AWS re:Invent

DECEMBER 2 – 6, 2024 | LAS VEGAS, NV

DAT420

Achieving scale with Amazon Aurora PostgreSQL Limitless Database

Anum Jang Sher
Senior Product Manager
Amazon Aurora

David Wein
Senior Principal Technologist
Amazon Aurora

Agenda

01 Scaling challenges

02 Overview

03 Architecture

04 Data distribution

05 Transactions and queries

06 Get started today!

Celebrating a decade of Amazon Aurora innovation for customers

2014

Aurora MySQL-Compatible Edition

2016

Cross-Region replicas

Cross-account snapshots

Reader endpoint

Zero-downtime patching

2017

Fast database cloning

Export to Amazon S3

Auto Scaling with Aurora Replicas

Aurora PostgreSQL-Compatible Edition

2018

Aurora Serverless v1

Parallel query (MySQL)

Backtrack (MySQL)

Global Database

2019

Cluster cache management

Cross-account cloning

Aurora ML

2020

128 TiB storage support

Global Database–managed RPO

AWS Backup integration

2021

Babelfish (PostgreSQL)

Global Database switchover

Federated querying

2022

Aurora Serverless v2

Blue/green deployment

Fast export to Amazon S3 (MySQL)

Trusted Language Extensions (PostgreSQL)

2023

I/O-Optimized

Optimized Reads

Global Database failover

Aurora MySQL zero-ETL with Amazon Redshift

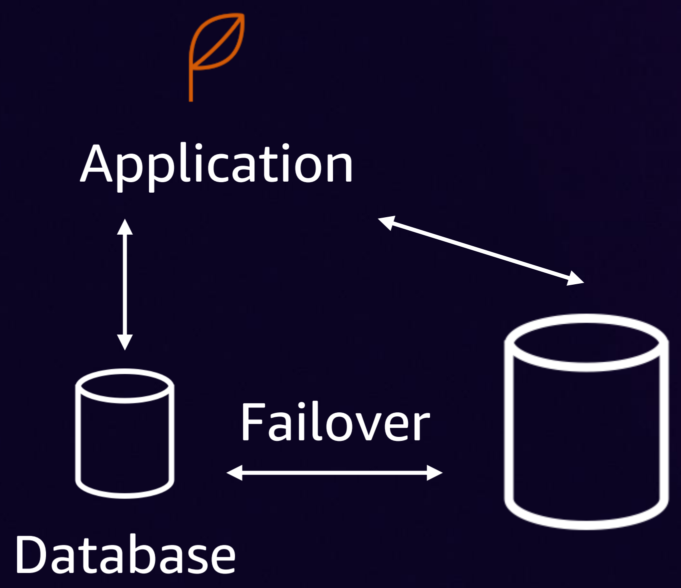
2024

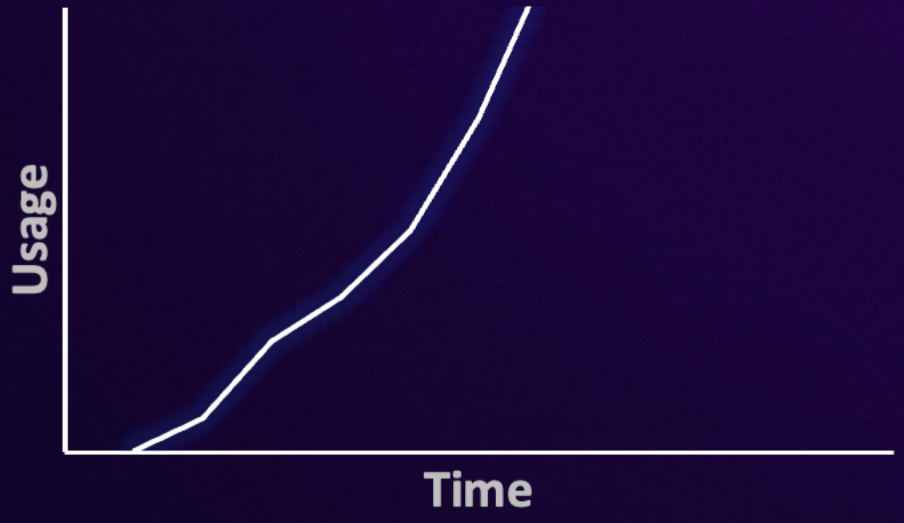
Aurora PostgreSQL Limitless Database

Aurora PostgreSQL zero-ETL with Amazon Redshift

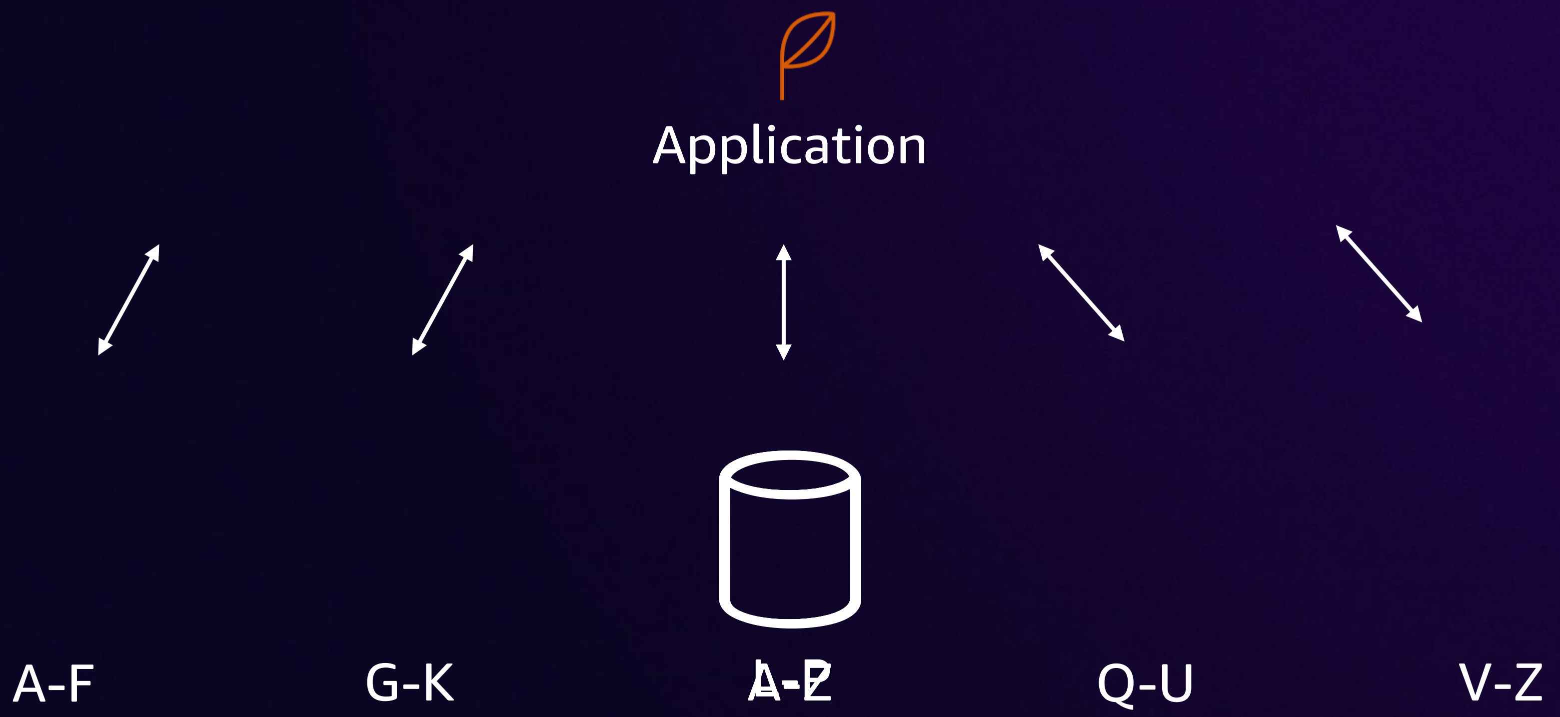
And more to come . . .

