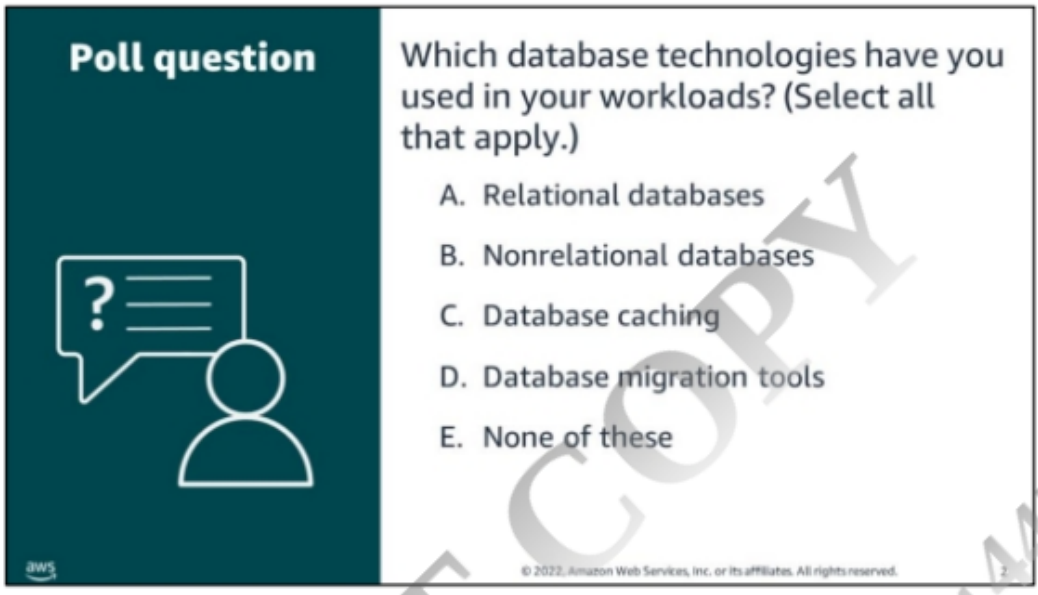
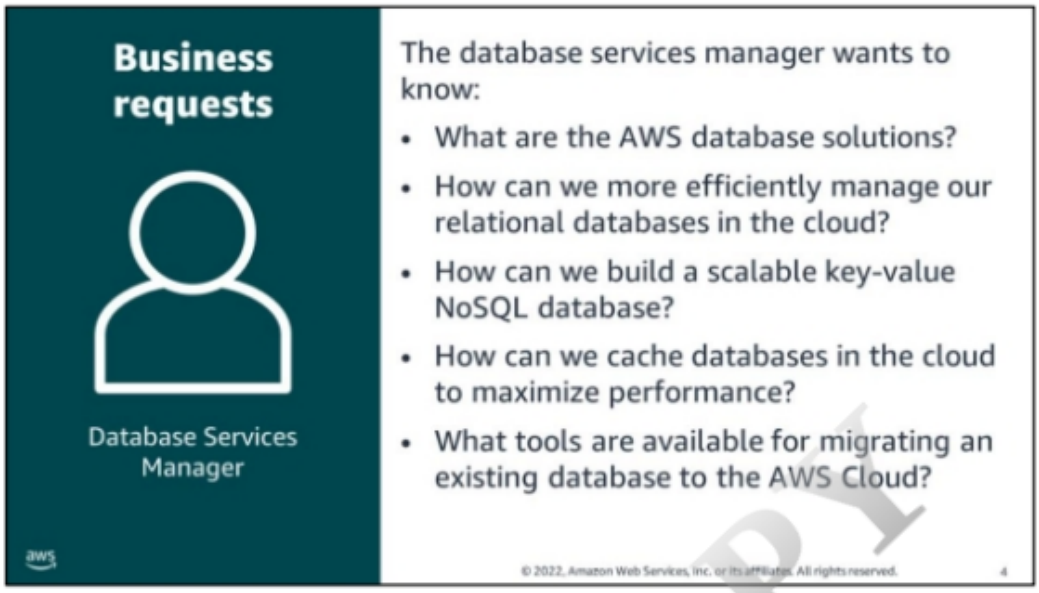
Printed by: 2d35e8483186bd2@placeholder.44518.edu. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.



Module overview

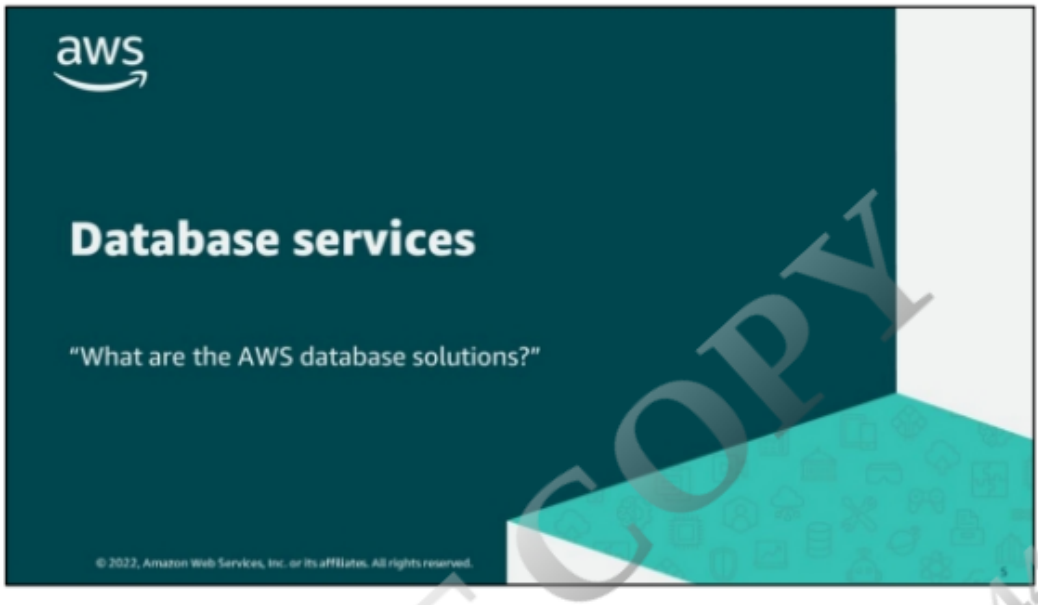
- Business requests
- Database services
- Amazon Relational Database Service (Amazon RDS)
- Amazon DynamoDB
- Database caching
- Database migration tools
- Present solutions
- Capstone check-in
- Knowledge check
- Lab 3: Create a database layer in your Amazon VPC infrastructure

 3



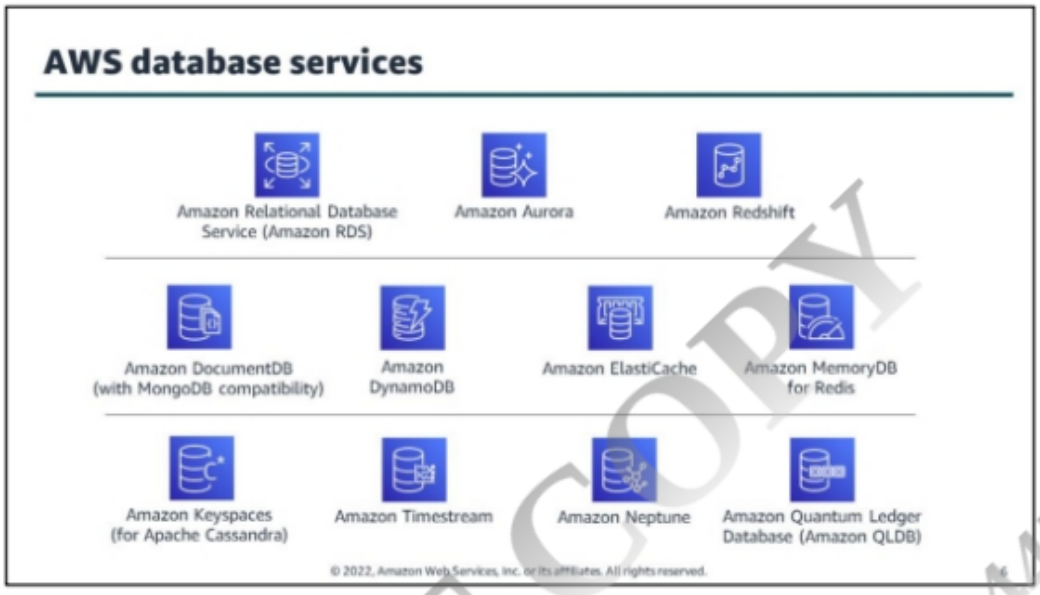
Imagine your database services manager meets with you to discuss how to manage databases in the cloud. Here are some questions they are asking.

At the end of this module, you meet with the database services manager and present some solutions.



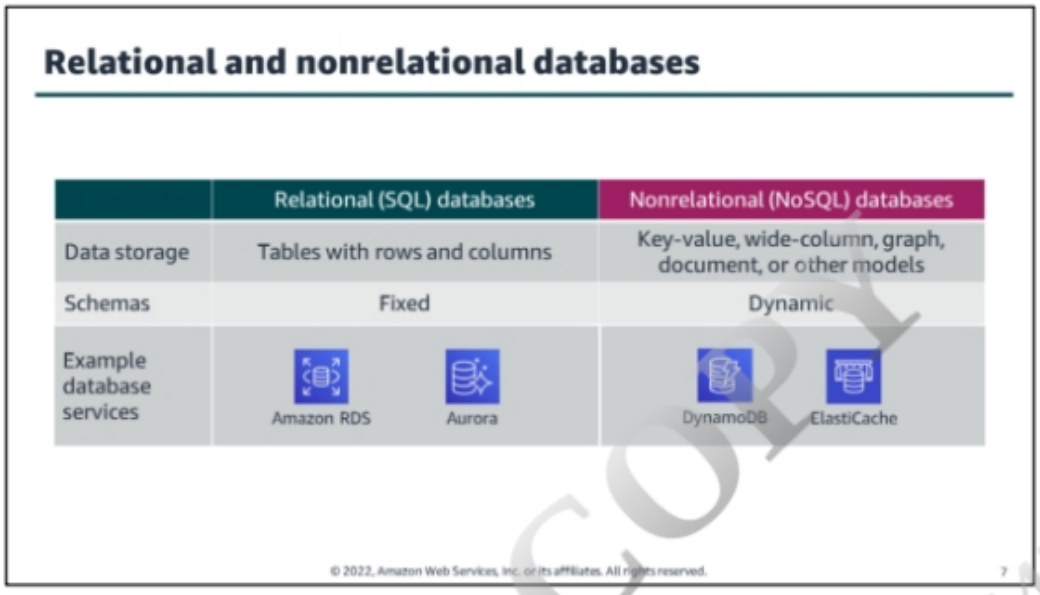
The database services manager asks, "What are the AWS database solutions?"

The database team is beginning to investigate opportunities for running databases in the cloud. The company wants you to identify which AWS database solutions they should consider.



AWS database engines are purpose-built and include relational, key-value, document, in-memory, graph, time series, wide-column, and ledger databases. We discuss how to pick the right database for your needs.

For more information, see "AWS Cloud Databases" (https://aws.amazon.com/products/databases/).



For decades, the predominant data model that was used for application development was the relational data model used by relational databases such as Oracle, IBM DB2, SQL Server, MySQL, and PostgreSQL. It wasn't until the mid to late 2000s that other data models began to gain significant adoption and usage. To differentiate and categorize these new classes of databases and data models, the term *NoSQL* was coined. Often the term NoSQL is used interchangeably with *nonrelational*.

A *relational database* is a collection of data items with predefined relationships between them. These items are organized as a set of tables with columns and rows. Each column in a table holds a certain kind of data and a field stores the actual value of an attribute. The rows in the table represent a collection of related values of one object or entity. This data can be accessed in many different ways without reorganizing the database tables themselves.

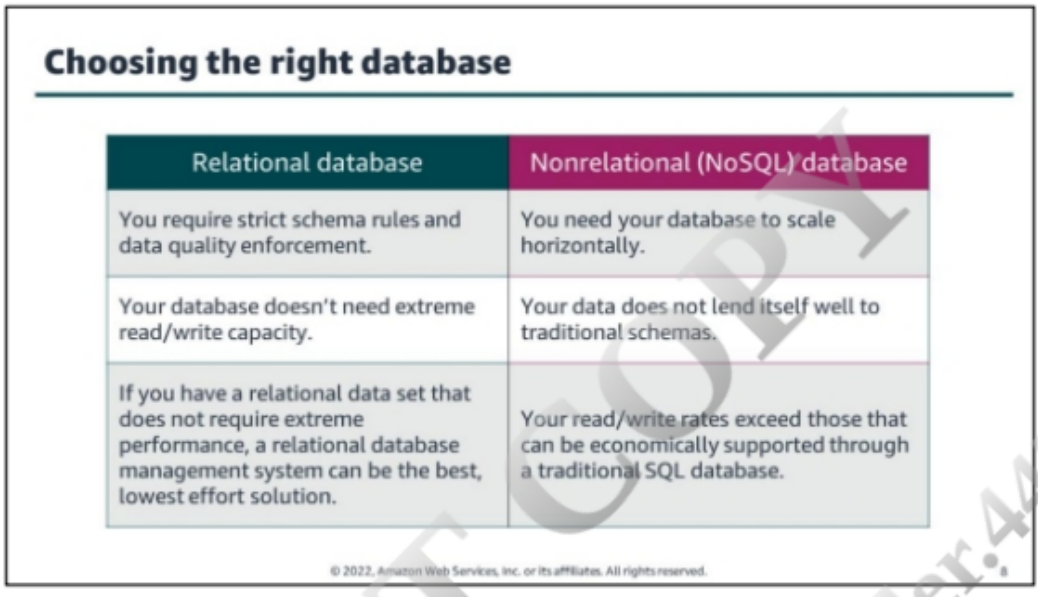
This module focuses on two SQL databases services, Amazon RDS and Amazon Aurora.

NoSQL is a term used to describe nonrelational database systems that are highly available, scalable, and optimized for high performance. Instead of the relational model, NoSQL databases use alternate models for data management, such as key-value pairs or document storage.

