MODULE 8 · STUDY NOTES

Databases

Amazon RDS, DynamoDB, Redshift, and Aurora — managed databases and how to pick the right one

Databases are the link between recording data and profiting from it, and cloud databases deliver that link without the server racks, patch cycles, and backup scripts of the on-premises world. This module tours AWS's four foundational database services — Amazon RDS for managed relational databases, DynamoDB for NoSQL at any scale, Redshift for data warehousing and analytics, and Aurora for enterprise-grade MySQL/PostgreSQL — and builds the judgment the exam actually tests: managed vs. unmanaged, SQL vs. NoSQL, availability (Multi-AZ) vs. read scaling (read replicas), and which database is the right tool for a given workload.

LEARNING OBJECTIVES

- Explain Amazon Relational Database Service (Amazon RDS)
- Identify the functionality in Amazon RDS
- Explain Amazon DynamoDB
- Identify the functionality in Amazon DynamoDB
- Explain Amazon Redshift
- Explain Amazon Aurora
- Perform tasks in an RDS database, such as launching, configuring, and interacting

8.1 Amazon Relational Database Service (Amazon RDS)

AWS solutions fall into two categories. An **unmanaged** service is provisioned in discrete portions you specify, and *you* decide how it responds to load changes, errors, and resource failures — a web server on an EC2 instance will not scale for traffic or replace unhealthy instances unless you wire up a scaling solution. The payoff is fine-tuned control. A **managed** service still needs configuring (you create an S3 bucket and set its permissions), but scaling, fault tolerance, and availability are handled automatically and internally — a static website hosted on S3 simply inherits them.

Run your own relational database and a long administrative to-do list follows: server maintenance and energy footprint, software installation and patches, database backups, high availability, planning for scalability, data security, and OS installation and patching — each demanding time and expertise. **Amazon RDS** is the managed answer: a service that sets up, operates, and scales a relational database in the cloud with cost-efficient, resizable capacity while automating those time-consuming tasks. Your primary focus becomes your data and optimizing your application.

TABLE 8.1 Who carries which responsibility – on-premises vs. EC2 vs. RDS

RESPONSIBILITY	ON-PREMISES DB	DB ON EC2	DB ON RDS/ AURORA
Application optimization	You	You	You
Scaling · high availability · backups	You	You	AWS
DB software installs and patches	You	You	AWS
OS installs and patches	You	You	AWS
Server maintenance · rack and stack	You	AWS	AWS
Power, HVAC, network	You	AWS	AWS

The basic building block of RDS is the **DB instance** — an isolated database environment that can contain multiple user-created databases and is reached with the same tools you'd use against a standalone database. Its compute resources come from the **DB instance class** (CPU, memory, network performance); its storage comes from the disk type — **magnetic**, **General Purpose (SSD)**, or **Provisioned IOPS** — so you tune performance and price independently. The first decision when creating an instance is the **database engine**: MySQL, IBM DB2, Microsoft SQL Server, PostgreSQL, MariaDB, or Oracle (the section's takeaway list counts **Amazon Aurora** among the six alongside PostgreSQL, MySQL, MariaDB, Oracle, and SQL Server).

RDS runs happily inside a **VPC**: you pick the IP address range, subnets, routing, and ACLs, and functionality is identical to running outside one. In practice the DB instance sits in a **private subnet**, directly accessible only to the application instances you designate. Because each subnet belongs to a single Availability Zone, choosing the subnet also chooses the physical location of your database.

> **MULTI-AZ: THE AVAILABILITY FEATURE**
>
> - RDS creates a **standby copy** of your DB instance in *another AZ* within the same VPC, then replicates transactions **synchronously** after seeding.
> - Protects during planned maintenance, DB instance failure, and AZ disruption.
> - If the main instance fails, RDS **automatically** brings the standby online as the new main; synchronous replication minimizes data loss.
> - Applications reference the database by name via the **RDS DNS endpoint** — failover requires **no application code change**.

Read replicas solve a different problem: read scaling. Available for MySQL, MariaDB, PostgreSQL, and Aurora, a read replica receives updates from the source **asynchronously**. Route read queries to the replica to cut load on the source, and scale out past the capacity of a single instance for read-heavy workloads. A replica *can* be promoted to primary, but that is a **manual** action precisely because replication is asynchronous. Replicas can also be created in a **different Region** — for disaster recovery, or to serve reads closer to users.

