# Salesforce Database: A Cloud-Native Multi-Tenant OLTP Database

Atish Agrawal, Vaibhav Arora, Gary Baker,
Milena Bergant, Benjamin Busjaeger, Subho Chatterjee, Terry Chong, Roy Chowdhuri, Doug Doole,
Haiyan Du, Thomas Fanghaenel, Pat Helland, JV Jujjuri, Jim Mace, Ashish Mahajan, Jamie Martin,
Nicolas Michael, Ayman Mobarak, Steven Raspudic, Randy Spalten, Simon Wong, Nat Wyatt
Salesforce

## ABSTRACT

Salesforce Database (SalesforceDB) is a multi-tenant relational database built by Salesforce to power the transactional workload of its Customer Relationship Management applications and platform. It is the system of record and an integral part of the Salesforce stack. Using its own database allows Salesforce to innovate from the database layer up and integrate the application's multi-tenancy into the database, enables the creation of new capabilities, optimizes operations, and improves security. Designed from the ground up to be cloud-native with a focus on high reliability and performance, it allows for simple, reliable cloud operations and scales to Salesforce's largest customers. The database manages customer data within the Salesforce infrastructure but is not directly accessible by customers. It is deployed in production across thousands of instances and dozens of regions around the world and holds many petabytes of data.

This paper describes the motivation and innovative design of SalesforceDB. Like many modern databases, SalesforceDB is cloud-native and decouples compute and storage. Its design reflects Salesforce-specific requirements: it natively supports multi-tenancy, employs a shared, immutable, distributed storage system, and uses a tenant-clustered log-structured merge tree (LSM) for data access. This enables it to support high levels of availability and durability, including tolerating availability zone (AZ) failures. SalesforceDB's architecture includes a novel LSM virtualization layer that facilitates tenant-level features that cannot be efficiently supported on existing database architectures. These include creating instant tenant copies, seamless tenant migration across database instances via metadata-only operations and efficient tenant-level encryption. Its shared-table multi-tenancy design minimizes per-tenant overhead. SalesforceDB's scale-out architecture leverages its shared LSM storage to achieve high scalability with minimal coordination.

**PVLDB Reference Format:**
Atish Agrawal, Vaibhav Arora, Gary Baker, Milena Bergant, Benjamin Busjaeger, Subho Chatterjee, Terry Chong, Roy Chowdhuri, Doug Doole, Haiyan Du, Thomas Fanghaenel, Pat Helland, JV Jujjuri, Jim Mace, Ashish Mahajan, Jamie Martin, Nicolas Michael, Ayman Mobarak, Steven Raspudic, Randy Spalten, Simon Wong, Nat Wyatt. Salesforce Database: A Cloud-Native Multi-Tenant OLTP Database. PVLDB, 19(12): 4426-4439, 2026.
doi:10.14778/3827998.3828043


Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828043

## 1 INTRODUCTION

Transactional workloads are a fundamental part of Salesforce's applications. The transition to a modern public cloud infrastructure [9] with increased growth and scale across multiple regions and mission critical applications requires increased resiliency and scalability throughout the stack. In particular, the database layer needs to provide high reliability in cloud environments, support automated and simplified operations, be deployable across multiple cloud providers, and meet the growing scale and emergent requirements of applications in an agentic environment. The ability to leverage a unified multi-tenant model directly within the database creates new product capabilities and improved operational and cost efficiencies. For example, co-designing database observability with application-layer service protection and background jobs enables intelligent throttling, admission control, and mitigation of noisy neighbors, all of which are best achieved in sympathy with the full application stack. The resultant ability to create better end-to-end outcomes is a key function of the deep integration of the database into functional, scale, security, and operational aspects of the Salesforce service.

### 1.1 Requirements for SalesforceDB

Trust is Salesforce's foremost value, and for a database system this implies ensuring high levels of availability, durability, and consistency. To satisfy its availability requirements, SalesforceDB must maintain a high uptime and be resilient to the failure of individual nodes or even entire availability zone failures, as well as accommodate operational activities like upgrades and security patching. Durability means guaranteeing zero data loss, under any hardware or software faults, as well as any accidental deletion from the application or even customer actions. The database needs to provide transactional consistency, ensuring that concurrent operations and failures never leave customer data in a logically corrupt state.

Salesforce provides a rich platform for Enterprise applications [9]. The application drives a read-heavy workload comprising high volumes of small read-write transactions, some large transactions, and a diverse mix of queries, including real-time analytics, queries for supporting page loads, and API-driven access.

The Salesforce application leverages the database to manage tenanted operations, where a "tenant" is essentially a single organization or customer of the Salesforce service. For example, the application uses the database to create isolated "sandboxes" — copies of a customer's production environment — for development, testing, and user training. On existing databases, creating a sandbox requires physically copying data, which can be time-consuming for large tenants, and lengthens the development cycle on the Salesforce platform. Beyond sandbox creation, compliance and regulatory obligations have introduced the need for per-tenant data encryption, where each tenant's data is protected by its own encryption key.

Salesforce makes it very easy to sign up new customers. So there are many small tenants — over 4 million at the time of writing this paper. The database must effectively handle this large number of tenants and minimize per-tenant overhead.

The database architecture must also be scalable to accommodate its largest tenants. Small database instances may have hundreds of thousands of tenants and if an instance is getting overutilized, the system must be able to seamlessly migrate individual tenants to another instance. But some large tenants can have an entire instance to themselves, and this dedicated instance has to scale to the entirety of the single tenant's workload, as well as accommodate seasonal peaks.

The Salesforce application operates at a significant scale worldwide, and SalesforceDB needs to support workloads at this magnitude. Fleetwide, SalesforceDB has thousands of instances that currently process a total of more than 4.8 trillion SQL statements per day and over a trillion record modifications (inserts, updates or deletes). Supporting a fleet of this size requires an architecture that in addition to being resilient, supports simple, scalable operations.

## 1.2 SalesforceDB Architectural Principles

These requirements drove the architecture of SalesforceDB. SalesforceDB is built upon PostgreSQL [57]. PostgreSQL's mature code base provides good basic query processing, transaction processing and schema management capabilities. SalesforceDB extensively evolved these PostgreSQL features to meet the performance and availability demands of the Salesforce application. However, there were significant drawbacks to using the Postgres storage and access methods for achieving the goals of scalability, simple cloud operations, and multi-tenancy. For those reasons, SalesforceDB has a proprietary LSM-based storage and access layer, upon which its scale-out architecture is built.

**Core Architectural Principles**: The core architectural principles that drove SalesforceDB's design are listed below:

- **Separation of Compute and Storage**: In SalesforceDB, stateless SQL compute nodes access data from a shared distributed storage system, enabling independent scaling of compute and storage, and high resiliency. This also allows multiple database instances to run on the same storage cluster, which underlies several of SalesforceDB's innovations.
- **Immutability**: SalesforceDB employs immutable LSM-based storage [51], and immutability [41] is leveraged in several places in the system design. Immutable storage eliminates an entire class of corruptions and concurrency issues that plague in-place update engines during B-tree splits and page modifications, simplifies backup and recovery, and enables simple cloud object storage interfaces. And, as will be discussed later, immutable storage significantly reduces the need to coordinate across nodes for efficient scale-out. We have found that the benefits of immutability have outweighed the read performance challenges inherent in LSMs.
- **Multi-tenancy**: The database natively supports multi-tenancy. Tenants are database objects and storage is clustered by tenants in a single LSM. This provides the foundation for building efficient operations at the granularity of the tenant.
- **Cloud-Native Design**: SalesforceDB is designed to run on commodity cloud infrastructure, leveraging redundant components for high availability and durability. Operations are simplified through Kubernetes-orchestrated deployments and autonomous failure recovery.

## 1.3 SalesforceDB's Design Overview

SalesforceDB is a distributed multi-tenant Online Transaction Processing (OLTP) database. It is built on the foundation of the Salesforce Database Store (SDBStore), a highly distributed, immutable storage system that originated as a fork of Apache BookKeeper [1, 44].

SDBStore is an append-only, non-POSIX storage system that uses cloud object storage as its durable system of record, with a distributed cache providing low-latency access. Quorum writes across three availability zones, together with immutability and flexible write units, deliver fault tolerance and low-latency transaction commits. A single SDBStore is shared by multiple SalesforceDB instances.

A core design principle of SalesforceDB is multi-tenancy, enabling efficient support of operations at the granularity of tenants. To effectively cater to these requirements and minimize per-tenant overhead, SalesforceDB chose the shared-table multi-tenancy model, rather than a database per-tenant [62], or a shared-process approach [28, 58]. A single instance of SalesforceDB supports multiple tenants, with all the tenants sharing the same physical schema.

The multi-tenancy is embedded into the database storage. SalesforceDB employs an LSM-based storage-engine, with the underlying persistent data files being immutable. A single leveled LSM [47, 55] is used for the whole database. SalesforceDB supports a large number of tables and tenants within a single database instance, and a single LSM-design helps minimize the metadata overhead, as compared to a per-table [13] or per-tenant LSM model. The storage engine can be viewed as an LSM key-value store with relational information mapped onto the keys (primary keys) and values (record payloads). The key encoding enables the LSM to store the data clustered by tenant, even though the logical view of the database is per table or index. Employing an LSM-based storage engine does create read-performance challenges [54], as the reads suffer from amplification and have to consolidate multiple versions of data across many levels. SalesforceDB employs several techniques including extensive use of filters to optimize for read-performance on LSMs [18].

