Aurora PostgreSQL Limitless Database: Building a Highly Scalable OLTP Database

Dmitry Arkhangelskiy*
Saikiran Avula
Sachit Batra
Amazon Web Services
Seattle, USA

Jin Chen
Radwan Deeb
Amazon Web Services
Seattle, USA

Alexey Gotsman
Amazon Web Services
Seattle, USA
IMDEA Software Institute
Madrid, Spain

Upendra Gowda
Haritabh Gupta†
Benoit Hudzia
Rishabh Jain
Kaumudi Kaushik
Amazon Web Services
Seattle, USA

Aravind Kumar Kumar
Sergey Melnik†
Saleem Mohideen
Sharique Muhammed
Davor Prugovecki
Amazon Web Services
Seattle, USA

Sanjay Shanthakumar
Sagar Shedge
Anand Kumar Thakur
David Wein†
Amazon Web Services
Seattle, USA

Abstract

We present Aurora Limitless Database, a cloud-native distributed database system that extends Amazon Aurora PostgreSQL with horizontal scaling capabilities while maintaining strong consistency guarantees. The system provides transparent scalability using a router layer for query distribution and a storage layer of PostgreSQL shards, which eliminates the need for application-level sharding. Our key technical contributions include a distributed transaction protocol that integrates time-based multi-version concurrency control with two-phase commit, an adaptive scaling framework that combines vertical and horizontal scaling, and a distributed query processing engine that maintains strong consistency across both DML and DDL operations.

Aurora Limitless Database achieves near-linear scalability for a variety of production workloads, reaching millions of transactions per second while maintaining millisecond-level latencies. The system automatically optimizes resource allocation through serverless capabilities and dynamic sharding, significantly reducing operational costs compared to static provisioning. This approach enables scalable OLTP applications to leverage PostgreSQL's rich feature set without sacrificing consistency and without the administrative burden of provisioning and managing multiple systems.

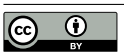
CCS Concepts

• **Information systems → Data management systems**.

Keywords

Sharding, Two-phase commit, Scalability, Snapshot isolation

*All authors are ordered alphabetically.
†Work done while at Amazon Web Services.



SIGMOD Companion '26, Bengaluru, India

ACM ISBN 979-8-4007-2450-3/2026/05
https://doi.org/10.1145/3788853.3803089

ACM Reference Format:
Dmitry Arkhangelskiy, Saikiran Avula, Sachit Batra, Jin Chen, Radwan Deeb, Alexey Gotsman, Upendra Gowda, Haritabh Gupta, Benoit Hudzia, Rishabh Jain, Kaumudi Kaushik, Aravind Kumar Kumar, Sergey Melnik, Saleem Mohideen, Sharique Muhammed, Davor Prugovecki, Sanjay Shanthakumar, Sagar Shedge, Anand Kumar Thakur, and David Wein. 2026. Aurora PostgreSQL Limitless Database: Building a Highly Scalable OLTP Database. In *Companion of the International Conference on Management of Data (SIGMOD Companion '26), May 31-June 05, 2026, Bengaluru, India.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3788853.3803089

1 Introduction

Modern cloud-native applications demand databases that can seamlessly scale to handle millions of transactions per second and petabytes of data without requiring complex application-level sharding or infrastructure management. However, customers operating high-throughput workloads — from financial services platforms processing real-time transactions to SaaS applications — face a fundamental constraint: traditional single-primary database architectures eventually hit hard limits on write throughput, storage capacity, and concurrent connections.

To address this problem, we present Aurora Limitless Database, a cloud-native distributed database system that extends Amazon Aurora PostgreSQL [36, 37] with horizontal scaling capabilities. The system achieves write scaling via data partitioning while maintaining strong consistency guarantees – Read Committed and Repeatable Read isolation levels extended with external consistency [10]. It furthermore leverages Aurora's existing disaggregated compute and distributed storage to provide durability, region-wide availability, I/O throughput scaling, and elastic disk capacity.

Aurora Limitless is designed to achieve several key objectives: (1) write scaling without reliance on a centralized transaction manager; (2) preservation of PostgreSQL transaction semantics even for multi-shard transactions; (3) single-system-like performance for well-sharded access patterns; (4) seamless horizontal scalability; and (5) single-system-like management experience. Aurora Limitless achieves these goals using the following techniques:

- The system provides a data model that allows customers to specify which tables need to be sharded and how (§2). It horizontally partitions the data in sharded tables using hash-based distribution on the columns chosen by the customer.
- Aurora Limitless separates *shards*, which store the data partitions and process queries on them, from *routers*, which serve the application traffic and orchestrate query processing across the different shards (§3). Both of these components leverage the Aurora storage system, which guarantees a high degree of availability and durability by replicating the data across 3 Availability Zones (AZs) in a single AWS region [36, 37].
- The router and shard fleets can be scaled independently to adapt to changing workload conditions. They can furthermore scale both vertically and horizontally (§4). Shards and routers scale vertically by adjusting their capacity via Aurora Serverless V2 [7]. Shards also scale horizontally by partitioning the data across more instances, while routers scale by adding extra instances to the router fleet.
- Aurora Limitless supports ACID distributed transactions that involve multiple shards yet conform to the semantics of PostgreSQL isolation levels (§5). The system achieves this using a multi-version concurrency-control algorithm that computes transaction snapshot and commit timestamps through the Amazon Time Sync service [21]. The service provides bounded clock uncertainty using a fleet of redundant satellite-connected atomic clocks. It thereby enables Aurora Limitless to maintain globally consistent snapshots across shards and to also guarantee external consistency [10], which prohibits reading stale data.

  To implement the new concurrency-control algorithm, we had to make major changes to the PostgreSQL transaction processing: in particular, we replaced traditional snapshots based on sets of transaction identifiers by scalar timestamps. We have also implemented a non-blocking variant of the two-phase commit (2PC) protocol for distributed transactions that is tightly integrated with time-based concurrency control.
- Aurora Limitless uses the timestamp-based nature of its transaction processing to allow customers to take consistent snapshots of the whole cluster at a particular time point (§6). The customers can then restore the database from the backups they created.
- Routers include a query processor that optimizes the evaluation of complex cross-shard queries by pushing down the computation closer to the data at shards (§7). This heavily builds on the existing PostgreSQL query planner while modifying it to be aware of data distribution.

Aurora Limitless has been in production at AWS for over a year, with customers using clusters that contain up to 64 shards and 32 routers. We illustrate its scaling properties using HammerDB [23], an open-source benchmark derived from TPC-C (§8). We find that, when vertical scaling reaches its limits, horizontal scaling can improve performance by distributing the workload across additional shards.

2 Data Model

Aurora Limitless supports three table types, each optimized for different access patterns – sharded, reference and standard. To explain them, we use a toy schema in Figure 1 with three tables – `customers`, `orders` and `tax_rates`. To keep data definitions compatible with PostgreSQL, Aurora Limitless leaves the `CREATE TABLE` syntax unchanged; the application indicates the type of each table using a session parameter `limitless_create_table_mode`.

```
SET rds_aurora.limitless_create_table_mode = 'sharded';
SET rds_aurora.limitless_create_table_shard_key = '{"cust_id"}';
CREATE TABLE customers (
  cust_id INT PRIMARY KEY NOT NULL,
  name TEXT,
  email VARCHAR(100) );

SET rds_aurora.limitless_create_table_collocate_with = 'customers';
CREATE TABLE orders (
  order_id INT NOT NULL,
  cust_id INT NOT NULL,
  amount DECIMAL(10,2) NOT NULL,
  tax_rate_id INT,
  PRIMARY KEY (order_id, cust_id) );

SET rds_aurora.limitless_create_table_mode = 'reference';
CREATE TABLE tax_rates (
  tax_rate_id INT PRIMARY KEY NOT NULL,
  state TEXT,
  country TEXT NOT NULL,
  tax_rate DECIMAL(5,4) NOT NULL );

SET rds_aurora.limitless_create_table_mode = 'standard';
...
```

Figure 1: An example schema for different table types.

Sharded tables are horizontally partitioned across shards using hash-based distribution on given *shard keys* – fields specified using a session parameter `limitless_create_table_shard_key`. For example, the `orders` table is sharded on the `cust_id` field. Tables that share the same shard key can be *co-located*, so that all the data for a given shard-key value resides on the same shard. This allows optimizing joins and is specified using a session parameter `limitless_create_table_collocate_with`. In our example, the `orders` table is co-located with the `customers` table, so that all the data for a given customer from both tables is located together.

Reference tables are replicated in full on every shard. They are useful for tables that need to be joined frequently with sharded tables but that don't lend itself naturally to sharding. In our example, `tax_rates` is declared as a reference table: this is natural, since it's relatively small and is modified relatively infrequently compared to the other tables. Whenever we join it with other tables, we are more likely to get a query that can be executed at a single shard.