Scaling databases





Sharding brings **scale**



Challenges

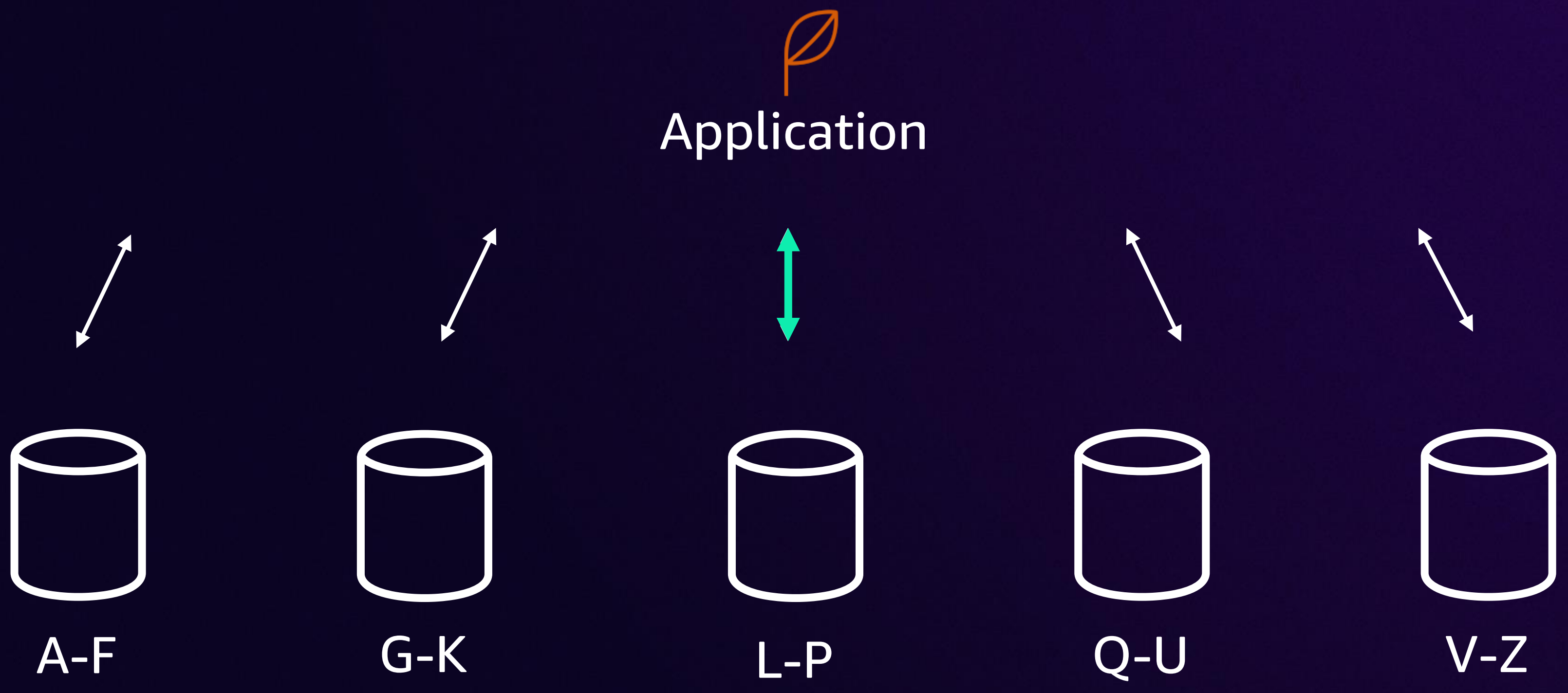
Querying

Consistency

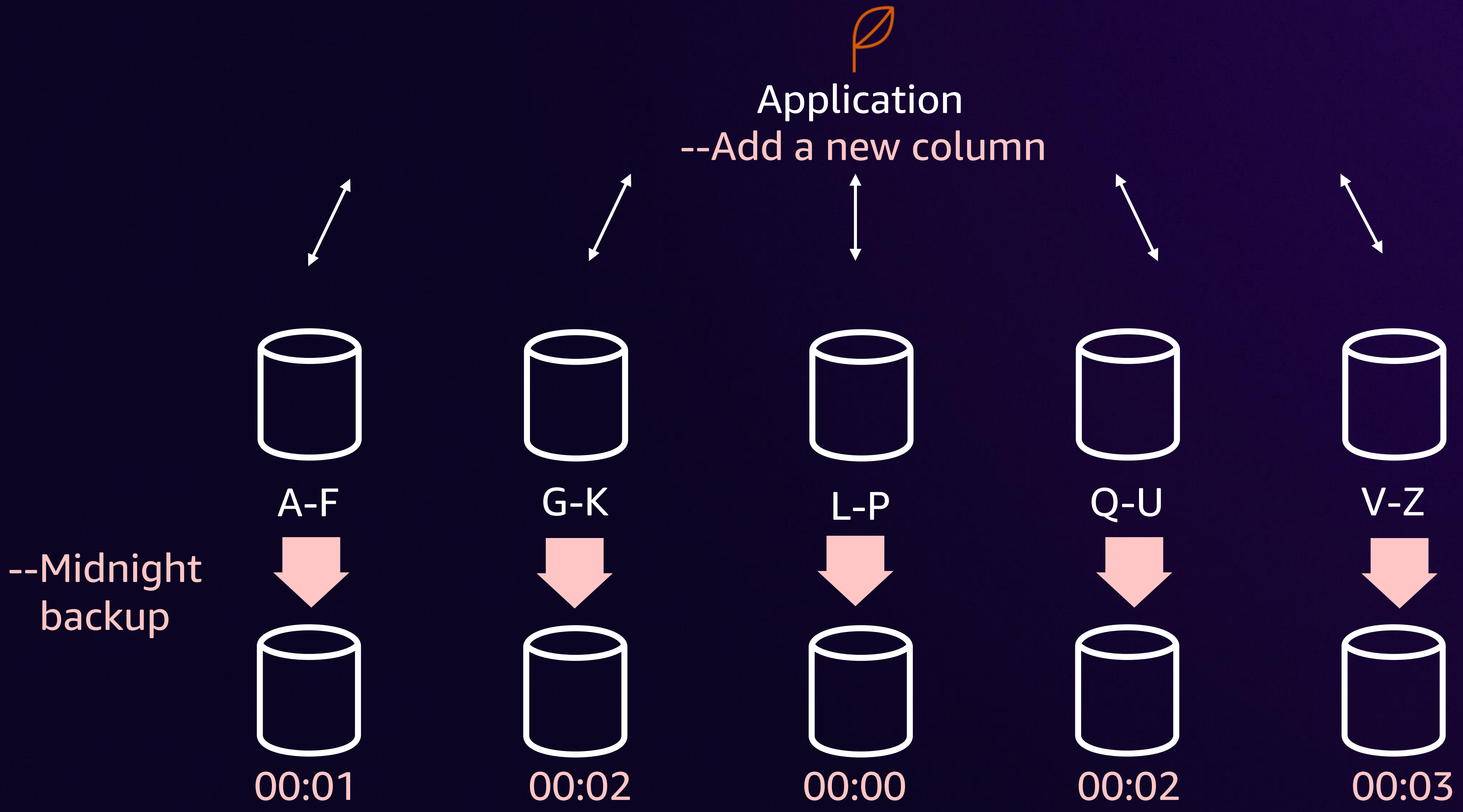
Re-sharding

Capacity management

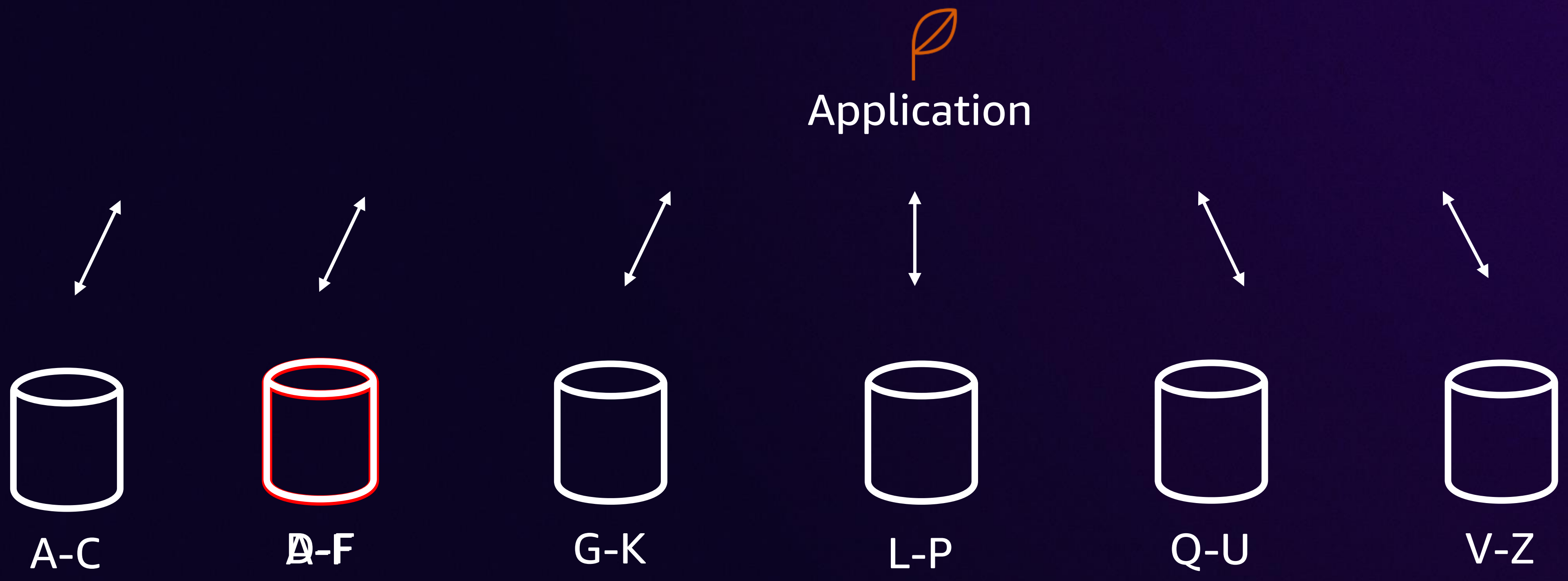
Challenges: Querying



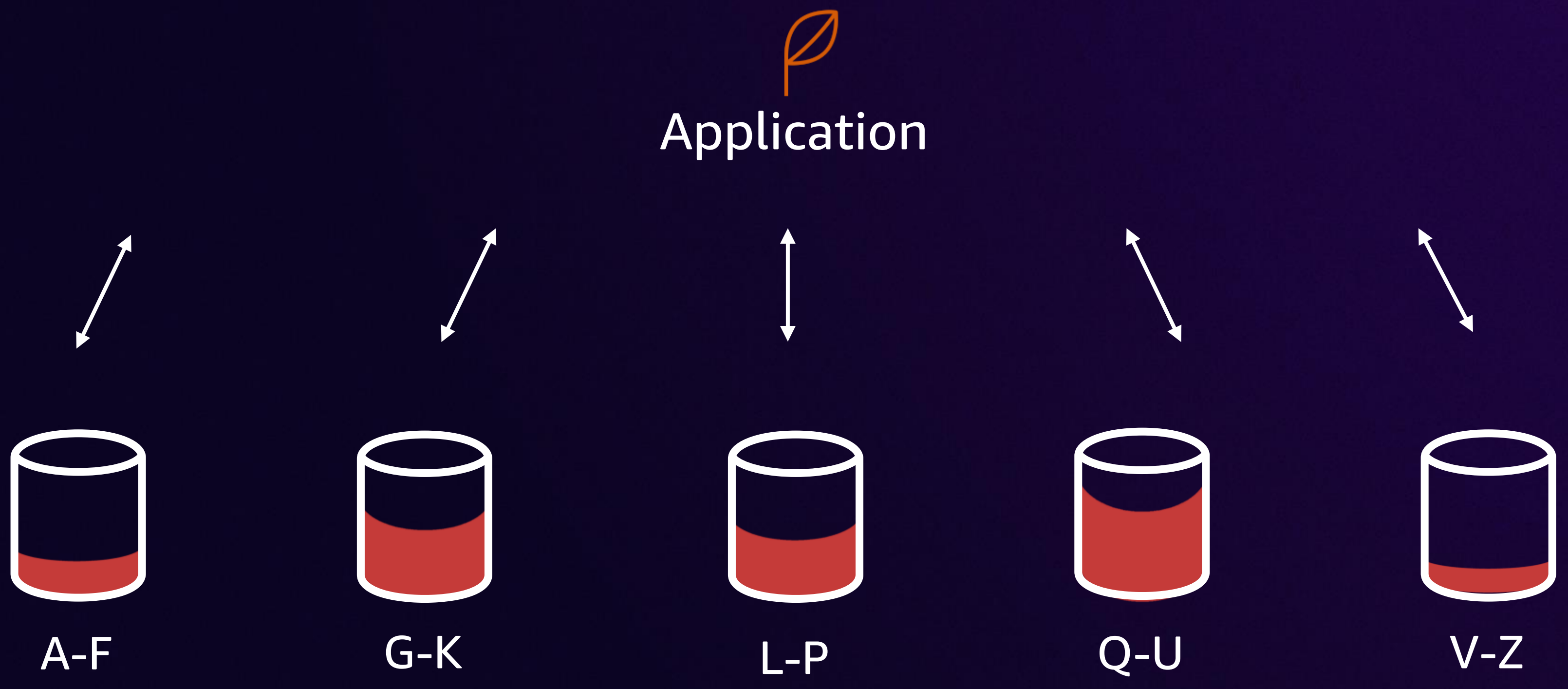
Challenges: Consistency



Challenges: Re-sharding



Challenges: Database capacity management



Aurora PostgreSQL Limitless Database

NEW

MANAGED HORIZONTAL SCALE-OUT BEYOND THE LIMITS OF A SINGLE INSTANCE

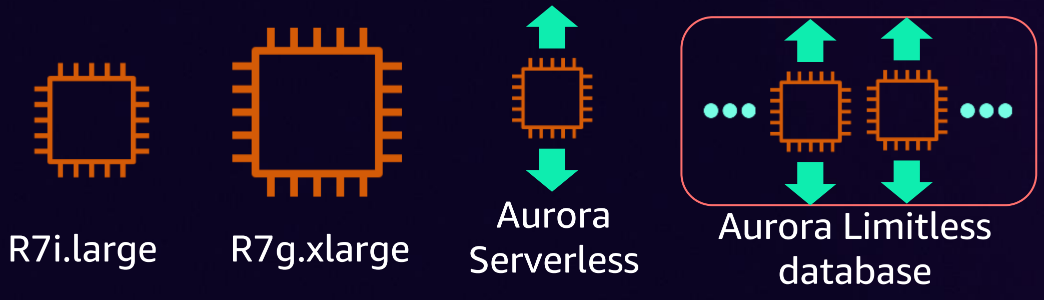
Scales to millions of write transactions per seconds

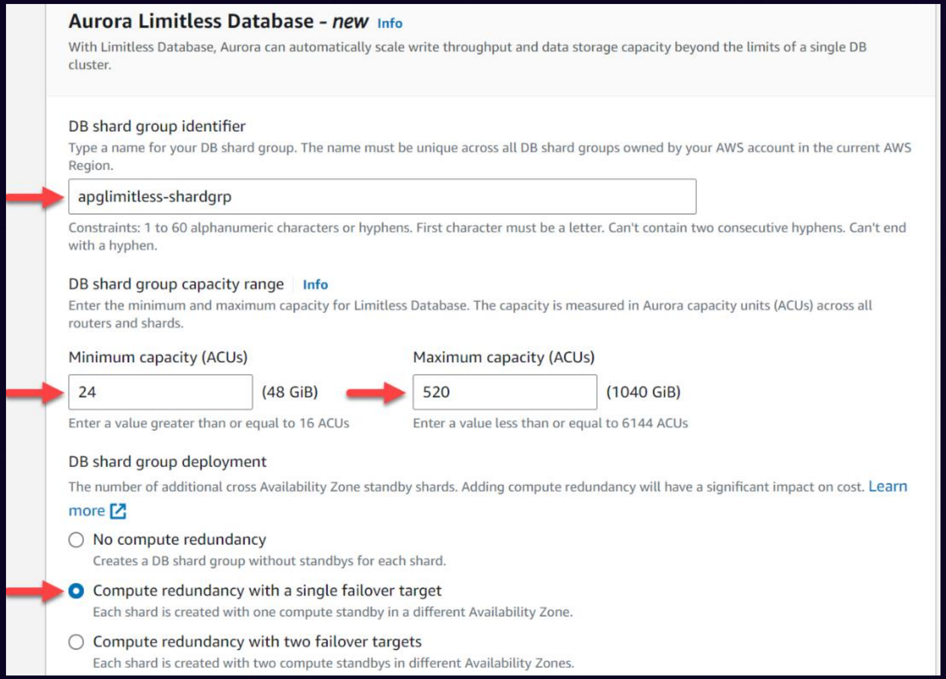
Manages petabytes of data

Operational simplicity of a single instance

Pay-per-use pricing

DB shard group





Aurora Limitless Database - *new* Info

With Limitless Database, Aurora can automatically scale write throughput and data storage capacity beyond the limits of a single DB cluster.

DB shard group identifier

Type a name for your DB shard group. The name must be unique across all DB shard groups owned by your AWS account in the current AWS Region.

apglimitless-shardgrp

Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

DB shard group capacity range Info

