DATABASES IN AWS

Databases: Background

In this section, we will discuss what databases are and their types. The topics to be discussed in this chapter are as follows:

- Introduction to databases
- SQL, its advantages and disadvantages
- NoSQL, its advantages and disadvantages

If the reader believes themselves to already be familiar with the listed concepts, then they can feel free to skip this section and move on to the rest of the chapter.

Introduction to Databases

Databases are the main method of storing and organizing volumes of related data in modern organizations, serving as the backbone of data management by providing structured environments for storing, retrieving, and updating information. Whether maintaining employee records, managing transactions in an e-commerce system, or supporting real-time data analytics, databases are essential for organized data storage and retrieval.

Today, databases come in a variety of forms, and are often tailored for specific needs. From traditional relational databases (SQL) designed to handle structured data with strict relationships, to flexible NoSQL databases, which excel at handling unstructured or semi-structured data in real-time applications.

SQL

Short for Structured Query Language, SQL databases organize data into tables that define relationships between different types of data. They offer a structured tabular format, providing a high level of consistency and reliability for applications where data integrity is crucial, such as finance and healthcare. It is ideal for data that benefits from adhering to the ACID (Atomicity, Conistency, Isolation and Durability) set of properties.

Having consistent schema makes it easy to run queries on the stored data, which is where SQL gets its name from. All SQL databases regardless of their differences store data in tables that can be queried according to similar commands and principles.

However, the structured format offered by SQL databases mean that the type of database is ill suited for the handling of unstructured data, data that cannot easily be organized into a rigid database schema.

NoSQL

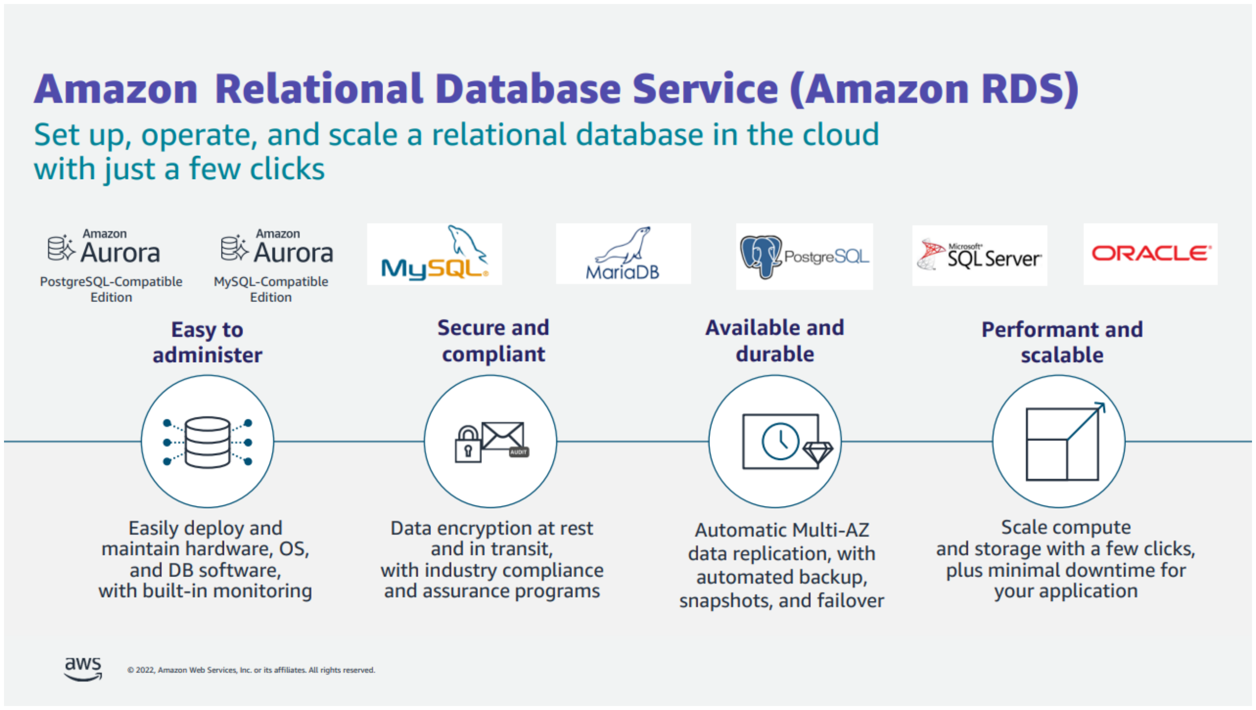
Short for Not Only SQL, NoSQL databases operate without a defined, rigid schema, the defining property of SQL databases. They store data in non-tabular formats, prioritizing flexibility and scalability, making them ideal for handling large volumes of unstructured databases.