SalesforceDB virtualizes the persistent LSM. LSM Virtualization is enabled through the introduction of virtual file references, which point to immutable data files in SDBStore, and contain metadata (keyspace, tenant, timestamp). These virtual file references, coupled with the fact that SalesforceDB has a uniform schema across instances, allow seamless tenant migration without physically copying the data. Virtual file references also allow for instant tenant cloning by associating source and cloned tenants within the file reference, with queries applying the tenant translation logic at runtime. Such instant cloning of tenants is not supported on existing databases and speeds up feature development and testing for customers through rapid sandbox creation. Additionally, tenant-clustered LSM storage facilitates tenant-specific encryption.

SalesforceDB uses a scale-out architecture consisting of a primary cluster for read-write transactions and multiple secondary clusters, for executing read-only transactions. Each cluster is made up of multiple database nodes. Within each node are a Memstore, an in-memory component for managing uncommitted and recently committed data, and a view of the persisted LSM, which holds committed data. The database needs to handle non-partitionable application workloads,

with high skew. To meet this challenge, any node in the primary cluster can process read or write requests for any database record, rather than partitioning the keyspace. This avoids two-phase commit or on-demand re-partitioning, and distributes load evenly. A designated node writes to the log and that node's Memstore coordinates uncommitted records, locks, and commits within the cluster. Because most data resides in the immutable shared LSM, only recently modified data requires coordination. SalesforceDB employs coordination-minimization techniques to enable efficient scale-out.

Many other modern databases separate compute from storage [27, 34, 45, 62], or are natively multi-tenant [28, 58] or built on immutable storage [48, 69]. The differentiating factor of SalesforceDB is in integrating its shared multi-tenant schema design, compute-storage separation and immutable tenant-clustered storage to build its unique LSM virtualization layer. This layer enables innovative features like instant tenant cloning and migration via metadata only operations that are not feasible on existing architectures, and supports efficient tenant-level encryption. SalesforceDB's distinctive scale-out architecture leverages the immutable shared LSM and scales horizontally with minimal coordination, without relying on data partitioning [27, 59], or coordination on mutable pages [61], or depending on specialized hardware solutions [65, 66].

The rest of this paper details the key architectural elements of SalesforceDB: high availability and durability (Section 2), distributed immutable store (Section 3), shared-table multi-tenancy design (Section 4), LSM Virtualization (Section 5), and scaling-out compute (Section 6). Evaluation, related work and lessons learnt are presented in Sections 7, 8, and 9, respectively. The paper concludes in Section 10.

## 2 AVAILABILITY AND DURABILITY

To ensure high levels of availability and durability, SalesforceDB has redundancy and automatic failover/repair for both compute and storage. The SalesforceDB service architecture runs across three availability zones, with compute and storage replicated across these zones to ensure the system remains available in the presence of node or zone failures. All services run in Kubernetes to enable automated failure recovery and service deployments.

SalesforceDB provides "AZ+1" availability — the system remains available if an availability zone is completely unavailable, while also allowing single-node failures in the healthy zones. Importantly, this allows software releases while a zone is unavailable.

To provide high levels of durability, the ultimate system of record for SalesforceDB is cloud storage such as AWS's S3 [4]. Due to high latency in cloud storage, SalesforceDB uses storage caches to access data. These caches are distributed storage systems that maintain temporary copies of storage objects in a cluster of nodes, ensuring replication as needed by the database.

The SQL compute tier consists of a primary database cluster and two standby clusters in three different availability zones. The primary cluster handles all modifications, while the standby clusters handle read-only operations and background jobs like LSM compaction. All the compute nodes share the same storage cluster.

Figure 1 illustrates the high-level architecture of SalesforceDB. Omitted for clarity from the diagram is a coordination tier that runs Apache ZooKeeper [43]. The coordination tier provides leader election and quorum services for compute and caching, as well as discovery for external clients.

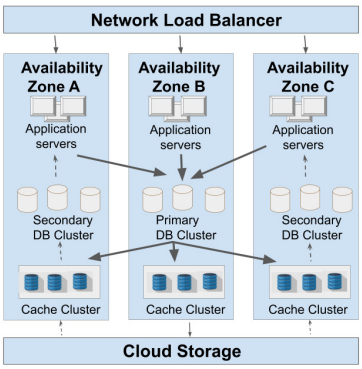


**Figure 1: SalesforceDB Architecture Overview**

### 2.1 SQL Compute

SQL compute nodes are stateless, with all state maintained in the storage tier. This allows any node to be terminated at any time without data loss: ongoing transactions are aborted, Kubernetes starts a replacement, and it recovers state from storage. This fully automated process handles hardware faults, software errors, and operational scenarios such as upgrades and patching. If a majority of nodes in the primary zone become unavailable, a secondary zone is promoted to primary and takes over read-write traffic.

In addition to software releases, the database has to apply schema changes without downtime. Schema changes can happen on a weekly basis, with major schema changes during three yearly major Salesforce releases. Schema management is tightly tied to the application release model and supported by schema versioning in the database. Each schema release is associated with a version, and connections see the schema as of that version. Before the release, a new schema version is installed, along with the schema and data modifications needed for the release. During the release, connections to both old and new schemas are active, with updates aligned by triggers as needed. Old schema versions can be retired after the release. DDL modifications of existing schema are non-blocking, and concurrent online index builds ensure no application brown-outs.

SQL compute nodes never overwrite storage. As the database employs LSM-based storage, logical changes to records are written as new versions within an LSM file. Each data file has multiple fixed-size blocks, and every write to storage appends a new block to a file. Each block gets a unique checksum including its file identifier (ID) and block number. This lets reads verify both that the block has not been corrupted and that the correct block was read. Blocks are immutable when cached in memory, which eliminates coordination and removes the possibility of corruptions during writes.

While checksums and immutability address physical corruptions, there is still the possibility of logical corruptions. SalesforceDB uses snapshot-based multiversion concurrency control (MVCC) [22] within the LSM, and that means there is the possibility of a corruption from an update that reads an out-of-date version. To guard against this, SalesforceDB performs lineage checking on updates. Lineage is embedded within the records in the form of previous record's commit timestamp; updates verify that the new version's lineage matches the most recently committed version of that record. This prevents changes based on an outdated version of a row.

This combination of features provides continuous availability over the application lifecycle and software and hardware faults.

## 2.2 Storage

Even though SalesforceDB uses cloud object storage that provides high levels of durability, there can be corruption and errors above the object storage layer. To mitigate such scenarios, SalesforceDB continuously takes and verifies backups from every database in the fleet. These backups are stored in a separate cloud provider account to protect against ransomware. Backups are based on periodic snapshots of the immutable database files, and continuous copying of closed log files. Since standbys are continuously replaying the transaction log, the possibility of an error during restore replay is low.

The storage layer detects corruptions using checksums at both the block and file levels, with file checksums computed hierarchically from block checksums to reduce computational overhead. On a mismatch, the system automatically re-replicates from healthy replicas — or from object storage if needed — leveraging the immutable storage architecture to ensure continuous data integrity.

Although cloud storage is the system of record, the tail latency of random reads for queries and appends to the transaction log are four orders of magnitude too slow. This means loss of caching capacity can impact the effective availability of the database. For this reason the cache nodes also have AZ+1 redundancy. Separate caches are used for transaction log storage and data file storage. The placement of replicas in both the caches is AZ-aware and each file in the store is distributed across three availability zones.

Log writes need to be durable until the log file is closed and moved to cloud storage. So, log writes are cached in persistent block storage devices. However, data cache is write-through and reads can be served from cloud storage as a last resort, so data nodes cache files in local NVMe.

## 2.3 Operations

Operations can also be a source of availability and durability issues. SalesforceDB employs several safeguards against this. Database and storage clusters are cellularized to limit the blast radius of failures. Software releases are staggered across the fleet — starting with Salesforce's own internal systems (engineering, support, sales), then a subset of sandbox instances, then the remaining sandboxes, and finally production. Pre- and post-deployment checks halt the rollout and auto-rollback to the previous version if any problem is detected.

# 3 SDBSTORE

SDBStore's architecture uses cloud object storage as the system of record, accessed through a distributed cache layer. This approach harnesses cloud storage's high durability while mitigating its latency limitations for database use. As described above, caches are replicated across three availability zones. A single cache can hold about a petabyte of data, and multiple database instances share a cache cluster, and the underlying cloud storage. SDBStore has an immutable storage model that aligns with SalesforceDB's LSM-tree architecture, where data files are write-once and transaction logs are append-only. This enables aggressive caching and simplified consistency protocols impossible with mutable semantics, while delivering the performance, durability, and integrity guarantees databases require.

SDBStore is built on Apache BookKeeper [1, 44] for its quorum-based writes, fencing mechanisms, and simple non-POSIX APIs. Its

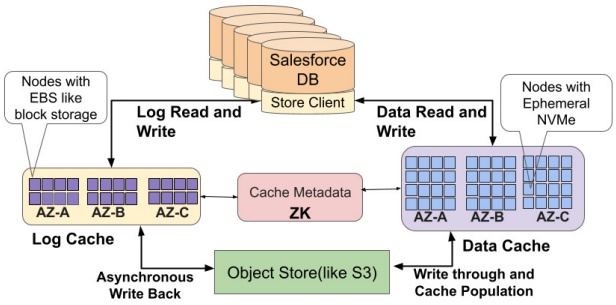


**Figure 2: SDBStore Overview**

immutable object semantics align with database and cloud storage, and supports read, append, and delete operations on immutable blocks in synchronous, asynchronous, and streaming modes.