Enter the minimum and maximum capacity for Limitless Database. The capacity is measured in Aurora capacity units (ACUs) across all routers and shards.

Minimum capacity (ACUs)

24 (48 GiB)

Enter a value greater than or equal to 16 ACUs

Maximum capacity (ACUs)

520 (1040 GiB)

Enter a value less than or equal to 6144 ACUs

DB shard group deployment

The number of additional cross Availability Zone standby shards. Adding compute redundancy will have a significant impact on cost. Learn more

- No compute redundancy
  Creates a DB shard group without standbys for each shard.
- Compute redundancy with a single failover target
  Each shard is created with one compute standby in a different Availability Zone.
- Compute redundancy with two failover targets
  Each shard is created with two compute standbys in different Availability Zones.

Create **DB shard group** instead of instances

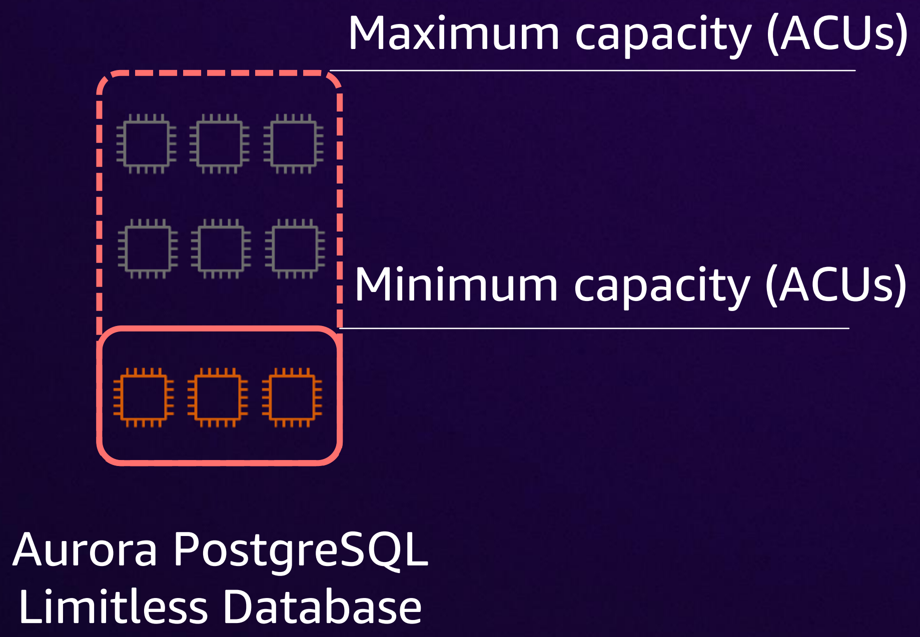
Automatic scaling based on workload

Specify **compute redundancy**

Supported by **99.99%** availability SLA

Capacity management

- DB shard group capacity is measured in **Aurora Capacity Units** (ACUs)
- One ACU is a combination of **2GiB** of memory
- Corresponding **CPU** and **networking**
- You set the **minimum** and **maximum** capacity



