefit substantially from index nested-loop joins, which are available in PostgreSQL but not yet implemented in Direct Access.

Under a concurrent 400 QPS OLTP workload, Direct Access scans slow down by 1.76× due to cache invalidation. However, Direct Access never affects write latency on the primary since it runs outside the compute.

8 RELATED WORK

Separation of storage from processing: First-generation database systems such as MySQL [38], PostgreSQL [4], and Oracle [3] tightly couple compute and storage within a single machine. Second-generation systems—including Aurora [46], Socrates [8], AlloyDB [1], SolarDB [51], Taurus [18], and Tell [32]—separate storage from compute internally [23], improving elasticity and availability. PolarDB serverless [50] scales up seamlessly via transaction migration and provides low-latency strongly consistent reads on primary and secondary nodes [48]. Snowflake Unistore [5] adds OLTP into Snowflake warehouses via a proprietary row store. However, their storage formats remain proprietary and data is accessible only through the primary database engine. Microsoft Fabric [2] stores data in OneLake as Delta tables while the OLTP engine format remains proprietary.

Lakebase represents a third-generation architecture: it externalizes all storage to cloud object storage in open PostgreSQL page format, enabling multi-engine access and removing structural vendor lock-in. WAL redo is offloaded to distributed Pageservers, and a storage hierarchy provides performance and availability. Lakebase further supports sub-second compute startup and zero-copy branching for agentic workloads.

LSM-tree data stores: LSM-tree data stores optimize for write-heavy workloads [15, 20, 21, 25, 27, 34, 43]. Chen et al. [35] survey alternative designs. Key implementations include LevelDB [24] and RocksDB [20], which buffer writes in memory and flush to immutable on-disk SSTables. Alternative compaction strategies [17, 27, 40] trade off read, write, and space amplification. WiscKey [33] separates keys from values to reduce compaction cost.

Lakebase's Pageserver uses a two-dimensional LSM-tree structure indexed by both page key and LSN. It buffers WAL records in memory, flushes them as immutable layer files, and uploads all files to object storage for recovery. Background compaction reduces the WAL redo chain for `GetPage` requests, and garbage collection reclaims data beyond the PITR window.

MicroVM and autoscaling: The popularity of serverless applications in the cloud has inspirited novel MicroVM systems such as Firecraker [6], which aimed at improving isolation, provisioning performance, and density for serverless containerized applications. NeonVM is one such MicroVM system with similar goals, with the added emphasis on supporting resource autoscaling.

Modern data systems scale their resources on demand based on the load changes [16, 22, 28, 29, 44]. Amazon Aurora Serverless [11] implements an on-demand, vertically elastic compute layer that separates compute from distributed storage. Aurora dynamically adjusts capacity in response to workload changes and enables fast scale-up/scale-down and pay-for-what-you-use billing. Elsewhere, there are descriptions of reactive and proactive techniques [36, 39, 42]. Some use statistical models [16], while others use machine learning models [36] to predict the resource usage. Lakebase scales computes with fractions of a CPU and memory with no downtime. The autoscaling agent monitors compute CPU usage and working set size. It scales up compute to sustain performance during peak traffic and scales to zero when compute is idle.

Cluster resource management: Cluster resource management has been extensively studied [12, 45, 47]. Recent work addresses oversubscription [10], workload forecasting [19], and hierarchical allocation [30]. Lakebase overprovisions VMs in warm pools and uses an ML model to predict cluster demand based on historical patterns, enabling fast compute startup.

9 CONCLUSION

This paper presented Lakebase, a third-generation transactional database architecture that separates compute from storage and persists all data in open PostgreSQL page format on cloud object storage. By externalizing storage in open formats, Lakebase enables multi-engine access and situates OLTP directly alongside the Lakehouse, unifying operational and analytical workflows. The architecture leverages cloud object storage for durability, a storage hierarchy for performance and availability, and consensus-replicated Safekeepers for write correctness. Autoscaling, scale-to-zero, automated failover, and built-in branching reduce operational complexity compared to traditional database deployments. These properties are particularly relevant for agentic workloads that require instant clones, sub-second startup, and ephemeral compute. Together, these capabilities make Lakebase an agent-ready transactional database. Production deployments show Lakebase sustaining deep branching patterns and rapid experimentation at scale. At the time of writing, Lakebase is the fastest-adopted product in Databricks history, creating and managing more than 10 million databases daily. Lakebase reflects what a transactional database looks like when designed for today's cloud infrastructure, where storage is durable and inexpensive, compute is elastic, and data openness is a requirement.