TABLE 8.2 Multi-AZ deployments vs. read replicas

	MULTI-AZ	READ REPLICAS
Purpose	High availability and durability	Offload and scale reads
Replication	Synchronous, to a standby in another AZ	Asynchronous, to one or more replicas
Failover	Automatic promotion; same DNS endpoint	Manual promotion
Placement	Second AZ in the same VPC	Same Region or cross-Region

RDS suits web and mobile applications needing high throughput, massive storage scalability, and high availability — with no licensing constraints, it fits variable usage patterns. Ecommerce gets a flexible, secure, low-cost store; mobile and online games get a platform where nobody has to provision, scale, or monitor database servers.

USE AMAZON RDS WHEN THE APP NEEDS	LOOK ELSEWHERE WHEN IT NEEDS
the relational sweet spot	*NoSQL — or your own engine on EC2*
▪ Complex transactions or complex queries ▪ Medium-to-high query/write rate — up to 30,000 IOPS (15,000 reads + 15,000 writes) ▪ No more than a single worker node or shard ▪ High durability	▪ Massive read/write rates (for example, 150,000 writes/second) ▪ Sharding driven by data size or throughput demands ▪ Simple GET or PUT requests a NoSQL database can handle ▪ RDBMS customization (run the engine on EC2 for full control)

TABLE 8.3 What you pay for in Amazon RDS

COST DIMENSION	WHAT DRIVES THE CHARGE
Clock-hours of service time	Resources bill while running — from instance launch until you terminate it
Database characteristics	Physical capacity: engine, size, and memory class
DB purchase type	On-Demand (per hour, no commitment) vs. Reserved (low one-time upfront payment for a 1- or 3-year term, discounted hourly rate)
Number of DB instances	Provision multiple instances for peak loads — each one bills
Storage	No charge for backup storage up to 100% of provisioned storage while the DB is active; per GB/month once terminated, and for backup beyond the provisioned amount
Requests	The number of input and output requests made to the database
Deployment type	Storage and I/O charges differ for single-AZ vs. Multi-AZ deployments
Data transfer	Inbound is **free**; outbound is tiered

LAB 5 – BUILD YOUR DB SERVER AND TALK TO IT FROM AN APP

The module's lab stands up a highly available relational backend: create a **VPC security group**, create a **DB subnet group**, launch an **RDS DB instance with Multi-AZ** high availability, configure it to permit connections from your web server, then interact with the database through a web application. The recorded console demo covers the same ground solo — configuring an RDS installation running **MySQL** and connecting with a MySQL client.

SECTION TAKEAWAYS

- RDS = set up, operate, and scale relational databases in the cloud; accessible via the **console, AWS CLI, or API calls**.
- Scale compute and storage with **no downtime**; automated redundancy and backup are available.
- Two SSD-backed storage options: one optimized for high-performance **OLTP**, one for cost-effective general-purpose use.
- Runs in your VPC for control and security, on the same infrastructure as other AWS services.

8.2 Amazon DynamoDB

A **relational database** works with structured data organized into tables, records, and columns, with well-defined relationships between tables and **SQL** as the standard interface. Its weak spots: scaling out horizontally, handling semistructured data, and the many joins normalized data can require. A **non-relational (NoSQL) database** is any database that abandons that model — designed precisely to overcome those limits, scaling out horizontally and working with unstructured and semistructured data.

RELATIONAL (SQL)	NON-RELATIONAL (NOSQL)
RDS · Aurora · Redshift ▪ Tables, records, columns; well-defined relationships ▪ SQL programming interface ▪ Hard to scale out horizontally ▪ Joins multiply for normalized data	*DynamoDB* ▪ No relational model; flexible structure ▪ Scales out horizontally ▪ Handles unstructured and semistructured data ▪ Items in one table can carry different attributes

Amazon DynamoDB is a fast, flexible NoSQL database service for applications that need **consistent, single-digit-millisecond latency at any scale**. AWS manages all the underlying data infrastructure and stores data redundantly across multiple facilities in a Region as part of its fault-tolerant architecture. You create tables and add items; DynamoDB automatically **partitions** your data and table storage to meet workload demands — there is no practical limit on items, and some production tables hold billions. Because items in the same table can have **different attributes**, you can evolve the data model as the application grows, storing new-format items beside old ones with no schema migration. All data lives on **SSDs**, and the simple query language delivers consistent low-latency performance. Throughput is yours to set: provision read/write capacity manually, or enable **auto scaling** and let DynamoDB watch the load and adjust. Rounding out the feature set: **global tables** (automatic replication across the Regions you choose), **encryption at rest**, and item **Time-to-Live (TTL)**.