Now, NoSQL databases come in many different flavors, and are often fundamentally different from one another. A full list of all the different types of NoSQL documents is beyond the scope of this book but some of the major ones include: **Document databases**, **Key-value databases** and **Graph databases**.

While this variety means NoSQL databases can handle a wider range and variety of data, it also means that there is no standard querying model/language that works across different types of NoSQL databases, with integration between different types of NoSQL databases often being a subject of particular hassle.

Relation Database Service (RDS)

Managing and maintaining the infrastructure required to run SQL servers can often be a troublesome task (to say the least). Amazon Relational Database Service (RDS) is a **managed SQL database service** provided by Amazon Web Services (AWS) which makes this task easier, allowing you to set up and operate SQL servers using AWS infrastructure, abstracting away a great deal of the headaches associated with having a relational database. Many organizations in fact, prefer managed database services like RDS over self-managed databases because of how hassle-free they make a lot of the database-adjacent tasks such as data migration, backup, recovery and patching.


Amazon Relational Database Service (Amazon RDS)
Set up, operate, and scale a relational database in the cloud with just a few clicks
Amazon Aurora PostgreSQL-Compatible Edition
Amazon Aurora MySQL-Compatible Edition
MySQL
MariaDB
PostgreSQL
Microsoft SQL Server
ORACLE
Easy to administer
Secure and compliant
Available and durable
Performant and scalable
Easily deploy and maintain hardware, OS, and DB software, with built-in monitoring
Data encryption at rest and in transit, with industry compliance and assurance programs
Automatic Multi-AZ data replication, with automated backup, snapshots, and failover
Scale compute and storage with a few clicks, plus minimal downtime for your application
aws
© 2022, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Source: AWS

Boasting support for a whole host of database engines (listed above), Amazon RDS is considered an industry standard for storing and organizing data in the cloud using databases for reasons mentioned in the illustration. Additionally, though this fact is probably clear to the reader by this point, it is widespread enough of a misunderstanding that I find it necessary to clearly re-iterate: **Amazon RDS itself is not a database; It is a service used to manage relational databases.**

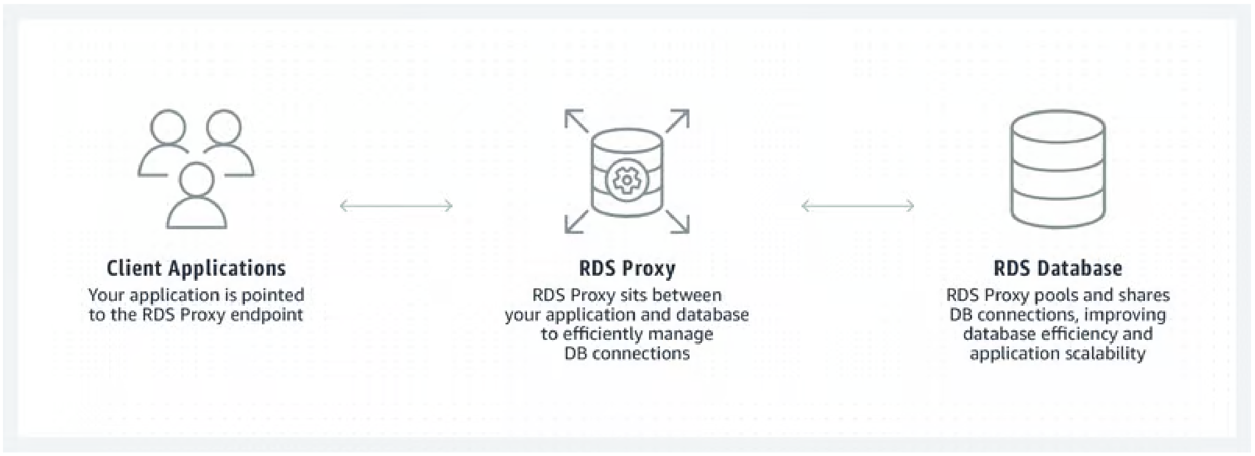
The main allure of RDS and its popularity in my personal opinion is that once we adopt and get used to the features of managed services such as Automatic scaling, encryption, Read Replicas, Multi-AZ databases, proxies, etc (Mentioned features will be discussed in later sections), the convenience factor is so great that it becomes hard to decouple entirely and go back to being self-managed.