ACKNOWLEDGMENTS

We thank the many engineers, managers, and product teams across Databricks and Neon who designed, built, and operate Lakebase, as well as the open-source Neon database community whose contributions this work builds upon. We are also grateful to our anonymous reviewers and shepherd for their thoughtful feedback, which substantially improved this paper.

REFERENCES

[1] Google Inc. 2026. *AlloyDB: 100% PostgreSQL-compatible Database that Runs Anywhere.* Google Inc. https://cloud.google.com/products/alloydb
[2] Microsoft Inc. 2026. *Microsoft Fabric.* Microsoft Inc. https://www.microsoft.com/enus/microsoft-fabric
[3] Oracle. 2026. *Oracle.* Oracle.
[4] The PostgreSQL Global Development Group. 2026. *PostgreSQL: The World's Most Advanced Open Source Relational Database.* The PostgreSQL Global Development Group.

[5] Snowflake Inc. 2026. *Snowflake Unistore.* Snowflake Inc. https://www.snowflake.com/en/product/features/unistore/

[6] Alexandru Agache, Marc Brooker, Andreea Iordache, Marc Liguori, Rolf Neugebauer, Phil Piwonka, and Jörg Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 419–434.

[7] Amazon Web Services. 2021. Launch Windows faster on EC2. https://aws.amazon.com/blogs/modernizing-with-aws/launch-windows-faster-on-ec2/. Accessed November 17, 2025.

[8] 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 *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1743–1756. https://doi.org/10.1145/3299869.3314047

[9] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *Proceedings of the 11th Annual Conference on Innovative Data Systems Research (CIDR 2021)*. Online. https://www.cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf

[10] Pankaj Arora, Surajit Chaudhuri, Sudipto Das, Junfeng Dong, Cyril George, Ajay Kalhan, Arnd Christian König, Willis Lang, Changsong Li, Feng Li, Jiaqi Liu, Lukas M. Maas, Akshay Mata, Ishai Menache, Justin Moeller, Vivek Narasayya, Matthaios Olma, Morgan Oslake, Elnaz Rezai, Yi Shan, Manoj Syamala, Shize Xu, and Vasileios Zois. 2023. Flexible Resource Allocation for Relational Database-as-a-Service. *Proc. VLDB Endow.* 16, 13 (Sept. 2023), 4202–4215. https://doi.org/10.14778/3625054.3625058

[11] Bradley Barnhart, Marc Brooker, Daniil Chinenkov, Tony Hooper, Jihoun Im, Prakash Chandra Jha, Tim Kraska, Ashok Kurakula, Alexey Kuznetsov, Grant McAlister, Arjun Muthukrishnan, Aravinthan Narayanan, Douglas Terry, Bhuvan Urgaonkar, and Jiaming Yan. 2024. Resource Management in Aurora Serverless. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 4038–4050. https://doi.org/10.14778/3685800.3685825

[12] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* 59, 5 (April 2016), 50–57. https://doi.org/10.1145/2890784

[13] Donald D Chamberlin, Morton M Astrahan, Michael W Blasgen, James N Gray, W Frank King, Bruce G Lindsay, Raymond Lorie, James W Mehl, Thomas G Price, Franco Putzolu, et al. 1981. A history and evaluation of System R. *Commun. ACM* 24, 10 (1981), 632–646.

