RDS, Aurora & ElastiCache

Amazon RDS Overview

- **RDS** stands for **Relational Database Service**
- It’s a managed DB service for databases that use **SQL** as a query language
- It allows you to create databases in the cloud that are managed by AWS, including:
  - **PostgreSQL**
  - **MySQL**
  - **MariaDB**
  - **Oracle**
  - **Microsoft SQL Server**
  - **IBM DB2**
  - **Aurora** (AWS Proprietary database)

Advantage over using RDS versus deploying DB on EC2

- **RDS is a managed service**, which provides:
  - Automated provisioning and OS patching
  - Continuous backups and restore to specific timestamp (Point in Time Restore)
  - Monitoring dashboards
  - Read replicas for improved read performance
  - Multi-AZ setup for Disaster Recovery (DR)
  - Maintenance windows for upgrades
  - Scaling capability (both vertical and horizontal)
  - Storage backed by EBS
- **BUT** you can't SSH into your instances

RDS – Storage Auto Scaling

- Helps you increase storage on your RDS DB instance dynamically
- When RDS detects you are running out of free database storage, it scales automatically
- Avoid manually scaling your database storage
- You have to set **Maximum Storage Threshold** (maximum limit for DB storage)
- Automatically modifies storage if:
  - Free storage is less than 10% of allocated storage
  - Low-storage condition lasts at least 5 minutes
  - 6 hours have passed since the last modification
- Useful for applications with **unpredictable workloads**
- Supports all RDS database engines

Architecture Explanation (example)

Users send requests to an **Application**, which performs **read/write** operations on **Amazon RDS**.
Amazon RDS automatically scales the **underlying storage layer** as needed, based on usage thresholds.

RDS Read Replicas for Read Scalability

- Up to **15 Read Replicas**
- Can be deployed **within an AZ**, **across AZs**, or **cross-region**
- Replication is **asynchronous (ASYNC)** — reads are **eventually consistent**
- Replicas can be **promoted** to become their own standalone database
- Applications must **update the connection string** to use read replicas

Architecture Explanation (example)

The **Application** performs:

- **Writes** only on the **main RDS instance**
- **Reads** on any of the **RDS read replicas**

Replication between the main RDS instance and its replicas happens asynchronously, enabling the system to **scale read operations horizontally**.

RDS Read Replicas – Use Cases

- You have a production database that is taking on normal load
- You want to run a reporting application to perform some analytics
- You create a **Read Replica** to handle the new workload
- The **production application** remains unaffected
- **Read replicas are used only for SELECT (read-only) operations**
  - Not suitable for **INSERT**, **UPDATE**, or **DELETE**

Architecture Explanation (example)

- The **Production Application** performs **writes and reads** on the main RDS instance.
- The **Reporting Application** performs **reads** on the RDS Read Replica.
- **Asynchronous replication** ensures the replica gets updates from the main DB.

RDS Read Replicas – Network Cost

- In AWS, there is a **network cost** when data is transferred **between Availability Zones (AZs)**
- For **RDS Read Replicas within the same region**, this fee **does not apply**

Replication Cost Comparison

- **Same Region, Different AZs (e.g. us-east-1a → us-east-1b)**:

  - ASYNC replication
  - **No network cost** (Free)

- **Cross-Region (e.g. us-east-1a → eu-west-1b)**:

  - ASYNC replication
  - **Network cost applies** ($$$)

Visual Summary (example)

- Replicating between AZs **within the same region** is cost-free.
- Replicating **across regions** introduces additional charges.

RDS Multi AZ (Disaster Recovery)

- Uses **synchronous (SYNC) replication**
- A **single DNS name** is used – automatic application failover to standby instance
- Increases **availability**
- Provides automatic failover in case of:
    - Availability Zone (AZ) failure
    - Network failure
    - Instance failure
    - Storage failure
- **No manual intervention** required in applications
- **Not used for scaling**

> ***Note:*** *Read Replicas can be set up as **Multi-AZ** for Disaster Recovery (DR)*

Architecture Explanation (example)

- The **Application** uses one DNS name to access the DB.
- Writes and reads go to the **primary RDS instance (AZ A)**.
- A **standby RDS instance (AZ B)** receives data via synchronous replication.
- In case of failure, DNS automatically redirects to the standby.