Standard tables are located on a single shard, specified as part of the configuration. They are useful for data that does not need to be sharded, and also allow customers to easily import data from traditional PostgreSQL: the data can be imported into standard tables, which can later be converted to sharded or reference tables as needs evolve. Queries can freely join standard tables with sharded or reference ones.

3 System Architecture

Aurora Limitless Database implements a multi-layered architecture that separates control operations from data processing. Building upon Aurora's distributed storage architecture [36, 37] and serverless management [7], the system extends these foundations with

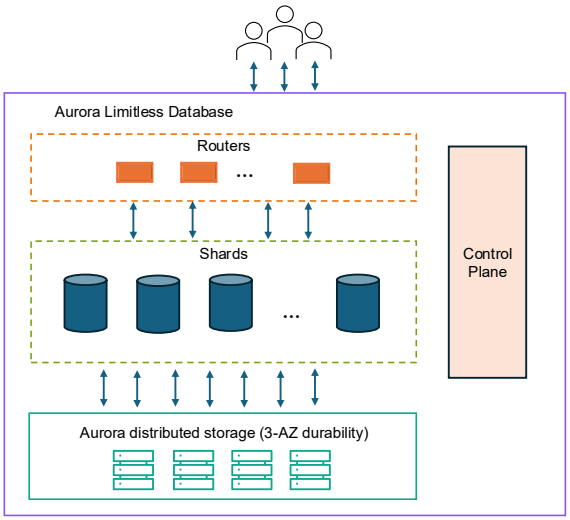


Figure 2: Aurora Limitless database architecture.

distributed processing capabilities while maintaining their high availability characteristics. Figure 2 illustrates the key components.

3.1 Aurora Distributed Storage

Aurora distributed storage [36, 37] achieves a high degree of availability and durability by replicating the data across 3 Availability Zones (AZs) in a single AWS region. The system splits a database volume into smaller segments and replicates each write to a segment to 6 storage nodes (2 per AZ). This ensures that the data stays durable even in case of a simultaneous failure of an entire AZ and one replica of the data in another AZ. The storage system also stays available for writes even in case of an entire AZ failure. The system automatically detects segment failures and repairs them using the data in healthy segments. It also dynamically adjusts to the amount of data that needs to be stored. The storage system enables Aurora Limitless to manage petabytes of data spread across multiple shards.

3.2 Data Plane

The data plane of Aurora Limitless consists of two independently scalable components: a *router* fleet and a *shard* fleet, each serving distinct roles in query processing (Figure 2). Each router or shard is implemented as a PostgreSQL cluster with its own storage volume, managed by the Aurora distributed storage. The set of routers and shards in a single Aurora Limitless database is called a *shard group*.

Shards own portions of the actual data, which they store in their associated Aurora storage volumes. Data in sharded tables is distributed across shards using hashing on shard keys. Shards execute queries on their portions of data given plan fragments received from routers, described next.

Routers serve all application traffic. They do not store the actual data, but maintain metadata including cluster topology (the set of active routers, shards and their health status), database schema definitions, and data placement mappings (which shard-key ranges reside on which shards). Routers serve as the authoritative source for schema definitions and data placement mappings, while cluster topology information is synchronized from the control plane's metadata service (§3.3).

Routers receive queries from client applications, plan their execution and send them to the shards. They are also responsible for setting transaction snapshots and driving the commit protocol, as we describe in §5. Client connections are distributed across the router fleet using DNS-based load balancing, with each connection establishing a session to a specific router. Once established, a connection remains bound to its router for the session's lifetime.

Each pair of a router and a shard establishes multiple connections. The router's connection manager then multiplexes client sessions onto the shard connections, dynamically allocating capacity based on the workload. The multiplexing is done at transaction granularity: once a transaction finishes, its connections to the shards can be reused by other transactions executing at the router. Session state (authentication, roles, variables) is preserved across transaction and shard boundaries through session-context passing.

3.3 Control Plane

The control plane orchestrates cluster operations through several components:

- *Cluster-management service* manages the life cycle of shards and routers. It maintains the authoritative registry of the shard-group topology, propagates changes in the topology to shard-group members, and manages the workflows for provisioning and deprovisioning of nodes. The latter builds heavily on the Aurora Serverless V2 infrastructure [1], as we detail in §4.
- *Monitoring service* includes a local agent at each node to monitor the local health status and trigger a failover if needed, and a cross-node monitoring service to detect network partitions. The service stores all health metrics in Amazon CloudWatch.
- *Heat-management service* for automated scaling is covered in §4.
- *Backup-and-restore service* is covered in §6.

3.4 High Availability

As we explained in §3.1, the Aurora storage system replicates data across 3 AZs. Aurora Limitless also spreads routers and shards across the same 3 AZs, which ensures that an AZ failure will not make the whole system unavailable. Router and shard failures are then handled as follows:

- ***Routers.*** Routers are identical and accessed via DNS resolution of a single endpoint. Thus, if one router is unavailable, other routers can handle the workload through DNS resolution while the faulty router recovers or a new router is added to the fleet. Thus, routers don't have dedicated standbys, which saves costs.
- ***Shards.*** If a shard fails, its data becomes unavailable until the host is re-provisioned. Hence, shards can have standbys in different AZs (from 0 to 2, as controlled by the customer). If a shard fails, a standby takes over its storage volume and workload.

This approach achieves high availability of Aurora Limitless even during component failures. The system also includes a backup-and-restore service, which we describe in §6.

Total nodes	Routers	Shards	Default min ACUs	Max ACU range
4	2	2	16	16–400
12	4	8	48	1101–1200
18	6	12	72	1701–1800
24	8	16	96	2301–6144

Table 1: Some of the mappings determining the initial number of nodes and the default minimum capacity from the maximum capacity.

4 Adaptive Scaling

Aurora Limitless implements two-dimensional automatic scaling. First, both routers and shards scale vertically by adjusting node capacity using Aurora Serverless V2 [1, 7]. Second, they also scale horizontally: routers by adding extra instances to the router fleet, and shards by partitioning the data across more instances.

During the creation of a shard group, customers can set its total minimum and maximum compute capacity. This is measured in *Aurora Capacity Units (ACUs)*: 1 ACU is a combination of approximately 2 GB of memory and the corresponding CPU and networking resources. The initial number of routers and shards is determined by the maximum capacity: the higher the capacity, the greater the numbers of routers and shards. Table 1 shows some of the mappings we use. For instance, if the customer sets the total maximum ACUs to 1200, the system will initially create 4 routers and 8 shards. Note that the total capacity set by the customer and the numbers of nodes in Table 1 exclude shard standbys that the customer may have configured (§3.4): their capacity always matches that of the corresponding primary shard, and is not counted towards the total capacity budget. The biggest shard group we have observed in production so far has 32 routers and 64 shards, while the most frequently used configuration has 4 routers and 8 shards.

4.1 Vertical Scaling

Each node (shard or router) has its own minimum and maximum capacity values, also measured in ACUs. Using Aurora Serverless V2, each node can scale up until reaching its maximum ACUs, and scale down until reaching its minimum ACUs. At the beginning, Aurora Limitless splits the ACU budget evenly among all members of the shard group. The system periodically adjusts new allocations in proportion to each node's actual consumed ACU usage, as this reflects their true resource demands. In a shard group with n nodes, the dynamically adjusted minimum and maximum ACUs for a node i are calculated as follows:

$$dynamicMinACU_i = shardGroupMinACU \cdot \frac{consumedACU_i}{\sum_{i=1}^{n} consumedACU_i}$$

$$dynamicMaxACU_i = shardGroupMaxACU \cdot \frac{consumedACU_i}{\sum_{i=1}^{n} consumedACU_i}$$

This proportional allocation strategy is based on the observation that nodes consuming more capacity usually demonstrate higher growth demand and need to scale faster.

Aurora Limitless manages shard and router scaling according to their distinct operational patterns. Shards process complex queries and, to this end, read a lot of data from the storage; this results in high compute and buffer-cache usage. In contrast, routers read most of the data from shards and may use a lot of memory to hold the data while processing a distributed query. Hence, the triggers for scaling shards pay more attention to their compute usage, whereas those for routers to their memory usage. We also use different memory allocation strategies when scaling shards and routers: shards get more buffer-cache memory, whereas routers get more heap memory.

Similarly to Aurora Serverless V2 [1], our system uses asymmetric scaling rates. Scale-up operations are triggered rapidly in response to load spikes to ensure responsive performance. Scale-down operations are more conservative, to prevent resource thrashing: they trigger after longer periods of low utilization.

4.2 Horizontal Scaling

Customers can also scale an Aurora Limitless database by splitting a shard in two or by adding a new router. The system provides functions to initiate these operations, which start an asynchronous workflow, and to check the completion status of the workflow.