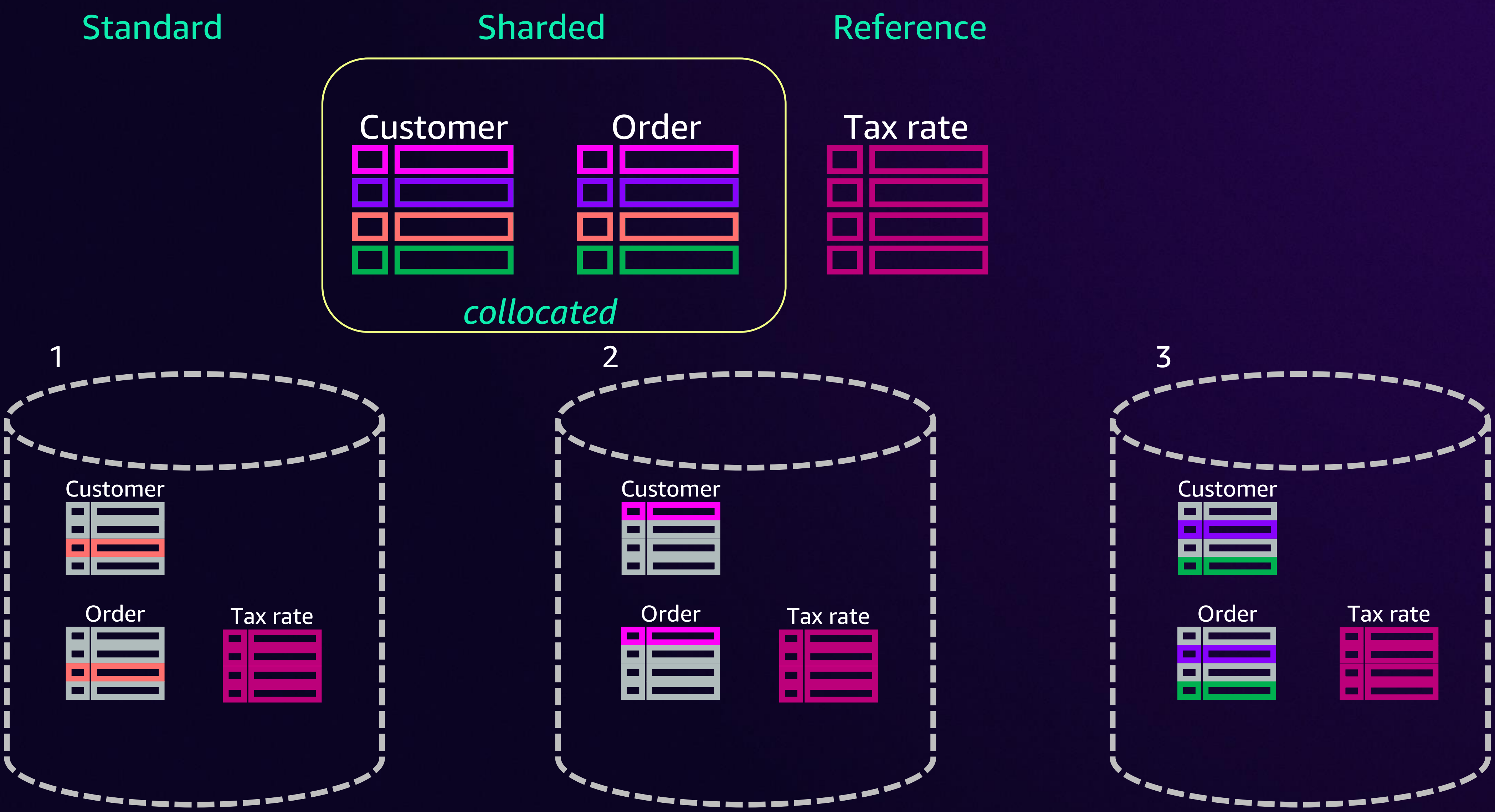
Scenario

Customer
cust_id
name
email

Order
order_id
cust_id
amount
tax_rate_id

Tax rate
tax_rate_id
city
state
country
tax_rate

Use limitless database



Create sharded customer table

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

SET rds_aurora.limitless_create_table_shard_key='{“cust_id"}';
```

```
CREATE TABLE customer (
	cust_id INT PRIMARY KEY NOT NULL,
	name    TEXT,
	email   VARCHAR(100)
	);
```

Create collocated order table

```
SET rds_aurora.limitless_create_table_mode='sharded';
SET rds_aurora.limitless_create_table_shard_key=‘{“cust_id”}';


SET rds_aurora.limitless_create_table_collocate_with='customer';

CREATE TABLE order (
      order_id    INT NOT NULL,
      cust_id     INT NOT NULL,
      amount      DOUBLE NOT NULL,
      tax_rate_id DOUBLE,
      PRIMARY KEY (order_id, cust_id)
      );
```

Create reference table tax_rate

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