RDS – From Single-AZ to Multi-AZ

- **Zero downtime operation** (no need to stop the database)
- You simply click on **"modify"** for the database in the AWS Console
- The following steps happen internally:
    - A **snapshot** of the database is taken
    - A new DB is **restored from the snapshot** in a different Availability Zone (AZ)
    - **Synchronous replication** is established between the two databases

Architecture Explanation (example)

- The **RDS DB instance (M)** creates a **DB snapshot**
- The snapshot is used to **restore a new standby DB (S)** in another AZ
- **SYNC replication** begins between the two, enabling Multi-AZ high availability

RDS Custom

- Managed **Oracle** and **Microsoft SQL Server** databases with OS and database customization

- **RDS** automates:

    - Setup
    - Operation
    - Scaling of the database in AWS

- **RDS Custom** gives you access to the **underlying database and OS**, so you can:

    - Configure settings
    - Install patches
    - Enable native features
    - Access the underlying EC2 instance using **SSH** or **SSM Session Manager**

- You must **deactivate Automation Mode** to perform customizations
  (It's recommended to take a DB snapshot beforehand)

RDS vs RDS Custom

- **RDS**: AWS manages the entire DB and OS
- **RDS Custom**: You get **full admin access** to the OS and DB

Amazon Aurora

- Aurora is a **proprietary technology** from AWS (not open source)
- **PostgreSQL** and **MySQL** are both supported as Aurora DB engines
  (you can use the same drivers as for Postgres or MySQL)
- Aurora is **AWS cloud optimized** and claims:
  - **5x performance** over MySQL on RDS
  - **3x performance** over Postgres on RDS
- **Aurora storage** automatically grows in increments of **10 GB**, up to **128 TB**
- Supports up to **15 read replicas**, with **sub-10 ms replication lag**
- **Instantaneous failover** — built-in **High Availability (HA)**
- Aurora is **20% more expensive** than standard RDS, but it's more efficient

Aurora High Availability and Read Scaling

- **6 copies** of your data are stored across **3 Availability Zones (AZs)**:
  - **4 out of 6 copies** are required for **writes**
  - **3 out of 6 copies** are required for **reads**
  - **Self-healing** via peer-to-peer replication
  - **Storage is striped** across hundreds of volumes
- **One Aurora instance** handles **writes** (master)
- **Automated failover** for the master occurs in **under 30 seconds**
- The system supports:
  - **1 master** + **up to 15 Aurora Read Replicas** for read operations
- Includes **support for Cross Region Replication**

Architecture Explanation (example)

- Master and read replicas are spread across **AZ1, AZ2, and AZ3**
- All instances use a **shared storage volume**, which supports:
  - Replication
  - Self-healing
  - Auto-expansion
- Writes go to the **master**, while all other instances serve **reads**

Aurora DB Cluster

- **Writer Endpoint**: points to the **master** instance (used for write operations)
- **Reader Endpoint**: provides **connection load balancing** across all read replicas
- The Aurora cluster consists of:
  - **1 master (M)** instance for **writes**

  - **Multiple read replicas (R)** for **reads**
  - All instances share the same **storage volume**

- The **shared storage**:

  - Is **auto-expanding** from **10 GB up to 128 TB**
  - Supports **concurrent reads** from all replicas
  - Allows **auto scaling** of read replicas based on demand

Features of Aurora

- **Automatic fail-over**
- **Backup and Recovery**
- **Isolation and security**
- **Industry compliance**
- **Push-button scaling**
- **Automated Patching with Zero Downtime**
- **Advanced Monitoring**
- **Routine Maintenance**
- **Backtrack**: restore data at any point in time without using backups

Aurora Replicas – Auto Scaling

- **Writer Endpoint**: used for all **write** operations, pointing to the **master** Aurora instance

- **Reader Endpoint**: used by the client to perform **read** operations

- When **read traffic increases**, Aurora automatically **scales the number of read replicas**

  - Triggered by **CPU usage** or other metrics
  - New replicas are added to handle the load via the **Endpoint Extended**

- All instances (writer and readers) share a **common storage volume**, which:

  - Is **auto-expanding**
  - Provides **low-latency access** to all replicas

- This architecture ensures **high availability**, **load balancing**, and **elastic read capacity**

Aurora – Custom Endpoints

- You can **define a subset of Aurora instances** as a **Custom Endpoint**
- Useful to run **analytical queries** on specific replicas without affecting others
- Once Custom Endpoints are defined, the **Reader Endpoint is typically not used**

Example Use Case

- Writer Endpoint: handles all **write operations** (points to the master)
- Reader Endpoint: handles general **read traffic**
- Custom Endpoint: used by the client to direct **analytical queries** to selected replicas (e.g., larger instance types like `db.r5.2xlarge`)

Benefits

- **Query isolation**: analytical workloads do not affect normal read operations
- **Instance targeting**: route traffic based on instance type or workload

- **Better resource utilization** and **load separation**

All instances share the same **storage volume** and benefit from Aurora's **auto-scaling and replication**.