KEY TERMS

Table
A collection of data.

Item
A group of attributes that is uniquely identifiable among all other items in the table.

Attribute
A fundamental data element — something that needs no further decomposition.

Partition key
A simple primary key composed of a single attribute.

Composite primary key
Partition key + sort key: two attributes together identify the item.

Query
Retrieval by primary key — exploits partitioning to locate items efficiently.

Scan
Retrieval by matching conditions on non-key attributes — flexible, but examines every item in the table.

As data grows, the table is partitioned and indexed **by the primary key**, so key design decides your performance. A **query** rides that partitioning straight to the matching items; a **scan** offers the flexibility of filtering on any attribute but pays for it by reading through the entire table.

CHOOSING A KEY

For a products table, a single attribute with uniform distribution — a **GUID** or the **product ID** — makes a clean simple primary key. For a books table, a **composite key** of *author* (partition) + *title* (sort) uniquely identifies each book **and** lets a frequent access pattern — "all books by this author" — run as an efficient query instead of a scan.

COMMON PITFALL

Scan feels convenient, but it touches every item in the table to find matches on non-key attributes. Design your keys around your access patterns so the hot paths run as **queries**.

SECTION TAKEAWAYS

- Runs **exclusively on SSDs**; supports **document and key-value** store models.
- **Global tables** replicate tables automatically across your choice of Regions — availability and business continuity with no replication plumbing.
- Great fit for **mobile, web, gaming, adtech, and IoT** — and for latency-sensitive apps, since query performance stays predictable even on huge tables.
- Accessible via the console, AWS CLI, and API calls; consistent single-digit-ms latency at any scale; **no limits on table size or throughput**.
- The recorded console demo creates a table, adds data, and queries it — including from the AWS CLI.

8.3 Amazon Redshift

Analytics matters to every business, but building a data warehouse the traditional way is complex and expensive — months of setup and significant capital. **Amazon Redshift** is a fast, fully managed data warehouse that is simple and cost-effective to set up, use, and scale. It runs **complex analytic queries against petabytes of structured data** using sophisticated query optimization, **columnar storage** on high-performance local disks, and **massively parallel** data processing — with most results returning in seconds, through **standard SQL** and the BI tools you already use.

The architecture has two node roles. The **leader node** manages communication with client programs and with the compute nodes: it parses queries, develops execution plans, compiles code for each plan element, and assigns that code out. The **compute nodes** run the compiled code and send intermediate results back to the leader for final aggregation. As with other AWS services, you pay for what you use — starting around **25 cents per hour**, and at scale roughly **$1,000 per terabyte per year** (3-year partial upfront Reserved Instance pricing). The **Redshift Spectrum** feature extends queries to **exabytes of data directly in Amazon S3**, no loading required.

> **AUTOMATION, SCALING, SECURITY, COMPATIBILITY**
>
> - Most common administrative tasks — managing, monitoring, scaling the cluster — can be automated, so you focus on the data.
> - Scale up or down with a few clicks and **no downtime**: Redshift adds nodes to the cluster and redistributes data for maximum performance.
> - Security is built in: strong **encryption at rest and in transit** — you only need to enable it. The cluster is also monitored and backed up automatically for easy restore.
> - Standard SQL plus high-performance **JDBC and ODBC** connectors — bring your own SQL clients and BI tools.

TABLE 8.4 Amazon Redshift use cases

USE CASE	WHY REDSHIFT FITS
Enterprise data warehouse	Migrate at a pace you're comfortable with; experiment without large upfront cost, commitment, or IT procurement cycles; respond faster to business needs
Big data	Low price point puts a warehouse within reach of smaller customers; the managed service absorbs deployment and maintenance, so teams focus on data rather than database administration
Software as a service	Scale capacity as demand grows; add analytic functionality to applications; some operators run a cluster per customer with tagging to manage SLAs and billing; reduces hardware and software costs

> **COMMON PITFALL**
>
> When a question says *analyze*, *data warehouse*, or *BI tools*, the answer is **Redshift**. When it describes a transactional application workload, Redshift is the wrong pick — that's RDS, Aurora, or DynamoDB territory.