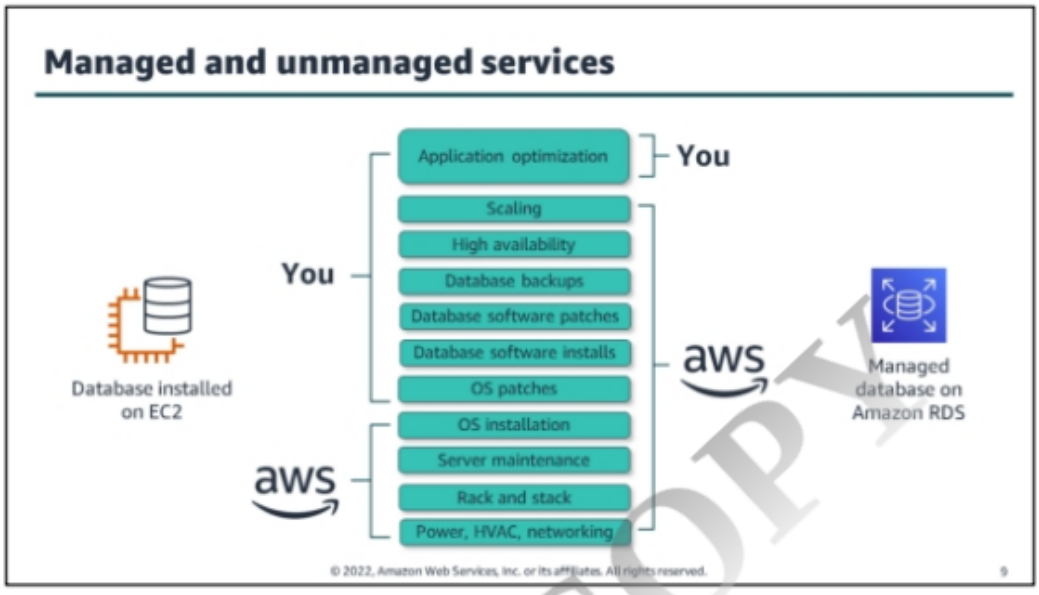
This module focuses on two NoSQL databases services, Amazon DynamoDB and Amazon ElastiCache.

For more information about relational databases, see "What is a Relational Database?" (https://aws.amazon.com/relational-database/).

For more information about nonrelational databases, see "What is NoSQL?" (https://aws.amazon.com/nosql/).



Though there are many types of databases with varying features, this table shows some of the differences between SQL (relational) and NoSQL (nonrelational) databases.



When building resources in the cloud, you want to consider the level of control you need and the resources you have to manage that resource.

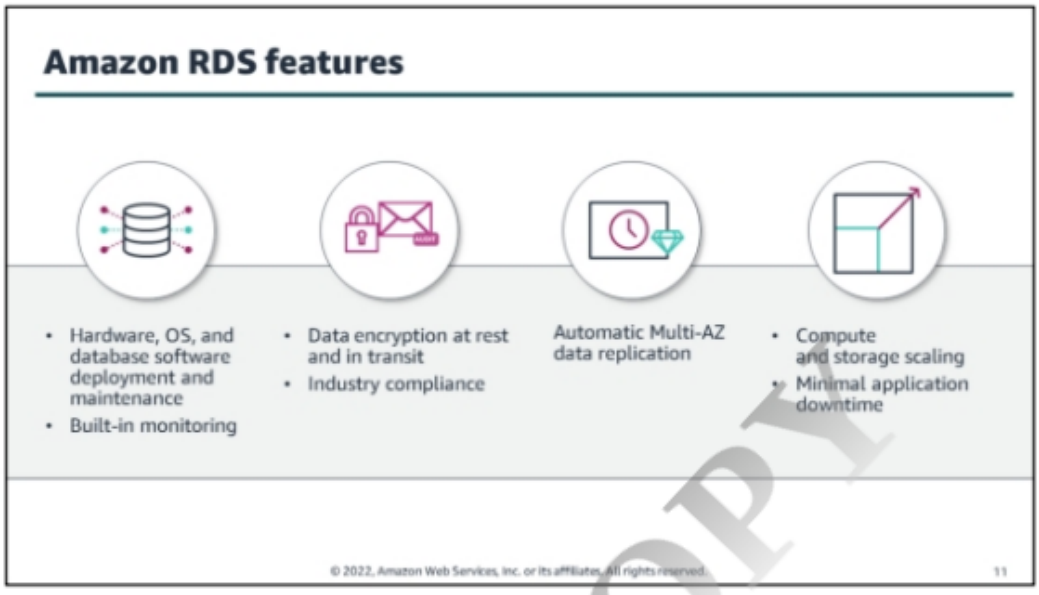
For example, when running a database in the cloud, you can install a database on an Amazon Elastic Compute Cloud (Amazon EC2) instance or you can choose a managed database option such as Amazon RDS.

- Installing a database on an EC2 instance gives you complete control over all aspects of the database except the hardware. The trade-off is that this increased amount of control requires more resources and expertise to manage.
- Using a managed database service removes the undifferentiated heavy lifting of managing your databases.

Databases managed by AWS provide systems for high availability, scalability, and backups. You can choose to use scaling, high availability, database backups, database software patches, database software installs, and operating system (OS) patches. In general, you are responsible only for optimizing your applications to make sure the database layer works as well as possible with your application.

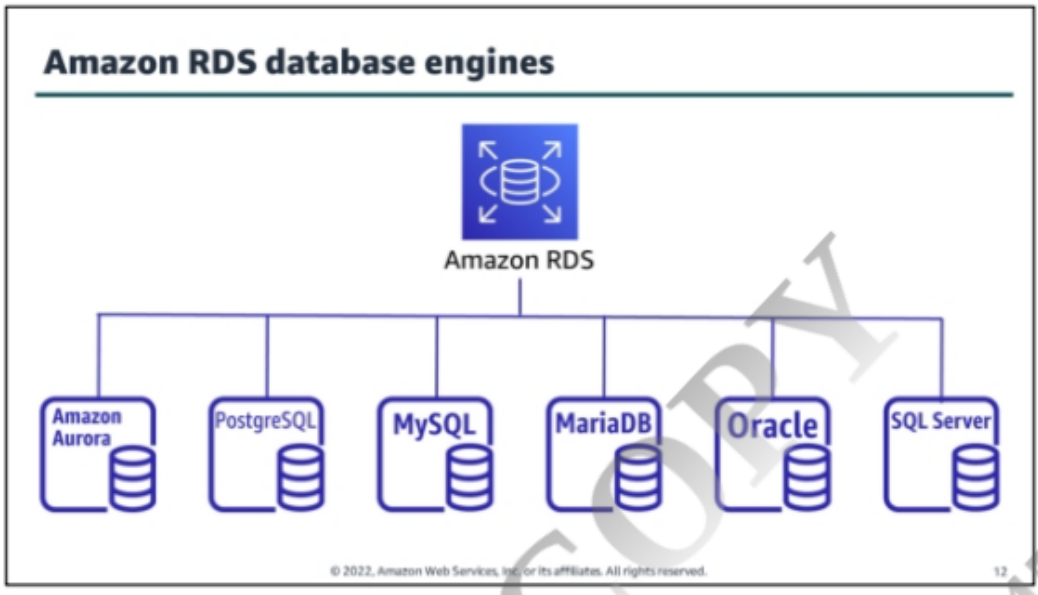
The database services manager asks, "How can we more efficiently manage our relational databases in the cloud?"

The database team has relational databases and are considering Amazon RDS. The company wants you to explain the benefits of running a database in a managed service and examine the features of Amazon RDS.



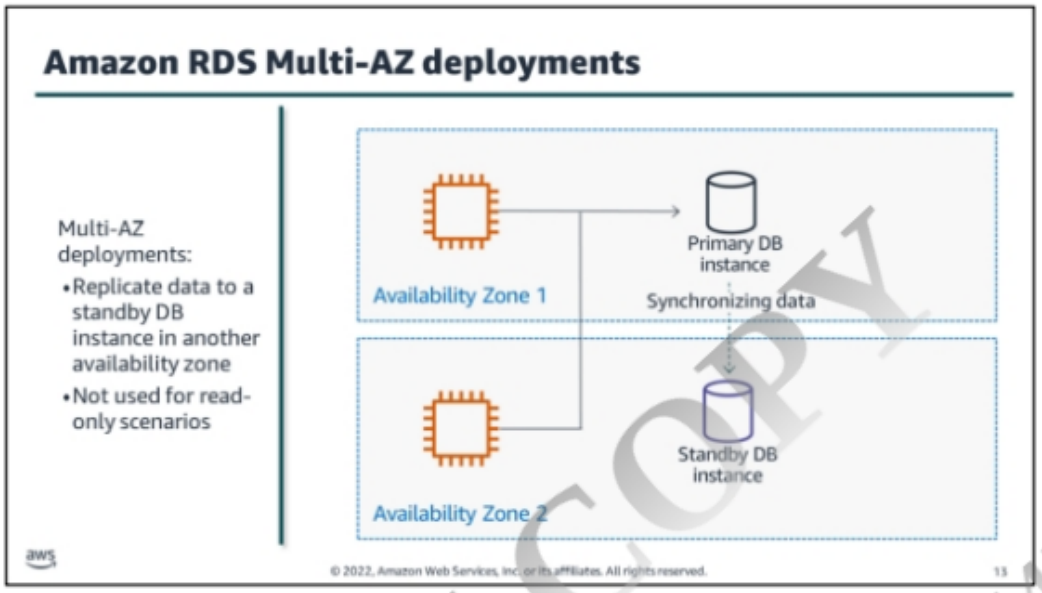
Amazon RDS is a web service that helps you to set up, operate, and scale a relational database in the cloud. It provides cost-efficient and resizable capacity, while managing time-consuming database administration tasks. By using Amazon RDS, you can focus on your applications and business. Amazon RDS provides you with six familiar database engines to choose from, including Amazon Aurora, PostgreSQL, MySQL, MariaDB, Oracle Database, and Microsoft SQL Server. This means that most of the code, applications, and tools you already use with your existing databases can be used with Amazon RDS.

Amazon RDS automatically patches the database software and backs up your database. It stores the backups for a user-defined retention period and provides point-in-time recovery. You benefit from the flexibility of scaling the compute resources or storage capacity associated with your relational DB instance with a single API call.



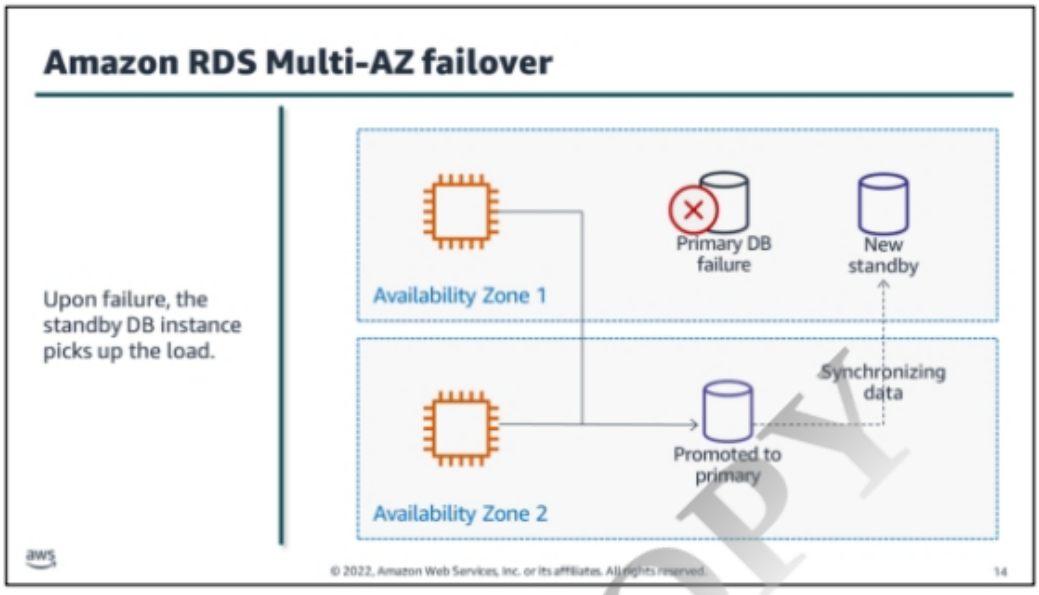
Amazon RDS is available on six database engines, which optimize for memory, performance, or I/O. The database engines include the following:

- Amazon Aurora
- PostgreSQL
- MySQL
- MariaDB
- Oracle Database
- SQL Server



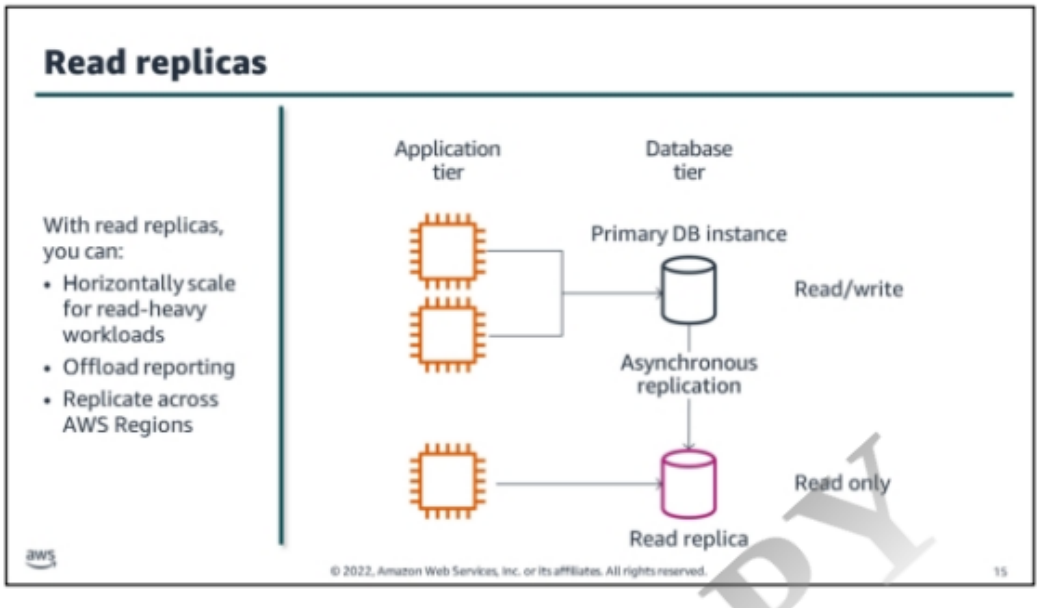
Amazon RDS Multi-AZ deployments provide enhanced availability and durability for database (DB) instances, making them a natural fit for production database workloads. When you provision a Multi-AZ DB instance, Amazon RDS synchronously replicates the data to a standby instance in a different Availability Zone.

You can modify your environment from Single-AZ to Multi-AZ at any time. Each Availability Zone runs on its own physically distinct, independent infrastructure and is engineered to be highly reliable.



In case of the primary instance failure, Amazon RDS performs an automatic failover to the standby instance.

In this example, two EC2 instances in separate Availability Zones are connected to the primary database in one Availability Zone. A standby database is hosted in the other Availability Zone. When the primary database fails, Amazon RDS promotes the secondary database to primary. Because it assumes the primary databases endpoint, the EC2 instances can resume traffic with the new primary database. Meanwhile, a new standby database is created in the other Availability Zone.