```
CREATE TABLE tax_rate (
      tax_rate_id INT PRIMARY KEY NOT NULL,
      city        TEXT NOT NULL,
      state       TEXT,
      country     TEXT NOT NULL,
      tax_rate    DOUBLE NOT NULL
      );
```

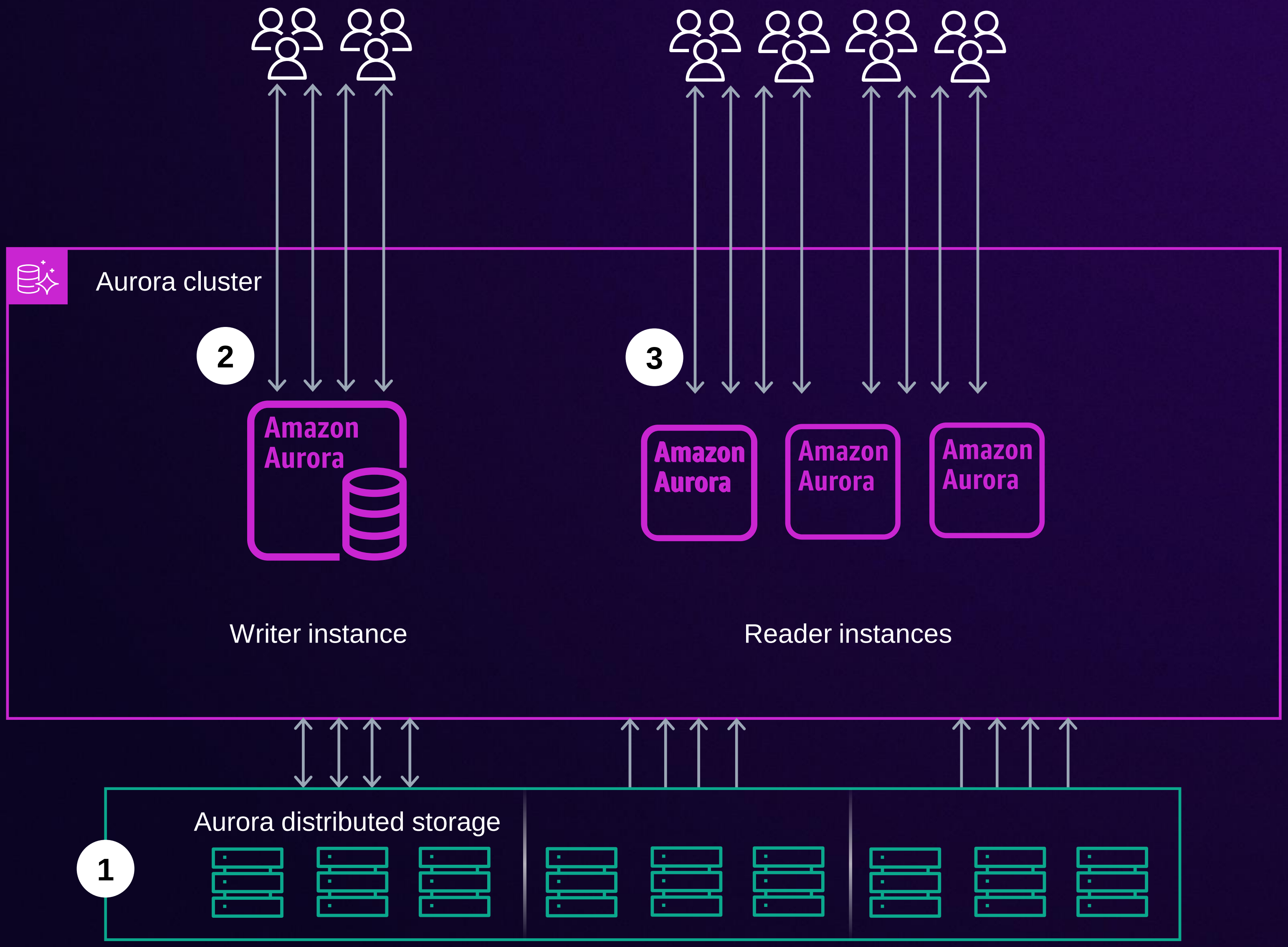
Amazon Aurora PostgreSQL Limitless Database architecture

David Wein

Senior Principal Technologist
Amazon Aurora

Standard Aurora architecture

1. Aurora volume on **distributed storage**
2. An Aurora **writer** instance
3. Optional **reader instances** for availability and read scaling



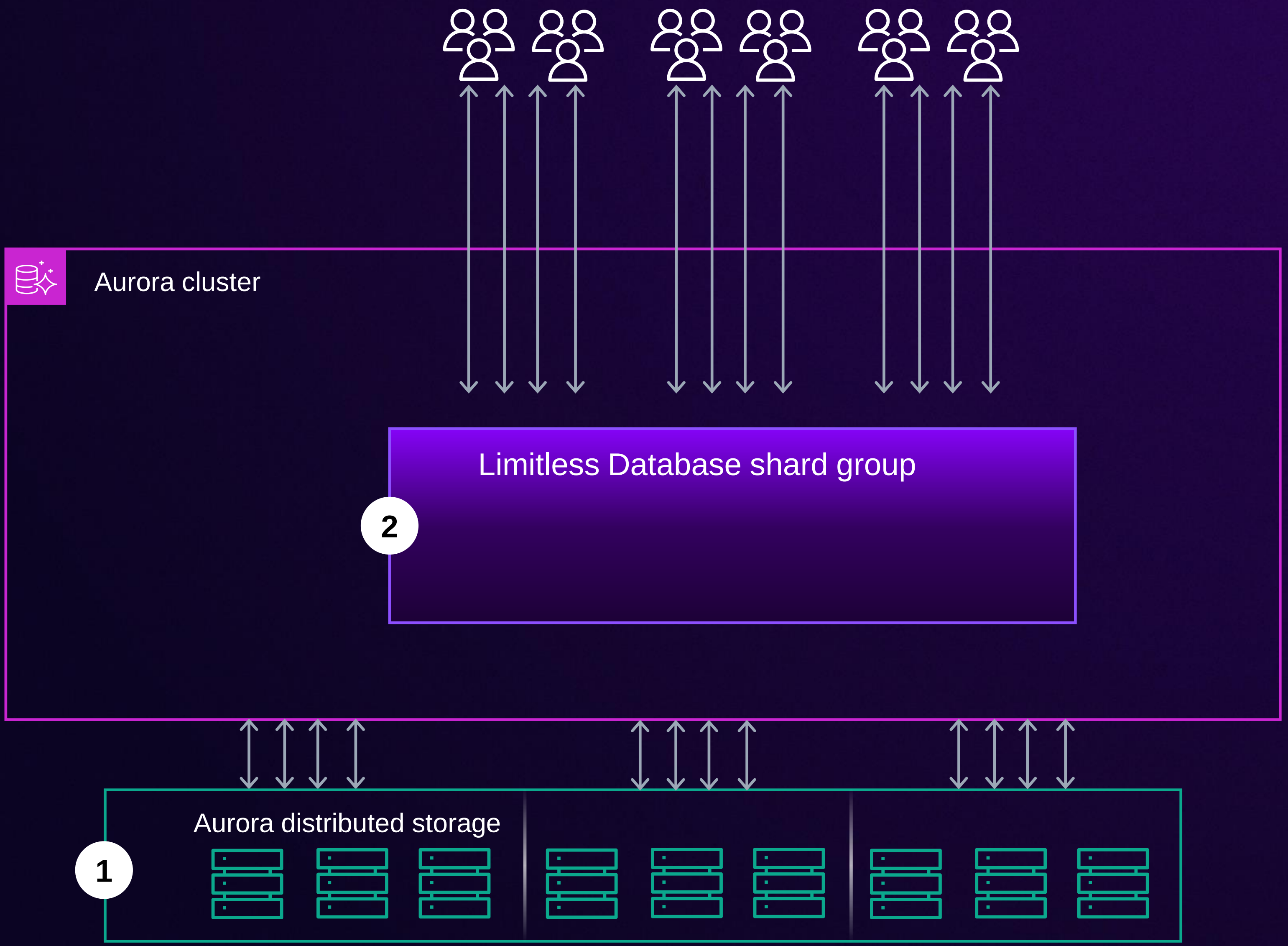
Standard Aurora architecture

1 Aurora volume on distributed storage

~~An Aurora writer instance~~

~~Optional reader instances for availability and read scaling~~

2 Limitless Database introduces the "shard group" concept



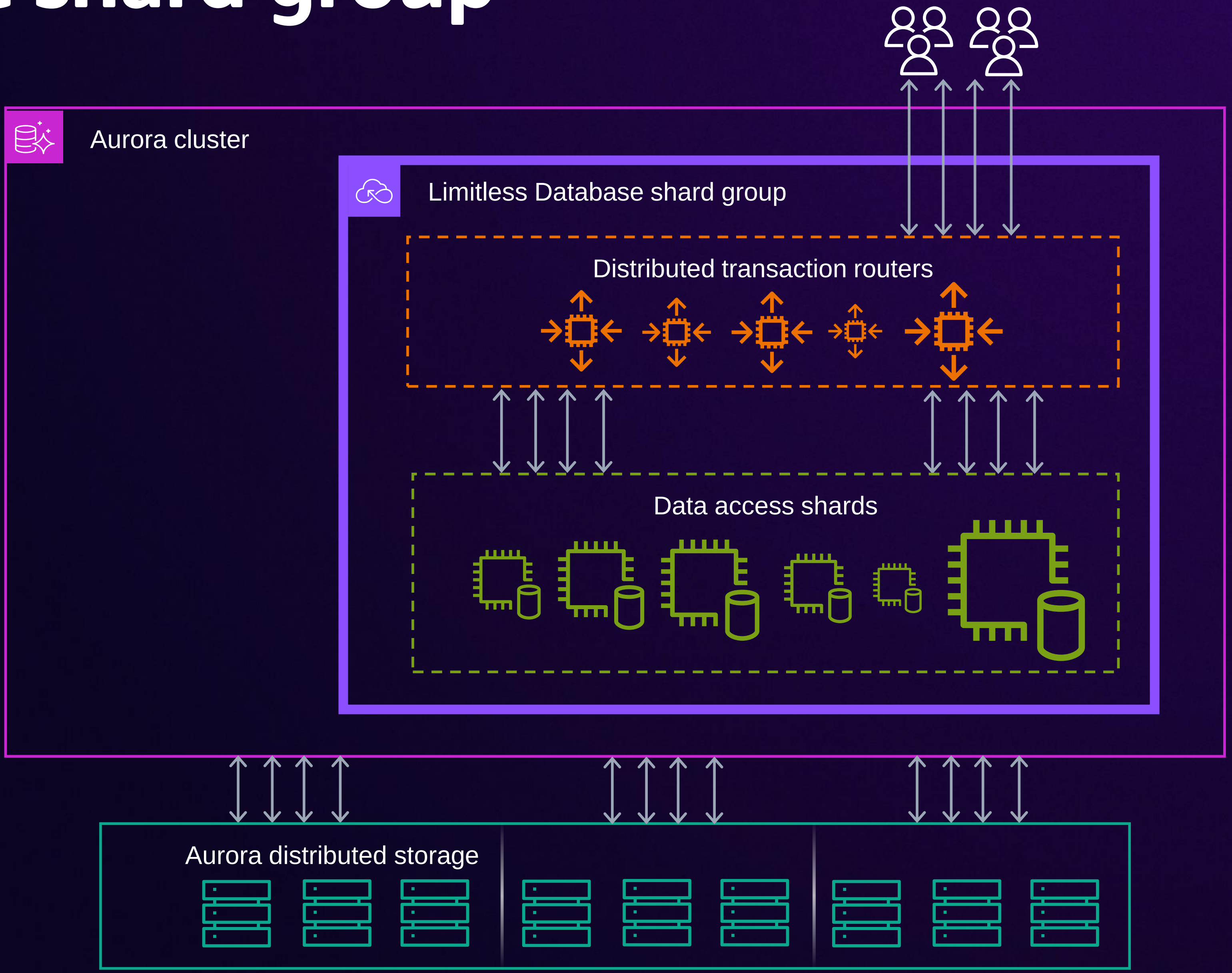
Limitless Database shard group

Contained within your Aurora cluster

Encapsulates limitless database infrastructure for your cluster