Aurora Limitless provides customers with monitoring information about ACU and storage consumption of routers and shards, which the customers can use to decide when horizontal scaling is needed. Customers can also allow Aurora Limitless to do automatic shard splits: in this case, a heat-management service will monitor the ACU and storage consumption at shards and split a shard if its metrics exceed predefined thresholds even after vertical scaling.

Capacity management. When adding a new node, the system allocates its capacity budget from the capacity not consumed by the existing members of the shard group. If there is not enough spare capacity, the customer request is rejected. The system estimates the expected initial ACU consumption of the new node from the data about the existing ones. It then repartitions the maximum capacity allocations for shard-group members according to the formulas given in §4.1.

Table slices. The foundation for shard splits is a fine-grained partitioning abstraction called *table slices*. As we explained in §3.2, each shard owns a hash range of the shard key for each sharded table. Aurora Limitless further subdivides the range into table slices: typically 512 slices per table are distributed among the shards. A shard represents its portion of a sharded table as a PostgreSQL partitioned table, with partitions corresponding to slices. A slice is the minimum migration unit during shard splits. In the case of co-located tables (§2), corresponding slices migrate together, thus preserving join optimizations. Standard tables never split: they always stay at the same shard.

During customer-initiated shard splits, half of the slices on the source shard are migrated to a new shard. Thus, the hash ranges of sharded tables assigned to the source shard are split midway. With automatic shard splits, the point where the hash range is split is decided based on which slices are hot, using the information provided by the heat-monitoring service.

Shard-split workflow. The shard-split workflow has several phases.

Phase 1: Storage-level cloning. A straightforward approach to shard splits would read the data from the source shard and write it

to new shard. This, however, would place even more load on an already overloaded shard. Instead, we employ Aurora's copy-on-write cloning mechanism [4]. This creates a clone of the source shard volume, so that the data is only copied when modifications occur. By offloading the cloning work to the storage layer, Aurora Limitless can split a shard even if it is operating at maximum capacity.

Phase 2: Redo-log replay. Cloning a volume takes some time, during which further updates may have happened at the source shard. Thus, by the time the cloning completes, the created copy may already be out of date. To deal with this, the newly created shard starts to apply pending updates from the source shard's redo logs, to catch up with it.

Phase 3: Switchover. Once the new shard nearly catches up with the source shard through the redo-log replay, the system initiates a switchover process. The tables whose slices are being migrated are locked at all the routers and shards, blocking new writes to them. If the required locks cannot be immediately acquired because ongoing transactions already hold conflicting locks, these transactions are terminated and are forced to release their locks. The locking effectively freezes the state of the affected tables, so the source shard does not produce any new redo-log records for them. This allows the new shard to finally complete the redo-log replay.

After this, the routers update their slice-to-shard mappings, assigning the migrated slices to the new shard. The source and the new shard also update their schemas: the source shard unlinks the migrated slices (using the `DETACH PARTITION` operation of PostgreSQL); the new shard unlinks the slices cloned from the source shard that it will not host. The locks taken during the switchover are released, and all subsequent transactions targeting the migrated slices are automatically directed to the new shard.

Phase 4: Clean-up. After the switchover, both shards temporarily hold slices that are no longer under their ownership: the source shard holds the migrated slices, and the new shard, the slices assigned to the source shard. Both shards then perform background cleanup by deleting the slices that are no longer mapped to them.

Customer workloads run normally throughout most of the shard-split process. The only customer impact happens during the switchover phase, when DDL operations and updates to the slices being migrated are temporarily blocked, and some ongoing transactions may be terminated.

Router addition. The router addition workflow is similar to the shard split one. It clones an existing router into a new instance, adds it to the topology through the cluster-management service and finally registers it on the DNS. After this, customers can connect to the newly added router.

Horizontal vs vertical scaling. Vertical and horizontal scaling provide customers with two complementary scaling dimensions. By considering their workload characteristics, customers can combine these approaches to either maximize performance or optimize the performance-cost ratio. We experimentally evaluate the effectiveness of these scaling approaches in §8.

5 Concurrency Control and Commit Protocol

Aurora Limitless preserves PostgreSQL transaction semantics in a distributed database through careful integration of time-based concurrency control with Aurora storage. The system provides Repeatable Read (RR), which in PostgreSQL means snapshot isolation (SI), and Read Committed (RC), as these are the most commonly used isolation levels in PostgreSQL. We first describe our SI implementation for transactions with only DML statements, which is similar to Clock-SI [17]. We then describe how we handle DDL statements (§5.7) and Read Committed (§5.9).

5.1 Timestamps via Amazon Time Sync

Snapshot isolation requires each transaction to read data from a snapshot of the database plus its own writes [8]. To ensure this, PostgreSQL exploits that fact that each transaction modifying the database is assigned a unique identifier (*xid*), which is generated from a global counter the first time the transaction does a modification. To compute a snapshot for a transaction T, PostgreSQL records the lowest identifier of a currently running transaction (*xmin*), the lowest as-yet-unassigned identifier (*xmax*), and the identifiers of all currently running transactions between *xmin* and *xmax* (*xip_list*). Apart from its own writes, T sees only writes made by the transactions committed at the time its snapshot was taken: i.e., all committed transactions with $xid < xmin$, and those with $xid < xmax$ such that $xid \notin xip\_list$. The need to record all running transactions makes the snapshot computation CPU-intensive, so it is a well-known contention inducer on high-throughput PostgreSQL systems. It is also poorly suited to a distributed database, where a single node may not be aware of all running transactions.

Instead, Aurora Limitless defines snapshots by timestamps generated using the Amazon Time Sync Service [5]. This service provides bounded clock uncertainty using a fleet of redundant satellite-connected atomic clocks in each region. A function `now()` returns the current time t as well as a *clock error bound (CEB)*. The real time is guaranteed to be inside the interval $[earliest, latest]$, where $earliest = t - CEB$ and $latest = t + CEB$. A typical *CEB* is under 1 millisecond in all AWS regions; this has recently been reduced to low double-digit microseconds in some regions [21]. The above API is provided by an open-source ClockBound daemon, available to all AWS customers [3].

When a transaction executes its first query, its router assigns it a *snapshot timestamp startTs* as the upper bound of the current Time Sync interval, `now()`.*latest*. This timestamp is piggybacked on every query the router sends to a shard to determine the data it should read. When a transaction commits, it is assigned a *commit timestamp commitTs*. This timestamp is also computed using Time Sync as we describe in the following sections. Snapshot and commit timestamps determine transaction snapshots as per the following property, aligned with the SI specification [8].

Property 1. *Given two distinct transactions T and T', the modifications by T' are included into the snapshot seen by T if and only if $T'.commitTs \leq T.startTs$.*

In the following sections we explain how Aurora Limitless ensures this property.

5.2 Time-Based Multi-Versioning

PostgreSQL maintains multiple versions of each row in a table. Each version is associated with the identifiers of the transaction that created it (the version's *xmin*) and the transaction that created

the next version of the row (the version's *xmax*); the latter is invalid when there is no next version. PostgreSQL decides whether a version is visible to a transaction T by comparing the version's *xmin* and *xmax* to T's *xid*-based snapshot. In contrast, Aurora Limitless decides visibility using snapshot and commit timestamps: a committed version of a row with given *xmin* and *xmax* is visible to a transaction T if

$$xmin.commitTs \leq T.startTs < xmax.commitTs$$

(the last check is omitted if *xmax* is invalid). The above check straightforwardly ensures that, once a transaction T' commits, its changes become visible to any transaction T with $T.startTs \geq T'.commitTs$, as required by Property 1.

To implement the visibility check without changing the PostgreSQL row format, a shard maintains a separate mapping from transaction identifiers to commit timestamps. This mapping is integrated into an existing PostgreSQL *commit log* data structure that records which transactions have committed.

5.3 Write-Conflict Detection

Snapshot isolation requires that a transaction T can only commit if no other transaction wrote to the same rows as T while T was executing. Aurora Limitless ensures this in the same way as PostgreSQL: when T modifies a row, it acquires an exclusive lock on this row on the corresponding shard and holds it until commit, thus preventing concurrent modifications. After acquiring the lock, the transaction T also checks that all currently existing version of the row are still covered by its snapshot: if there is a newer version, the transaction aborts. If the check passes, T creates a new version of the row being modified at the corresponding shard.