With Amazon RDS, you can create read replicas of your database. Amazon automatically keeps them in sync with the primary DB instance. Read replicas are available in Amazon RDS for Aurora, MySQL, MariaDB, PostgreSQL, Oracle, and Microsoft SQL Server. Read replicas can help you do the following:

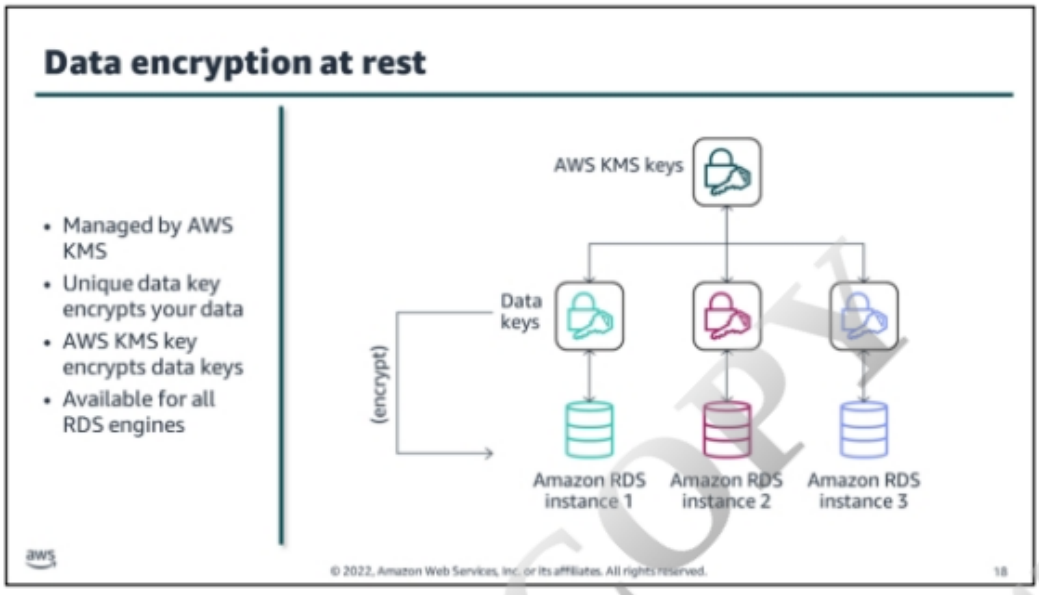
- Relieve pressure on your primary node with additional read capacity.
- Bring data close to your applications in different AWS Regions.
- Promote a read replica to a standalone instance as a disaster recovery (DR) solution if the primary DB instance fails.

You can add read replicas to handle read workloads so your primary database doesn't become overloaded with read requests. Depending on the database engine, you can also place your read replica in a different Region from your primary database. This gives you the ability to have a read replica closer to a particular location.

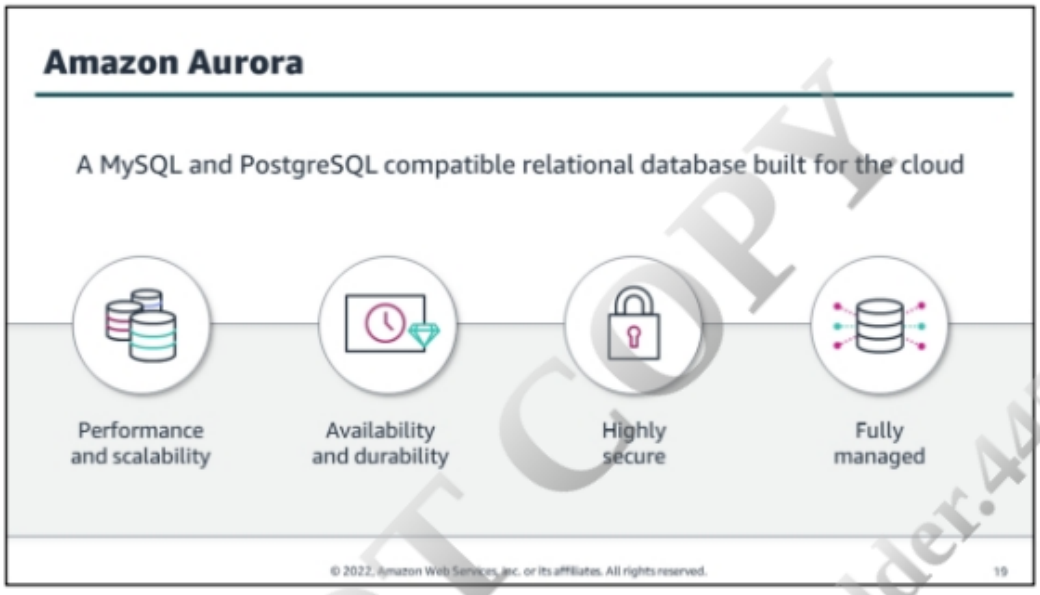
You can configure a source database as Multi-AZ for high availability and create a read replica in Single-AZ for read scalability. With Amazon RDS for MySQL and MariaDB, you can also set the read replica as Multi-AZ, and as a DR target. When you promote the read replica to be a standalone database, it will be replicated to multiple Availability Zones.

For more information about read replicas, see "Working with read replicas" in the *Amazon Relational Database Service (RDS) User Guide* (https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_ReadRepl.html#USER_ReadRepl.PostgreSQL).

**For Accessibility: Two instances in the application tier connect to a primary database instance in the database tier. This database can perform read/write operations. Amazon RDS creates a read replica of this database using asynchronous replication. A third instance in the application tier connects to the read replica, which can only perform read operations. End Description

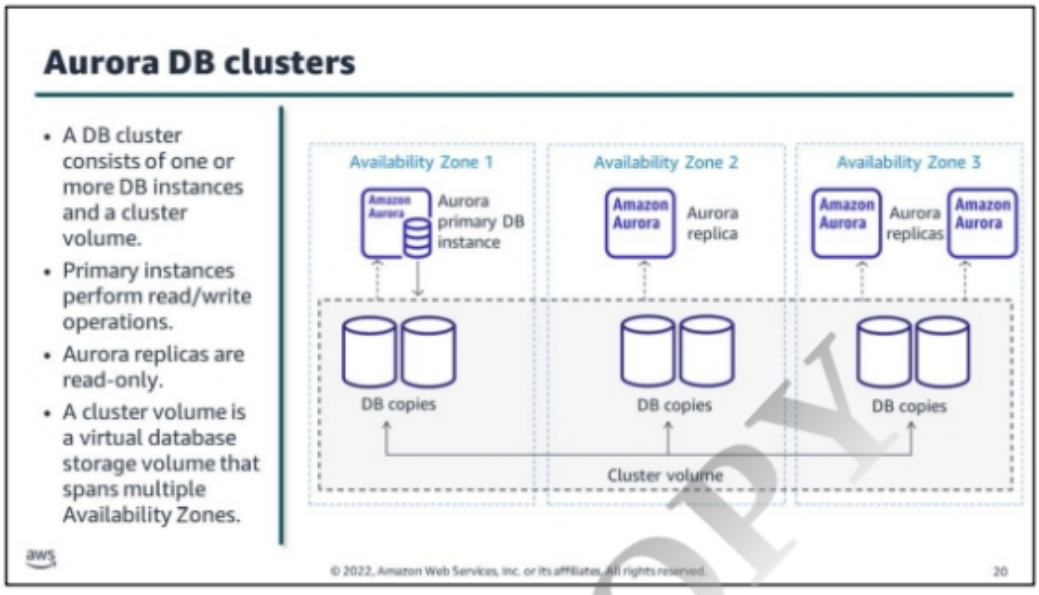


Amazon RDS provides encryption of data at rest using the AWS Key Management Service (AWS KMS). AWS KMS is a managed service that provides the ability to create and manage encryption keys and then encrypt and decrypt your data using those keys. All of these keys are tied to your AWS account and are fully managed by you. AWS KMS provides an additional layer of protection against unauthorized access to the underlying storage of your Amazon RDS instance. AWS KMS uses industry-standard AES-256 encryption to protect data stored on the underlying host that your Amazon RDS instance is running on.



Amazon Aurora is an enterprise-class relational database. It is compatible with MySQL and PostgreSQL relational databases. It is up to five times faster than standard MySQL databases and up to three times faster than standard PostgreSQL databases. Aurora helps to reduce your database costs by reducing unnecessary I/O operations, while ensuring that your database resources remain reliable and available. Consider Aurora if your workloads require high availability. It replicates six copies of your data across three Availability Zones and continuously backs up your data to Amazon Simple Storage Service (Amazon S3).

Aurora supports network isolation, encryption at rest and in transit, and compliance and assurance programs. Aurora is managed by Amazon RDS, so it requires no server provisioning, software patching, setup, configuration, or backups.

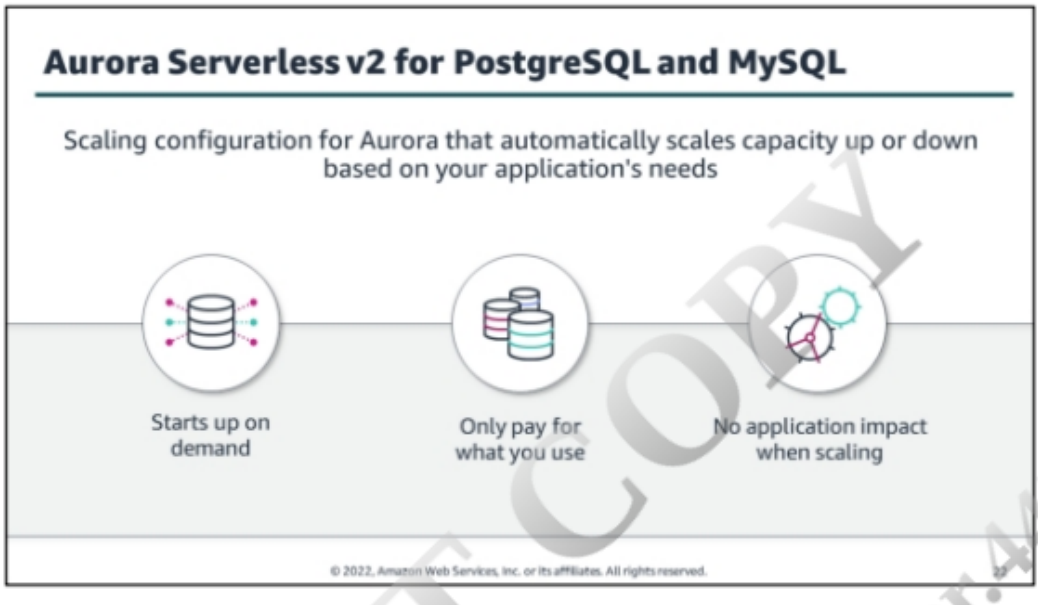


An Amazon Aurora DB cluster consists of one or more DB instances and a cluster volume that manages the data for those DB instances. The instances perform the compute functions of the database while the cluster volume stores the actual data.

Aurora offers two instance types:

- **Primary instance** – Supports read and write operations and performs all the data modifications to the cluster volume. Each Aurora DB cluster has one primary instance.
- **Aurora replica** – Supports read operations only. Each Aurora DB cluster can have up to 15 Aurora replicas in addition to the primary instance. Multiple Aurora replicas distribute the read workload. You can increase availability by locating Aurora replicas in separate Availability Zones. You can have a read replica in the same Region as the primary instance.

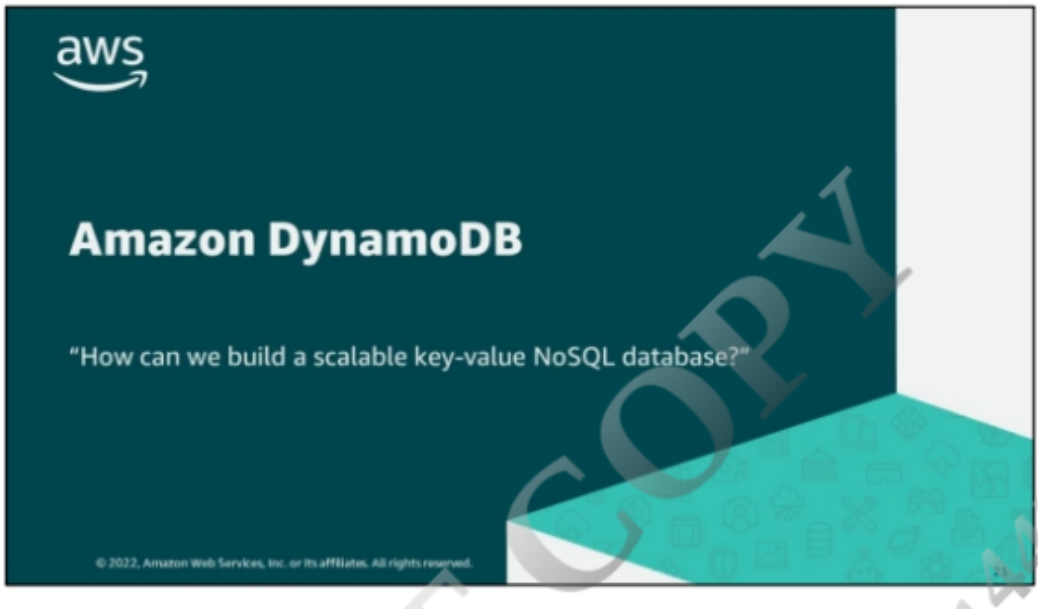
An Aurora *cluster volume* is a virtual database storage volume that spans multiple Availability Zones, with each Availability Zone having a copy of the DB cluster data. Storage in the cluster volume is replicated across hundreds of storage nodes. Aurora presents the cluster volume as a single, logical volume to the primary instance and to Aurora replicas in the DB cluster. Write operations to the cluster volume are often available to Aurora replicas in less than 100 milliseconds.



Aurora Serverless v2 is an on-demand, auto scaling configuration for Amazon Aurora. Aurora Serverless v2 helps to automate the processes of monitoring the workload and adjusting the capacity for your databases. Capacity is adjusted automatically based on application demand. You're charged only for the resources that your DB clusters consume, so Aurora Serverless v2 can help you to stay within budget and avoid paying for computer resources that you don't use.