Provides an endpoint for applications

Scales resources within configured capacity based on load and data size



Distributed transaction routers

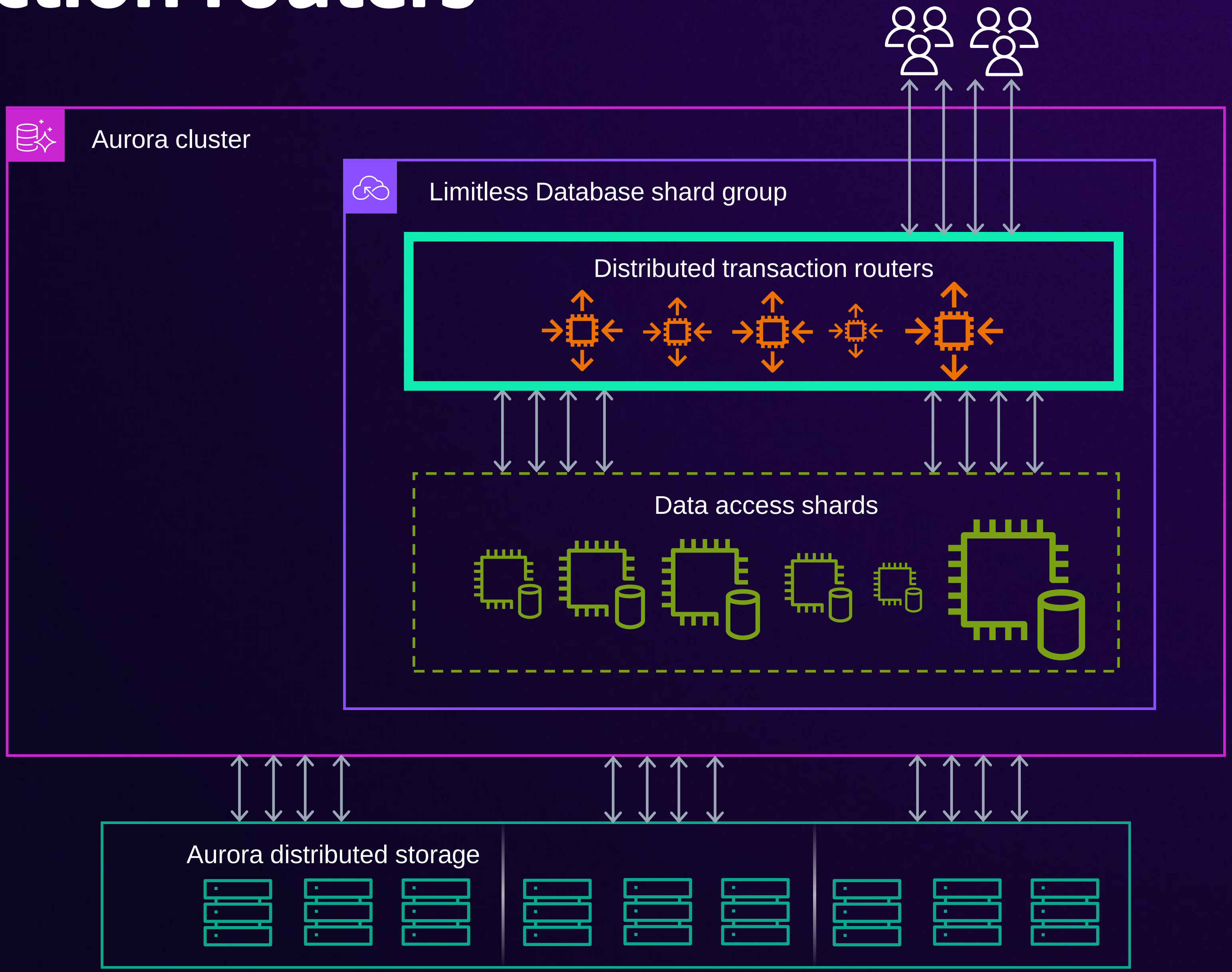
Serve all application traffic

Scale vertically and horizontally based on load

Know schema and key range placement

Assign time for transaction snapshot and drive distributed commits

Perform initial planning of query and aggregate results from multi-shard queries



Data access shards

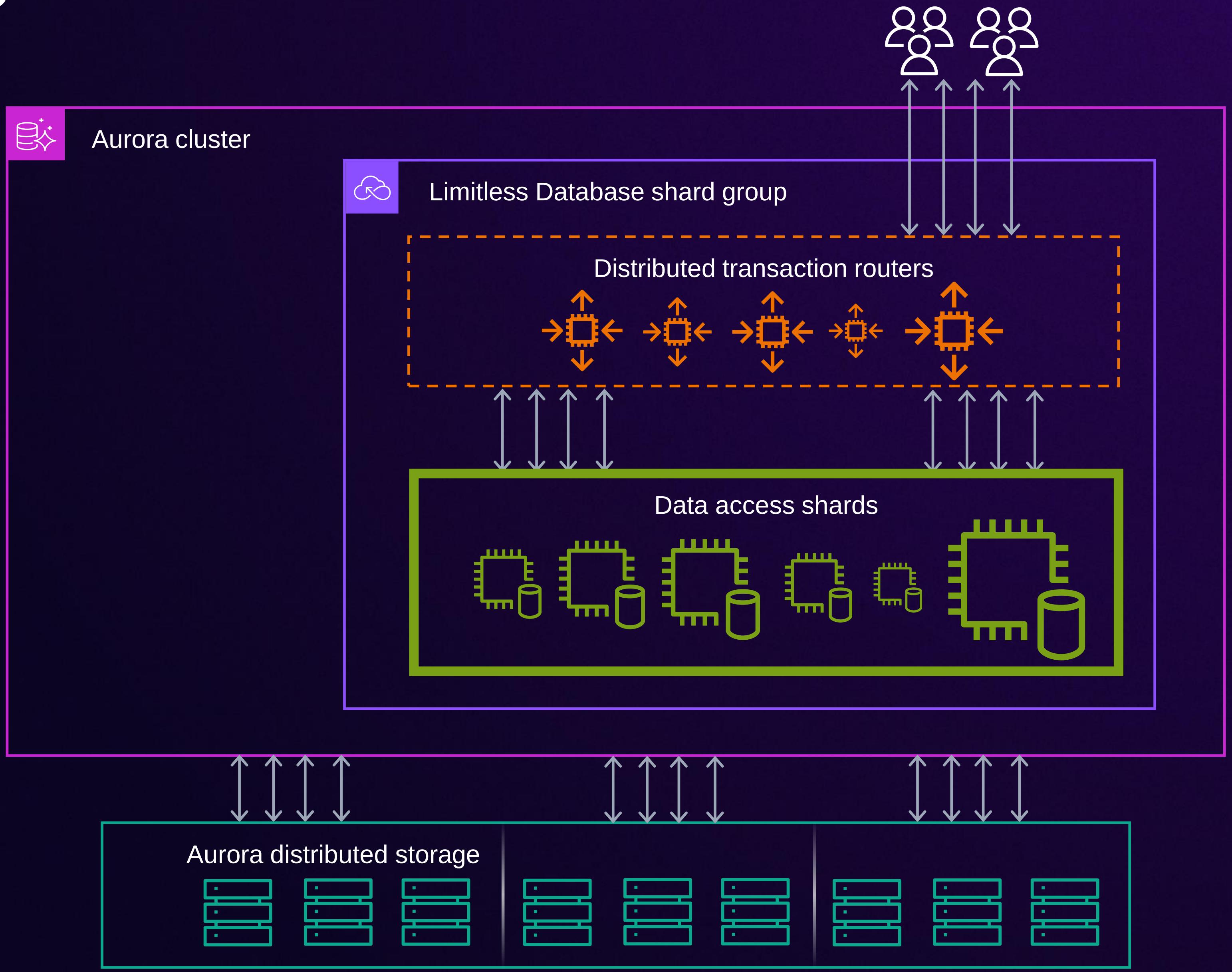
Own portion of sharded table key space and have full copies of reference tables

Scale vertically and split based on load

Perform local planning and execution of query fragments

Execute local transaction logic

Backed by Aurora distributed storage

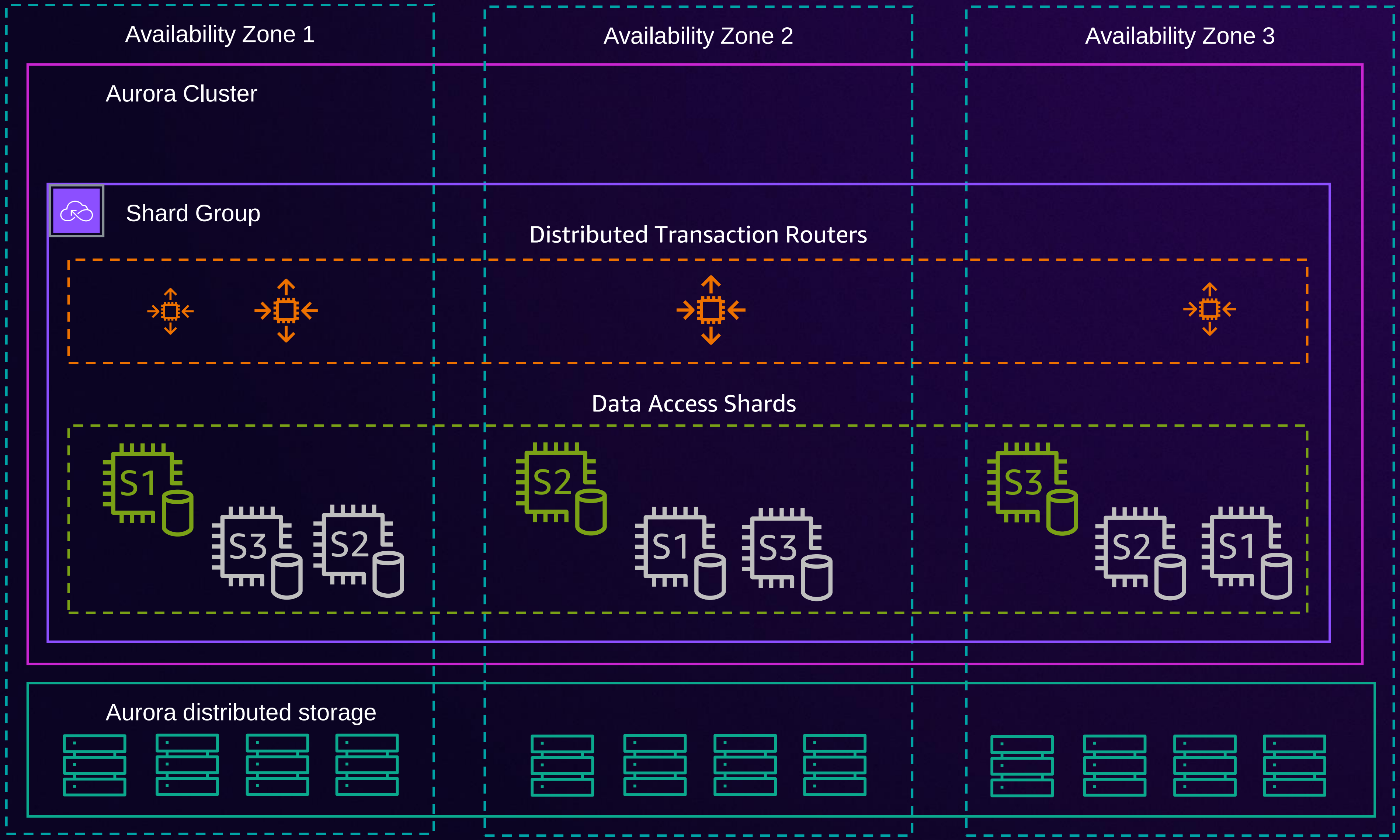


Topology and availability

Topology distributed across Availability Zones

Compute redundancy 0, 1, and 2

Routers are fungible and can be replaced as necessary



Data distribution

Sharded tables