8.4 Amazon Aurora

Amazon Aurora is a MySQL- and PostgreSQL-compatible relational database **built for the cloud**, combining the performance and availability of high-end commercial databases with the simplicity and cost-effectiveness of open source. It can reduce database costs while *improving* reliability and availability, and as a fully managed service it automates the time sinks: **provisioning, patching, backup, recovery, failure detection, and repair**.

SERVICE BENEFITS

- Highly available, with a fast **distributed storage subsystem**.
- Straightforward to set up; speaks standard **SQL**.
- **Drop-in compatibility** with MySQL and PostgreSQL engines — existing database tools work with little or no change.
- **Pay-as-you-go** — only for the services and features you use.
- Migration help built in: integrates with **AWS Database Migration Service (DMS)** and the **AWS Schema Conversion Tool**.

Why pick Aurora over, say, MySQL on RDS? Mostly its **high availability and resilient design**. Aurora stores **multiple copies of your data across multiple Availability Zones** and **continuously backs up to Amazon S3**. Up to **15 read replicas** reduce the possibility of losing data — and it is designed for **instant crash recovery** if the primary becomes unhealthy: after a crash, Aurora does *not* replay the redo log from the last checkpoint (it performs that work on every read operation), which cuts restart time to **under 60 seconds** in most cases. The **buffer cache is moved out of the database process**, so it is available immediately at restart — no throttling access while a cache slowly repopulates to avoid brownouts.

SECTION TAKEAWAYS

- High performance and scalability; high availability and durability; fully managed **by Amazon RDS**.
- Multiple levels of security: **network isolation** with Amazon VPC, **encryption at rest** with keys you create and control in AWS KMS, and **SSL encryption in transit**.
- Fault-tolerant, self-healing storage; data replicated across AZs with continuous backup to S3.
- Compatible with existing MySQL and PostgreSQL databases, adding compatibility for new releases regularly.

COMMON PITFALL

Aurora shows up two ways on the exam: as its own service and as one of RDS's engines. Either way it is **relational** — if the keyword is *NoSQL*, the answer is DynamoDB, never Aurora.

8.5 The right tool for the right job

The cloud keeps driving down the cost of storage and compute, and a new generation of applications has emerged with a new set of database requirements: store **terabytes to petabytes** of new data types, serve it with **millisecond latency**, process **millions of requests per second**, and scale to **millions of users anywhere in the world**. No single engine does all of that — you need relational *and* non-relational databases, each **purpose-built** for the specific needs of the application. AWS offers a broad range of databases for exactly this reason.

TABLE 8.5 Matching workloads to AWS databases

SERVICE	MODEL	REACH FOR IT WHEN	SCALING STORY
RDS	Relational (OLTP)	Complex transactions/queries; medium-to-high rates; one node; high durability	Compute + storage scale with no downtime; read replicas for reads; Multi-AZ for availability
Aurora	Relational, MySQL/PostgreSQL-compatible	Commercial-grade availability at open-source cost	Distributed storage across AZs; up to 15 read replicas
DynamoDB	NoSQL key-value + document	Simple GET/PUT at any scale; semistructured data; latency-sensitive apps	Automatic partitioning; provisioned or auto-scaled throughput; global tables
Redshift	Data warehouse	Analytic SQL over petabytes with BI tools	Add nodes with no downtime; MPP + columnar; Spectrum reaches into S3

ACTIVITY – DATABASE CASE STUDIES

The module's group activity hands you three real AWS customer scenarios and asks you to defend a database choice. **Case 1**: a data protection company with **55 PB** of data needs a *relational* store for configuration data plus a store for *unstructured* deduplication metadata (deduplicated data lands in S3, then S3 Glacier) — the relational half points to RDS/Aurora, the unstructured half to DynamoDB. **Case 2**: a shipping company migrating an on-premises, **Oracle-based** legacy system to a serverless ecosystem while *decomposing highly structured relational data into semistructured data* — Oracle compatibility during migration suggests RDS for Oracle; the semistructured target suggests DynamoDB. **Case 3**: a payment processor handling **1 million+ transactions per day** whose flash-sale customers spike demand **30×** in a short window (IAM and KMS authenticate transactions) — that spiky, high-throughput profile is what DynamoDB's scalable throughput is built for. In each case, name the key factors behind the pick — and what would change your recommendation.