RDS Proxy and Read Replicas

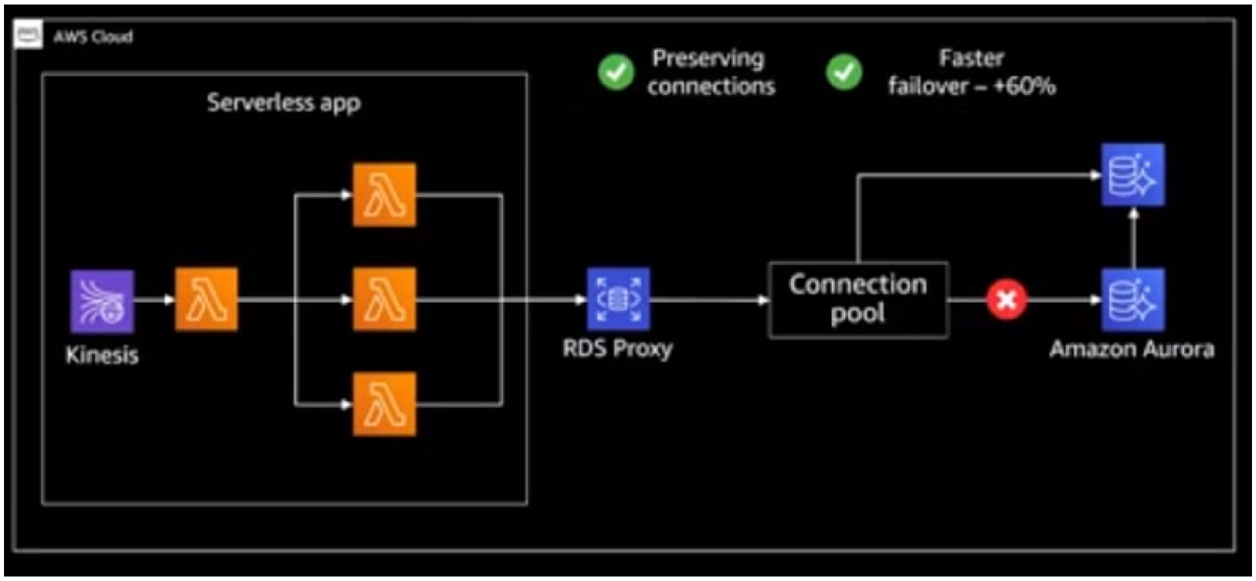
A database proxy is best understood as a middle-man that resides between an application and a database, receiving all the requests sent by the application before forwarding it to the desired database. Now, why is it exactly that we need this middle-man? What possible benefits could there be after all, to adding a extra hop between the client and the database? Well, there are a few, such as:

- **Connection Pooling**: Without a proxy, every time an application makes a request, it might need to establish a new connection to the database. This is resource-intensive, especially with high traffic. The proxy manages a pool of connections, reusing them efficiently, reducing the time and resources spent on opening and closing connections.
- **Load Balancing**: If multiple database instances exist (e.g., for replication or sharding), the proxy can distribute traffic evenly across them. This reduces the risk of overloading any single instance and ensures a smoother application performance.
- **Failover and High Availability**: In the event that a database instance goes down, the proxy can detect this and reroute traffic to a healthy instance automatically, without requiring changes on the application side. This ensures continuous availability.
- **Security**: The proxy can centralize access controls, authentication, and encryption, acting as a gatekeeper to protect the database from direct exposure. This adds an extra layer of security while streamlining how applications interact with the database.