```
SET rds_aurora.limitless_create_table_mode='sharded';
SET rds_aurora.limitless_create_table_shard_key='{"cust_id"}';

CREATE TABLE customer (
        cust_id INT PRIMARY KEY NOT NULL,
        name    TEXT,
        email   VARCHAR(100)
        );
```

```
postgres_limitless=> \d+ customer
                                    Partitioned table "public.customer"
  Column  |      Type      | Collation | Nullable | Default | Storage  | Compression | Stats target | Description
----------+----------------+-----------+----------+---------+----------+-------------+--------------+-------------
 cust_id  | integer        |           | not null |         | plain    |             |              |
 name     | text           |           |          |         | extended |             |              |
 email    | char..var(100)|           |          |         | extended |             |              |
Partition key: HASH (cust_id)
Partitions: customer_fs00001 FOR VALUES FROM (MINVALUE) TO ('-4611686018427387904'),
            customer_fs00002 FOR VALUES FROM ('-4611686018427387904') TO ('0'),
            customer_fs00003 FOR VALUES FROM ('0') TO ('4611686018427387904'),
            customer_fs00004 FOR VALUES FROM ('4611686018427387904') TO (MAXVALUE)
```

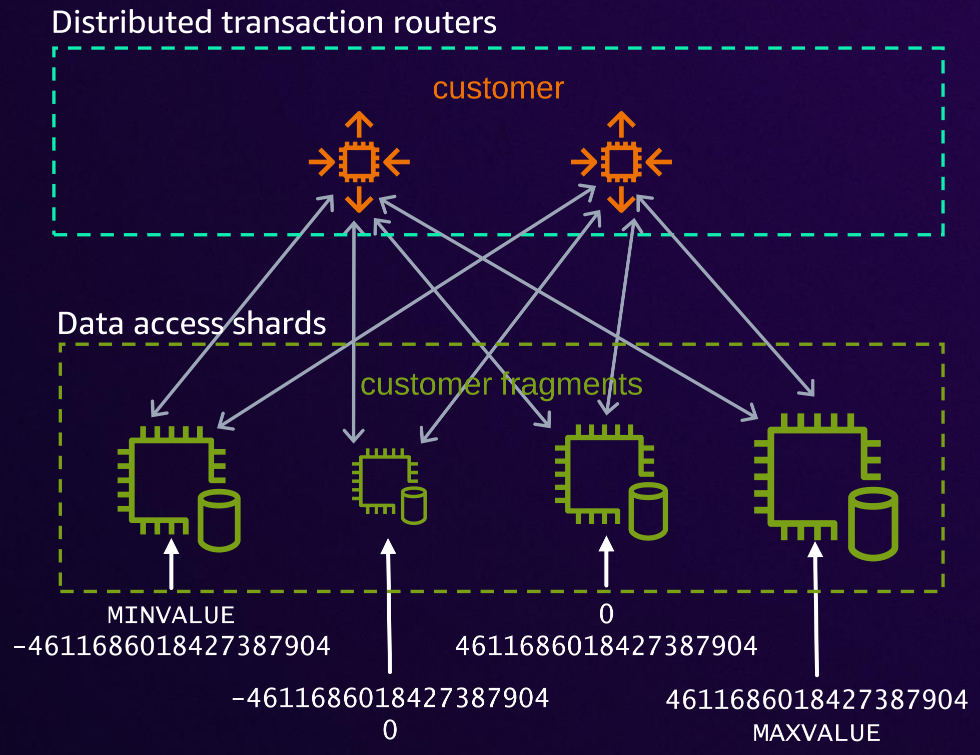
Hash-range partitioning

Shard key is hashed to 64-bits

Ranges of 64-bit space are assigned to shards

Shards own table fragments

Routers have table fragment references, but no data



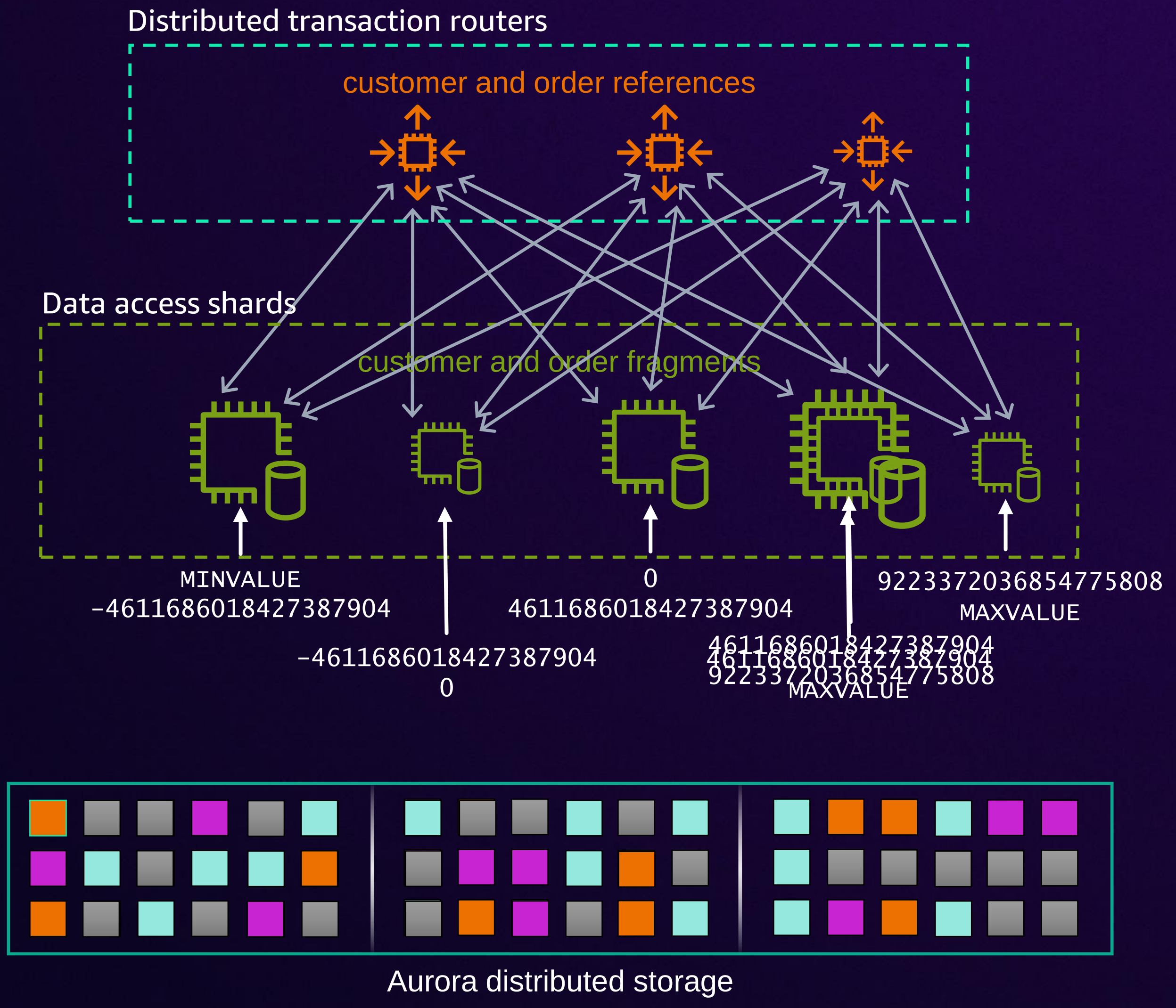
Horizontal scale out

“Shard split” occurs due to utilization or storage size

Collocated key ranges are moved together

Leverages Aurora storage level cloning and replication

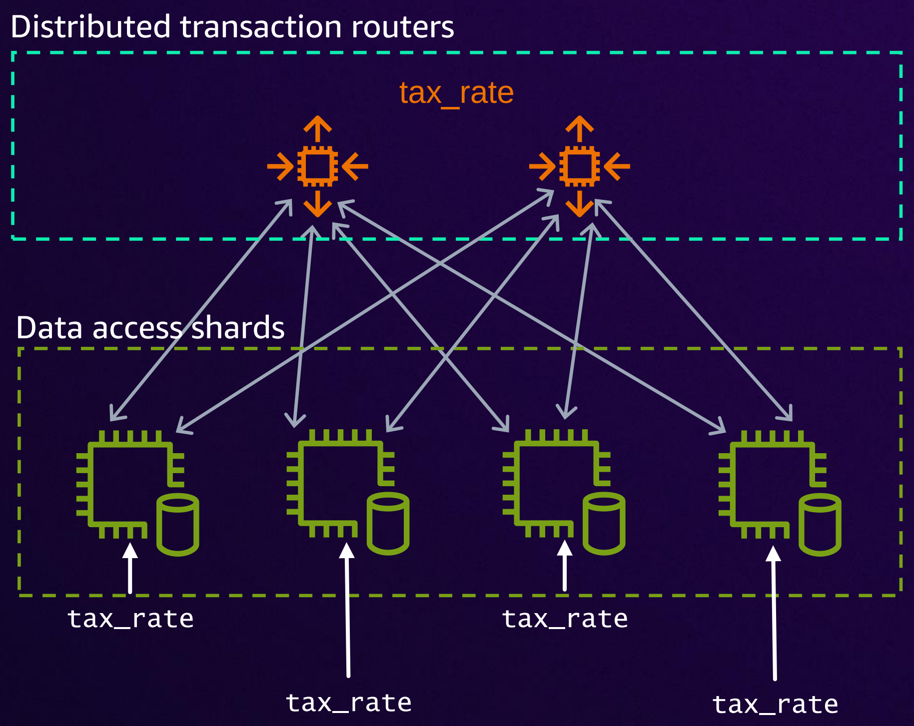
Routers can be added



Reference tables

```sql
SET rds_aurora.limitless_create_table_mode='reference';

CREATE TABLE tax_rate (
        tax_rate_id INT PRIMARY KEY NOT NULL,
        city        TEXT NOT NULL,
        state       TEXT,
        country     TEXT NOT NULL,
        tax_rate    DOUBLE NOT NULL
        );
```



Strongly consistent (ACID writes)

Enables join pushdown

Frequent read/join, infrequent write

Transactions

Transaction design goal

PostgreSQL semantics for **READ COMMITED** and **REPEATABLE READ**

…with a consistent view as in a single system

aws

Challenges in a distributed database

Coordination limits scalability

Transaction scope unknown until commit

Query fragments execute at different times

Maintain order

Consistent restores

Limitless ACID properties

PostgreSQL *read committed, repeatable read (SI)*

- Same semantics as single node PostgreSQL
- External consistency

Multi-shard writes are atomic

- All participants commit at same effective time

DDLs are transactional and strongly consistent

- DDLs are always RC, same as PostgreSQL
- Can be initiated from any SQL connection

Single system backup

- Point-in-time-restore fully consistent

PostgreSQL MVCC snapshots simplified

PostgreSQL snapshots are taken as of ***now***

- Knows the current XID . . . a transaction start sequence number
- Records the XID of all running transactions
- Compares this with XID recorded on the tuples it is scanning
    - Tuple header has XID of inserter, deleter

Illustrative only - full implementation is more complex

PostgreSQL snapshots simplified

Invisible Tuples

- Written by an XID > snapshot XID
- Written by an XID that isn’t committed
- Written by an XID running when the snapshot was taken

Visible Tuples

- Your own uncommitted writes
- Everything else

Illustrative only - full implementation is more complex

Limitless does snapshots as of *then*

Transaction router establishes the snapshot time ***now***

Router passes this time to shards along with query fragment

Shards create their local snapshot as of ***then***

Multi-shard snapshots will use the same time on all shards…***then***

Is transaction commit time earlier than snapshot time? Visible.

Distributed clocks

Historically, relying on wall time between multiple systems doesn't work

Innovation and major investment in time infrastructure makes this possible in AWS

Database algorithms built on highly reliable, drift bounded clocks

Extremely scalable design

Bounded clocks in EC2

Amazon Time Sync Service provides high quality time to EC2 instances

ClockBound is an open source daemon that provides *{earliest,latest}* uncertainty bounds, typically < 1msec

Actual true time guaranteed between *{earliest,latest}*

New architecture has clock source on Nitro card, <50 usec uncertainty

Repeatable read – distributed (with clocks)

Transaction T1

1) router gets time t100
2) execute on shard w/cust_id 619 using snapshot@t100

```
BEGIN TRANSACTION ISOLATION LEVEL
REPEATABLE READ;

SELECT status FROM order WHERE cust_id
= 619 and order_id = 61890340;
filling
```

1) execute on shard w/cust_id 801 using snapshot@t100

```
SELECT status FROM order WHERE cust_id
WHERE cust_id = 801 and order_id =
80044011;
filling
```

Transaction T2

1) router gets time t103
2) execute on shard w/cust_id 801 using snapshot@t103

```
BEGIN;

SELECT status FROM order WHERE cust_id
WHERE cust_id = 801 and order_id =
80044011;
filling

UPDATE order
cust_id = 80
COMMIT;
```

1) router uses 1PC on shard
2) shard assigns commit@t110
3) acks commit when
   a) writes durable on disk
   b) earliest possible time > t110

Transaction T3

```
SELECT status FROM order WHERE cust_id
WHERE cust_id = 801 and order_id =
80044011;
shipped
```

1) router gets time t125
2) execute on shard w/cust_id 801 using snapshot@t125

Multi-shard writes

Build on modified two-phase commit protocol

Router coordinates distributed commit

All shards will commit the transaction with the same commit time

Commit latency is roughly 2–3x single shard commit

Transactions conclusion

Same RC/RR semantics as PostgreSQL

All reads are consistent, w/o quorum, even on failover

Commits w/single shard writes scale linearly (millions/sec)

Distributed commits are atomic

Queries & Performance

Fundamentally Aurora PostgreSQL

PostgreSQL wire compatible

PostgreSQL parser and semantics

Broad surface area coverage

Selected extensions

Query execution basics

PostgreSQL foreign tables foundation

Enhancements in core engine

A custom foreign data wrapper

Query flow

Router

1. Receives query from client
2. Plans what can be sent to shards and any joins that must be done
3. Sends partial queries to shards with transaction context

Shard

4. Receives partial query from router
5. Plans local joins and scans
6. Execute and sent results to router

7. Router does final joins, filters, and aggregations as necessary

Locality is key to performance

Lowest latency and best scalability when locality is maintained

Push execution close to the data

Reduce messaging, leverages caching

Collocated and reference tables are key building blocks

Single shard optimization

Best performance when router determines query goes to a single shard