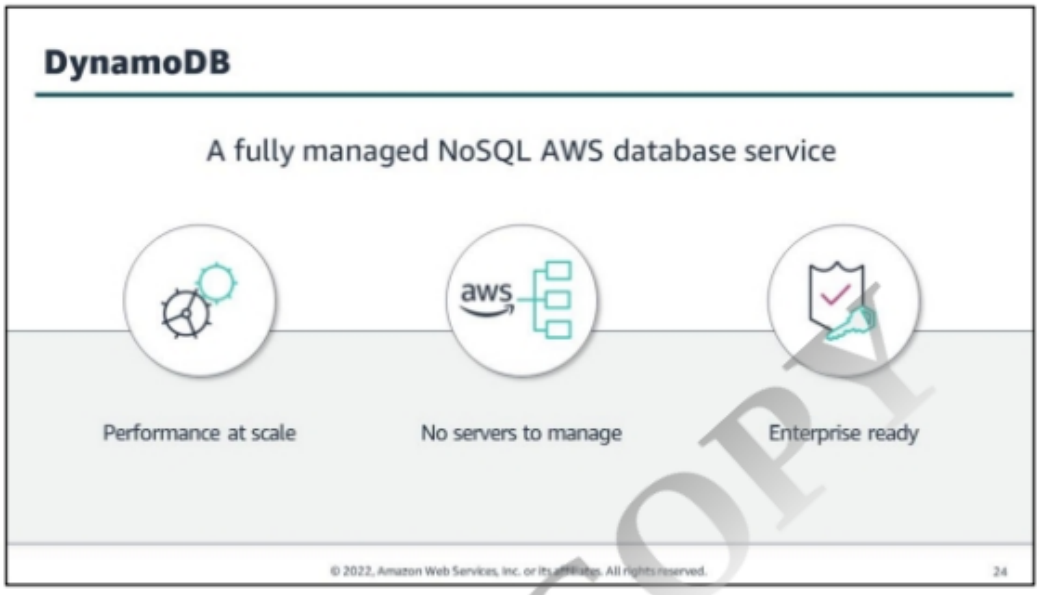
With Aurora Serverless v2, your database automatically scales capacity to meet the needs of the application's peak load and scales back down when the surge of activity is over. With Aurora Serverless v2, you no longer need to provision for peak or average capacity. You can specify an upper capacity limit to handle your most demanding workloads, and that capacity isn't used unless it's needed.

You also set your minimum capacity setting. The scaling rate for an Aurora Serverless v2 DB instance depends on its current capacity. The higher the current capacity, the faster it can scale up. If you need the DB instance to quickly scale up to a very high capacity, consider setting the minimum capacity to a value where the scaling rate meets your requirement. This type of automation is especially valuable for multitenant databases, distributed databases, development and test systems, and other environments with highly variable and unpredictable workloads.



The database services manager asks, "How can we build a scalable key-value NoSQL database?"

The database team is considering using NoSQL databases in some of their workloads. The company wants you to identify a high-performance key-value database solution.

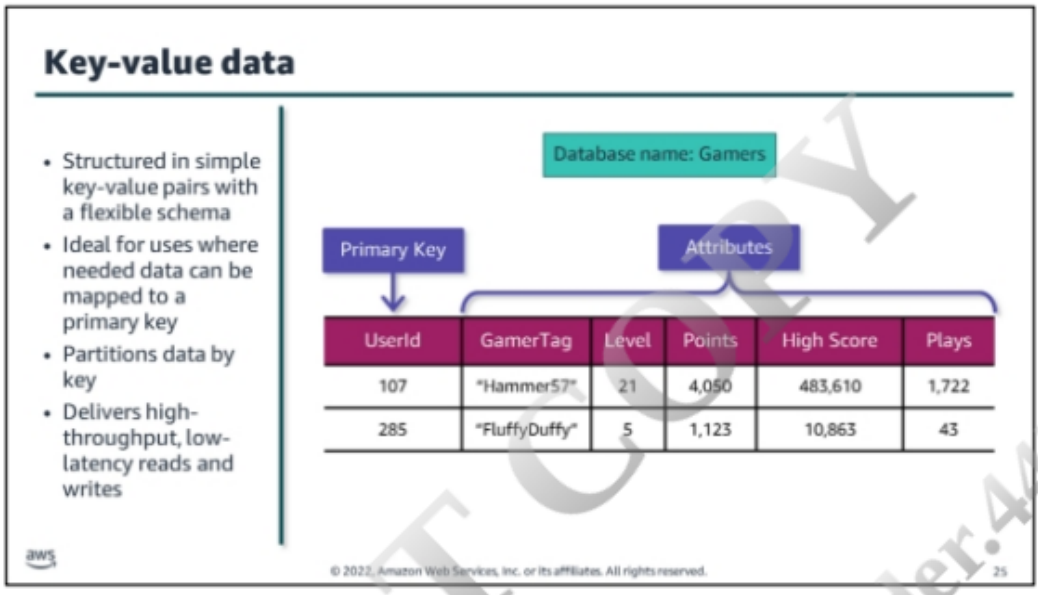


DynamoDB is a fully managed NoSQL database service. The complexity of running this massively scalable, distributed NoSQL database is managed by the service itself. The software developers can focus on building applications instead of managing infrastructure.

NoSQL databases are designed for scale, but their architectures are sophisticated and there can be significant operational overhead in running a large NoSQL cluster. DynamoDB removes the need to become an expert in advanced distributed computing concepts. You only need to learn DynamoDB's straightforward API using the SDK for the programming language of choice.

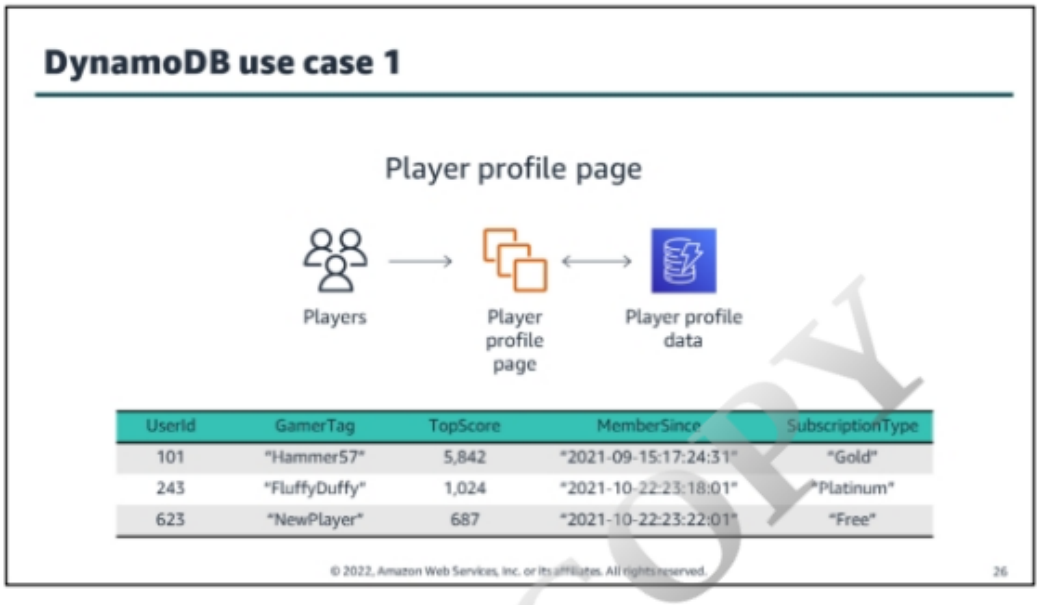
DynamoDB is cost effective. You pay for the storage you are consuming and the I/O throughput you have provisioned. It is designed to scale elastically while maintaining high performance. You can choose to provision a small amount of capacity when the storage and throughput requirements of an application are low. If you choose auto scaling, additional capacity is provisioned when the required I/O throughput increases, within limits set by you. The on-demand choice permits an application to seamlessly grow to support millions of users making thousands of concurrent requests to the database every second.

DynamoDB supports end-to-end encryption and fine-grained access control.



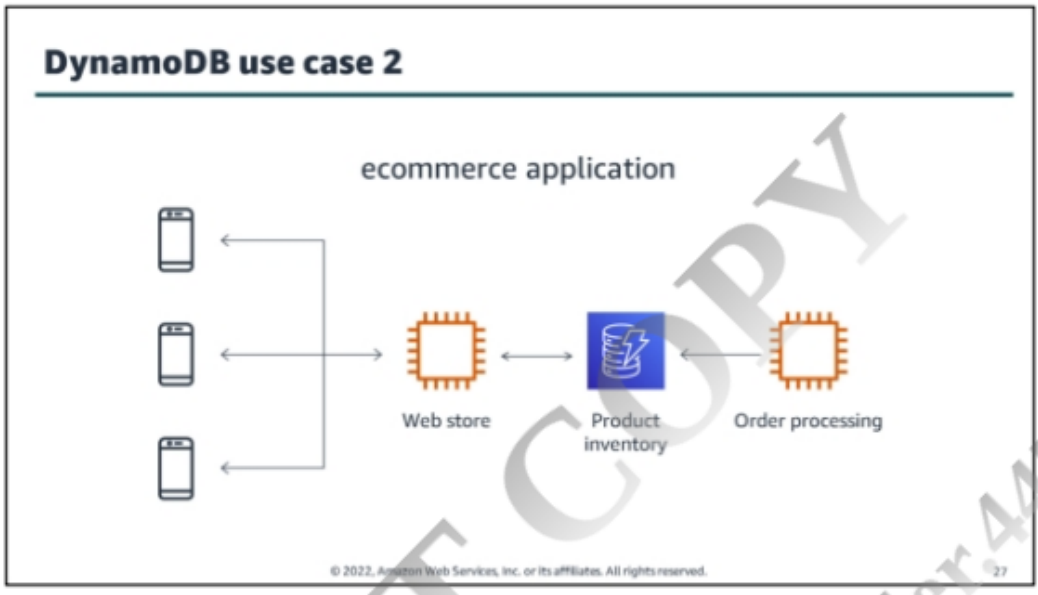
NoSQL databases use a variety of data models to access and manage data. These types of databases are optimized specifically for applications that require large data volume, low latency, and flexible data models. NoSQL databases are optimized by relaxing some of the data consistency restrictions of other databases. One common model is key-value data.

Key-value databases are good for use cases when the requested data can be associated with a single key. For example, you can store an application's user profile data in a key-value database and use the "UserID" value as the key. You can rapidly retrieve a user's data simply by requesting a specific "UserID."

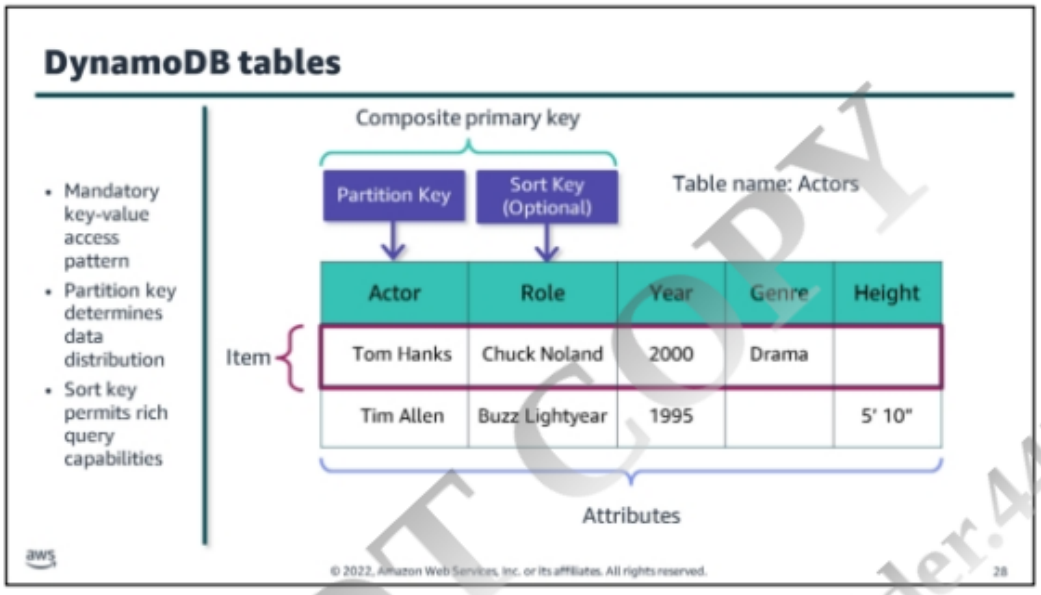


Game makers can support simple player profile pages by using DynamoDB. You can store user profile data in DynamoDB using "UserId" as the key. When a profile page loads, the application only needs to make a single read request to DynamoDB using the "UserId" value. In addition, when a new game launches, DynamoDB can scale rapidly to provide enough storage and throughput to support spikes in traffic.

For more information about gaming use cases, see "Amazon DynamoDB: Gaming use cases and design patterns" in the *AWS Database Blog* (https://aws.amazon.com/blogs/database/amazon-dynamodb-gaming-use-cases-and-design-patterns/).



Imagine you have an ecommerce application to sell products to your customers. You need the web store to display an accurate inventory. This becomes a significant challenge during peak traffic and holiday sales. DynamoDB supports a flexible schema to accommodate data for a variety of products. It can also manage many concurrent read and write operations while maintaining the accuracy of stored data.



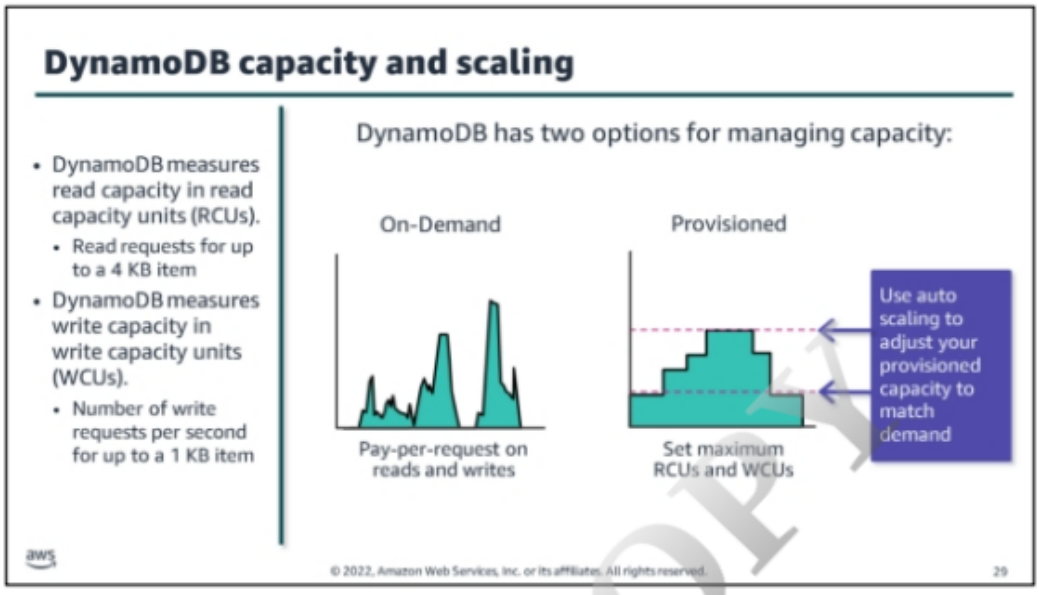
DynamoDB stores data in tables. When creating a table, you must specify a table name and a partition key. These are the only two required entities.