COMMON PITFALL

The module's sample exam question is pure keyword work: *"Which of the following is a fully managed NoSQL database service?"* RDS and Aurora are relational; Redshift is a data warehouse. **NoSQL ⇒ DynamoDB.**

Module summary

- Unmanaged services leave scaling, fault tolerance, and availability to you; managed services build them in and only ask for configuration.
- Amazon RDS sets up, operates, and scales relational databases — you keep only application optimization. Six engines: Aurora, PostgreSQL, MySQL, MariaDB, Oracle, SQL Server.
- Multi-AZ = synchronous standby in another AZ with automatic failover through the same DNS endpoint — availability, not read scaling.
- Read replicas = asynchronous copies that offload reads, promoted manually, cross-Region capable — read scaling, not automatic failover.

- RDS is for complex transactions and queries up to ~30,000 IOPS on a single node; massive rates, sharding, or simple GET/PUT push you to DynamoDB (or EC2 for engine customization).
- DynamoDB: NoSQL key-value/document store — tables → items → attributes, partition or partition+sort keys, single-digit-ms latency, no table size or throughput limits, global tables, encryption at rest, TTL.
- Redshift: fully managed data warehouse — leader + compute nodes, columnar storage, massively parallel processing, standard SQL and BI tools, Spectrum querying S3 directly.
- Aurora: MySQL/PostgreSQL-compatible and fully managed by RDS — data copied across multiple AZs, continuous S3 backup, up to 15 read replicas, sub-60-second crash recovery. Pick each database per workload: the right tool for the right job.

Review questions

1. Your team runs MySQL on an EC2 instance today. What do you still manage, and what disappears if you move the database to RDS?

Answer: On EC2 you still handle OS patches, database software installs and patches, backups, high availability, and scaling (AWS already covers the hardware, racking, and power/HVAC/network). Move to RDS and all of that shifts to AWS — you keep only application optimization.

2. A production database must survive an AZ outage, and the app also needs more read capacity. Which two RDS features solve this, and how do they differ?

Answer: Multi-AZ for the outage: a synchronous standby in another AZ with automatic failover through the same DNS endpoint. Read replicas for read capacity: asynchronous copies that offload read queries, manually promotable, and optionally cross-Region.

3. Why does promoting a read replica require manual action while Multi-AZ fails over automatically?

Answer: Read-replica replication is asynchronous, so the replica may lag the source — a human decides when promotion is acceptable. Multi-AZ replicates synchronously, so RDS can promote the standby automatically with minimal risk of data loss.

4. Name four workload signals that push you off RDS, and where each points.

Answer: Massive read/write rates (e.g., 150,000 writes/second), sharding for data size or throughput, and simple GET/PUT request patterns all point to a NoSQL service like DynamoDB; a need for RDBMS customization points to running the engine yourself on EC2.

5. Explain DynamoDB's two primary-key types and when a composite key earns its keep.

Answer: A partition key is a single attribute — pick one with uniform distribution, like a GUID or product ID. A composite key adds a sort key: author + title on a books table means each item is unique and the frequent 'all books by this author' lookup runs as an efficient query instead of a scan.

6. What is the difference between a DynamoDB query and a scan?

Answer: A query uses the primary key and partitioning to jump straight to matching items. A scan examines every item in the table and filters on non-key attributes — flexible, but far less efficient.

7. How does Redshift turn one SQL statement into fast results over petabytes?

Answer: The leader node parses the query, builds and compiles an execution plan, and assigns the pieces to compute nodes, which run them in parallel over columnar storage and return intermediate results for the leader to aggregate. Spectrum extends the same queries to exabytes sitting in S3.

8. What makes Aurora recover from a crash in under a minute?

Answer: It skips replaying the redo log from the last checkpoint at startup — that work happens on every read operation instead — and its buffer cache lives outside the database process, so the cache survives the restart and access doesn't need throttling while it repopulates.