RDS Proxy is an AWS service that provides us with a fully managed, highly available database proxy just like the one discussed above, acting as an intermediary between our application (usually also being hosted on an EC2 instance) and Amazon RDS database instances, enhancing their relationship with all the benefits mentioned above.


Client Applications
Your application is pointed to the RDS Proxy endpoint
RDS Proxy
RDS Proxy sits between your application and database to efficiently manage DB connections
RDS Database
RDS Proxy pools and shares DB connections, improving database efficiency and application scalability

The ability of the RDS Proxy to pool connections is particularly useful for application developers and cloud architects as it allows us to preserve connections and/or failover to an alternative database instance during cases where the connections between the database and the application are interrupted due to, for example, too many agents trying to connect at the same time, as shown in the architecture diagram below:



RDS Read Replicas

RDS Read Replicas are duplicates of your primary Amazon RDS database that are specifically designed to handle read-only queries. They play a key role in scaling database performance, and are particularly useful in scenarios where an application has a read-heavy workload but doesn't require comparably as many write operations such as for example, querying tools and programs.

Though they do cost extra, many organizations do not mind paying the cost due to the myriad of benefits associated with having Read Replicas such as:

1. **Offloading Read Traffic**: By creating one or more Read Replicas, organizations can direct read-only queries (like SELECT statements) to these replicas instead of burdening the primary database. This reduces the load on the primary instance, allowing it to focus more on write operations (like INSERTs, UPDATEs, or DELETEs).
2. **Horizontal Scaling**: Read Replicas enable horizontal scaling of read operations. If the number of read requests increases as traffic grows, organizations can create additional replicas to handle the increased load without affecting the performance of the primary database.
3. **High Availability for Reads**: In the event of a failure or downtime of the primary database, the Read Replicas can continue serving read requests, ensuring continuous availability for read operations.
4. **Geographic Distribution**: Read Replicas can be deployed in different regions to reduce latency for users in various geographic locations. This allows users to access data from the closest replica, improving read query response times.

Aurora

Considered as a part of the RDS extended family of offerings, Amazon Aurora is a proprietary relational database engine built by Amazon to take full advantage of AWS's vast cloud infrastructure. Aurora was designed from the ground up to leverage the distributed, scalable, and fault-tolerant capabilities of the cloud. While it is a proprietary database engine, it is compatible with MySQL and PostgreSQL at the relational database layer, meaning that applications written for those databases can run on Aurora with little to no modification.

With features like Aurora Global Database allowing for cross-region replication and Multi-AZ deployments, Aurora can efficiently power modern, scalable applications. However, the privilege of having Amazon manage your servers for you does come at an extra cost, thereby making the service more expensive sometimes, especially in comparison to RDS. This cost is however worth it as Aurora is considered to be superior to RDS, delivering higher performance and scalability than traditional database engines with Aurora providing upto five times the throughput of normal MySQL and upto three times the throughput of normal PostgreSQL databases.

Aurora is also better suited for the handling of large, distributed workloads due to its superior availability and performance, with built-in support for features such as automatic scaling of storage (upto 128TB) and automatic addition of read replicas (upto 15 separate instances). Aurora DB clusters are also highly fault tolerant by nature, where cluster volumes span multiple Availability Zones (AZs; will be discussed in further sections) with each Availability Zone containing a copy of the data, allowing said clusters to tolerate the failure of an Availability Zone without any major interruptions.

Database Cloning

A database clone is an exact copy of an existing database, created usually for the purpose of testing, development, troubleshooting, or scaling. It replicates the entire database, including its structure, data, indexes, and sometimes user privileges, depending on the system. In the context of RDS, Database cloning is a method that allows us to create a database clone of an RDS database instance, with all the same settings and configurations.