DynamoDB uses *primary keys* to uniquely identify each item in a table and *secondary indexes* to provide more query flexibility. There are two types of primary keys supported:

- **Simple primary key** – A simple primary key is composed of just one attribute designated as the *partition key*. If you use only the partition key, no two items can have the same value.
- **Composite primary key** – A composite primary key is composed of both a partition key and a *sort key*. In this case, the partition key value for multiple items can be the same, but their sort key values must be different.

You work with the core components: tables, items, and attributes. A *table* is a collection of *items*, and each item is a collection of *attributes*. In this example, the table includes two items, with simple primary keys *Tom Hanks* and *Tim Allen*. The item with the primary key *Tom Hanks* includes three attributes: *Role*, *Year*, and *Genre*. The primary key for *Tim Allen* includes a *Height* attribute, and it does not include the *Genre* attribute.

For more information about the components of DynamoDB, see "Core Components of Amazon DynamoDB" in the *Amazon DynamoDB Developer Guide* (https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.CoreComponents.html).



When planning for capacity in DynamoDB, you must consider the expected number of read/write requests per second as well as the size of those requests. This helps you choose the capacity mode for your table. It also helps you design your table to manage message size.

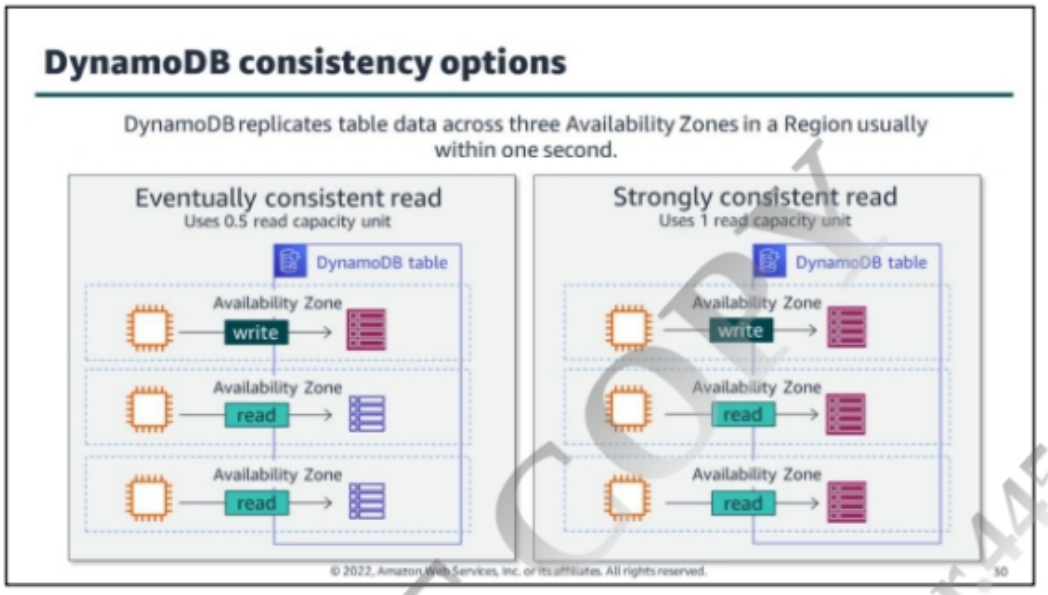
On-demand capacity mode is a pay-per-request model. On-demand capacity mode is best when you:

- Have unknown workloads
- Have unpredictable traffic
- Prefer to pay for only what you use

With provisioned capacity mode, you set a maximum number of RCUs and WCUs. When traffic exceeds those limits, DynamoDB throttles those requests to control your costs. You can adjust your provisioned capacity using auto scaling. Provisioned capacity mode is best when you:

- Have predictable application traffic
- Have traffic that is consistent or changes gradually
- Can forecast capacity requirements to control costs

For more information about DynamoDB auto scaling, see "Amazon DynamoDB auto scaling: Performance and cost optimization at any scale" in the *AWS Database Blog* (https://aws.amazon.com/blogs/database/amazon-dynamodb-auto-scaling-performance-and-cost-optimization-at-any-scale/).

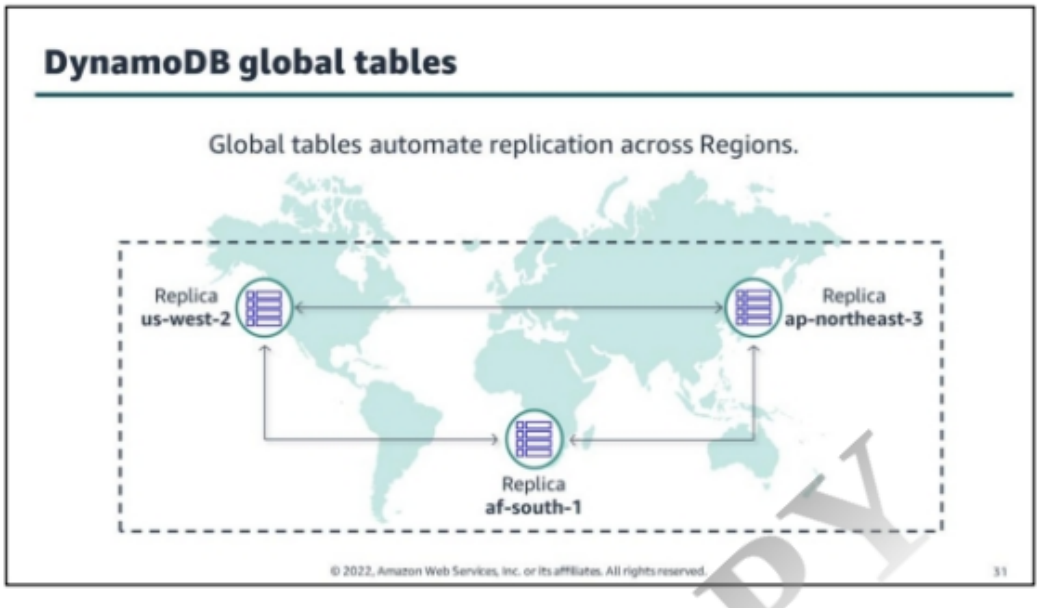


When your application writes data to a DynamoDB table and receives an HTTP 200 response (OK), the write has occurred and is durable. The data is eventually consistent across all storage locations, usually within one second or less. DynamoDB supports eventually consistent and strongly consistent reads.

Eventually consistent reads
When you read data from a DynamoDB table, the response might not reflect the results of a recently completed write operation. The response might include some stale data. If you repeat your read request after a short time, the response should return the latest data.

Strongly consistent reads
When you request a strongly consistent read, DynamoDB returns a response with the most up-to-date data, reflecting the updates from all prior write operations that were successful. A strongly consistent read might not be available if there is a network delay or outage.

DynamoDB uses eventually consistent reads, unless you specify otherwise. Read operations (such as GetItem, Query, and Scan) provide a ConsistentRead parameter. If you set this parameter to true, DynamoDB uses strongly consistent reads during the operation.



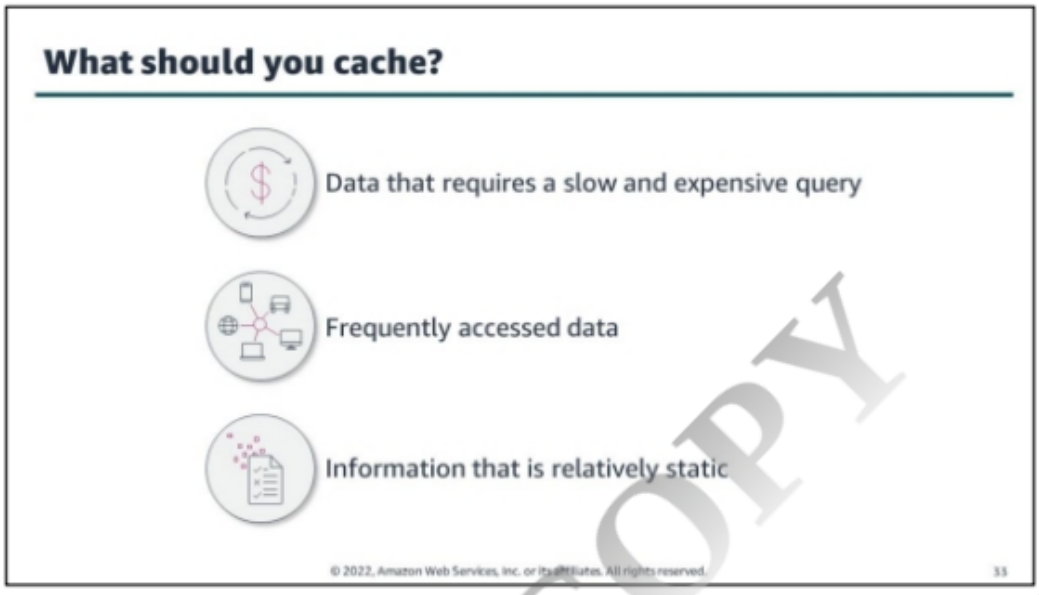
A *global table* is a collection of one or more DynamoDB tables, all owned by a single AWS account, identified as replica tables. A *replica table* (or *replica*, for short) is a single DynamoDB table that functions as part of a global table. Each replica stores the same set of data items. Any given global table can only have one replica table per Region, and every replica has the same table name and the same primary key schema.

DynamoDB global tables provide a fully managed solution for deploying a multi-Region, multi-active database, without having to build and maintain your own replication solution. When you create a global table, you specify the Regions where you want the table to be available. DynamoDB performs all the necessary tasks to create identical tables in these Regions and propagate ongoing data changes to all of them. DynamoDB communicates these changes over the AWS network backbone.

For more information about Amazon DynamoDB global tables, see "Amazon DynamoDB global tables" (https://aws.amazon.com/dynamodb/global-tables/).

The database services manager asks, "How can we cache databases in the cloud to maximize performance?"

The database team has identified some common queries that are causing a lot of read traffic. The company is looking for your advice on how to cache this commonly accessed data to improve performance and decrease the load on the databases.

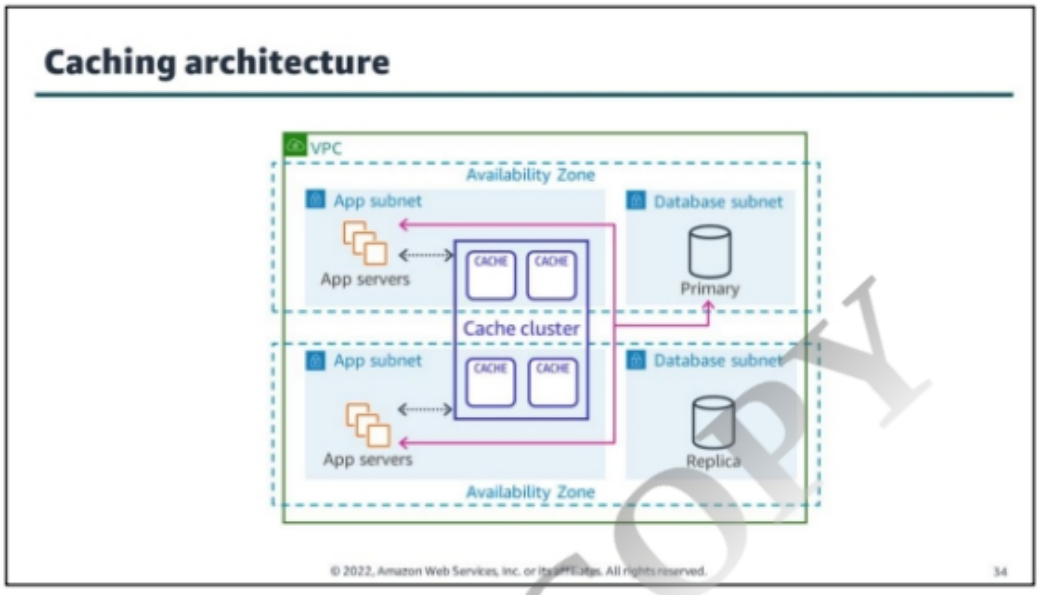


To determine if your application should use database caching, consider the following:

Speed and expense –Some database queries are inherently slower and more expensive than others. For example, queries that perform joins on multiple tables are significantly slower and more expensive than simple, single-table queries. If requesting data requires a slow and expensive query, it's a candidate for caching.

Data and access pattern – Determining what to cache also involves understanding the data itself and its access patterns. For example, it doesn't make sense to cache data that is rapidly changing or is seldom accessed. For caching to provide a meaningful benefit, the data should be relatively static and frequently accessed, such as a personal profile on a social media site.

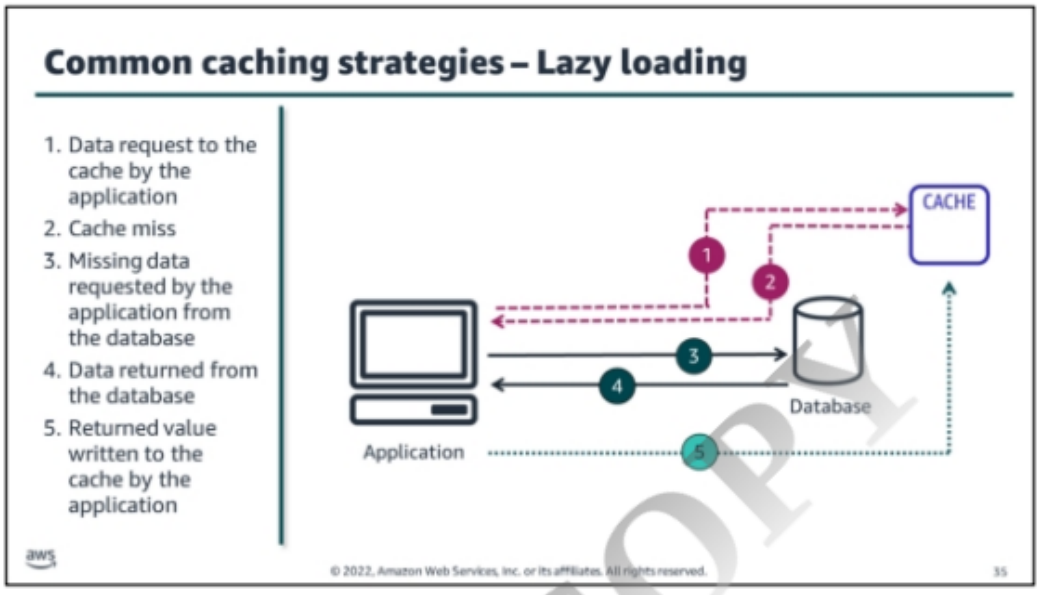
Cache validity – Data can become out-of-date while it is stored in cache. Writes that occur on that data in the database may not be reflected in that cached data. To determine whether your data is a candidate for caching, you need to determine your application's tolerance for occasionally inaccurate cache data.