5.4 Time-Aware Two-Phase Commit

Aurora Limitless commits a multi-shard transaction using a two-phase commit protocol (2PC), which ensures that all participants agree on the outcome of the transaction and, if it is committed, on its commit timestamp. High availability is a key consideration in this protocol, and achieving it requires some care. While the transaction router is a natural choice for the 2PC coordinator, to reduce costs, the router does not have dedicated standby nodes (§3.4). Hence, replacing the router in case of a host failure could take minutes, during which time the 2PC protocol would be blocked. To deal with this, in our 2PC protocol the state of the transaction is persisted not at the router, but at a designated *lead shard*. The lead shard can have standby nodes, in which case it can be replaced and its state recovered in seconds rather than minutes. The flow of the 2PC protocol in the absence of failures is illustrated in Figure 3:

- Upon receiving a request to commit a transaction, the router selects one of the shards updated by the transaction as the lead shard and sends a `PREPARE TRANSACTION` command with the lead shard's identifier to all other shards where the transaction made updates.
- Each shard computes its *prepare timestamp* for the transaction using `now()`.*latest* returned by Time Sync (the actual computation is slightly more subtle, and we describe it in §5.5). The shard then persists the prepare information and the lead shard's identifier in its storage volume, and sends the timestamp back to the router.
- Upon receiving all the replies, the router sends the maximum of the prepare timestamps to the lead shard.
- The lead shard calculates the commit timestamp of the transaction as the maximum of the timestamp received from the router and its own timestamp proposal (computed in the same way as a prepare timestamp). It then persists the commit timestamp, commits the transaction locally and sends an acknowledgment to the router.
- Upon receiving this, the router notifies the client and sends a `COMMIT PREPARED` command with the commit timestamp to the other participating shards, which also commit the transaction.

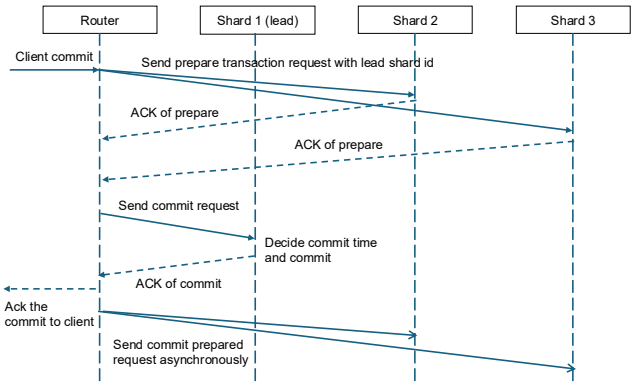


Figure 3: 2PC protocol with a lead shard.

If during the 2PC protocol some shard is unable to prepare the transaction, e.g., due to a failed write to storage, it notifies the router, which sends a `ROLLBACK PREPARED` command to all shards.

If the router fails during 2PC, the protocol leverages the lead shard to ensure a consistent resolution of in-progress transactions. This handles transactions differently depending on their progress through the commit protocol. Transactions that have not yet prepared at any shards are aborted, as no durable state has been recorded for them. For a transaction that has prepared but not committed at some shards, the lead shard's state becomes authoritative, with other shards querying the lead shard about the transaction's status. If the lead shard has already committed the transaction, it becomes committed at all shards; otherwise, it is aborted. In particular, the latter case will arise if the router failed before asking the lead shard to commit the transaction.

This protocol guarantees that, once a transaction's commit is acknowledged to the client, its effects remain durably visible regardless of subsequent failures. It also guarantees that all participants agree on the commit time of the transaction, which is no lower than the maximum of prepare timestamps.

Aurora Limitless avoids the 2PC protocol for read-only and single-shard transactions, which are dominant in production workloads. First, the router commits read-only transactions immediately, since they observe consistent data during their execution. Second, the router commits a transaction that updates a single shard by forwarding it to the shard the transaction updated. This shard then

manages the commit process locally, and in particular, picks a commit time for the transaction. This optimization is applicable even if the transaction read from multiple shards.

5.5 Reading Data from Shards

The distributed nature of Aurora Limitless makes it challenging to preserve consistency when reading data from shards. As we explained in §5.2, a transaction T reading data from a shard can determine which versions *currently committed* at the shard it should see by comparing the *xmin* and *xmax* of these versions to its snapshot timestamp. But as we now explain, to ensure Property 1 (§5.1), T also needs to take into account versions written by transactions that the shard may commit *in the future*.

Handling Clock Skews. Clock skew between routers and shards may lead to a situation where a transaction reading from a shard has a snapshot timestamp that is *in the future* with respect to the local time at the shard. For example, a transaction T with *startTs* of 100 may attempt to read data from a shard whose local Time Sync interval is $[20, 40]$. In this case T cannot just read the data currently committed at the shard, because after the read, the shard could commit a transaction T' with *commitTs* of 60. Then the read by T could miss the modifications by T' that should fall into its snapshot (Property 1).

Traditional approaches handle this situation by delaying the read by T until the shard's local time advances past its snapshot timestamp [17]. Aurora Limitless instead allows T to read the data immediately by using a hybrid logical clock (HLC) [20]. Each shard maintains a logical clock C, which is advanced to $\max\{C, T.startTs+1\}$ on each read by a transaction T. Whenever the shard is asked to prepare a distributed transaction T', it computes its prepare timestamp as $\max\{C, \texttt{now}().latest\}$; the shard computes the commit time for single-shard transactions in the same way. Recall that the commit time of a distributed transaction is computed as the maximum of its prepare timestamps at different shards (§5.4). Thus, once a transaction T reads from a shard, any transaction the shard commits after this will have a commit timestamp $> T.startTs$. This allows T to read the data from the shard without the extra delay while preserving consistency.

Handling Prepared Transactions. Another tricky situation is when a transaction T is trying to read a row at a shard S, and a distributed transaction T' updating this row has already prepared at the shard with a prepare timestamp $< T.startTs$. In this case the shard S does not know whether the final commit timestamp of T' will be below or above $T.startTs$. In the former case, allowing T to read the row could again ignore the modifications by T' that should fall into T's snapshot by Property 1. To resolve this situation, S sends an inquiry to the lead shard of T' about the status of this transaction, which includes the snapshot timestamp of T. If T' has already been committed, its lead shard responds with its commit timestamp, which allows S to determine whether T should see the modification by T'. Otherwise, the lead shard advances its logical clock C to $\max\{C, T.startTs+1\}$ and replies to S. This ensures that T' will get a commit timestamp higher than $T.startTs$, so that its modifications will be invisible to T.

Handling Committing Transactions. When a transaction is committing at its lead shard (if the transaction is distributed) or the shard it is pushed down to (if it is single-shard), there is a relative short period of time during which the transaction has already obtained its commit time but has not yet committed, e.g., because the shard is still flushing log records to storage. During this period, we mark the transaction as having a special `COMMITTING` status. A `COMMITTING` transaction may end up being aborted (e.g., if the write to storage fails), and thus its updates cannot be read safely. Hence, a transaction T reading a row at a shard must take care if the shard is `COMMITTING` another transaction T' that updates this row: if $T'.commitTs \le T.startTs$, then T must wait until T' commits or aborts before reading the row.

5.6 Preserving the Real-Time Order

Aurora Limitless implements a *strong* version of snapshot isolation that respects the real-time order between transactions [14]: if a transaction T_2 starts after a (non-read-only) transaction T_1 returns to the client, then $T_2.startTs > T_1.commitTs$. Thus, T_2 is guaranteed to see the modifications by T_1, regardless of the originating nodes of the two transactions[1]. This guarantee improves customer experience, since transactions always see up-to-date data.

Aurora Limitless implements the above guarantee using the Time Sync service. When committing a transaction T_1 that made modifications, once the lead shard determines the commit timestamp $T_1.commitTs$, it doesn't send an acknowledgment to the router immediately, but first waits until $\texttt{now}().earliest > T_1.commitTs$. This *commit wait* [11] preserves the real-time order. Indeed:

- Let t_1 be the *real* time when T_1 exits the commit wait. Since t_1 must fall inside the interval that Time Sync returns at this point, we have $t_1 \ge \texttt{now}().earliest > T_1.commitTs$.
- Let t_2 be the *real* time when T_2 computes its snapshot (§5.1). Since t_2 must fall inside the interval that Time Sync returns at this point, we have $T_2.startTs = \texttt{now}().latest \ge t_2$.
- If T_2 starts after T_1 finishes, then $t_2 > t_1$, so $T_2.startTs \ge t_2 > t_1 > T_1.commitTs$, as required.

To minimize the impact of the commit wait on latency for transactions with updates, the lead shard executes the commit wait in parallel with writing the commit log to storage. Since the storage write latency often exceeds the Time Sync error bound (typically under 1 ms), the wait rarely adds latency to the critical path.

5.7 DDL Statements

DDL operations in Aurora Limitless follow the PostgreSQL semantics; in particular, multiple DDL and DML operations can be packaged inside a single transaction to ensure the ACID properties.