SDBStore's architecture (Figure 2) treats the cache as the primary interface, with cloud object storage handling persistence [16, 70]. There are two storage caches, one for data, and one for the transaction log. They share the same API but have different implementations. The data cache is write-through and is optimized for high-throughput sequential writes and low-latency random reads. The transaction log cache is write-back, and is optimized for low-latency durable writes and high-throughput streaming reads. Specializing caches has other benefits: each cache is independently scalable and uses storage optimized for its specific requirements.

Both log and data cache clusters dynamically scale in and out based on demand through automated monitoring of cache utilization, request patterns, and performance metrics. Scale-out events automatically integrate new nodes into the cache cluster with parallel data rehydration, while scale-in operations safely redistribute cached content before node removal to maintain data availability during capacity reduction.

SDBStore's immutable storage model is defined in terms of *Extents* that contain Fragments. Extents are conceptually similar to files and fragments are conceptually similar to blocks within a file. Extents are identified by an Extent identifier (ID), which is a randomly generated 128-bit UUID. UUID-based global namespace enables multiple databases to share the namespace without coordination. Extents can only have a single writer. They go through a lifecycle of being created, having fragments appended, being closed, and finally being deleted once no longer referenced by a database.

It is not possible to modify a fragment once written, or append more fragments to the Extent once it is closed. Fragments are variable-sized, numbered from 1 to N, with fragment 0 indicating the last fragment written (for finding metadata at the end of a data extent). Clients can "seek" a fragment by its number. Clients can "tail" an extent — receive fragments as they are appended until the extent is closed.

The file-to-node metadata is implemented by BookKeeper and stored in ZooKeeper. The placement of extents on cache nodes is made via a zone-aware weighted algorithm that takes zones, upgrade domains and the storage load on cluster nodes into account.

## 3.1 Write-Through Data Cache

The data cache is a write-through cache on ephemeral NVMe storage backed by cloud object storage such as S3. This design leverages cost-efficient local NVMe drives to provide sub-ms low latency for reads while ensuring durability through object storage.

Writes to the data cache are always from background flush and compaction processes. The flush process can be a particular bottleneck for overall database ingest. As the data extents are not made visible to the LSM until closed, the data write process needs high throughput. Cloud storage enables this high throughput through parallel upload, and the writes to the NVMe caches are done in parallel.

Reading from cloud storage incurs high tail latencies, with the time taken to read a fragment going into seconds. This high latency significantly hurts database performance. Queries touching multiple fragments or even a single slow fragment increase overall latency. This also strains data structures like the buffer cache, raising lower-level latch contention and reducing database throughput. To minimize access to cloud storage for random reads, the entire database working set is kept in cache, and each extent is cached three times with multiple replicas distributed across availability zones. The data cache also uses hedged reads [33] to further reduce tail latencies.

In addition to the extent-to-cache-node mapping stored in ZooKeeper, each data cache node's extent list is persisted in the cloud object store to enable rehydration after ephemeral storage loss.

## 3.2 Write-Back Log Cache

SDBStore's log cache is a write-back cache backed by durable block storage like AWS EBS [3]. Transactions are written to multiple log storage nodes using quorum replication which ensures low commit latencies in the range of 1-3 ms while maintaining strong durability guarantees. The transaction log data is asynchronously written back to object storage for long-term durability.

The log cache implements an ensemble-based quorum system to achieve low tail latencies while providing strong consistency guarantees during database failover. An ensemble represents the set of storage nodes (Bookies) that collectively store replicas of each fragment [44]. The system operates with an ensemble size (En) of 6 nodes distributed across 3 AZs, implementing a write quorum (Qw) of 6, acknowledge quorum (Qa) of 4, and read quorum (Qr) of 3 replicas. A read quorum of 3 ensures that the database nodes can determine the end of the log even in the AZ+1 fault scenario. If Bookies fail during writing the next immutable fragment, the ensemble transparently replaces them with live Bookies from the remaining zones, making storage node failures transparent to the database.

The log cache supports SalesforceDB's single log writer model by enabling database nodes to tail the transaction log for fast failover. During failover, the log cache fences [44] the previous log writer and accurately detects the end of the log, even during AZ+1 failures. This quorum-based mechanism ensures zero-data-loss failovers and prevents split-brain scenarios. Regular log-tailing operations avoid quorum reads, reading from a single storage node only up to the last quorum-acknowledged fragment ID. This fragment ID, maintained at every replica, indicates the point up to which a quorum of nodes has confirmed acknowledgment of all preceding fragments [44].

# 4 SHARED-TABLE MULTI-TENANCY

SalesforceDB employs a shared-table multi-tenancy design [25]. The motivation for this approach is to efficiently support a large number of tenants per database instance and align with the Salesforce application schema, which co-locates the data for multiple tenants in the same table. The Salesforce application provides extensible and customizable behavior over these tables for different tenants [64].

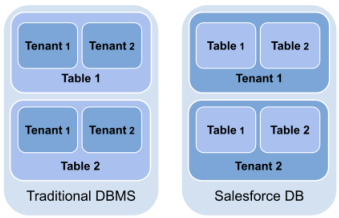


**Figure 3: SalesforceDB Multi-tenancy**

SalesforceDB defines a tenant lifecycle, and uses a key-encoding that organizes the storage by tenant. This encoding ensures all data for a single tenant forms a contiguous key range in the LSM (Figure 3), enabling the database to treat a tenant's data as a single mobile logical unit. The key encoding also colocates all records for a tenant within each table, making scans more efficient.

## 4.1 Tenants as Database Objects

Tenants in SalesforceDB are first class database objects. Each tenant is identified by a unique Tenant ID. The application ensures that these identifiers are unique across the database fleet. Tenants follow a create/modify/drop life cycle that is supported by DDL (Data Definition Language) commands. Tenants have associated states: Creating (initial state), Active (permits reads and writes), Quiesced (read-only), or Maintain (for internal database operations). These states enable a wide range of database operations. Any transaction modifying a tenant's data takes a shared lock on the tenant object, which is used to synchronize with operations attempting to change the tenant state as well as identify ongoing transactions on a tenant.

Records stored in SalesforceDB may not be associated with a tenant. SalesforceDB distinguishes between tenanted data, i.e., records associated with a single specific tenant, and non-tenanted data, i.e., records not associated with any tenant.

In addition to CREATE and DROP commands, SalesforceDB supports creation of point-in-time snapshots of tenants. Those tenant snapshots are the foundation for tenant mobility and tenant cloning.

## 4.2 Tenant Clustering

SalesforceDB's storage layer is based on a single LSM key/value store that supports a linear keyspace (referred to as the galactic keyspace because including unique Tenant IDs makes keys unique across all SalesforceDB instances). The LSM layer is implemented on immutable storage.

**Multi-Tenant Tables and Indexes.** SalesforceDB implements two different types of tables. Non-tenanted data is stored in regular tables, while tenanted data is stored in designated tables called multi-tenant tables.

Multi-tenant tables must have a Tenant ID column as the leading column in their primary key. All secondary indexes on multi-tenant tables must also reference the Tenant ID column as the leading column of the index key. For records in multi-tenant tables, the galactic key is composed of the Tenant ID, followed by the table ID and the remaining primary key columns. Keys for records in non-tenanted tables are formed by concatenating the table ID and the columns of the record's primary key in that order.

Secondary indexes are implemented as standalone mapping tables between secondary index keys and primary keys of the associated base table records [53]. Secondary indexes are mapped onto the same

linear galactic keyspace as tables. Galactic keys for index records are composite keys and formed using a combination of Tenant ID (if the index is on a multi-tenant table), index ID, the index key, and the galactic key of the associated base table record.

As a consequence:

- Storage of tenanted and non-tenanted data is mapped into two different segments of the linear galactic keyspace.
- Multi-tenant tables and indexes on multi-tenant tables are discontiguous in the linear galactic keyspace.
- All data associated with a single tenant forms a contiguous key range in the linear keyspace.
- A single tenant's data in a multi-tenant table or index is contiguous.

Records for a tenant in a multi-tenant table are stored contiguously and are fast to scan, whereas the table as a whole is discontiguous and inefficient to scan. Scanning an entire multi-tenant table without a Tenant ID is an anti-pattern at Salesforce; however, when a full table scan is submitted to SalesforceDB without a Tenant ID qualifier, instead of issuing a keyspace scan, it transparently transforms the query into a join with the table of tenants.

**Tenant Isolation** Isolation between tenants is enforced by predicates added by the Salesforce application. SalesforceDB will check and enforce the existence of these predicates when the tenant context is set in the session.

Performance isolation is achieved through resource tracking in the database engine and throttling at the Salesforce application level. Database requests return resource usage (CPU, Memstore, connection, I/O) to the application layer. This layer allocates usage to operations (Orgs, Users, URIs) and throttles requests when usage exceeds a resource threshold.

### 4.3 Tenant Encryption

The tenant-clustered LSM also enables the encryption of data with rotatable tenant-specific keys. To facilitate this encryption, all data stored within a fragment belongs exclusively to a single tenant. Data extents are written to the LSM during flush and compaction, and each fragment is encrypted using a tenant-specific key, with the key ID embedded in the fragment. When a fragment is subsequently accessed, whether for a query, LSM compaction, or any other process, it is decrypted using the corresponding tenant key. Both encryption and decryption times are negligible as compared to the reading and writing of the fragment to the SDBStore. By restricting a fragment to a particular tenant's data and performing encryption at fragment boundaries, SalesforceDB achieves tenant-level encryption while limiting the overhead on database operations.