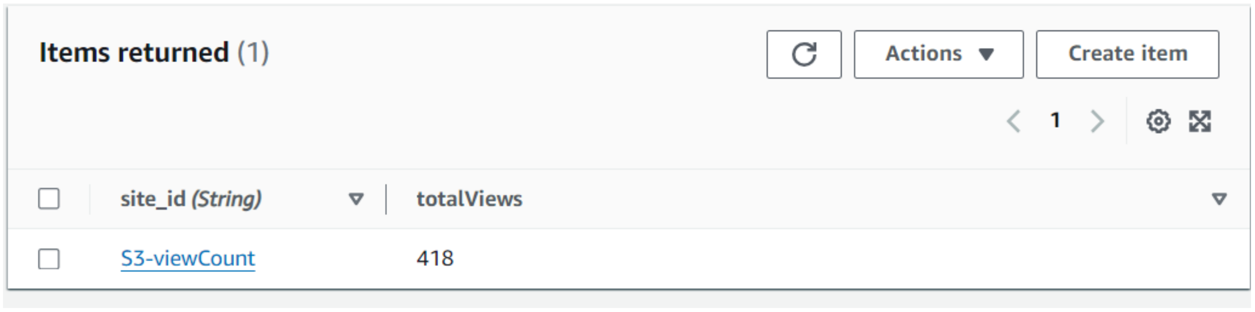
RDS database cloning is also particularly useful when creating **staging databases** (i.e. temporary databases created to serve an Extract, Transform and Load, aka ETL procedure; ETL procedures will be discussed in-depth later) on-demand in a streamlined, efficient and almost real-time manner without the overhead of traditional backup and restore processes.

However, it is worth remembering that database clones are (by deliberate design) isolated from their source databases, meaning any modifications made to the cloned database do not affect the original, and the clone can be queried and updated independently. This isolation is critical for scenarios where testing or development needs to mimic production data without risking changes to the live environment.

DynamoDB and DynamoDB Streams

As discussed in the start of this section, databases can broadly be categorized into two categories: SQL and NoSQL, with NoSQL offerings being famous for their flexibility and ability to store and organize structured as well as unstructured and semi-structured data, well DynamoDB is one such a database, offered to us by Amazon as a fully managed DBaaS, meaning that we will not have to handle tasks like hardware provisioning, setup, configuration, or scaling.

With support for both key-value and document data models, DynamoDB organizes data into tables, with each table consisting of items (analogous to rows) and attributes (akin to columns). It is especially great for the handling of large amounts of data with high availability, and is ideal for applications requiring consistent, low-latency data access without compromising on performance or scalability.



site_id (String)	totalViews
S3-viewCount	418

A simple example of a DynamoDB table with two key-value fields, site_id and totalViews

Querying in DynamoDB also works quite differently than with conventional SQL-based relational databases, using fields called Partition Keys and Sort Keys. Partition keys are the primary key of the DynamoDB table, uniquely identifying each item in the table while sort keys are an extension of primary keys, allowing us to uniquely identify items with the same primary key. Sort keys can be thought of as a sub-primary key, as they are a composite key consisting of both a partition key and a sort key enabling us to differentiate between different items with the same partition key.

Additionally, DynamoDB also has multiple interesting additions that can enhance the capabilities and functionality of the service, of which there are two in particular which I feel deserve special attention: **DynamoDB Streams** and **DynamoDB Accelerator**.

DynamoDB Streams is a feature of DynamoDB that captures real-time data modifications in DynamoDB tables, enabling applications to respond promptly to updates. This feature facilitates use cases such as real-time analytics, cross-region replication. Note however, that DynamoDB streams is not enabled by default and instead must be manually enabled if it is to be taken advantage of.

DynamoDB Accelerator or DAX for short (as it is often commonly referred to), on the other hand is not a feature of DynamoDB but rather a separate service designed to improve the performance of DynamoDB tables by providing an in-memory caching layer specifically for said tables. This is particularly useful for read-heavy workloads as frequently requested data can be stored in the DAX cache and applications can fetch said data without having to repeatedly ping DynamoDB for it, greatly reducing the load on the DynamoDB tables.

OLTP and OLAP

Another manner of categorizing databases is through database processing systems (the methods used to create, manage and manipulate databases), of which there are two major ones: **OLTP** and **OLAP**.