Without caching, EC2 instances read and write directly to the database. With caching, an instance first attempts to read from a cache, which uses high performance memory. For example, database caches on AWS such as Amazon ElastiCache and DynamoDB Accelerator (DAX) are in-memory databases. They use a cache cluster that contains a set of cache nodes distributed between subnets. Resources within those subnets have high-speed access to those nodes.

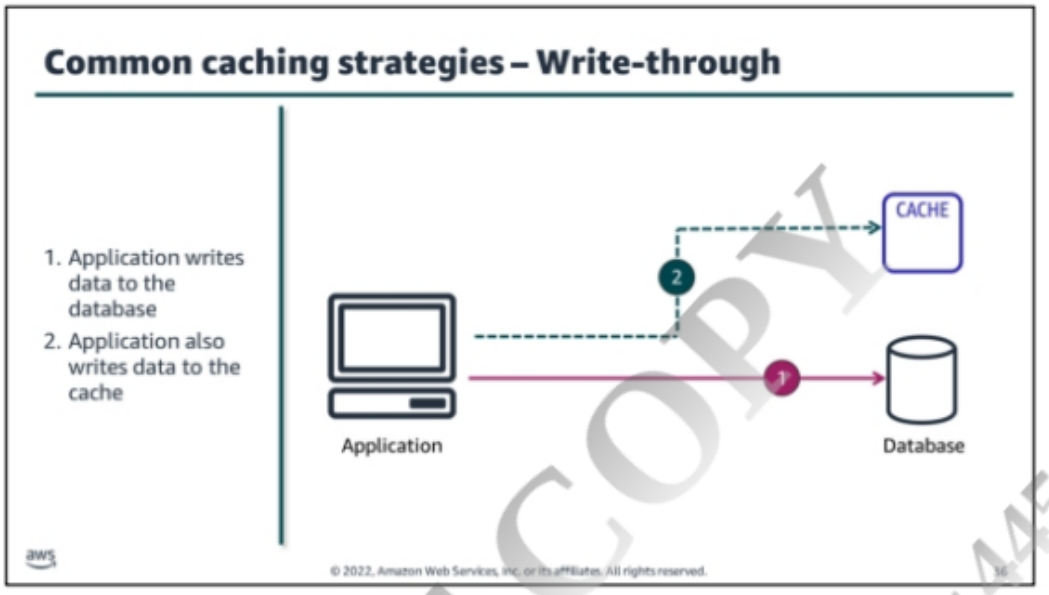
In this example, a VPC has an app subnet and data subnet in each of two Availability Zones. The application servers in both app subnets connect to a primary database in one data subnet. The other data subnet holds a replica database. A cache cluster spans both app subnets with cache nodes in each.

When you're using a cache for a backend data store, a side-cache is a common approach. Canonical examples include both Redis and Memcached. These are general-purpose caches that are decoupled from the underlying data store and can help with both read and write throughput, depending on the workload and durability requirements.



There are multiple strategies for keeping information in the cache in sync with the database. Two common caching strategies include *lazy loading* and *write-through*.

In lazy loading, updates are made to the database without updating the cache. In the case of a cache miss, the information retrieved from the database can be subsequently written to the cache. Lazy loading loads data needed by the application in the cache, but it can result in high cache-miss-to-cache-hit ratios in some use cases.



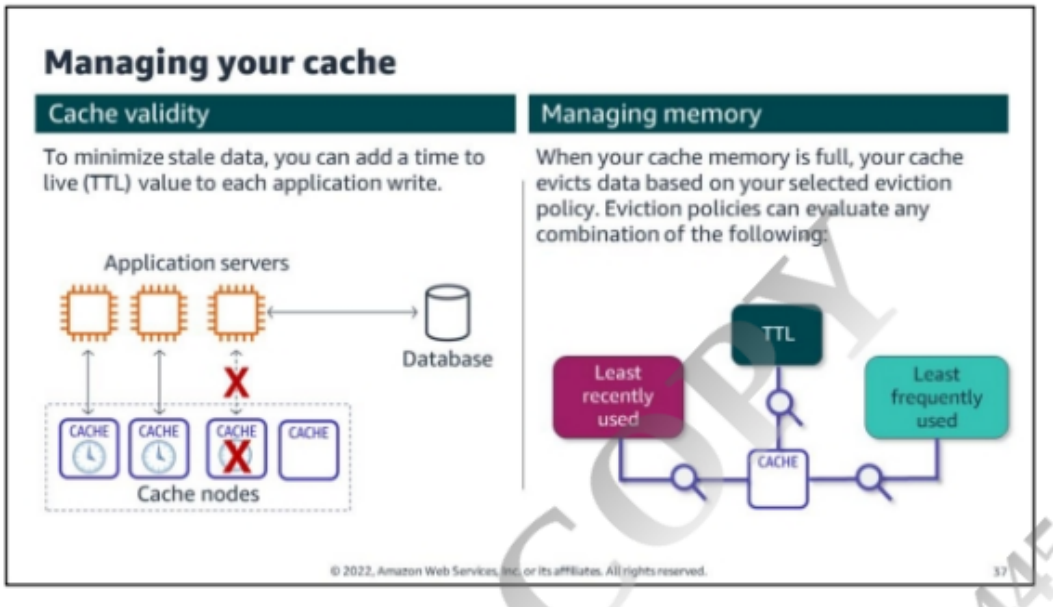
An alternative strategy is to write through to the cache every time the database is accessed. This approach results in fewer cache misses. This improves performance, but requires additional storage for data that may not be needed by the applications.

The best strategy depends on your use case. It is critical to understand the impact of stale data on your use case. If the impact is high, then consider maintaining freshness with write-throughs. If the impact is low, then lazy loading may be sufficient. It also helps to understand the frequency of change of the underlying data, because this affects the performance and cost tradeoffs of the caching strategies.

Once you decide on a strategy for maintaining your cache, you will need to implement this approach within your application.

For more information about ElastiCache, see "Caching strategies" in the following:

- *Amazon ElastiCache for Redis User Guide* (https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/Strategies.html).
- *Amazon ElastiCache for Memcached User Guide* (https://docs.aws.amazon.com/AmazonElastiCache/latest/mem-ug/Strategies.html).

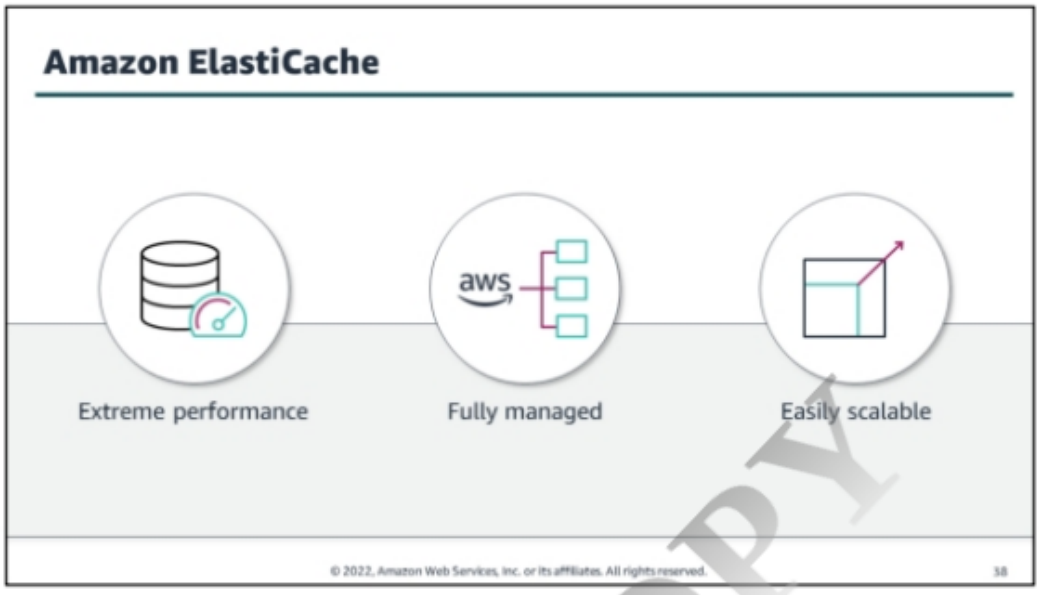


Lazy loading allows for stale data, but doesn't fail with empty nodes. Write-through maintains fresh data, but can fail with empty nodes and can populate the cache with superfluous data. By adding a time to live (TTL) value to each write to the cache, you can maintain fresh data without cluttering up the cache with extra data.

TTL is an integer value that specifies the number of seconds or milliseconds until the key expires. When an application attempts to read an expired key, it is treated as though the data is not found in cache, meaning that the database is queried and the cache is updated. This keeps data from getting too stale and requires that values in the cache are occasionally refreshed from the database.

When cache memory is full, the cache engine removes data from memory to make space for new data. It chooses this data based on the eviction policy you set. An eviction policy evaluates the following characteristics of your data:

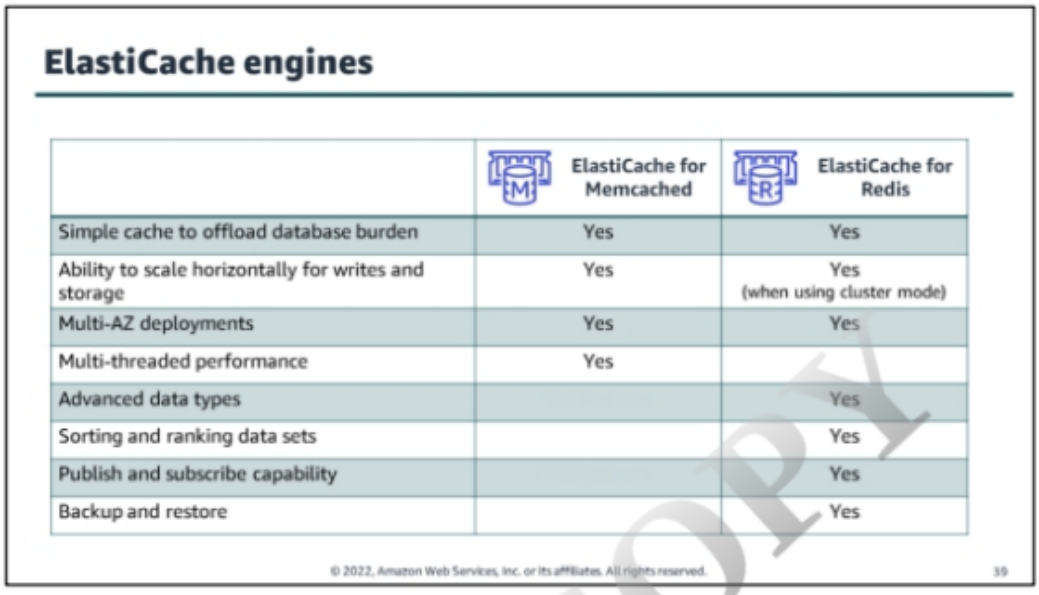
- Which were accessed least recently?
- Which have been accessed least frequently?
- Which have a TTL set and the TTL value?



Amazon ElastiCache is a web service that makes it easy to set up, manage, and scale a distributed in-memory data store or cache environment in the cloud. It provides a high-performance, scalable, and cost-effective caching solution. At the same time, it helps remove the complexity associated with deploying and managing a distributed cache environment.

ElastiCache supports two open-source in-memory engines: [in-memory compared to disk]
- Redis
- Memcached

For more information about ElastiCache, see "Amazon ElastiCache" (https://aws.amazon.com/elasticache/).

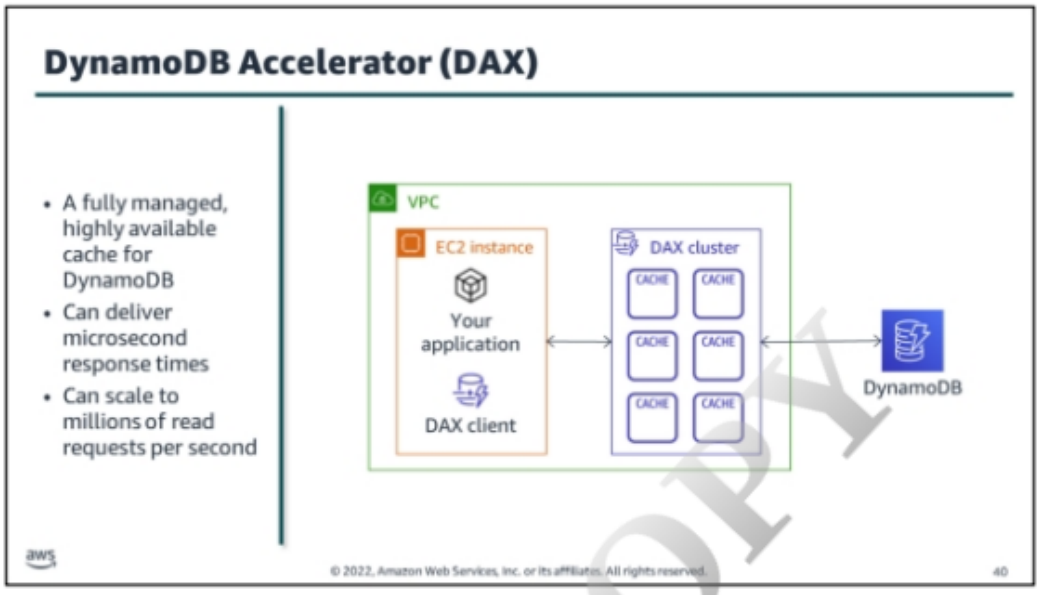


ElastiCache offers fully managed capabilities for two popular, open-source compatible in-memory data stores: Memcached and Redis.

With ElastiCache for Memcached, you can build a scalable caching tier for data-intensive apps. The service works as an in-memory data store and cache to support the most demanding applications requiring sub-millisecond response times. ElastiCache for Memcached is fully managed, scalable, and secure, making it an ideal candidate for use cases where frequently accessed data must be in memory. This engine offers a simple caching model with multi-threading. The service is a popular choice for use cases such as web, mobile apps, gaming, ad tech, and ecommerce. The Memcached-compatible service also supports Auto Discovery.

ElastiCache for Redis is an in-memory data store that provides sub-millisecond latency at internet scale. ElastiCache for Redis combines the speed, simplicity, and versatility of open-source Redis with manageability, security, and scalability from Amazon. It can power the most demanding real-time applications in gaming, ad tech, ecommerce, healthcare, financial services, and Internet of Things (IoT).

For more information, see "Comparing Redis and Memcached" (https://aws.amazon.com/elasticache/redis-vs-memcached/).