A key enabler of tenant-level encryption is the immutable persistence. In traditional databases, any page mutation and subsequent eviction from the buffer pool would result in re-encrypting the page.

A key management system (KMS) is responsible for managing the encryption keys in Salesforce. Encryption keys can be rotated, and the data rotation is triggered via LSM compaction. Since a key's identifier is embedded in the fragment, data encrypted with the old encryption key can still be read, until the data rotation is completed.

## 5 LSM VIRTUALIZATION

SalesforceDB uses tenant clustering and the immutable LSM to make cloning and moving tenants a metadata-only operation, via what we call "LSM Virtualization".

### 5.1 Virtual File References

Data in SalesforceDB is persistently stored within the immutable LSM as extents. However, it is the extents that are immutable; the LSM itself changes shape over time as new extents are flushed from memory, and extents are combined by the compaction process. The shape and contents of the LSM is maintained in a per-database instance storage manifest. The metadata is made up of Extent References (ER) that describe an extent, and Extent Reference Sets (ERS) that relate groups of ERs.

In SalesforceDB, ERs serve as virtual file references. An ER is an abstraction layer over an extent, and includes the extent's ID and metadata defining the visible keyspace (minKey, maxKey), timestamp range, and tenant mapping (old tenant, new tenant).

Multiple ERs can point to the same extent, and the metadata in the ER can be used to restrict visibility to specific keys, time, or tenants. Furthermore, the tenant mapping in the ER enables virtualizing the underlying immutable extent data, allowing it to be shared across tenants. This mapping acts as a link between the source tenant of the data, and the new tenant to which the storage is mapped. The tenant mapping information is embedded in the access layer of the database which transparently performs tenant translation during query processing and LSM compaction.

Because records are clustered by tenant, an ERS can be defined to exactly match the records for a given tenant in the LSM. Then, SalesforceDB can perform metadata operations on the storage manifest to drop, migrate, or clone a tenant.

### 5.2 Tenant Snapshots

A tenant ERS can be saved into the storage manifest as a Tenant Snapshot. As extents are immutable, these snapshots guarantee a stable representation of the tenant's data at a given point in time. In addition to the tenant's ERS, a tenant snapshot also includes metadata that captures the database state, such as the schema fingerprint.

Figure 4 demonstrates how tenant clustering facilitates the construction of a tenant snapshot by combining Extent References from various LSM levels. In this example, the ERS for the database comprises Extent References ER1 through ER10. To form the tenant snapshot for Tenant B, the eight references that intersect Tenant B's keyspace are duplicated as virtual references ER1′, ER2′, ER3′, ER4′, ER7′, ER8′, ER9′, and ER10′. The references with overlapping data from other tenants are modified to restrict them to Tenant B's keyspace: the minimum keys of ER1′, ER2′, and ER7′ are adjusted to align with the beginning of Tenant B's keyspace, and the maximum keys of ER4′ and ER10′ are adjusted to align with its end. These eight virtual Extent References collectively constitute the tenant snapshot.

Creating, exporting, importing, and deleting tenant snapshots are metadata-only operations. The underlying data extents only need to be stable and do not need to be moved or copied. Since the metadata is several orders of magnitude smaller than the data, these effectively become constant-time operations.

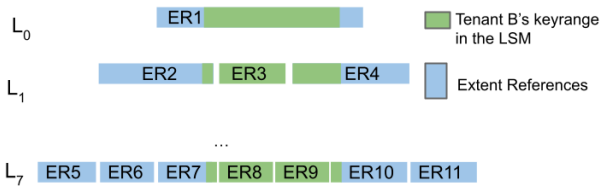


**Figure 4: Virtual References**

### 5.3 Enabling Cross-Database Storage Sharing

To allow migrating or cloning a tenant across databases, the system needs to ensure that it is safe to share persistence across databases. First, the differences in schemas need to be taken into account. For instance, consider a scenario where a tenant is migrated to a target database that possesses a multi-tenant index (index-T) on table T, but the source database lacks this index. Following migration, a query accessing the tenant might incorrectly attempt to utilize index-T, potentially leading to inaccurate results. The second challenge is to ensure that extents shared across databases don't get prematurely deleted while one of the databases is still actively accessing the extent.

For directly mapping one instance's storage into another, database schemas have to be identical across the participating database instances. The schema evolution process implemented by the Salesforce application layer ensures that schemas evolve in a coordinated fashion across the fleet. Furthermore, the application has logic to pick suitable database targets with matching schemas and compatible database versions. But the database still needs to perform validations to account for any ongoing schema changes or application bugs.

SalesforceDB verifies schema compatibility by computing and maintaining a schema fingerprint on each database instance. It is maintained as a compact hash value capturing the logical shape of the database (tables, indexes, attributes). The schema fingerprint of an instance is compared against the schema fingerprint in the tenant snapshot to ensure that it's safe to import it into the database.

The extent deletion process carefully accounts for the sharing of extents. Specifically, an extent is only deleted when it is no longer referenced by any ERSes in any database's storage manifest within the same storage cluster. In addition, the application manages the sequencing of creation, exporting and snapshots in a way that shared extents remain referenced throughout the process of being shared across databases.

### 5.4 Tenant-Migration

When migrating a tenant across SalesforceDB instances, the application first selects compatible source and target database instances sharing the same storage cluster. The migration process is orchestrated by the application, and the corresponding databases do not communicate directly with each other. The application first sends a request to move the tenant to Quiesced state; ongoing write transactions are drained, with any that exceed a configured time period being aborted. Next, a tenant snapshot is created to capture the tenant's point-in-time state. As part of creating the tenant snapshot, any data that the tenant has in memory is flushed to the persistent LSM. The snapshot's Extent References are then exported to the target database instance.

First, an empty tenant is created in the initial "Creating" state, invisible to the application. The set of ERs in the tenant snapshot is then added to the target's storage manifest, splicing the tenant into the instance's keyspace. The tenant is then activated on the target, and the application can start routing customer traffic to it. Upon migration completion, the source tenant and the snapshot on the source database are dropped. If errors occur or the migration process exceeds the configured timeout, the tenant is re-activated on the source, by moving it back to the Active state.

### 5.5 Sandbox Creation

Sandbox creation is a multi-stage process resembling migration; LSM virtualization handles the core data cloning between the source and sandbox tenants, while additional application-side stages prepare the sandbox for use, including post-copy cleanup (e.g., row-by-row deletion of records not needed in the sandbox and PII sanitization). Like migration, the application creates a tenant snapshot on the source and transfers it to the target instance designated to host the sandbox. Then the target instance creates a new Tenant ID for the sandbox and splices the snapshot's Extent Reference set into this new tenant's keyspace. Unlike migration, this process clones data from the source tenant to a new, distinct Tenant ID. Cloning is achieved by adding a tenant mapping from the source to the clone for all imported Extent References. The minKey and maxKey of these references are updated to point to the cloned tenant, while the physical extent data still references the source.

Queries on the cloned tenant perform tenant translation at runtime based on the tenant mapping information. When a probe or a scan is performed for a cloned tenant, the access layer reads data corresponding to the Extent References in the cloned tenant's keyspace. The imported Extent References in this keyspace still point to the extents whose data records carry source Tenant IDs. The tenant information in the records is translated, allowing the results to be returned as if the data belonged to the cloned tenant. In addition to the automatic translation of the primary Tenant ID column, the schema allows explicitly designating other columns containing Tenant IDs for on-the-fly translation.

Similarly, LSM compaction on keyspaces with tenant-mapped Extent References will also perform the translation. The compaction will write out the data records with cloned Tenant IDs. Any new write on the sandbox tenant will lead to records being encoded with the cloned tenant's ID.

## 6 COMPUTE SCALING

SalesforceDB compute scaling is constrained by the following requirements:

- **External Consistency**: The system must provide external consistency, ensuring that the multi-node database cluster behaves identically to a single, centralized database instance.
- **Read/Write OLTP Workloads**: Salesforce applications are inherently transactional, and typically have a roughly 30%/70% ratio of writes to reads with reads and writes intermingled in the same transaction. Salesforce applications cannot be scaled out by sending reads to secondary read replicas.
- **Non-Partitionable Workload**: SalesforceDB must support complex schemas with high data skew that aren't amenable to partitioning. SalesforceDB cannot impose application-level partitioning or rely on assumptions of data locality.
- **Commodity Cloud Infrastructure**: SalesforceDB is designed to run in cloud environments using standard cloud infrastructure

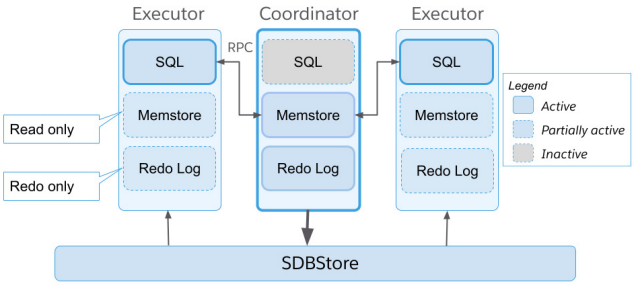


**Figure 5: SalesforceDB Scale-out**

and commodity SKUs, it is not possible to utilize specialized hardware (like SANs or Infiniband) to achieve scaling requirements.