[14] Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Bowen Wang, Alex Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, Jeff Chen, Lakshya Agrawal, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. 2025. Barbarians at the Gate: How AI is Upending Systems Research. arXiv:2510.06189 [cs.AI] https://arxiv.org/abs/2510.06189

[15] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 155–171. https://www.usenix.org/conference/osdi20/presentation/dai

[16] Sudipto Das, Feng Li, Vivek R. Narasayya, and Arnd Christian König. 2016. Automated Demand-driven Resource Scaling in Relational Database-as-a-Service. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1923–1934. https://doi.org/10.1145/2882903.2903733

[17] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 505–520. https://doi.org/10.1145/3183713.3196927

[18] 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 *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1463–1478. https://doi.org/10.1145/3318464.3386129

[19] Yanlei Diao, Dominik Horn, Andreas Kipf, Oleksandr Shchur, Ines Benito, Wenjian Dong, Davide Pagano, Pascal Pfeil, Vikram Nathan, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Forecasting Algorithms for Intelligent Resource Scaling: An Experimental Analysis. In *Proceedings of the 2024 ACM Symposium on Cloud Computing* (Redmond, WA, USA) *(SoCC '24)*. Association for Computing Machinery, New York, NY, USA, 126–143. https://doi.org/10.1145/3698038.3698564

[20] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings* (Chaminade, California, USA). www.cidrdb.org, Chaminade, California, USA. http://cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf

[21] Robert Escriva. 2020. *HyperLevelDB.* HyperDex. https://github.com/rescrv/HyperLevelDB

[22] Guilherme Galante, Luis Carlos Erpen De Bona, Antonio Roberto Mury, Bruno Schulze, and Rodrigo da Rosa Righi. 2016. An Analysis of Public Clouds Elasticity in the Execution of Scientific Applications: a Survey. *Journal of Grid Computing* 14, 2 (2016), 193–216. https://doi.org/10.1007/s10723-016-9361-3

[23] Shahram Ghandeharizadeh, Philip A. Bernstein, Dhruba Borthakur, Haoyu Huang, Jai Menon, and Sumit Puri. 2022. Disaggregated Database Management Systems. In *Performance Evaluation and Benchmarking: 14th TPC Technology Conference, TPCTC 2022, Sydney, NSW, Australia, September 5, 2022, Revised Selected Papers* (Sydney, NSW, Australia). Springer-Verlag, Berlin, Heidelberg, 33–48. https://doi.org/10.1007/978-3-031-29576-8_3

[24] Sanjay Ghemawat and Jeff Dean. 2020. *LevelDB.* Google. https://github.com/google/leveldb

[25] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling Concurrent Log-Structured Data Stores. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 32, 14 pages. https://doi.org/10.1145/2741948.2741973

[26] GD Held, MR Stonebraker, and Eugene Wong. 1975. INGRES: a relational data base system. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*. 409–416.

[27] Haoyu Huang and Shahram Ghandeharizadeh. 2021. Nova-LSM: A Distributed, Component-based LSM-tree Key-value Store. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3448016.3457297

[28] Abdul R. Hummaida, Norman W. Paton, and Rizos Sakellariou. 2016. Adaptation in cloud resource configuration: a survey. *Journal of Cloud Computing: Advances, Systems and Applications* 5, 1 (2016), 7. https://doi.org/10.1186/s13677-016-0057-9

[29] Rong Kang, Yanbin Chen, Ye Liu, Fuxin Jiang, Qingshuo Li, Miao Ma, Jian Liu, Guangliang Zhao, Tieying Zhang, Jianjun Chen, and Lei Zhang. 2025. ABase: the Multi-Tenant NoSQL Serverless Database for Diverse and Dynamic Workloads in Large-scale Cloud Environments. In *Companion of the 2025 International Conference on Management of Data* (Berlin, Germany) *(SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 471–484. https://doi.org/10.1145/3722212.3724426

[30] Ji You Li, Jiachi Zhang, Wenchao Zhou, Yuhang Liu, Shuai Zhang, Zhuoming Xue, Ding Xu, Hua Fan, Fangyuan Zhou, and Feifei Li. 2023. Eigen: End-to-End Resource Optimization for Large-Scale Databases on the Cloud. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 3795–3807. https://doi.org/10.14778/3611540.3611565

[31] Nong Li, Shoumik Palkar, Shant Hovsepian, Mostafa Mokhtar, and Reynold Xin. 2026. *Introducing Lakehouse//RT: Real-Time Performance on the Unified Lakehouse.*

[32] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. 2015. On the Design and Scalability of Distributed Shared-Data Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 663–676. https://doi.org/10.1145/2723372.2751519

[33] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 133–148. https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu

[34] Chen Luo and Michael J. Carey. 2020. Breaking down Memory Walls: Adaptive Memory Management in LSM-Based Storage Systems. *Proc. VLDB Endow.* 14, 3 (Nov. 2020), 241–254.