The execution of most DDL operations and the commit of their enclosing transactions involves not just shards storing the relevant tables, but also all the routers. This ensures the atomic visibility of DDL statements across the entire cluster. In more detail, DDL operations in PostgreSQL often acquire heavyweight locks that prevent some of the DML operations from executing concurrently (e.g., `ALTER TABLE` would generally acquire an `ACCESS EXCLUSIVE`

[1] In fact, we guarantee an even stronger property that called *external consistency* [10]: if T_2 starts to commit after T_1 finishes, then $T_2.commitTs > T_1.commitTs$.

lock on the table being altered). In Aurora Limitless, such locks need to be acquired at all the shards storing the corresponding table as well as the routers. Then the commit of the enclosing transaction is done via a variant of the 2PC protocol from §5.4 that additionally involves all the routers. For example, the DDL operation `CREATE TABLE customers` from Figure 1 will create partitions of the table `customers` on the shards and will also add it to the schema at the routers. This change will be visible atomically to concurrently executing transactions.

DDL operations that create global sequences are implemented a special way. When a client submits a `CREATE SEQUENCE` statement, the corresponding router first picks one shard and asks it to create a global sequence object for persistence. The router then propagates to all other routers a command to initialize a new sequence, assigning disjoint sets of ranges to different routers. Each router then hands out sequence numbers from its pre-allocated range without contacting a shard, which improves performance. A router only contacts the dedicated shard storing the global sequence to get a new range when its current range is used up.

5.8 Distributed Deadlock Management

In Aurora Limitless, deadlocks can occur among multiple transactions distributed among different routers and shards. Customers can set a timeout after which a transaction waiting on a resource suspects a distributed deadlock. In this case, the system selects a router to gather waits-for graphs from the shards and to check for cycles in their union. Upon encountering a cycle, the router selects a random transaction as a victim and sends a command to the corresponding node to abort it.

As we explained in §5.7, DDL statements may need to acquire locks at multiple nodes. To prevent deadlocks among DDL statements due to different orders of contacting the nodes, such statements always first try to acquire the locks at a distinguished node. If this acquisition succeeds, requests to acquire the locks at the other nodes are then all sent in parallel; otherwise the transaction is aborted. This strategy avoids some of the deadlocks at the cost of adding an extra roundtrip during DDL statement execution.

5.9 Read Committed Implementation

The Read Committed implementation is similar to that of Repeatable Read, with the following differences. First, whereas under Repeatable Read all statements in a transaction use the same snapshot timestamp, under Read Committed every statement computes a new snapshot timestamp using $\texttt{now}().latest$ (§5.1). The reads by the statement then observe the database snapshot determined by the timestamp and the transaction's prior writes. Thus, as the transaction executes, its statements see increasingly up-to-date snapshots of the database.

Read Committed also turns off the write-conflict detection check (§5.3). When a transaction T modifies a row, it still acquires an exclusive lock on this row and holds it until commit. However, T does *not* check if another transaction modified the row between T starting and acquiring the lock. Whereas a Repeatable Read transaction aborts if there is such a modification, a Read Committed transaction proceeds as usual.

6 Failover, Backups and Recovery

Shard failover and read leases. Aurora storage accepts writes to a given volume from only one Aurora PostgreSQL instance at a time [37]. We exploit this property to avoid split-brain scenarios in case of shard failures: when a shard is suspected of a failure, only its replacement will be able to write to the corresponding volume. If the previous instance is actually alive (i.e., is a *zombie*), it will exit when its writes fail and it realizes that it has been replaced. Even with this protection, we still need to take care that a zombie shard does not serve stale reads: it will not be aware of the writes performed after the time when it has been replaced, and allowing it to serve stale data would violate external consistency (§5.6).

We use a form of leases to ensure that such problems do not happen. Every shard holds a lease, which is established and renewed upon a successful write to the storage. To this end, each write is tagged with the current clock value t at the shard. If the write is successful, it grants a lease to the shard until the time $t + TTL$, where TTL is a parameter determining the duration of the lease. This lease allows the shard to serve a read at snapshot times $\leq t + TTL$. If a shard cannot renew the lease after multiple attempts, the shard concludes that it is a zombie and terminates itself.

Upon a failover to a new instance, this instance first recovers the start time of the last granted lease t_{last} from the Aurora storage volume. It then waits until Time Sync returns $\texttt{now}().latest > t_{last} + TTL$, obtains a new lease, and only then starts processing transactions. In practice, we set the lease duration TTL less than the typical failover time. Thus, when new instance starts after a failover, in most cases the previous lease has already expired and no extra wait is needed.

The rationale for the above failover protocol is as follows. Recall that the shard computes its prepare timestamp for a transaction from $\texttt{now}().latest$, and the final commit timestamp is guaranteed to be no lower than this prepare timestamp. Hence, after the above wait is over, any transaction committed by the new shard instance will get a commit timestamp $> t_{last} + TTL$. This guarantees that any read that can be served by the zombie instance is valid: the read can only miss transactions that it is not supposed to see anyway.

Backup and recovery protocol. All Aurora Limitless data is stored in the Aurora storage system, which guarantees a high degree of durability (§3.1). Customers can additionally create backups for the whole shard group – including all shards and routers – and use them for recovering the database. The backups are integrated with the time-based commit protocol from §5.4: every backup is associated with a Time Sync timestamp t and reflects exactly the transactions with commit timestamps $\leq t$, regardless of the shards at which they were committed.

To implement this, every log record a node writes to Aurora storage is tagged by the same timestamp that the transaction commit protocol associates with the record; in particular, the commit record for a transaction is associated with the commit timestamp. When a backup with a timestamp t is created, it will only include the log entries with timestamps $\leq t$ in all storage volumes of the shard-group members. The recovery loads all the volumes from a backup and restarts the system from this state.

Note that, when the system creates a backup at a timestamp t, a distributed transaction with a commit timestamp $t' \leq t$ may be committed at some shards, but only prepared at others. During

recovery, the system will be restored to the same state. The prepared shards resolve this situation like in case of router failures, by querying the lead shard about the state of the transaction (§5.4).

7 Query Processing and Optimization

Table representation. Query execution is driven by routers, which construct query plans and orchestrate their execution at shards. As we explained in §2, Aurora Limitless offers three table types: sharded, reference, and standard. Routers internally represent sharded tables as PostgreSQL partitioned tables, with the partitioning mirroring that of the hash-space among the shards[2]. Aurora Limitless also builds on the PostgreSQL interface of a Foreign Data Wrapper (FDW), which allows the query planner and executor to access and query data stored in external sources as if they were local tables. Namely, a router represents each partition of a sharded table a PostgreSQL foreign table that references the actual portion of the table on the corresponding shard. The system implements a custom FDW that allows the router to access shard data.

Reference tables are replicated across all shards and are also represented as foreign tables. The implementation modifies the PostgreSQL query processing to respect their non-standard placement. Finally, standard tables are represented as foreign tables that point to a single shard. Aurora Limitless supports various index types on the tables, mirroring PostgreSQL capabilities. Representing Limitless tables using existing mechanisms understood by PostgreSQL – partitioned and foreign tables – makes it easier for the system to optimize query execution, as we describe next.

Executing multi-shard queries. Query execution in Aurora Limitless may in general require data from multiple shards. In such cases, the router executing the query generates a plan that tries to push compute closer to the data on the shards. To this end, it exploits the fact that sharded tables are represented as PostgreSQL partitioned tables, so the PostgreSQL planner naturally optimizes the query execution, creating subplans for separate partitions. The router sends these sub-plans to the corresponding shards, which execute them in parallel. In the end, the router collects the intermediate results and, if necessary, executes post-processing, such as sorting or aggregation. Whenever possible, the optimizer pushes down partial aggregation and sorting to shards to reduce data that needs to be transferred to the router. For example, consider a query:

```
SELECT count(*) FROM orders;
```

on the sharded table `orders` from Figure 1. The planner at the router will generate a plan with the following structure:

```
Finalize Aggregate
 Output: count(*)
  -> Append
     -> Async Foreign Scan
         Output: (PARTIAL count(*))
     -> Async Foreign Scan
         Output: (PARTIAL count(*))
     ...
```

[2]As we described in §4.2, a shard also represents its portion of a sharded table as a PostgreSQL partitioned table, with partitions corresponding to table slices. Thus, sharded tables have two layers of PostgreSQL-style partitioning overall.

This evaluates the `count(*)` query at separate shards in parallel (via asynchronous foreign scans), appends the results and then sums them up.

Predicate pushdown. When executing a query, the router classifies predicates associated with tables as *foreign* or *local*. Foreign predicates are those which are safe to be evaluated at the shards, such as predicates involving only built-in operations and `IMMUTABLE` functions. For example, when processing the following query:

```
SELECT name FROM customers WHERE email = 'abc@xyz.com';
```

the router determines that the predicate in the `WHERE` clause is foreign. It thus pushes down its evaluation to the shards and then aggregates the results.

Local predicates are those which are unsafe to evaluated at the shards, so the system brings the data required for them to the router. They include predicates with mutable functions (`STABLE` and `VOLATILE`) and definer functions, which need to be executed using a different user from the current one.