Despite these constraints, SalesforceDB scales compute effectively by leveraging immutability, snapshot isolation [12, 21, 42], filtering, and log replication to minimize coordination.

## 6.1 Cluster Components and Roles

The primary read/write SalesforceDB cluster (Figure 5) consists of a single Coordinator node and multiple Executor nodes. Application servers distribute and balance load across the cluster.

**Coordinator.** The Coordinator is a single, dedicated node responsible for processing all low level database record writes and serving reads of recently committed data. This asymmetry is central to the design, as the Coordinator hosts the single active Memstore for the cluster. The Coordinator's Memstore holds uncommitted data, maintains row locks, adjudicates lock conflicts, and serves reads of recently committed data for the entire cluster. Each uncommitted record acts as an implicit row lock, blocking concurrent writers to that row. Writes sent to the Coordinator's Memstore are logged in the Coordinator's local transaction log, which is persisted in SDBStore.

The Coordinator also hosts the Epoch Coordinator, which functions as SalesforceDB's timestamp oracle (TSO). SalesforceDB's read and commit timestamps are encoded as Transaction Commit Numbers, aka XCNs, which represent a logical clock roughly approximating system time in nanoseconds. XCNs are implemented as 64-bit unsigned integers. The Coordinator maintains two distinct XCNs: the current commit epoch and the current visible epoch. The commit epoch is used by the Coordinator to assign commit timestamps to committing transactions. The visible epoch is a read timestamp assigned to statements running at Snapshot Isolation across the cluster. The Epoch Coordinator guarantees that the current commit epoch is monotonically increasing and is necessarily greater than the current visible epoch. The Epoch Coordinator advances epochs roughly every 1 ms, at which time the visible epoch advances to be greater than or equal to the maximum commit timestamp assigned in the previous epoch. The Epoch Coordinator broadcasts the visible epoch to all cluster members. Executors use the most recently received visible epoch to assign snapshot timestamps locally.

When the Coordinator fails, a new Coordinator is elected from the remaining Executor nodes in the cluster using ZooKeeper as an external coordination service. Normal transaction log recovery guarantees that all in-flight transactions not yet durably committed are aborted, and the newly elected Coordinator's Memstore has the complete state of the recently committed transactions. After recovery is complete, a new monotonically increasing commitXCN is established based on the end of the log, and writes can be resumed. Other than transaction aborts, a Coordinator failure is transparent to SalesforceDB clients: RPCs issued by Executors may briefly stall, but errors do not surface to database clients, and ongoing operations like scans transparently resume across different Coordinator instances. A full Coordinator failover and recovery typically completes in less than 5 seconds.

In large clusters the Coordinator is dedicated to processing reads and writes on behalf of Executors (Figure 5). In smaller clusters the Coordinator also serves customer traffic like the Executors.

**Executors.** Executors are peer nodes in the primary cluster that accept read-write application connections, and tail the Coordinator's transaction log. They are designed to process work locally, carrying out the bulk of query planning and execution without communicating to other nodes. When an Executor needs to read data residing in the Coordinator's active Memstore or needs to write data to the Coordinator, it must issue a Remote Procedure Call (RPC) to the Coordinator. RPCs are issued over TCP/IP. Executors use a number of techniques described later to minimize the amount of RPCs issued to the Coordinator, and to minimize the latency impact of RPCs.

## 6.2 Write and Commit Flow

SalesforceDB's query execution layer is unaware of the clustering topology. All coordination between Executors and the Coordinator happens below the access methods. When DML (INSERT, UPDATE, DELETE) requests are received by Executors, the Executor processes the DML locally and invokes access methods which do the work of inserting the record(s) into the Coordinator's Memstore transparently.

Because the majority of time for even a simple statement like a single row INSERT is spent in query execution — for example generating the new records to be inserted in the base table and secondary indexes, validating table constraints such as uniqueness or referential integrity, executing triggers, generating return values, and so forth — the overhead of the corresponding RPCs for sending Memstore inserts to the Coordinator is quite low (~5%) relative to the entire execution of a simple INSERT statement.

To further reduce the execution impact of RPCs and to maximize the efficiency of the Coordinator, SalesforceDB batches RPCs and sends RPCs asynchronously where possible. For example, a single row SQL INSERT may insert several records into secondary indexes in addition to inserting the base row. These separate Memstore inserts are always batched into a single RPC. Similarly, some related read operations such as snapshot violation checks and duplicate key checks are processed as part of the logical insert operation rather than sending multiple distinct read and write RPCs. SalesforceDB also issues writes asynchronously to reduce the impact of the latency of the RPC: if a single DML modifies multiple rows, those modifications are submitted asynchronously with a blocking synchronization point at the end of the statement. This keeps the latency impact of RPCs on DML to a minimum.

When a commit is submitted to an Executor, a commit RPC is sent to the Coordinator. The commit RPC blocks until the transaction has been made fully durable in the transaction log in SDBStore. The commit RPC returns the commit timestamp (XCN) of the transaction. In order to maintain external consistency, before the commit notification is returned to the client, SalesforceDB must guarantee that the commit is visible to any future query to any node in the

cluster. To accomplish this, the Executor checks the current local read timestamp. If the local read timestamp has advanced beyond the commit XCN, and advanced once more, then any subsequent query made to any node in the cluster is guaranteed to see the committed data so the commit status can be returned to the caller. This additional commit stalling is only borne by the committer, and is typically minimal because the duration of making a commit durable (1-3 ms) is approximately the same time that it takes to advance the global epoch twice (1 ms * 2 = 2 ms).

### 6.3 Read Flow

Salesforce queries always run at Read Committed Snapshot Isolation (RCSI) which is implemented in SalesforceDB using MVCC [22]. This means queries in SalesforceDB, other than a SELECT FOR UPDATE, never need to RPC to the Coordinator unless the query needs to access very recently committed data as required by its statement's snapshot timestamp.

Reads are broken down into probes, the retrieval of a single record identified by a unique key, and scans over a range of keys. Probes are processed by logically looking through the LSM in time order: probes first look for a record in Memstore, then proceed through the LSM level by level for each extent that may have matching data until a record is found with a commitXCN <= snapshotXCN. Scans are logically processed by opening sub scans across Memstore and all the levels of the LSM and deduping the results in a merge.

Because data is written to persistence shortly after it is committed, approximately 99.9% of the data accessed by queries is retrieved through shared persistence. Since SalesforceDB's persistence is immutable, there is no need to coordinate persistence access across nodes in a cluster. This is radically different than traditional page-oriented databases where writes are made to mutable pages. A scale-out page-oriented database typically has to coordinate all page readers with page writers using some form of distributed page cache with inter-node invalidations and distributed latch management. In SalesforceDB each node has its own buffer manager and can load in and read data without any coordination with any other database node. This means the majority of reads for a SalesforceDB Executor happen locally without any coordination between the Executor and other Executors or with the Coordinator. This in turn means that query optimization needn't take remote accesses into account during costing.

Each Executor tails the Coordinator's log through SDBStore and has a full representation of the data between the maximum XCN in persistence and its redoXCN, the most recent transaction commit timestamp below which all transactions have been fully replayed into its local Memstore (Figure 6). Scans and probes for data between redoXCN and the maximum XCN in persistence can be served from local Memstore. To guarantee that an uncommitted transaction can always read its uncommitted writes locally, Executors also apply transaction writes to their local Memstore in addition to RPCing them to the Coordinator. Executor writes to local Memstore are reconciled during redo.

The Executor also leverages Bloom filters [23] to avoid unnecessary RPCs to the Coordinator for data that cannot be present in the Coordinator's Memstore. For the time period between the Executor's local redoXCN and a query's snapshot time, the Executor consults Bloom filters it has retrieved from the Coordinator, which tell it about the data that is present in the Coordinator's Memstore. Each record

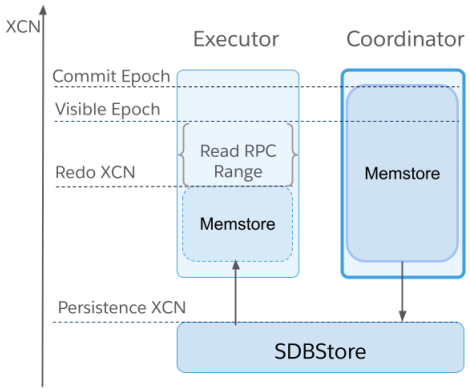


**Figure 6: Reads at Executor Nodes**

that is committed in Memstore on the Coordinator is added to a Memstore Bloom filter. To avoid unnecessary probe RPCs, the Coordinator tracks primary keys and secondary unique keys in the Memstore Bloom filters, which are shipped to Executors. For example, consider an Employee table with a unique index on EmailAddress. A query submitted to the Executor *WHERE EmailAddress = jacksmith@gmail.com* with snapshotXCN greater than redoXCN would probe the Memstore Bloom filter obtained from the Coordinator for that specific key. If the Bloom filter indicates that the key *may* exist in the Coordinator's Memstore, then an RPC to the Coordinator is necessary; otherwise, the remote Memstore probe can be bypassed entirely. Memstore Bloom filters eliminate ~99% of potential Coordinator probe RPCs.

To avoid unnecessary scan RPCs the Coordinator additionally stores a key prefix for the relation (index or base table) that was mutated and the key prefix up to the first column of the key for the mutated record for secondary indexes in its Bloom filters. Memstore Bloom filters eliminate ~60-70% of potential scan RPCs.