OLTP, short for Online Transaction Processing, typically involves managing and processing more atomized transactional data in real-time. It is optimized for rapid and efficient querying and updating of individual data records. OLTP databases are typically used for day-to-day transactional operations such as order processing, financial transactions, and customer relationship management (CRM). AWS services that support OLTP include:

- **Amazon RDS (Relational Database Service)**: Provides managed database services for relational databases like MySQL, PostgreSQL, Oracle, and SQL Server, optimized for OLTP workloads.
- **Amazon DynamoDB**: A fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. It is ideal for applications requiring low-latency data access and high throughput, such as gaming and mobile apps.

OLAP, short for Online Analytical Processing, on the other hand, focuses on processing large volumes of data for complex analytical queries supporting operations such as aggregations, calculations, and data mining to derive insights and make informed business decisions. AWS services that support OLAP include:

- **Amazon Redshift**: A fully managed data warehouse service that allows you to run complex queries across petabytes of structured data. It has built-in integration with Business Intelligence and reporting tools for data visualization and analysis.

IAM DB Authentication

When dealing with databases, having systems put in place for the secure, reliable authentication of agents (such as a user or an application) which have access to the database are legitimate, and prevent unwanted actors from engaging with the database especially when it deals with sensitive information.

RDS instances therefore, utilize and support a whole suite of established traditional authentication methods such as Usernames and Passwords, SSL / TLS certificates, etc. However, AWS has its own set of services that facilitate easy and simple methods of authentication and identity management under their IAM umbrella (will be elaborated on in future sections) of services and are especially critical when connecting AWS resources with RDS databases.

EC2 instances for example, are not allowed permission to connect to RDS tables and databases by default, and we must both enable an option with RDS called IAM DB Authentication as well as assign an IAM role to the EC2 instance if we desire to circumvent the restriction.

Disaster Recovery

Databases are an integral component of any modern business, and their failure can often be quite devastating and disrupting to its operations. Planning for disasters and setting up contingencies for database related disasters is hence considered as an important task by many companies, particularly those related to database recovery.

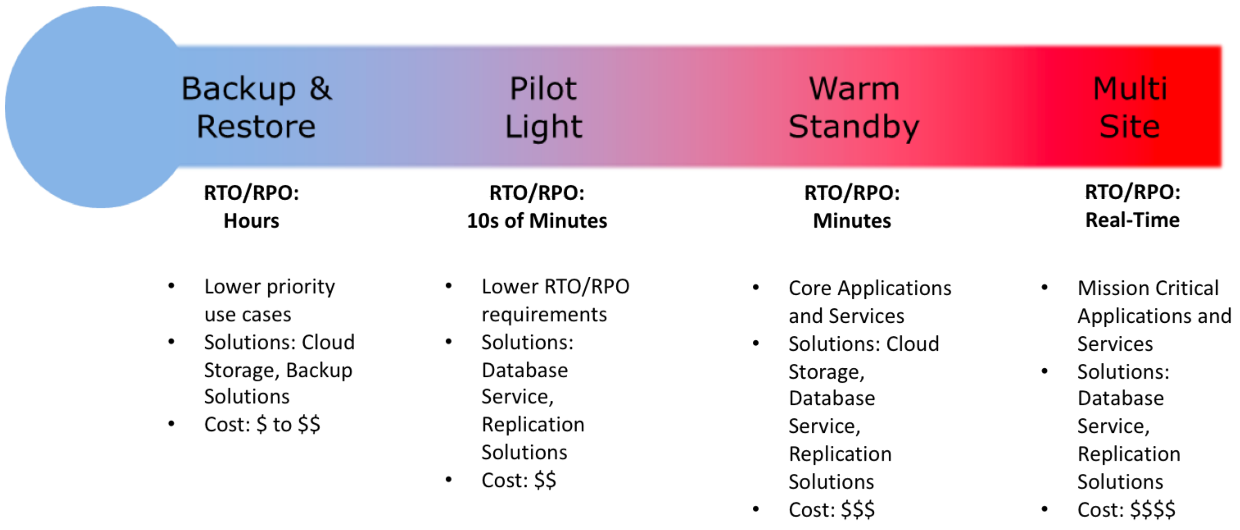
There exist however, a myriad of different approaches and practices for disaster recovery which are measured using the metrics **Recovery Point Objective (RPO)** and **Recovery Time Objective (RTO)**.