Join pushdown. Joins can be between any combination of table types. For joins between sharded tables, we heavily rely on the PostgreSQL planner to generate optimized plans that leverage existing partition-aware join algorithms. This is particularly effective for inner joins between co-located tables (§2): in this case, partial joins can be executed at each shard in parallel, and the results aggregated at the router. The PostgreSQL cost model may not always select this optimization path, since it is not aware of the fact that the data resides at the shards; we have thus modified it appropriately. For example, consider the following query, which joins co-located sharded tables `customers` and `orders` from Figure 1:

```
SELECT c.cust_id, c.name, c.email, SUM(o.amount) AS total_spent
FROM customers c
INNER JOIN orders o ON c.cust_id = o.cust_id
GROUP BY c.cust_id, c.name, c.email;
```

Aurora Limitless will execute this query by evaluating the joins on different shards in parallel (as PostgreSQL foreign scans) and then appending the results at the router.

We similarly optimize joins between a sharded table and reference table. For example, consider the following query, which joins the sharded table `orders` with the reference table `tax_rates`:

```
SELECT o.order_id, o.amount, t.country, t.tax_rate
FROM orders o
INNER JOIN tax_rates t ON o.tax_rate_id = t.tax_rate_id;
```

Since `tax_rates` is replicated at all shards, the system will again execute this query by evaluating the joins on the shards in parallel and then appending the results. Note, however, that this optimization technique is applicable to only specific join types: Cartesian products, inner joins (both on shard keys and others), and outer joins where the reference table is the null-padded side. It is not suitable for outer joins with a sharded table as the null-padded side or anti-joins.

Finally, joins between standard tables are pushed down to the distinguished shard storing them. Joins between standard and other types of tables are currently just executed at the router.

Function distribution. Aurora Limitless also allows *distributing* SQL functions whose invocations only access data on a single shard. The execution of such functions is pushed down to the corresponding shard and the result is propagated back to the router, thus improving efficiency. To enable this, the declaration of a function to be distributed specifies the corresponding sharded table and the function arguments that correspond to the shard key. For example, given the schema in Figure 1, the following code distributes a function on the sharded table `customers` whose first argument `id` corresponds to the shard key `cust_id`:

```
CREATE FUNCTION func(id INTEGER, n TEXT) RETURNS TEXT
    LANGUAGE SQL
    VOLATILE
    AS $$
    UPDATE customers SET name = n WHERE cust_id = id RETURNING name;
    $$;

SELECT rds_aurora.limitless_distribute_function(
    'func(integer, text)', ARRAY['id'], 'customers');
```

Optimizing single-shard queries. When executing a query, a router leverages the PostgreSQL partition pruning mechanism to determine if all the required source tuples reside on the same shard. If this is the case, such a *single-shard* query is pushed down to the corresponding shard. This results in fewer round trips between routers and shards and lower latency, making single-shard queries the sweet spot of the system. Choosing the right shard keys and tagging functions appropriately as distributed can increase the percentage of queries that are executed as single-shard, thus improving the overall performance.

8 Evaluation

We conducted the experiments on Aurora Limitless Database clusters deployed in the us-east-1 AWS region. Both router and shard nodes utilized Serverless V2 instances, capable of scaling up to 256 Aurora Capacity Units (ACU), i.e., approximately 512 GB of memory and the corresponding CPU and other resources.

To evaluate performance, we employed a customized version of HammerDB [23], an open-source benchmark derived from TPC-C. Our modifications adapt table creation scripts to support sharded tables. The tables `customer`, `stock`, `history`, `warehouse`, `order_line`, `new_order`, `orders`, and `district` are sharded by their `id` column, and the `item` table is a reference table. We optimize query execution by distributing functions that could operate on individual shards, enabling them to be pushed down directly to the relevant shard. We use 12000 as the total number of warehouses. Around 10% of transactions in the workload are distributed.

We evaluated performance across five configurations, varying the number of shards, routers and maximum ACUs, as shown in Table 2. We maintained identical workload parameters across all tests: 1000 concurrent clients (virtual users) and 10 million total iterations. Each configuration is tested for 1 hour, excluding a 20-minute ramp up phase. Figures 4 and 5 give New Orders per Minute (NOPM) and latency measurements across the five different configurations.

8.1 Vertical Scaling

We observed that for non-saturated systems, increasing the maximum ACU allocation while maintaining the same number of routers and shards leads to improved performance. This is illustrated by comparing configurations r3 and r5. In a cluster with 8 routers and 16 shards, doubling the maximum capacity from 1,536 to 3,072 ACU results in significant performance improvements: vertical scaling yields a 41.6% increase in throughput (from 2,042,201 to 2,891,718 NOPM) and a 40.8% reduction in average latency of the NEWORD procedure (from 16.42ms to 9.72ms). In another pair of configuration r2 and r4 (4 routers and 8 shards), when doubling the maximum capacity from 1,536 to 3,072 ACU, the throughput increases only by 4.7% (from 2,012,763 to 2,107,539 NOPM), but the latency decreases more significantly, by 37.7% (from 21.06ms to 13.13ms).

Figures 6 and 7 illustrate the ACU allocations during the experiments, with each curve representing the allocation of an individual shard. These demonstrate a consistent pattern across the workloads: the ACU allocation increases as the workload begins, stabilizes during peak workload periods, and gradually decreases as the workload completes. This reflects the system's ability to automatically adjust resource allocation to changing workload demands.

Configuration Name	Routers	Shards	Max ACU
r1	2	4	1536
r2	4	8	1536
r3	8	16	1536
r4	4	8	3072
r5	8	16	3072

Table 2: Configurations with routers, shards, and Max ACU.

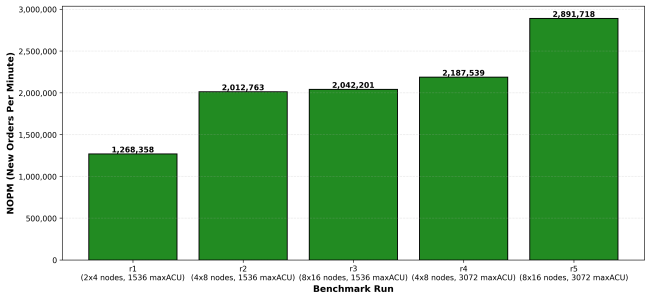


Figure 4: New Orders per Minute (NOPM) in HammerDB.

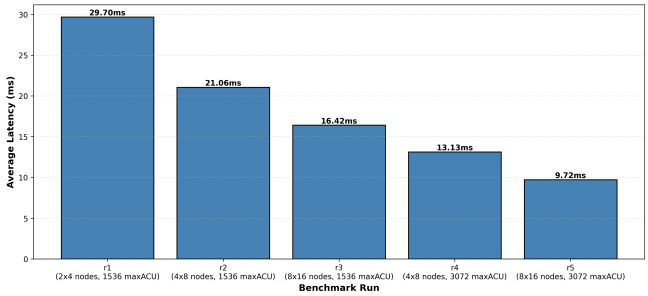
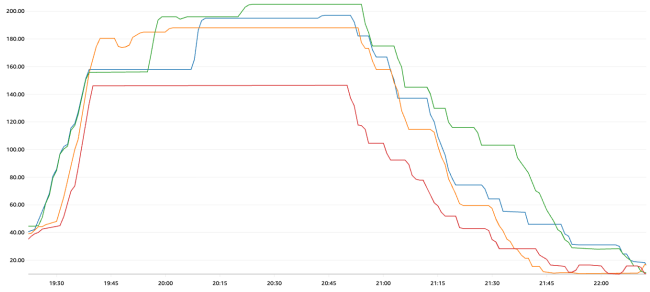
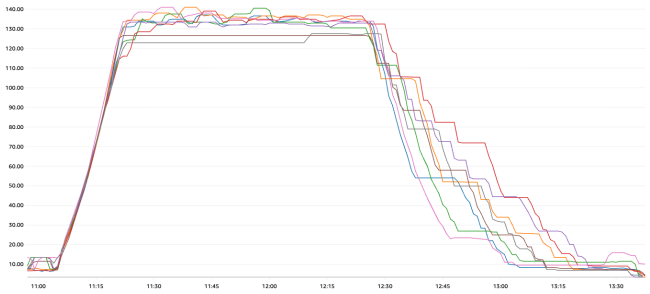


Figure 5: Average latency of the NEWORD procedure in HammerDB.