## 7 EVALUATION

This section presents data from both production clusters as well as detailed experimental evaluation to highlight the effectiveness and benefits of SalesforceDB's architecture. This section illustrates that SalesforceDB meets the requirements it had set forth to achieve:

- High availability and durability in presence of zone failures
- A distributed storage system with low-latency writes for transactions and low latency reads for queries
- Efficient multi-tenancy as well as enabling new multi-tenant features for customers: faster sandbox creation, seamless org-migration, and encrypting data using per-tenant keys.
- Scaling-out effectively to support large tenants

The SalesforceDB fleet comprises thousands of database instances running production workloads. The shared-table multi-tenancy model with one shared schema efficiently consolidates small tenants, with each instance hosting from one to hundreds of thousands of tenants. Each database instance is typically tens of terabytes in size, with the large instances reaching up to hundreds of terabytes. Unless otherwise noted, production fleet results are from April to June 2026.

### 7.1 Operational Resilience and Availability

**Production Availability.** Unavailability for SalesforceDB is defined as an interval of 30 seconds or more during which the cluster was not processing writes. The application can withstand errors during

this period and retry unsuccessful operations. The unavailability events include issues caused by software bugs, hardware failures, and routine maintenance such as database or storage upgrades. On this measure, the SalesforceDB fleet achieved 99.99% availability rate.

**Redundancy in Practice.** Since 2022, AZ redundancy has prevented six major correlated-failure events from impacting customers.

- Five involved thermal events or power outages that took down a subset of instances in a single AZ. There was no database "AZ failover" but individual services (ZooKeeper, SQL compute, storage) failed over to instances in another zone without impacting the overall service. Services on instances on different power or cooling systems within the AZ continued running normally.
- In an incident early in SalesforceDB's deployment history, a subset of storage cache nodes failed to recover and rejoin the cluster, leaving the cache functional but under-provisioned. That problem has since been fixed and serves to illustrate that it is important to learn from incidents.

**Network Partition Handling.** In another AZ-level incident, a fiber cut left a single AZ isolated from the other AZs. During the duration of the network partition, the system remained operational as the storage quorums reestablished themselves in the two connected AZs.

**Architectural Benefits.** Employing LSM-based immutable storage has resulted in zero physical corruption incidents in over five years of production deployments. Another design decision that has proven beneficial for operational resilience is the log-tailing design, which enables fast failovers. Forced database failovers (average: 7.91 seconds, p95: 11 seconds) are frequently used to handle production incidents caused by cloud provider issues such as slow or sick nodes and suboptimal network placements.

**Proactive Corruption Prevention and Detection.** While no physical corruptions have occurred, logical corruptions can happen (Section 2.1). To detect such corruptions, an index checker tool continuously runs on secondary database clusters. This tool identifies index records that point to incorrect base table records or are missing or dangling. The Index Checker has proven highly effective in identifying a diverse range of bugs, including those in index building, recovery, default column values, and ICU library updates. SalesforceDB can also repair corrupt records online without taking the entire index offline. To prevent logical corruptions, the database tracks lineage in data records (2.1) which rejects invalid updates.

## 7.2 Storage at Scale

A single SDBStore cluster can hold petabytes of data. In total, there are tens of thousands of data cache and log cache nodes across hundreds of store production clusters. Fleetwide observations demonstrate SDBStore can support low latency access at scale: group commits averaged 3.03 ms (p95: 3.74 ms), log writes 2.32 ms (p95: 3.60 ms), and random data reads 0.49 ms (p95: 1.20 ms).

## 7.3 LSM Virtualization Efficiency

The numbers below detail the application of SalesforceDB's LSM virtualization technology in practice.

**Tenant Cloning.** LSM virtualization accelerates sandbox creation. This includes creating a sandbox from either a production tenant or an existing sandbox tenant. The gains grow with tenant size — physical data copy dominates sandbox creation time for larger tenants, so eliminating it yields a bigger relative speedup: for tenants $\geq$ 10 TB, LSM virtualization was 2.5× faster than row-by-row copy, reaching a 7.3× speedup for one of the largest tenants (from 22 days to 3 days for an 85+ TB tenant). Across the fleet, 89,163 sandboxes were created this way between April and June 2026, including 584 over 1 TB, with average creation time reduced by 31% (from 8.02 to 5.51 hours). The LSM virtualization step itself, from snapshotting the tenant on the source to activating the clone on the target, averages just 84.4 seconds (p95: 279.9 s), only 0.69% of end-to-end sandbox creation time at p95. The remaining time is application-side work (Section 5.5), now dominated by post-copy cleanup — most notably deletion of records not needed in the sandbox — which is being optimized using range-deletion. LSM-based cloning also improves resource isolation: by bypassing Memstore and transaction coordination, it reduces cell load and interference for customers load-testing their sandboxes.

**Zero-Copy Mobility.** LSM-based migrations are currently utilized to transition trial tenants who convert to paying customers from signup to production environments. Between April and June 2026, 6,596 tenants have been seamlessly migrated across SalesforceDB production instances via this technology with the average migration time of 63 seconds, and a p95 of 151 seconds.

**Encryption with Tenant Keys.** SalesforceDB encrypts more than 7,000 tenants, with sizes up to 80+ TB. Encryption is performed when flushing data to LSM or during compaction. Overhead is minimal: encryption adds 13.96 $\mu$s per fragment write (p95: 49.58 $\mu$s), negligible compared to the 2.7 ms total write time; decryption adds 12.52 $\mu$s per fragment read (p95: 25.77 $\mu$s), low compared to 0.49 ms total read.

## 7.4 Experimental Evaluation of Compute Scaling

The following experimental evaluation characterizes the compute scaling of SalesforceDB. For evaluation, we created a synthetic workload that resembles production traffic, and evaluated it on clusters with different numbers of SQL compute nodes. Throughput increases significantly with more nodes, with minimal impact on latency.

**Workload and Benchmark Description.** The workload consists of 6 multi-tenanted tables, 4 read-only and 4 read-write business transactions at a 70%-to-30% read-to-write ratio. Queries vary from point lookups to range scans of hundreds of thousands of records to joins returning up to 500 rows. Data accesses are skewed towards recent writes. Write transactions range from single-record to bulk updates with one to many database transactions per business transaction. The workload ran on AWS r6i.32xlarge instances in a production-like environment, with compute nodes scaled from 1 to 16. For each compute node, the dataset adds 10 large and 100 small tenants sharing the multi-tenanted tables; large tenants have 5× the size and request rate of small tenants. Each configuration ran at 60% CPU utilization on query-processing nodes, with request rates normalized to 60% CPU utilization across configurations via interpolation.

Figure 7 shows normalized throughput and latency scaling behavior when scaling from 1 to 16 nodes. SalesforceDB achieves 11.79× capacity gain (throughput at identical utilization) at 16 nodes, with the throughput scaling up to 516K log records written per second (55 MB/sec), and 17,968 transactional commits per second. The commit latency stays stable going from 1.84 ms at 1 node to 2.01 ms at 16 nodes. At 16 nodes, the system was able to process 677K RPC/s.

Figure 8 compares Coordinator and Executor CPU utilization during scale-out. Up to 5 nodes, the Coordinator handles both write coordination and SQL processing, with slightly higher CPU usage than

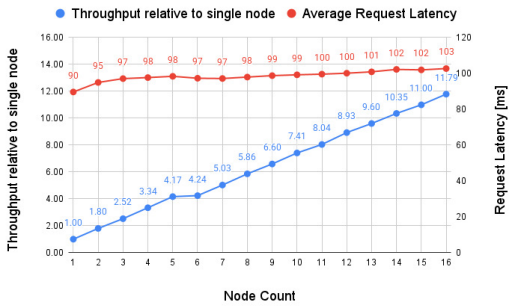


**Figure 7: SalesforceDB Scale-out**

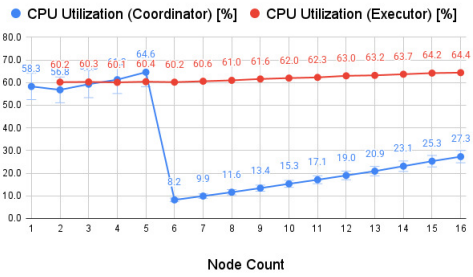


**Figure 8: CPU Utilization — Coordinator vs. Executor**

Executors. At 6+ nodes, the Coordinator handles only write coordination, reducing its CPU utilization and enabling further scale-out. As a result, 6-node and 5-node clusters achieve similar throughput.

Request latency rises slightly when moving from 1 node (RPC-less) to 2 nodes, where half of all requests are executed on Executors and may RPC. This trend continues until 6+ nodes, where all requests execute on Executors and involve potential RPCs. Figure 7 illustrates that the incurred RPC latency overhead is small, resulting in only a 15% increased latency when comparing between 1 and 16 nodes.

For the given workload, as well as for typical production traffic, 16 nodes is far from the scalability limit of the architecture. With Coordinator CPU utilization being less than half of Executor's, and RPC and write throughput far below their limits, further scaling is achievable.

To test potential scaling limits under high skew, we introduced a ninth business transaction that generates many writes to the same key. This causes expensive record chain traversal in the Coordinator's Memstore, increasing CPU usage for RPC requests. While throughput scaled 12.63× from 1 to 16 nodes, the Coordinator's CPU utilization exceeded 92% at 16 nodes, limiting further scaling. Bloom filters eliminate most of the probe RPCs but the high-skew increased the CPU footprint needed to serve RPC requests on the Coordinator, resulting in a 48% increase in average request latency.