Aurora Serverless

- **Automated database instantiation** and **auto-scaling** based on actual usage
- Ideal for:
  - **Infrequent**
  - **Intermittent**
  - **Unpredictable** workloads
- No need for **capacity planning**
- **Pay-per-second** billing model — potentially more **cost-effective**

Architecture Explanation (diagram)

- The **client** connects to a **Proxy Fleet** managed by Aurora
- The proxy dynamically routes requests to **Aurora instances**
- Aurora instances automatically **scale in/out** based on demand
- All instances share the same **underlying storage volume**

Global Aurora

Aurora Cross Region Read Replicas

- Useful for **disaster recovery**
- **Simple to implement**

Aurora Global Database (recommended)

- **1 Primary Region** (supports **read/write**)
- Up to **5 secondary regions** (read-only), with **replication lag < 1 second**
- Up to **16 read replicas** per secondary region
- Helps **reduce latency** for global applications
- **Disaster recovery**: another region can be promoted with **RTO < 1 minute**
- **Cross-region replication** is typically **< 1 second**

Architecture Explanation (diagram)

- The **Primary Region** (e.g., `us-east-1`) handles full read/write operations
- One or more **Secondary Regions** (e.g., `eu-west-1`) receive replicated data
- Applications in secondary regions perform **read-only** operations
- Data is continuously replicated from the primary to the secondary

Aurora Machine Learning

- Enables you to add **ML-based predictions** to your applications via **SQL**
- Provides a **simple, optimized, and secure** integration between Aurora and AWS ML services

Supported Services

- **Amazon SageMaker**: use with any ML model
- **Amazon Comprehend**: for sentiment analysis

Key Points

- You **don't need prior ML experience**

- Queries are written in **standard SQL**
- Aurora handles the **invocation of ML models** and returns predictions

Use Cases

- Fraud detection
- Ads targeting
- Sentiment analysis
- Product recommendations

Architecture Overview (diagram)

- The application sends a **SQL query** to Aurora (e.g. "what to recommend?")
- Aurora sends relevant **data** to the ML service (e.g. user profile, shopping history)
- Aurora receives and returns **predictions** as part of the query results

Babelfish for Aurora PostgreSQL

- **Babelfish** allows **Aurora PostgreSQL** to understand commands written for **Microsoft SQL Server** (e.g., **T-SQL**)
- Enables **MS SQL Server-based applications** to run on Aurora PostgreSQL with **minimal or no code changes**
- Applications can continue using the **same MS SQL Server client drivers**
- Useful for **migrating** existing SQL Server databases to Aurora PostgreSQL using:
  - **AWS SCT** (Schema Conversion Tool)
  - **AWS DMS** (Database Migration Service)

Key Benefits

- Reduces effort and complexity in migration
- Applications can interact using **T-SQL** or **PL/pgSQL**, depending on the driver
- Facilitates **faster adoption** of Aurora PostgreSQL for teams used to SQL Server

Architecture Overview (diagram)

- SQL Server apps send T-SQL queries via their original drivers
- Babelfish intercepts and processes T-SQL within Aurora PostgreSQL
- PostgreSQL apps continue using PL/pgSQL natively
- This enables a **smooth transition** with support for both SQL dialects

RDS Backups

Automated Backups

- Daily full backup of the database (during the backup window)
- Transaction logs are backed up by RDS every 5 minutes
- This enables **point-in-time restore** (from the oldest backup to 5 minutes ago)
- Retention period: **1 to 35 days**
  - Set to `0` to disable automated backups

Manual DB Snapshots

- Manually triggered by the user
- Backup is retained **as long as you want**

Cost Optimization Tip

- If an **RDS instance is stopped**, you **still pay for storage**
- If you plan to stop it for a long time, **snapshot and restore** instead

Aurora Backups

Automated Backups

- Retention period: **1 to 35 days** (cannot be disabled)
- Supports **point-in-time recovery** within that timeframe

Manual DB Snapshots

- **Manually triggered** by the user
- Backup is retained **as long as you want**

RDS & Aurora Restore Options

- **Restoring a RDS / Aurora backup or a snapshot** creates a new database

Restoring MySQL RDS database from S3

- Create a backup of your on-premises database
- Store it on Amazon S3 (object storage)
- Restore the backup file onto a new RDS instance running MySQL

Restoring MySQL Aurora cluster from S3

- Create a backup of your on-premises database using Percona XtraBackup
- Store the backup file on Amazon S3
- Restore the backup file onto a new Aurora cluster running MySQL