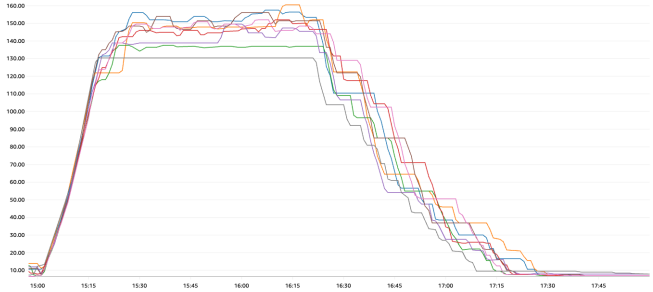


(a) r1: ACU allocations for different shards

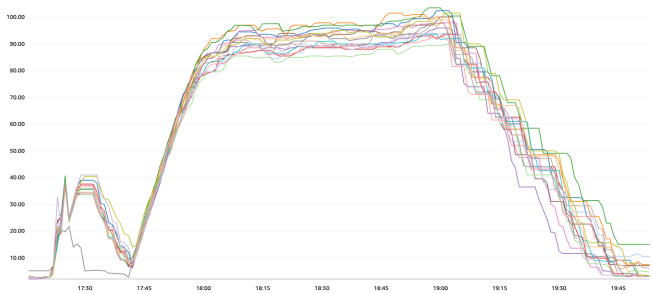


(b) r2: ACU allocations for different shards

Figure 6: ACU allocation in HammerDB (r1 vs r2).



(a) r4: ACU allocations for different shards



(b) r5: ACU allocations for different shards

Figure 7: ACU allocation in HammerDB (r4 vs r5).

8.2 Horizontal Scaling

When vertical scaling reaches its limits due to system saturation, horizontal scaling can improve performance by distributing the workload across additional shards. This is demonstrated by comparing configurations r1 and r2. While maintaining the same maximum capacity of 1,536 ACU, we expand the cluster from 2 routers and 4 shards to 4 routers and 8 shards. This horizontal scaling leads to a substantial improvement in performance: throughput increases by 58.7% (from 1,268,350 to 2,012,763 NOPM), while the average latency decreases by 29.1% (from 29.70ms to 21.06ms).

The analysis of ACU allocation patterns provides an insight into these improvements. Figure 6a shows the ACU allocation for r1, where individual curves represent different shards. During peak workload periods we observe significant imbalances, with some shards reaching 200 ACU while others operating at 150 ACU. This uneven distribution indicates that the performance is constrained by the most heavily loaded shards, while the total ACU allocation remains well below the maximum limit. This stands in contrast to the ACU allocations for r2, shown in Figure 6b. Here the 8 shards maintain a more balanced capacity usage between 125 and 140 ACU. This results in more efficient capacity utilization and, consequently, better overall performance compared to the configuration r1.

Similar trend is observed when comparing r4 to r5, where we expand from 4 routers 8 shards to 8 routers 16 shards while keeping the same maximum capacity of 3072 ACU. The throughput increases by 41.6% (from 2,042,201 to 2,891,718 NOPM), and the average latency decreases by 26% (from 13.13ms to 9.72ms). As shown in Figure 7, in r4 each shard's capacity allocation is in the range of 130 to 160 ACU during peak workloads, while in r5 it is in the range of 85 to 100. Thus, the load is more evenly distributed in r5.

These results demonstrate that Aurora Limitless offers flexible scaling options to support dynamic workloads – through vertical scaling by increasing ACU and horizontal scaling by adding routers and shards.

9 Related Work

Distributed PostgreSQL systems. Citus [12] (now part of Microsoft Azure) is a PostgreSQL-based distributed database supporting sharding. While it guarantees atomicity, consistency, and durability for distributed transactions through two-phase commit, it does not provide snapshot isolation to these transactions. It thus exposes users to a wide range of transaction anomalies.

Greenplum [24] is a PostgreSQL-based system for serving both OLTP and OLAP workloads. It provides snapshot isolation using a centralized coordinator that assigns distributed transaction identifiers. This coordinator can easily become a performance bottleneck as the workload intensity increases. Additionally, the system needs to maintain mappings between local and distributed transactions for visibility checks, which causes performance overhead and increases implementation complexity.

In contrast, Aurora Limitless eschews a centralized coordinator by using a time-based concurrency control combined with two-phase commit. It furthermore provides both vertical and horizontal scaling mechanisms – a combination absent from the above systems.

Geo-distributed SQL databases. Google's Spanner [11] pioneered transaction ordering through hardware-based synchronized clocks,

accessed through its TrueTime API [10]. Clock-SI [17] demonstrated that snapshot isolation could be achieved using loosely synchronized clocks without a centralized timestamp authority, an approach similar to our design. CockroachDB [35] supports serializability and Read Committed, but omits snapshot isolation. It determines commit timestamps for distributed transactions using hybrid logical clocks (HLC) [20], and does not in general guarantee external consistency [25]. YugabyteDB [40] supports serializability and snapshot isolation, also using HLC for commit timestamps. The HLC approach sometimes requires waiting out the maximum clock skew in the cluster, which without TrueTime [10] or Time Sync [5], has to be conservatively set in hundreds of milliseconds.

Aurora DSQL [2] is a serverless distributed SQL database that supports multi-region reads and writes. It implements optimistic concurrency control to provide snapshot isolation. Systems like Azure Cosmos DB [27] and MongoDB Atlas [29] provide multi-model support but often trade consistency for availability.

Unlike these geo-distributed databases, Aurora Limitless is specifically designed for cross-Availability Zone (AZ) rather than cross-region operations. This focused scope allows it to optimize performance for cross-AZ requirements. It leverages Aurora's existing cross-AZ storage layer for durability, eliminating the need for additional consensus protocols like Paxos or Raft.

PolarDB [38] allocates commit timestamps using either an RDMA-based timestamp oracle or HLC; the latter provides lower latency at the expense of relaxed consistency guarantees. Aurora Limitless achieves strong consistency through careful integration with Amazon Time Sync service [5] while maintaining high availability, building upon insights from both Spanner and Clock-SI.

Scaling and architecture. Traditional cloud databases like Azure Database for PostgreSQL [28] and Google Cloud SQL [18] focus on vertical scaling within single instances. Vitess [34] and Apache ShardingSphere [22] offer horizontal scaling but require complex operational management. Our architecture extends Aurora's storage-compute separation [36], similar to Snowflake's approach [13] but optimized for OLTP workloads. We integrate both vertical and horizontal scaling capabilities to support dynamic workloads.

Our query processing techniques draw from classic distributed database research [13, 15, 16, 19, 39] while focusing on optimizing single shard queries. While systems like F1 [33] have demonstrated distributed SQL processing at scale, Aurora Limitless contributes techniques for minimizing cross-shard communication. Our transaction protocol builds upon classic two-phase commit [9] with modifications to tolerate with router failures and support time-based concurrency control.

10 Conclusion and Future Work

Aurora Limitless Database was built to provide transparent scalability while maintaining a high degree of PostgreSQL compatibility. Key innovations that set it apart from other systems are a hybrid scale-up/scale-out model and deep integration of time-based concurrency control, offering external consistency. We now share some lessons we learned running Aurora Limitless in production as a managed AWS service and outline opportunities for future research.

Migrating from standalone PostgreSQL or self-managed sharding to distributed PostgreSQL has been one of the biggest challenges faced by customers. Many existing database applications were developed with no table or schema partitioning in mind. To make such applications scale, customers have to figure out whether their workload is partitionable, design a scale-out-ready schema, migrate their data to the new schema, and port their existing SQL queries and code. Choice of sharding keys is a critical consideration: it is a one-way door that requires data migration upon resharding. Building tools that help with these challenges is essential.

PostgreSQL is a phenomenally flexible foundation for building distributed databases. However, in our implementation we tripped over numerous PostgreSQL behaviors that could benefit from modernization and adaptation to distributed settings. One is schema evolution. Highly available distributed applications require mechanisms such as schema versioning [32], canarying, and multi-version concurrency control for schemas [6].

As another example, distributed transactions are much easier to implement correctly in a system where the order in which transactions commit coincides with the order in which they become visible to other transactions. Unfortunately, this is not the case in community PostgreSQL, which has led to anomalies in clusters with read replicas [26]. In Aurora Limitless we forced commit and visibility orders to match by defining both of them using timestamps (cf. Property 1 in §5.1). We stopped short of supporting serializability due to limited customer demand and the fact that, in its PostgreSQL implementation [31], the serialization order of transactions does not coincide with their commit order [30].

Finally, we found that hybrid scaling needs to be carefully tuned to maximize its performance and cost benefits for customers.