## 7.5 Scalability and Coordination in Practice

The numbers below are collected from production and illustrate that the scale-out behavior in practice matches experimental results in Figure 7, and SalesforceDB can scale-out with minimal coordination.

**Scalability Observations.** SalesforceDB instances are scaled in and out based on customer traffic patterns to support varying degrees of load. For example, a large customer's instance was scaled from 5 to 9 to 16 nodes to accommodate the customer's holiday traffic. Normalized to 30% CPU utilization, this instance served 515K query executions per second on 5 nodes, 874K query executions per second on 9 nodes, and 1,519K query executions per second on 16 nodes, illustrating the near-linear scaling of the database architecture.

**Effectiveness of Coordination Minimization Mechanisms.** On a 16 node cluster, only 0.88% of total fetches and 3.23% of all scans issued RPCs, illustrating the efficiency of the coordination minimization mechanisms. Memstore Bloom filters prevented RPCs for 99.22% of fetches and 71.06% of scans checking the filter. Although the Bloom filters were less effective for scans, other mechanisms (skipping RPC if local Memstore is caught up, piggybacking Memstore scan of duplicate unique index checks on inserts and ignoring Memstore for sampling queries) kept the overall RPC percentage low.

**Stalls for Guaranteeing Consistency.** Monitoring a 16 node cluster showed that only 0.67% of commits had to stall, averaging 560 $\mu s$ (p95: 990 $\mu s$). These numbers confirm that SalesforceDB's external consistency mechanisms add minimal performance overhead.

# 8 RELATED WORK

SalesforceDB employs a decoupled compute and storage architecture similar to OLTP databases such as Amazon Aurora [62], Google Spanner [27], PolarDB [45], Microsoft Socrates [16], and Neon [70], as well as OLAP databases such as Redshift [17] and Snowflake [29].

Storage systems like GFS [40], Colossus [6], HDFS [56], and Ceph [5] support replicated durable storage, but their primary replica or chain-replication based mechanisms optimize for throughput and block I/O. They lack native support for replicated log abstraction or quorum commits, and are less suitable for supporting low tail-latency demands of an OLTP database like SalesforceDB.

Many databases like Spanner [19, 27], CockroachDB [59] and others[10, 11, 69] scale database compute using range-partitioning to split data across multiple nodes. Transactions spanning multiple partitions have to perform a two-phase commit (2PC) across nodes. This approach doesn't work well for non-partitionable workloads. In Aurora [62, 63] and PolarDB-SCC [68] the database scales reads via read-only replicas and by offloading storage materialization to storage nodes, but all write requests must be serviced by a single node. PolarDB-MP [67] supports multiple writer nodes by using local buffer pools coordinated through a distributed buffer pool stored in Disaggregated Shared Memory over RDMA. SalesforceDB requires multiple nodes to support its largest tenants, and builds on its shared LSM storage to scale out. It does not require workload locality or complex coordination protocols that transfer key ownerships [30, 46] or page ownerships [20, 35, 61, 67]. It also runs on commodity infrastructure rather than specialized hardware solutions [36, 65, 66]. The limited set of coordination metadata allows SalesforceDB to add or remove nodes in seconds without any dynamic remastering of data page ownership [61] across the cluster.

Like Spanner, SalesforceDB is designed to be externally consistent. Rather than using highly coordinated clocks, SalesforceDB uses a single timestamp oracle and hybrid logical clocks as the foundation for maintaining external consistency. Timestamp epochs are broadcast across the cluster at 1 ms intervals. Spanner instead relies on time services with tight clock skew bounds. SalesforceDB may adopt cloud providers' time services as they become more widely available.

LSMs [51] are the bedrock of various key-value stores (AsterixDB [13, 14], LevelDB [7] and others [24, 52]) and relational databases (MyRocks [48], CockroachDB [59], OceanBase [69], YugabyteDB [11],

TiDB [10]). Unlike SalesforceDB, these systems employ an LSM per table [14] or per column family [8] or per partition [69]. A single tenant-clustered LSM supports a large number of tables and tenants, without the metadata overhead of managing numerous files.

SalesforceDB builds multi-tenancy into the database, defines a tenant lifecycle and uses a shared-table model [25]. Relational Cloud [28] and Elastras [31] employ the shared-process model with multiple tenants sharing the database server. CockroachDB Serverless [58] disaggregates its database node and employs a per tenant SQL layer and shared-process kv-layer with tenant-id key prefixes, but tenants don't share the same schema. SalesforceDB's shared-table multi-tenancy limits the per-tenant overhead and allows for maximum resource pooling as compared to other models, but offers a lower degree of performance isolation [49]. Approaches like SQLVM [50] use per-tenant quotas for CPU, buffer pool etc. In SalesforceDB, tenant performance isolation is done at the application layer using the database's observability signals. Models like Pythia [39] efficiently characterize tenant-behavior based on tenant size, transaction throughput etc., for guiding how to co-locate tenants. In SalesforceDB, the application places tenants on database instances based on historical CPU consumption and other characteristics. Apple's FoundationDB Record Layer [26] also clusters data by tenant, but it does so through logical tenant-scoped subspaces above the storage layer, rather than embedding multi-tenancy into the storage engine like SalesforceDB.

Snowflake [29] provides zero-copy cloning of tables and databases by manipulating metadata over immutable shared storage. In addition to metadata manipulation, SalesforceDB virtualizes tenant identity within the LSM layer. Tenant-clustered storage, virtual file references, and runtime tenant translation enable tenant cloning and migration across database instances that share a common schema.

Live migration approaches [32, 37, 38] move partitions or hot records within a database cluster. Like SalesforceDB, Albatross [32] leverages shared storage for tenant migration, but only across partitions within a cluster. SalesforceDB, however, uses its fleetwide schema uniformity to move tenants across different databases.

Unlike other tenant-level encryption approaches (column encryption [15] or per-tenant database [2]), SalesforceDB leverages its LSM's tenant co-location and compaction to enable tenant encryption and key rotation within its shared-table multi-tenancy model. Shield [60] also encrypts LSM files at flush and compaction boundaries, but it doesn't support per-tenant encryption keys.

## 9 LESSONS LEARNED

**Immutability changes everything**. The choice of immutable LSM-based storage was one of the most consequential architectural decisions in SalesforceDB's design. While it introduced well-known read-amplification challenges that required sustained engineering investment [18], it eliminated an entire class of corruptions that plague mutable engines. Beyond correctness, immutability created compounding leverage across the stack in ways not fully anticipated at the outset. It fits naturally with cloud object storage's write-once semantics. It makes caching trivially correct, since cached objects are never invalidated by writes. It allows multiple database nodes and instances to share storage without coordination, which is the foundation for both our scale-out and tenant-management capabilities. Expensive derived operations, such as building Bloom and range filters, calculating integrity checksums, and applying fragment-level encryption, are computed once during asynchronous flush and compaction and reused indefinitely. The LSM Virtualization layer would not have been tractable on a mutable storage engine: virtual file references are only meaningful because the files they point to never change. The lesson is that immutability is not just a storage optimization, it is a design principle that benefits the entire system.

**Filtering is a cross-cutting architectural pattern**. A recurring theme in SalesforceDB's evolution is the use of cheap filters to avoid expensive operations. At the storage layer, per-extent Bloom filters and range filters reduce unnecessary work across LSM levels, mitigating LSM read-amplification as described in [18]. Executor nodes apply the same principle to avoid RPCs: per-node Bloom filters over Memstore state allow an Executor to determine locally whether recently committed data is relevant to a query, eliminating inter-node coordination in the common case.

**Relational execution has substantial scalable compute**. A key factor in this architecture's scalability is how little work the Coordinator performs on behalf of an Executor: even a simple write-heavy DML statement is mostly CPU-bound on the Executor. We were concerned that the single Coordinator would become a scalability bottleneck, but this asymmetric architecture has scaled well in production. That said, the current architecture does have scaling limits with a single Coordinator; we are addressing this in the next revision of our architecture by disaggregating Coordinator responsibilities across independently scalable tiers.

## 10 CONCLUSION

SalesforceDB demonstrates a new approach to designing cloud-native databases for large-scale multi-tenancy and scale-out. SalesforceDB employs tenancy as a first-class database construct and embeds it within the LSM-based storage engine. Together with compute-storage separation and immutable storage, this architecture enables efficient and novel tenant-level capabilities. SalesforceDB is built upon SDBStore, a distributed immutable storage system that supports multiple databases while providing high availability and tolerance to zone failures. The key architectural innovations of SalesforceDB are:

- LSM-native shared-table multi-tenancy: By using a galactic keyspace to cluster data by tenant within a single LSM, SalesforceDB maximizes resource pooling while enabling fragment-level, tenant-specific encryption.
- LSM Virtualization: By introducing virtual file references to decouple physical persistence from logical data management, SalesforceDB transforms traditionally heavy, row-by-row copy operations — like tenant migration and sandbox cloning across database instances — via lightweight metadata-only operations.
- Scale-out with limited coordination: SalesforceDB exploits the immutable shared LSM and coordination-minimization techniques such as Memstore Bloom filters to scale horizontally, allowing every node to serve reads and writes for any record. This eliminates the need for partitioning and avoids the coordination overhead typical of traditional page-oriented engines.

Results from thousands of production instances and microbenchmarks show that SalesforceDB is operationally resilient, scales effectively, and enables efficient tenant-level capabilities for the Salesforce application. SalesforceDB runs the majority of Salesforce CRM customers and all of the largest ones.