[35] Chen Luo and Michael J. Carey. 2020. LSM-based Storage Techniques: A Survey. *The VLDB Journal* 29, 1 (2020), 393–418. https://doi.org/10.1007/s00778-019-00555-y

[36] Chenghao Lyu, Qi Fan, Fei Song, Arnab Sinha, Yanlei Diao, Wei Chen, Li Ma, Yihui Feng, Yaliang Li, Kai Zeng, and Jingren Zhou. 2022. Fine-grained modeling and optimization for intelligent resource management in big data processing. *Proc. VLDB Endow.* 15, 11 (July 2022), 3098–3111. https://doi.org/10.14778/3551793.3551855

[37] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 305–320.

[38] Oracle. 2026. *MySQL.*

[39] Olga Poppe, Qun Guo, Willis Lang, Pankaj Arora, Morgan Oslake, Shize Xu, and Ajay Kalhan. 2022. Moneyball: proactive auto-scaling in Microsoft Azure SQL database serverless. *Proc. VLDB Endow.* 15, 6 (Feb. 2022), 1279–1287. https://doi.org/10.14778/3514061.3514073

[40] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 497–514. https://doi.org/10.1145/3132747.3132765

[41] Michael Reeves, Dave Jing Tian, Antonio Bianchi, and Z.B̈erkay Celik. 2021. Towards Improving Container Security by Preventing Runtime Escapes. In *2021 IEEE Secure Development Conference (SecDev)*. IEEE, 38–46. https://doi.org/10.1109/SecDev51306.2021.00022

[42] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: workload autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 16, 16 pages. https://doi.org/10.1145/3342195.3387524

[43] Russell Sears and Raghu Ramakrishnan. 2012. BLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 217–228. https://doi.org/10.1145/2213836.2213862

[44] Ahmed A. Soror, Umar Farooq Minhas, Ashraf Aboulnaga, Kenneth Salem, Peter Kokosielis, and Sunil Kamath. 2008. Automatic virtual machine configuration for database workloads. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 953–966. https://doi.org/10.1145/1376616.1376711

[45] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (Santa Clara, California) *(SOCC '13)*. Association for Computing Machinery, New York, NY, USA, Article 5, 16 pages. https://doi.org/10.1145/2523616.2523633

[46] 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 *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. ACM, New York, NY, USA, 1041–1052. https://doi.org/10.1145/3035918.3056101

[47] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 18, 17 pages. https://doi.org/10.1145/2741948.2741964

[48] Xinjun Yang, Yingqiang Zhang, Hao Chen, Chuan Sun, Feifei Li, and Wenchao Zhou. 2023. PolarDB-SCC: A Cloud-Native Database Ensuring Low Latency for Strongly Consistent Reads. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3754–3767. https://doi.org/10.14778/3611540.3611562

[49] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65. https://doi.org/10.1145/2934664

[50] Yingqiang Zhang, Xinjun Yang, Hao Chen, Feifei Li, Jiawei Xu, Jie Zhou, Xudong Wu, and Qiang Zhang. 2024. Towards a Shared-Storage-Based Serverless Database Achieving Seamless Scale-Up and Read Scale-Out. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 5119–5131. https://doi.org/10.1109/ICDE60146.2024.00384

[51] Tao Zhu, Zhuoyue Zhao, Feifei Li, Weining Qian, Aoying Zhou, Dong Xie, Ryan Stutsman, Haining Li, and Huiqi Hu. 2019. SolarDB: Toward a Shared-Everything Database on Distributed Log-Structured Storage. *ACM Trans. Storage* 15, 2, Article 11 (June 2019), 26 pages. https://doi.org/10.1145/3318158