Acknowledgements. Aurora Limitless is a result of a collaborative effort by a large team. We particularly thank the following contributors: Abhishek Suresh Kumar, Ahmad Alsmair, Alex Friedman, Alexandre Verbitski, Alison Qiu, Amit Krishnan, Amogh Joshi, Aniket Rathore, Anna Grinzewich, Anum Sher Jang, Atanu Ghosh, Ayush Gupta, Ayush Vatsa, Bijivemula Sai Rakesh Reddy, Callis Chen, Charles Ren, Chaynika Saikia, Chen Gao, Chiya Ma, Christopher Heim, Christopher Sien, Daeseob Lim, Damir Kharisov, David Pacheco, Derek Zhang, Dylan Supenchek, Edison Zhao, Edward Shi, Erdem Yilmaz, Floria Peng, Frederick Yao, Frederico Dib, Gajendran Majepari, Gaurav Kumar Gupta, Geeta Sravanthi, Gia Zhou, Glenn LeBlanc, Grant McAlister, Hannes Rauhe, Harshank Vengurlekar, Huy Nguyen, Ian Gu, Ivan Skhundalev, Jacob Schweitzer, James Zhao, Jason Zhang, Jasper Wu, Jay Katariya, Jerry Wang, Jerry Tang, Jim Finnerty, Jingeng Yang, Jungkook Lee, Justin Lin, Kai Tang, Kaibo Zhang, Ken Zhang, Kiran Pillarisetty, Konstantin Dubinets, Krishna Sankaran, Kuntal Ghosh, Kushaal Shroff, Matthew Zhang, Mert Anil Hasret, Mohamed Ali, Nelson Truong, Nikunj Jha, Nirmit Shah, Nitin Veera, Nozim Islamov, Peng Gu, Philip Cai, Pradeep Roy, Pramod Vemula, Praveen Kannan, Rachel Fu, Rahul Ramachandran, Raj Aryan Srivastava, Rajan Pandey, Ram Yerramilli, Rob Verschoor, Sachin Kumar, Sambhavi Raghuraman, Sami Imseih, Shanshan Li, Shesan Balachandran, Sohil Nadarajan, Stefan Karlsson, Suiyi Fu, Sujeet Kumar, Suprio Pal, Surendra Vishnoi, Syam Haque, Takuya Sugimoto, Teja Vemula, Tim Galvin, Tingyi Guo, Utkarash Kumar Singh, Vidushi Gupta, Vinay Khandelia, Vinith Venkatesan, William McKinnon, Xiangyu Huang, Xiao Pan, Xingyu Su, Yash Jain, Yifei Jin, Zhangsu Wen, Zijie Zhu.

References

[1] Amazon Web Services. 2026. Amazon Aurora Serverless. https://aws.amazon.com/rds/aurora/serverless/. Accessed: April 1, 2026.

[2] Amazon Web Services. 2026. Aurora DSQL. https://aws.amazon.com/rds/aurora/dsql/. Accessed: April 1, 2026.

[3] Amazon Web Services. 2026. ClockBound daemon. https://github.com/aws/clockbound. Accessed: April 1, 2026.

[4] Amazon Web Services. 2026. Cloning a volume for an Amazon Aurora DB cluster. https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Aurora.Managing.Clone.html. Accessed: April 1, 2026.

[5] Amazon Web Services. 2026. Precision clock and time synchronization on your EC2 instance. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/set-time.html. Accessed: April 1, 2026.

[6] Panagiotis Antonopoulos, Mansi Chauhan, Shailender Dabas, Rajat Jain, Darshan Kattera, Wonseok Kim, Hanuma Kodavalla, Nikolas Ogg, Prashanth Purnananda, Rahul Ranjan, Alex Swanson, and Divyesh Tikmani. 2025. MD-MVCC: Multi-version concurrency control for schema changes in Azure SQL database. *Proc. VLDB Endow.* 18, 12 (2025).

[7] Bradley Barnhart, Marc Brooker, Daniil Chinenkov, Tony Hooper, Jihoun Im, Prakash Chandra Jha, Tim Kraska, Ashok Kurakula, Alexey Kuznetsov, Grant McAlister, Arjun Muthukrishnan, Aravinthan Narayanan, Douglas Terry, Bhuvan Urgaonkar, and Jiaming Yan. 2024. Resource management in Aurora Serverless. *Proc. VLDB Endow.* 17, 12 (2024).

[8] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. In *International Conference on Management of Data (SIGMOD)*.

[9] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.

[10] Eric Brewer. 2017. *Spanner, TrueTime and the CAP theorem.* Technical Report. Google Inc. https://static.googleusercontent.com/media/research.google.com/en/pubs/archive/45855.pdf.

[11] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.* 31, 3 (2013).

[12] Umur Cubukcu, Ozgun Erdogan, Sumedh Pathak, Sudhakar Sannakkayala, and Marco Slot. 2021. Citus: Distributed PostgreSQL for data-intensive applications. In *International Conference on Management of Data (SIGMOD)*.

[13] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake elastic data warehouse. In *International Conference on Management of Data (SIGMOD)*.

[14] Khuzaima Daudjee and Kenneth Salem. 2006. Lazy database replication with snapshot isolation. In *International Conference on Very Large Data Bases (VLDB)*.

[15] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008).

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *Symposium on Operating Systems Principles (SOSP)*.

[17] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. 2013. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Symposium on Reliable Distributed Systems (SRDS)*.

[18] Google Cloud. 2026. Google Cloud SQL. https://cloud.google.com/sql. Accessed: April 1, 2026.

[19] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: a Raft-based HTAP database. *Proc. VLDB Endow.* 13, 12 (2020).

[20] Sandeep S. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. 2014. Logical physical clocks. In *International Conference on Principles of Distributed Systems (OPODIS)*.

[21] Josh Levinson and Julien Ridoux. 2023. It's about time: Microsecond-accurate clocks on Amazon EC2 instances. (2023). https://aws.amazon.com/blogs/compute/its-about-time-microsecond-accurate-clocks-on-amazon-ec2-instances/.

[22] Ruiyuan Li, Liang Zhang, Juan Pan, Junwen Liu, Peng Wang, Nianjun Sun, Shanmin Wang, Chao Chen, Fuqiang Gu, and Songtao Guo. 2022. Apache ShardingSphere: A holistic and pluggable platform for data sharding. In *International Conference on Data Engineering (ICDE)*.

[23] HammerDB Ltd. 2026. HammerDB: The industry standard open-source database benchmark. https://www.hammerdb.com/. Accessed: April 1, 2026.

[24] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. 2021. Greenplum: A hybrid database for transactional and analytical workloads. In *International Conference on Management of Data (SIGMOD)*.

[25] Andrei Matei. 2021. CockroachDB's consistency model. (2021). https://www.cockroachlabs.com/blog/consistency-model/.

[26] Sergey Melnik. 2025. Understanding transaction visibility in PostgreSQL clusters with read replicas. https://aws.amazon.com/blogs/database/understanding-transaction-visibility-in-postgresql-clusters-with-read-replicas/.

[27] Microsoft Azure. 2026. Azure Cosmos DB. https://azure.microsoft.com/en-us/products/cosmos-db. Accessed: April 1, 2026.

[28] Microsoft Azure. 2026. Azure Database for PostgreSQL. https://azure.microsoft.com/en-us/products/postgresql. Accessed: April 1, 2026.

[29] MongoDB Inc. 2026. MongoDB Atlas Database. https://www.mongodb.com/products/platform/atlas-database. Accessed: April 1, 2026.

[30] Elizabeth J O'Neil and Patrick E O'Neil. 2016. Determining serialization order for serializable snapshot isolation. *Information Systems* 58 (2016).

[31] Dan R. K. Ports and Kevin Grittner. 2012. Serializable snapshot isolation in PostgreSQL. *Proc. VLDB Endow.* 5, 12 (2012).

[32] Ian Rae, Eric Rollins, Jeff Shute, Sukhdeep Sodhi, and Radek Vingralek. 2013. Online, asynchronous schema change in F1. *Proc. VLDB Endow.* 6, 11 (2013).

[33] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. 2013. F1: A distributed SQL database that scales. *Proc. VLDB Endow.* 6, 11 (2013).

[34] Sugu Sougoumarane and Mike Solomon. 2012. Vitess: Scaling MySQL at YouTube using Go. https://www.usenix.org/conference/lisa12/vitess-scaling-mysql-youtube-using-go.

[35] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The resilient geo-distributed SQL database. In *International Conference on Management of Data (SIGMOD)*.

[36] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design considerations for high throughput cloud-native relational databases. In *International Conference on Management of Data (SIGMOD)*.

[37] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvilli, and Xiaofeng Bao. 2018. Amazon Aurora: On avoiding distributed consensus for I/Os, commits, and membership changes. In *International Conference on Management of Data (SIGMOD)*.

[38] 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. *Proc. VLDB Endow.* 18, 12 (2025).

[39] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, Huang Yu, Bin Liu, Yi Pan, Boxue Yin, Junquan Chen, and Quanqing Xu. 2022. OceanBase: A 707 million tpmC distributed relational database system. *Proc. VLDB Endow.* 15, 12 (2022).

[40] YugabyteDB Inc. 2026. Fundamentals of distributed transactions. https://docs.yugabyte.com/stable/architecture/transactions/transactions-overview/. Accessed: April 1, 2026.