## REFERENCES


[1] [n.d.]. Apache Bookkeeper. https://bookkeeper.apache.org/.
[2] [n.d.]. AWS Aurora Encryption. https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Overview.Encryption.html.
[3] [n.d.]. AWS EBS. https://aws.amazon.com/ebs/.
[4] [n.d.]. AWS S3. https://aws.amazon.com/pm/serv-s3/.
[5] [n.d.]. Ceph. https://ceph.io/.
[6] [n.d.]. Colossus. https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system.
[7] [n.d.]. LevelDB. https://github.com/google/leveldb.
[8] [n.d.]. RocksDB. https://github.com/facebook/rocksdb.
[9] [n.d.]. The Salesforce Platform. https://architect.salesforce.com/fundamentals/platform-transformation/.
[10] [n.d.]. TiDB. https://www.pingcap.com/.
[11] [n.d.]. YugabyteDB. https://yugabyte.com/.
[12] Atul Adya, Barbara Liskov, and Patrick O'Neil. 2000. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*. IEEE, 67–78.
[13] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, et al. 2014. AsterixDB: A scalable, open source BDMS. *arXiv preprint arXiv:1407.0454* (2014).
[14] Sattam Alsubaiee, Alexander Behm, Vinayak R Borkar, Zachary Heilbron, Young-Seok Kim, Michael J Carey, Markus Dreseler, and Chen Li. 2014. Storage Management in AsterixDB. *Proc. VLDB Endow.* 7, 10 (2014), 841–852.
[15] Panagiotis Antonopoulos, Arvind Arasu, Kunal D Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, et al. 2020. Azure SQL database always encrypted. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 1511–1525.
[16] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, et al. 2019. Socrates: The new sql server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data*. 1743–1756.
[17] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. 2022. Amazon Redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*. 2205–2217.
[18] Vaibhav Arora, Subho Chatterjee, Terry Chong, Thomas Fanghaenel, Pat Helland, Jamie Martin, Kaushal Mittal, and Nat Wyatt. 2026. A Multi-tenant Relational OLTP Database at Salesforce. *Conference on Innovative Data Systems Research (CIDR)* (2026).
[19] David F Bacon, Nathan Bales, Nico Bruno, Brian F Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, et al. 2017. Spanner: Becoming a SQL system. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 331–343.
[20] Vlad Barshai, Yvonne Chan, Hua Lu, Satpal Sohal, et al. 2012. *Delivering continuity and extreme capacity with the IBM DB2 pureScale feature*. IBM Redbooks.
[21] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record* 24, 2 (1995), 1–10.
[22] Philip A Bernstein and Nathan Goodman. 1983. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)* 8, 4 (1983), 465–483.
[23] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.
[24] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.
[25] Frederick Chong, Gianpaolo Carraro, and Roger Wolter. 2006. Multi-tenant data architecture. *MSDN Library, Microsoft Corporation* (2006), 14–30.
[26] Christos Chrysafis, Ben Collins, Scott Dugas, Jay Dunkelberger, Moussa Ehsan, Scott Gray, Alec Grieser, Ori Herrnstadt, Kfir Lev-Ari, Tao Lin, et al. 2019. Foundationdb record layer: A multi-tenant structured datastore. In *Proceedings of the 2019 International Conference on Management of Data*. 1787–1802.
[27] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.
[28] Carlo Curino, Evan PC Jones, Raluca Ada Popa, Nirmesh Malviya, Eugene Wu, Sam Madden, Hari Balakrishnan, and Nickolai Zeldovich. 2011. Relational cloud: A database-as-a-service for the cloud. (2011).
[29] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*. 215–226.
[30] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2010. G-store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*. 163–174.
[31] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2013. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Transactions on Database Systems (TODS)* 38, 1 (2013), 1–45.
[32] 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. *Proceedings of the VLDB Endowment* 4, 8 (2011), 494–505.
[33] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.
[34] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, et al. 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*. 1463–1478.
[35] Alex Depoutovitch, Chong Chen, Per-Ake Larson, Jack Ng, Shu Lin, Guanzhu Xiong, Paul Lee, Emad Boctor, Samiao Ren, Lengdong Wu, et al. 2023. Taurus MM: bringing multi-master to the cloud. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3488–3500.
[36] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th symposium on operating systems principles*. 54–70.
[37] Aaron J Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. 2015. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proceedings of the 2015 ACM sigmod international conference on management of data*. 299–313.
[38] Aaron J Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 301–312.
[39] Aaron J Elmore, Sudipto Das, Alexander Pucher, Divyakant Agrawal, Amr El Abbadi, and Xifeng Yan. 2013. Characterizing tenant behavior for placement and crisis mitigation in multitenant dbmss. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 517–528.
[40] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 29–43.
[41] Pat Helland. 2015. Immutability changes everything. *Commun. ACM* 59, 1 (2015), 64–70.
[42] Pat Helland. 2024. Scalable OLTP in the Cloud: What's the BIG DEAL?. In *CIDR*.
[43] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. 2010. {ZooKeeper}: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*.
[44] Flavio P Junqueira, Ivan Kelly, and Benjamin Reed. 2013. Durability with bookkeeper. *ACM SIGOPS operating systems review* 47, 1 (2013), 9–15.
[45] Feifei Li. 2019. Cloud-native database systems at Alibaba: Opportunities and challenges. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2263–2272.
[46] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. 2016. Towards a non-2pc transaction management in distributed database systems. In *Proceedings of the 2016 International Conference on Management of Data*. 1659–1674.
[47] Chen Luo and Michael J Carey. 2020. LSM-based storage techniques: a survey: C. Luo, MJ Carey. *The VLDB Journal* 29, 1 (2020), 393–418.
[48] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-tree database storage engine serving Facebook's social graph. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3217–3230.
[49] Vivek Narasayya and Surajit Chaudhuri. 2022. Multi-tenant cloud data services: State-of-the-art, challenges and opportunities. In *Proceedings of the 2022 International Conference on Management of Data*. 2465–2473.
[50] Vivek Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri. 2013. Sqlvm: Performance isolation in multi-tenant relational database-as-a-service. In *CIDR 2013*.
[51] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta informatica* 33, 4 (1996), 351–385.
[52] Nitin Padalia. 2015. *Apache Cassandra Essentials*. Packt Publishing Ltd.
[53] Mohiuddin Abdul Qader, Shiwen Cheng, and Vagelis Hristidis. 2018. A comparative study of secondary indexing techniques in LSM-based NoSQL databases. In *Proceedings of the 2018 International Conference on Management of Data*. 551–566.
[54] Subhadeep Sarkar, Niv Dayan, and Manos Athanassoulis. 2023. The LSM design space and its read optimizations. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 3578–3584.
[55] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2022. Constructing and analyzing the LSM compaction design space (updated version). *arXiv preprint arXiv:2202.04522* (2022).
[56] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 1–10.

[57] Michael Stonebraker and Lawrence A Rowe. 1986. The design of Postgres. *ACM Sigmod Record* 15, 2 (1986), 340–355.

[58] Jeff Swenson, Andy Kimball, Raphael'kena' Poss, Rebecca Taft, Jay Lim, Adam Storm, Sumeer Bhola, Paul Bulkley-Logston, Pj Tatlow, Rachael Harding, et al. 2025. CockroachDB Serverless: Sub-second Scaling from Zero with Multi-region Cluster Virtualization. In *Companion of the 2025 International Conference on Management of Data*. 648–661.

[59] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 1493–1509.

[60] Viraj Thakkar, Dongha Kim, Yingchun Lai, Hokeun Kim, and Zhichao Cao. 2025. SHIELD: Encrypting Persistent Data of LSM-KVS from Monolithic to Disaggregated Storage. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–28.

[61] Murali Vallath. 2004. *Oracle real application clusters*. Digital Press.

[62] 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*. 1041–1052.

[63] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvilli, et al. 2018. Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes. In *Proceedings of the 2018 International Conference on Management of Data*. 789–796.

[64] Craig D Weissman and Steve Bobrowski. 2009. The design of the force. com multitenant internet application development platform. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 889–896.

[65] Xinjun Yang, Feifei Li, Yingqiang Zhang, Hao Chen, Qingda Hu, Panfeng Zhou, Qiang Zhang, Shuai Li, Zongzhi Chen, Zheyu Miao, et al. 2025. From Scale-Up to Scale-Out: PolarDB's Journey to Achieving 2 Billion tpmC. *Proceedings of the VLDB Endowment* 18, 12 (2025), 5059–5072.

[66] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Gerry Fan, Yang Kong, Bo Wang, Jing Fang, Yuhui Wang, Tao Huang, et al. 2025. Unlocking the potential of cxl for disaggregated memory in cloud-native databases. In *Companion of the 2025 International Conference on Management of Data*. 689–702.

[67] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Bo Wang, Jing Fang, Chuan Sun, and Yuhui Wang. 2024. PolarDB-MP: A Multi-Primary Cloud-Native Database via Disaggregated Shared Memory. In *Companion of the 2024 International Conference on Management of Data (SIGMOD)*. ACM, 295–308. https://doi.org/10.1145/3626246.3653377

[68] 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

[69] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, et al. 2022. OceanBase: a 707 million tpmC distributed relational database system. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3385–3397.

[70] Matei Zaharia. 2025. Bringing the Operational and Analytical Worlds Together with Lakebase. *Proceedings of the VLDB Endowment* 18, 12 (2025), 5539–5539.