RPO defines the maximum allowable data loss in terms of time; for instance, if a business has an RPO of 30 minutes, it means that, in the event of a disaster, they can only tolerate losing data from the last 30 minutes.

RTO on the other hand, specifies the maximum time allowed for system recovery after a failure. For example, if a company has an RTO of 2 hours, it indicates that they must restore their database and resume normal operations within that timeframe following an outage.

As such, a low RTO/RPO method of disaster recovery is acceptable when downtime is not a dealbreaker, and the business can handle some amount of lost data whilst a high RTO/TPO method of disaster recovery is more suited for databases that must be maximally available and cannot handle the loss of data.

An in-depth discussion of all the disaster recovery methods is beyond the scope of this book, and hence I will be relying on a diagram made available by AWS. Consult the diagram for a brief overview of all the disaster recovery methods and their respective RTO/RPOs:



Backup & Restore
Pilot Light
Warm Standby
Multi Site
RTO/RPO: Hours
RTO/RPO: 10s of Minutes
RTO/RPO: Minutes
RTO/RPO: Real-Time
• Lower priority use cases
• Solutions: Cloud Storage, Backup Solutions
• Cost: $ to $$
• Lower RTO/RPO requirements
• Solutions: Database Service, Replication Solutions
• Cost: $$
• Core Applications and Services
• Solutions: Cloud Storage, Database Service, Replication Solutions
• Cost: $$$
• Mission Critical Applications and Services
• Solutions: Database Service, Replication Solutions
• Cost: $$$$

I encourage the readers to research the different disaster recovery methods listed above on their own, searching the internet for their accompanying architecture diagrams as well if desired.

Primary Database and CNAME

Cloud architects and engineers often like to have multiple instances of the same database running in multiple Availability Zones (or AZs; discussed in the next section) in order to maintain high availability and low-latency, among a whole host of other reasons. When spinning off RDS databases into a multi-AZ setup however, we must designate one of the database instances as the primary or main database instance.

This so-called primary database instance is also the holder of something called a Canonical name, or CNAME for short. A CNAME is a type of DNS record that allows one domain name to act as an alias for another, essentially mapping an alias name to a true or canonical domain name. This means that when an application connects to the CNAME, it will resolve to the actual endpoint of the database instance, ensuring smooth communication.

In the event that the primary database fails, the Canonical Name (CNAME) record in the Domain Name System (DNS) is automatically updated to redirect database requests to the standby instance, a process that is considered essential for seamless failover. By updating the CNAME, the system ensures that applications continue to connect to the standby instance without interruption. This process minimizes downtime and enhances the overall reliability and resilience of your database services, allowing for high availability even in the event of a failure.

Redshift and AWS Lake Formation

Finally, though technically a cloud storage solution, I do find it necessary to at least mention data warehouses and data lakes in the storage section as they are often utilized to store and manage large volumes of data (usually data that is miscellaneous in nature). A purpose that keen readers may recall was mentioned as the defining characteristic of storage in the background section of the chapter.

Now, data warehouses are designed to assist with business intelligence and data analytics tasks and usually is responsible for storing data, performing simple quantitative analyses on the data and visualizing data. Amazon's data warehousing solution is called **Redshift** (discussed already in the OLAP section), and focuses on operating more so as a business and data analysis tool than as a service for the storage of data. Redshift is built around a cluster of computing nodes which sit on top of Redshift's data store allowing us to prepare, perform simple arithmetic on and visualize quantitative data at the petabyte-scale.

Data lakes on the other hand, allow an organization to store virtually any type of data at an almost unlimited scale. They are used when storing data pertaining to a wide range of services and belonging to a diverse set of formats (such as databases, BI tools and multimedia files) all in a single location. They are highly adaptable and can change with ease to meet new requirements. **AWS Lake Formation** is Amazon's managed service for the creation and deployment of data lakes, and is usually utilized when organizations using AWS require a repository to hold vast amounts of both processed and unprocessed data from a myriad of unrelated sources, which companies sometimes have to do, in order to comply with regulatory standards and policies.