Aurora Database Cloning

- Create a new Aurora DB Cluster from an existing one
- Faster than snapshot & restore
- Uses *copy-on-write* protocol
  - Initially, the new DB cluster uses the same data volume as the original DB cluster (fast and efficient — no copying is needed)
  - When updates are made to the new DB cluster data, then additional storage is allocated and data is copied to be separated
- Very fast & cost-effective
- Useful to create a "staging" database from a "production" database without impacting the production database

RDS & Aurora Security

- **At-rest encryption**:
  - Database master & replicas encryption using AWS KMS – must be defined at launch time
  - If the master is not encrypted, the read replicas cannot be encrypted
  - To encrypt an un-encrypted database, go through a DB snapshot & restore as encrypted
- **In-flight encryption**: TLS-ready by default, use the AWS TLS root certificates client-side
- **IAM Authentication**: IAM roles to connect to your database (instead of username/password)
- **Security Groups**: Control network access to your RDS / Aurora DB

- **No SSH available** except on RDS Custom

- **Audit Logs** can be enabled and sent to CloudWatch Logs for longer retention

Amazon RDS Proxy

- Fully managed database proxy for RDS
- Allows apps to pool and share DB connections established with the database
- **Improves database efficiency** by reducing the stress on database resources (e.g., CPU, RAM) and minimizes open connections (and timeouts)
- Serverless, autoscaling, highly available (multi-AZ)
- **Reduces RDS & Aurora failover time by up to 66%**
- Supports:
  - RDS: MySQL, PostgreSQL, MariaDB, MS SQL Server
  - Aurora: MySQL, PostgreSQL

- No code changes required for most apps
- Enforces **IAM Authentication** for DB, securely storing credentials in AWS Secrets Manager
- **RDS Proxy is never publicly accessible** (must be accessed from within a VPC)

Amazon ElastiCache Overview

- The same way RDS is to get managed Relational Databases…
- ElastiCache is to get managed Redis or Memcached
- Caches are in-memory databases with really high performance, low latency
- Helps reduce load off of databases for read intensive workloads
- Helps make your application stateless
- AWS takes care of OS maintenance / patching, optimizations, setup, configuration, monitoring, failure recovery and backups
- *Using ElastiCache involves heavy application code changes*

ElastiCache Solution Architecture – DB Cache

- Applications queries ElastiCache, if not available, get from RDS and store in ElastiCache.
- Helps relieve load in RDS
- Cache must have an invalidation strategy to make sure only the most current data is used in there.

Diagram Flow

1. Application queries ElastiCache:
   - If **cache hit** → return result from ElastiCache.
   - If **cache miss** → fetch from RDS, then:
     - Write result to ElastiCache.
     - Return result to application.

ElastiCache Solution Architecture – User Session Store

- User logs into any instance of the application.
- The application writes the session data into ElastiCache.
- The user accesses another instance of the application.
- That instance retrieves the session data from ElastiCache, allowing the user to remain logged in.

Flow Summary

1. **Write session** → User logs in → session data saved in ElastiCache.
2. **Retrieve session** → User accesses another app instance → session fetched from ElastiCache.

ElastiCache – Redis vs Memcached

Redis

- **Multi-AZ** with Auto-Failover
- **Read Replicas** to scale reads and ensure high availability
- **Data Durability** using AOF persistence
- **Backup and restore** features
- Supports **Sets** and **Sorted Sets**

Memcached

- **Multi-node** for partitioning of data (**sharding**)
- **No high availability** (no replication)
- **Non-persistent**
- **Backup and restore** (serverless)
- **Multi-threaded architecture**

ElastiCache – Cache Security

- ElastiCache supports **IAM Authentication for Redis**
- IAM policies on ElastiCache are only used for **AWS API-level security**

Redis AUTH

- You can set a “password/token” when you create a Redis cluster
- This provides an **extra level of security** for your cache (in addition to security groups)
- Supports **SSL in-flight encryption**

Memcached

- Supports **SASL-based authentication** (advanced)

Patterns for ElastiCache

- **Lazy Loading**: all the read data is cached; data can become stale in cache.
- **Write Through**: adds or updates data in the cache when written to a DB (no stale data).
- **Session Store**: stores temporary session data in a cache (using TTL features).

> *Quote: There are only two hard things in Computer Science: cache invalidation and naming things*

ElastiCache – Redis Use Case

- Gaming Leaderboards are computationally complex
- **Redis Sorted Sets** guarantee both uniqueness and element ordering
- Each time a new element is added, it's ranked in real time, then added in the correct order