```
postgres_limitless=> EXPLAIN (VERBOSE, COSTS OFF) SELECT * FROM customers WHERE
customer_id = 100;

                       QUERY PLAN
---------------------------------------------------------
 Foreign Scan
   Output: customer_id, other_id, customer_name, balance
   Remote SQL:  SELECT customer_id,
     other_id,
     customer_name,
     balance
    FROM public.customers
   WHERE (customer_id = 100)
 Single Shard Optimized
```

Single shard join pushdown

```
postgres_limitless=> EXPLAIN (VERBOSE, COSTS OFF) SELECT * FROM orders
    LEFT JOIN zipcodes ON orders.zipcode_id = zipcodes.zipcode_id
    WHERE customer_id = 11;

                                              QUERY PLAN
------------------------------------------------------------------------------------------
Foreign Scan
   Output: customer_id, order_id, zipcode_id, customer_name, balance, zipcodes.zipcode_id,
zipcodes.city
   Remote SQL:  SELECT orders.customer_id,
     orders.order_id,
     orders.zipcode_id,
     orders.customer_name,
     orders.balance,
     zipcodes.zipcode_id,
     zipcodes.city
    FROM (public.orders
      LEFT JOIN public.zipcodes ON ((orders.zipcode_id = zipcodes.zipcode_id)))
   WHERE (orders.customer_id = 11)
 Single Shard Optimized
```

Function distribution

Collection of statements that operate on the same key value can be wrapped in a function

Significant improvement in latency and reduction in router CPU

See "Function distribution" and `limitless_distribute_function` in the docs

Parallel operations

Parallel operations speed up via multi-shard execution

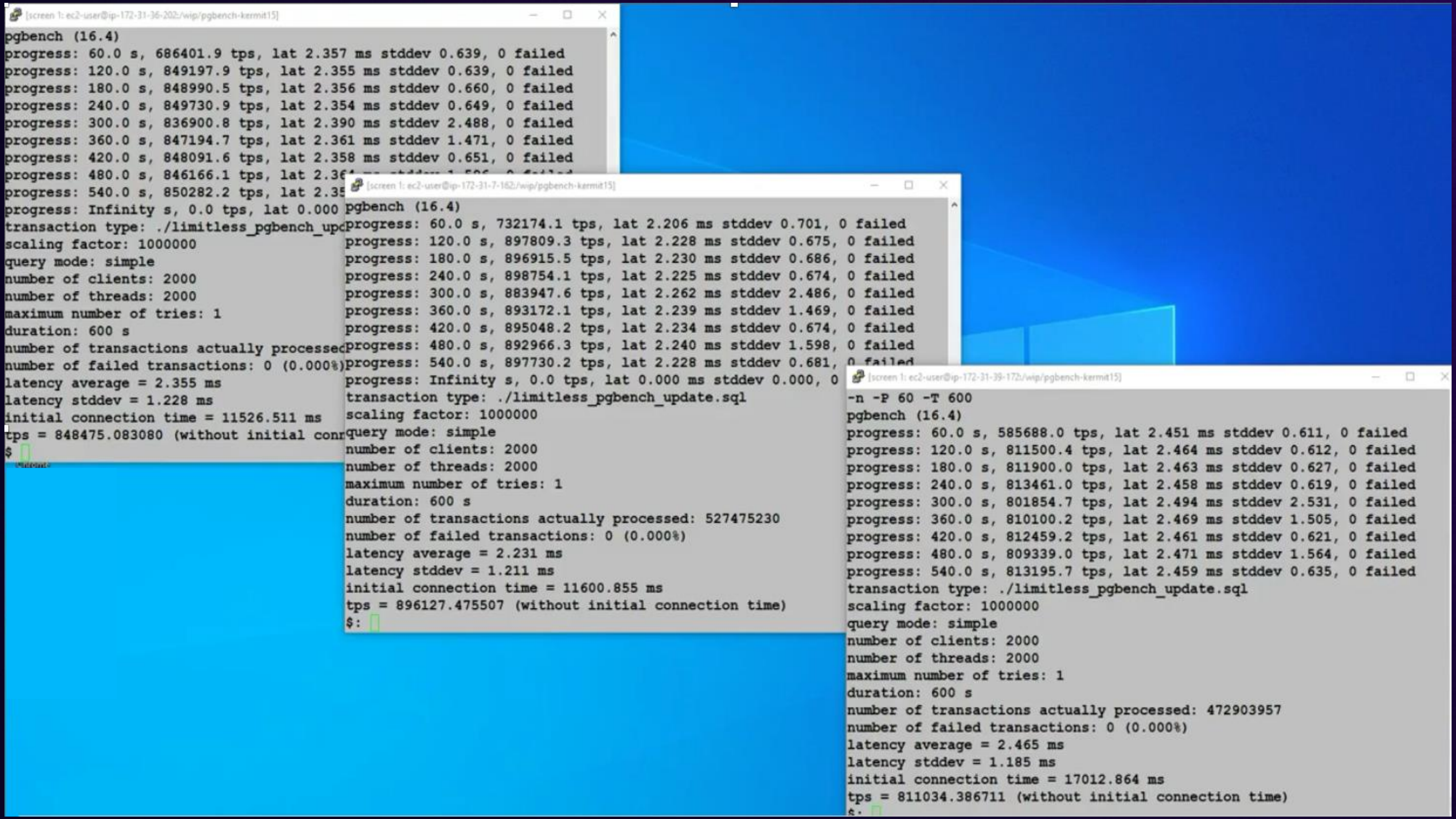
Some examples:

- Create index
- Analyze
- Vacuum
- Aggregates (count, sum, min, max)

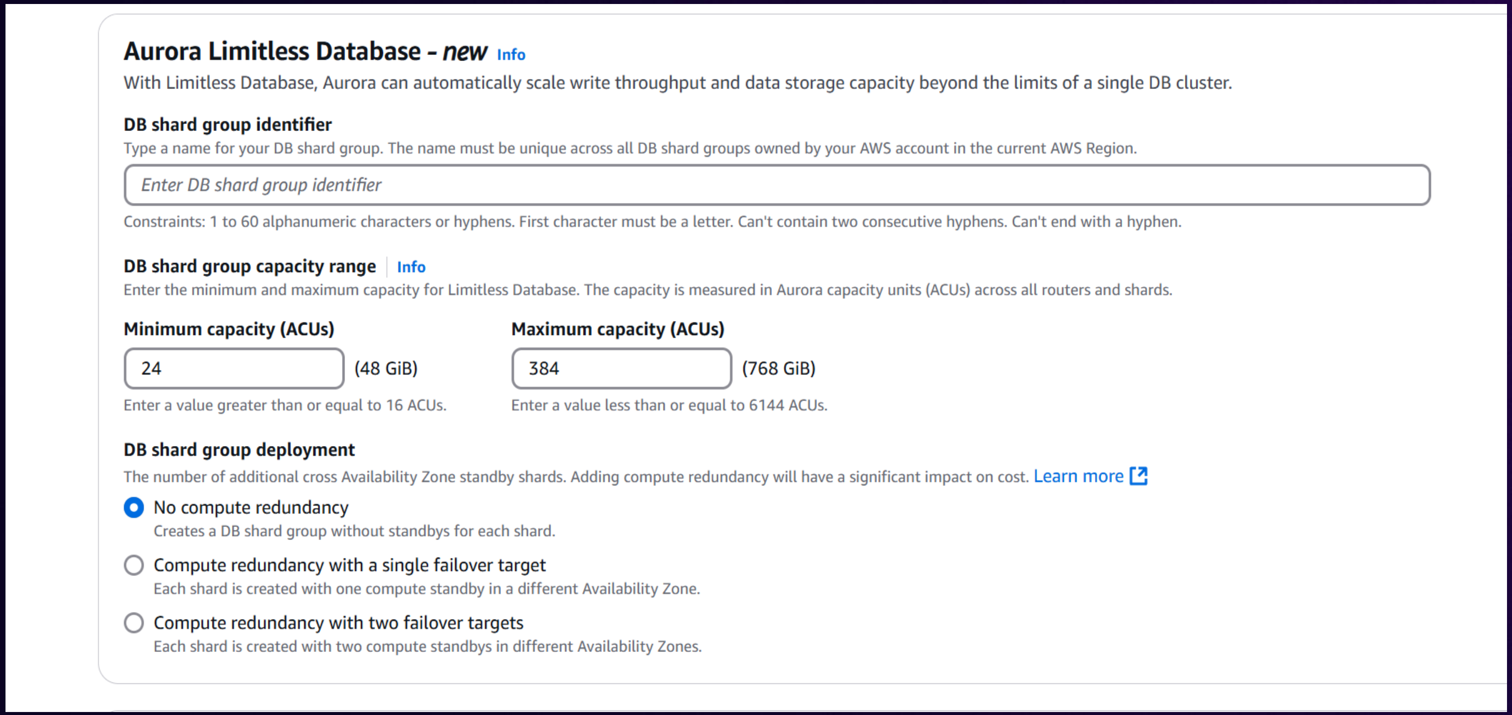
Low latency at scale

Experiment in us-east-1

Three big client drivers distributing random updates across 100B rows



Get started today!




```
aws rds create-db-shard-group
        --db-cluster-identifier proddb
        --db-shard-group-identifier proddb-sg
        --min-acu 150
        --max-acu 600
        --compute-redundancy 2
```

Summary

- ✓ Challenges for scaling
- ✓ Scales to millions of write transactions per second
- ✓ Manages petabytes of data
- ✓ Scalable architecture
- ✓ Data distribution
- ✓ Query and transactions

Get started today with AWS console: https://console.aws.amazon.com/
Learn more: https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/limitless.html

Related sessions

Session ID	Session Title
DAT416	Scalable database solutions with Aurora PostgreSQL Limitless Database
DAT316	Build scalable and cost-optimized apps with Amazon Aurora Serverless
DAT424	Get started with the latest Amazon Aurora innovations
DAT405	Deep dive into Amazon Aurora and its innovations
DAT304	Amazon Aurora HA and DR design patterns for global resilience

Thank you!

Please complete the session survey in the mobile app

Anum Jang Sher
anujangs@amazon.com

David Wein
dcw@amazon.com