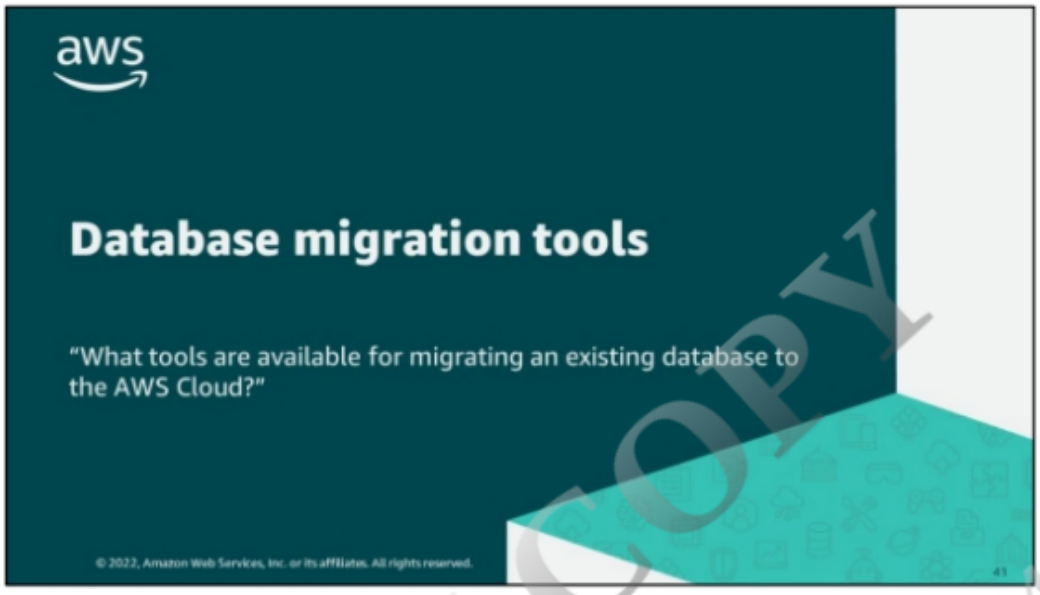


DynamoDB is designed for scale and performance. In most cases, the DynamoDB response times can be measured in single-digit milliseconds. However, there are certain use cases that require response times in microseconds. For those use cases, DynamoDB Accelerator (DAX) delivers fast response times for accessing eventually consistent data.

DAX is a caching service compatible with DynamoDB that provides fast in-memory performance for demanding applications.

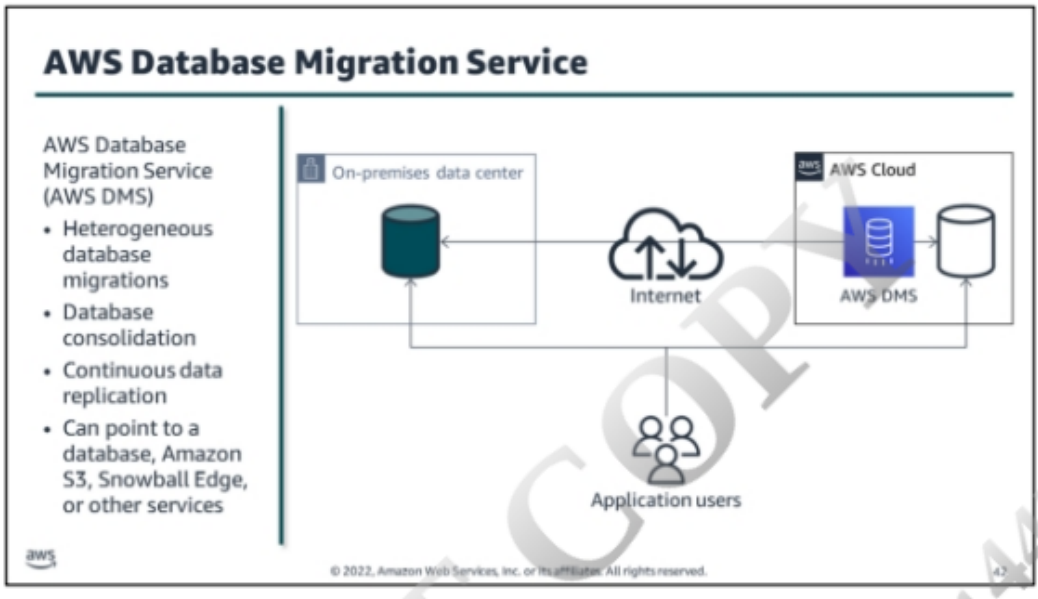
You create a DAX cluster in your Amazon VPC to store cached data closer to your application. You install a DAX client on the Amazon EC2 instance running your application in that VPC. At runtime, the DAX client directs all of your application's DynamoDB requests to the DAX cluster. If DAX can process a request directly, it does so. Otherwise, it passes the request through to DynamoDB.

For more information about memory acceleration, see "In-Memory Acceleration with DynamoDB Accelerator (DAX)" in the *Amazon DynamoDB Developer Guide* (https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DAX.html).



The database services manager asks, "What tools are available for migrating an existing database to the AWS Cloud?"

The database team is planning to move some of their on-premises databases to the cloud. The company wants you to identify tools that can help them with this process and minimize downtime.



AWS Database Migration Service (AWS DMS) replicates data from a source to a target database in the AWS Cloud. You create a source and a target connection to tell AWS DMS where to extract from and load to. Then you schedule a task that runs on this server to move your data. AWS DMS creates the tables and associated primary keys if they don't exist on the target.

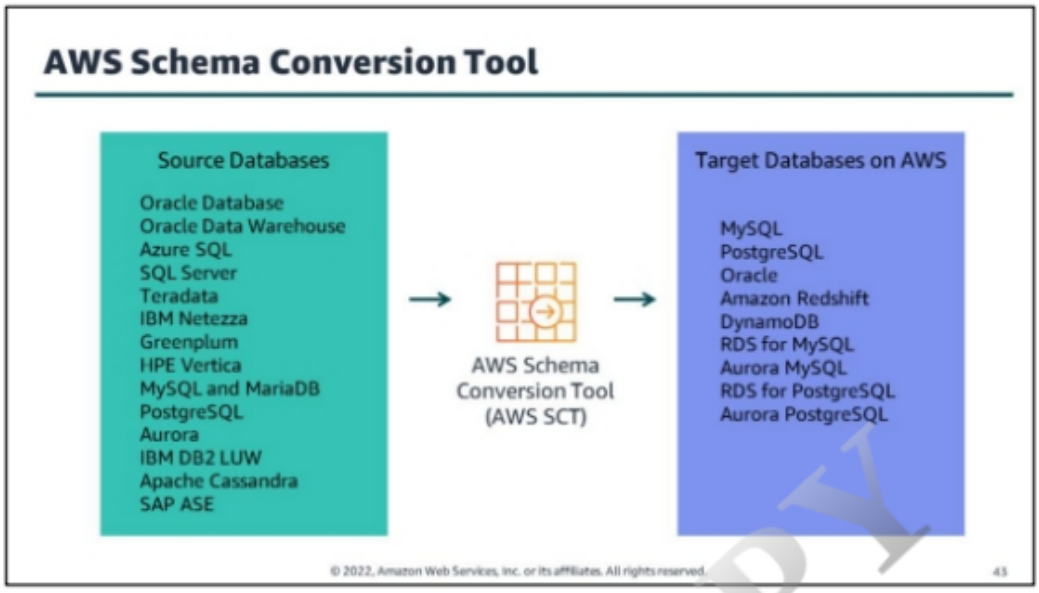
AWS DMS supports migration between the most widely used databases, which includes Oracle, PostgreSQL, SQL Server, Amazon Redshift, Aurora, MariaDB, and MySQL. It also supports homogenous (same engine) and heterogeneous (different engines) migrations. You can use the service to migrate between on-premises databases, Amazon EC2 databases, and Amazon RDS databases. However, you cannot migrate between two on-premises databases. Either the source or the target database (or both) need to reside in Amazon RDS or on Amazon EC2.

With AWS DMS, you can also use a Snowball edge device as a migration target. You would use this method if your environment has poor internet connectivity, the source database is too large to move over the internet, or if your organization has privacy or security requirements.

AWS DMS automatically handles formatting of the source data for consumption by the target database. It does not perform schema or code conversion.

For homogenous migrations, you can use native tools to perform these conversions. For heterogeneous migrations, you can use the AWS Schema Conversion Tool (AWS SCT).

For more information about AWS DMS, see "What is AWS Database Migration Service?" in the *AWS Database Migration Service User Guide* (https://docs.aws.amazon.com/dms/latest/userguide/Welcome.html).

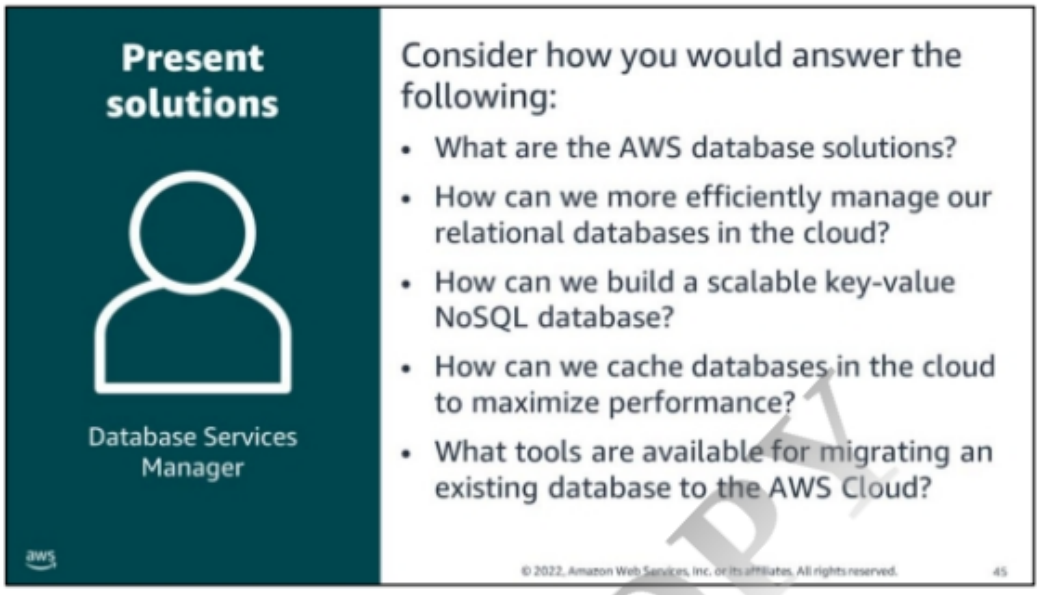


The AWS Schema Conversion Tool (AWS SCT) makes heterogeneous database migrations predictable. It automatically converts the source database schema and a majority of the database code objects. The conversion includes views, stored procedures, and functions. They are converted to a format that is compatible with the target database. Any objects that cannot be automatically converted are marked so that they can be manually converted to complete the migration.

The AWS SCT can also scan your application source code for embedded structured query language (SQL) statements and convert them as part of a database schema conversion project. During this process, the AWS SCT optimizes code built for the cloud by converting legacy Oracle and SQL Server functions to their equivalent AWS service, modernizing the applications at the same time of database migration.

Once the schema conversion is complete, the AWS SCT can help migrate data from a range of data warehouses to Amazon Redshift using built-in data migration agents.

For more information about the AWS SCT, see "AWS Schema Conversion Tool" (https://aws.amazon.com/dms/schema-conversion-tool/).



Imagine you are now ready to talk to the database services manager and present solutions that meet their architectural needs.

Think about how you would answer the questions from the beginning of the lesson.

Your answers should include the following solutions:
- AWS offers services that support relational and nonrelational databases.
- You can manage your relational databases efficiently with Amazon RDS and Amazon Aurora.
- You can manage your nonrelational key-value databases with Amazon DynamoDB.
- You can use Amazon ElastiCache and Amazon DynamoDB Accelerator for database caching.
- You can use AWS Database Migration Service to migrate your databases to the cloud.

Module review

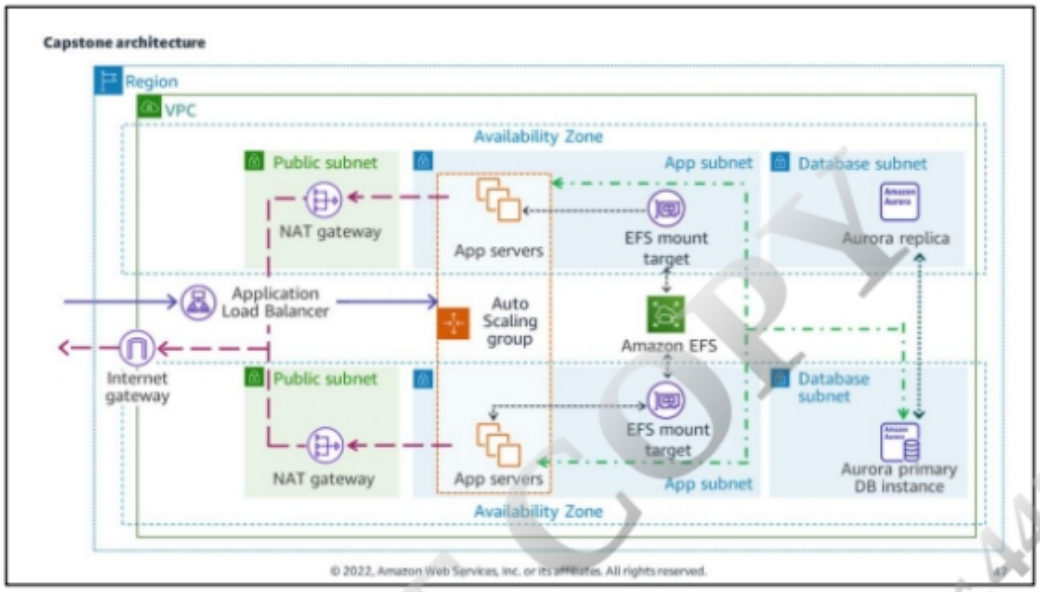
In this module you learned about:

- ✓ Database services
- ✓ Amazon RDS
- ✓ Amazon DynamoDB
- ✓ Database caching
- ✓ Database migration tools

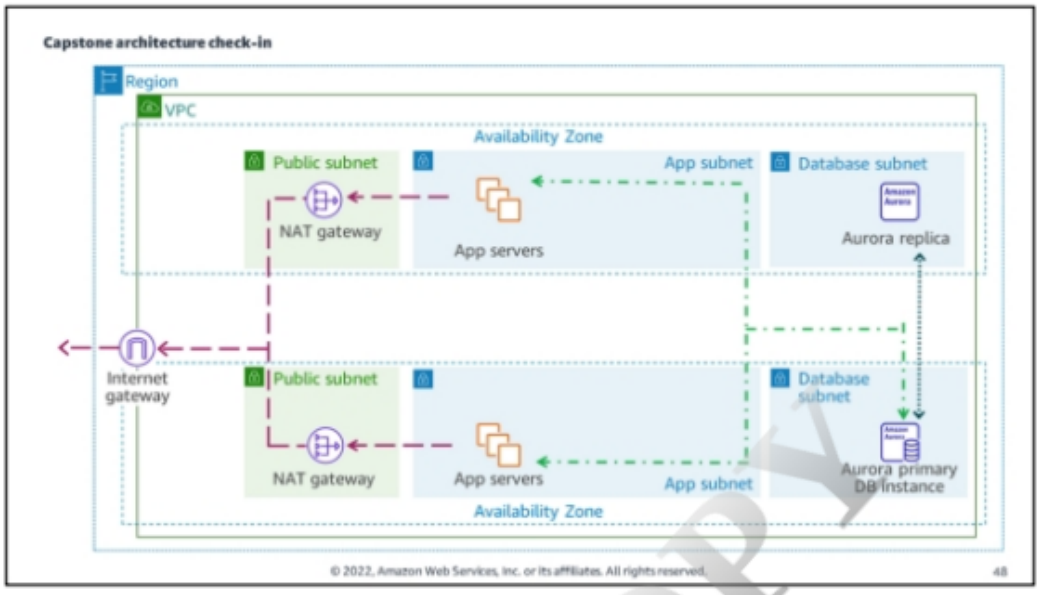
Next, you will review:

- Capstone check-in
- Lab introduction
- Knowledge check

 46



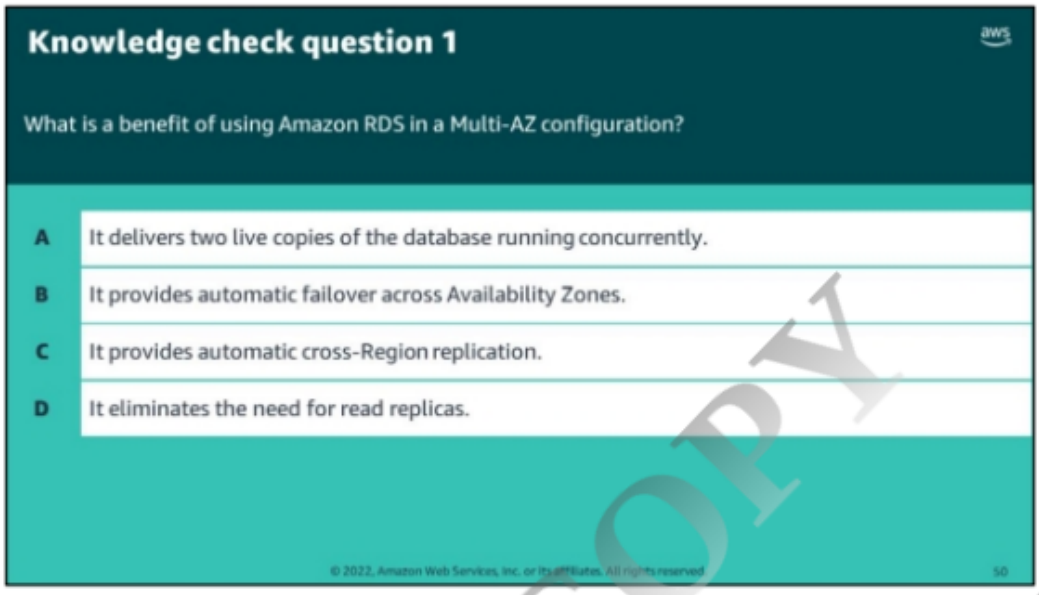
At the end of this course is a Capstone Lab project. You will be provided a scenario and asked to build an architecture based on project data, best practices, and the Well-Architected Framework.

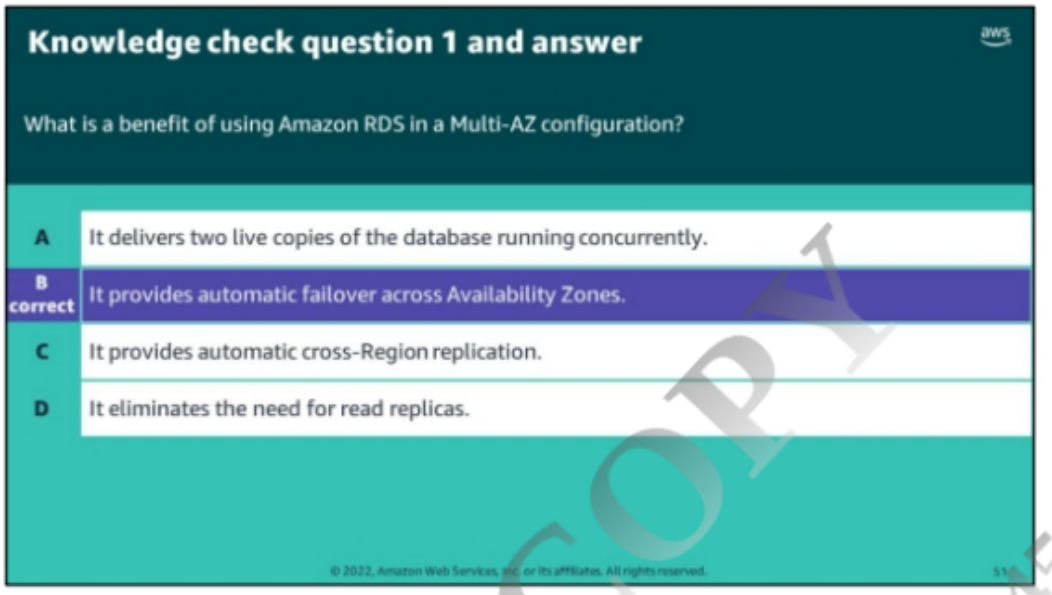


In this module, you explored AWS database services and resources.

Review the capstone architecture to explore some of the design decisions. This architecture helps you provide the following benefits:

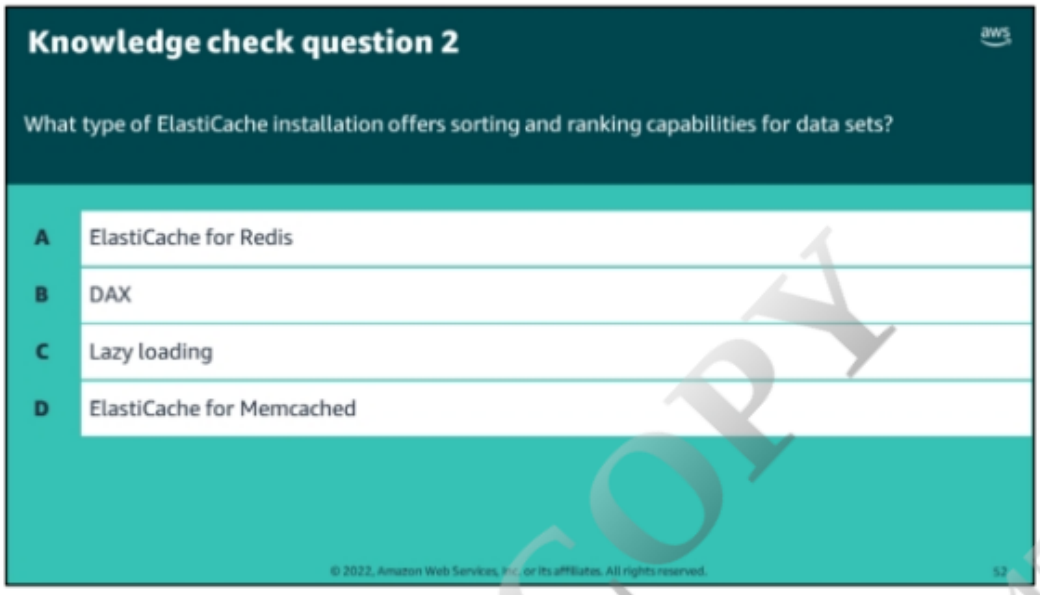
- In the capstone, you set up an application that requires a MySQL database. Amazon Aurora supports this engine. Using a managed service for hosting the database also provides the following benefits:
  - You do not have to manage server OS patches or database software updates.
  - Amazon Aurora helps you configure scaling, backups, and high availability.
- The capstone architecture achieves resiliency by using an Aurora replica. If the primary instance fails, the replica can be promoted to primary.

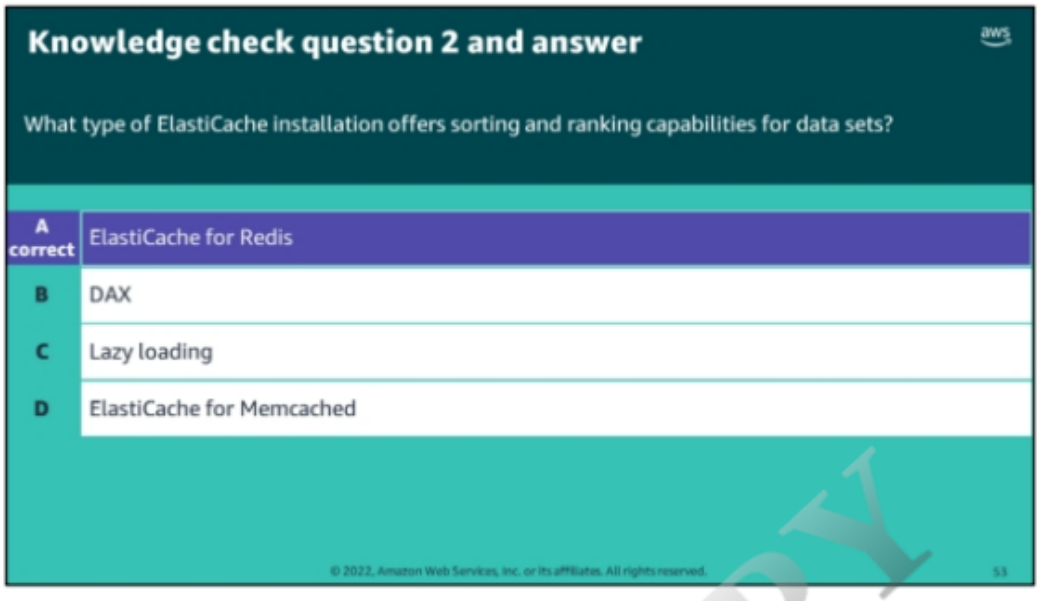




The correct answer is B, it provides automatic failover across Availability Zones.

When you provision a Multi-AZ DB instance, Amazon RDS synchronously replicates the data to a standby instance in a different Availability Zone. In case of the primary instance failure, Amazon RDS performs an automatic failover to the standby instance. Database operations can be resumed as soon as the failover is complete. Because the endpoint for your DB instance remains the same after a failover, your application can resume database operation without you needing to manually intervene in the administration.

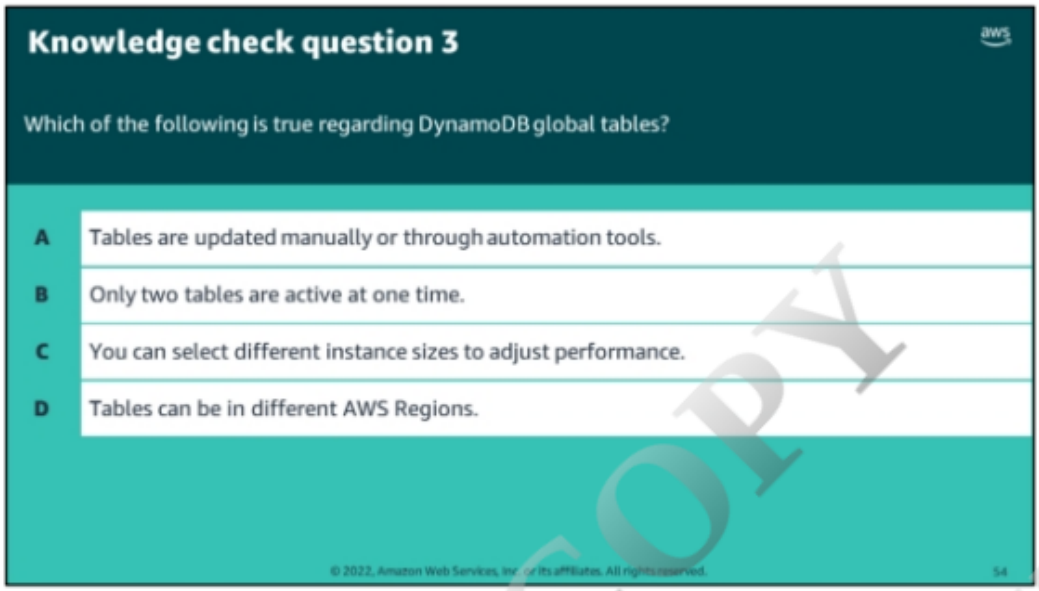


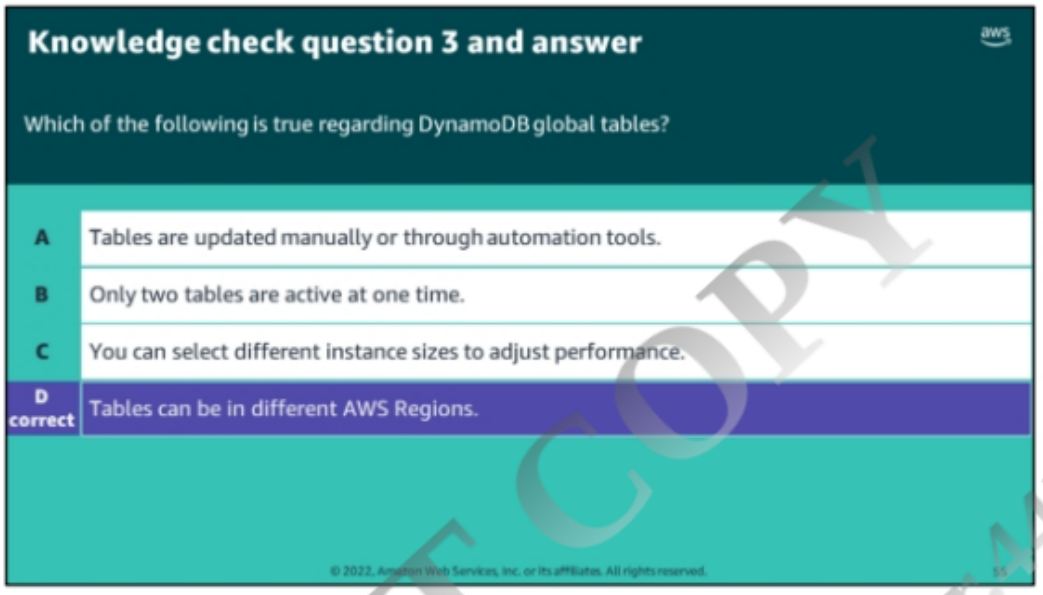


The correct answer is A, ElastiCache for Redis.

ElastiCache for Redis sorts and ranks data sets.

For more information about Redis and Memcached, see "Comparing Redis and Memcached" (https://aws.amazon.com/elasticache/redis-vs-memcached/) and the "Performance at Scale with Amazon ElastiCache" white paper (https://d0.awsstatic.com/whitepapers/performance-at-scale-with-amazon-elasticache.pdf).

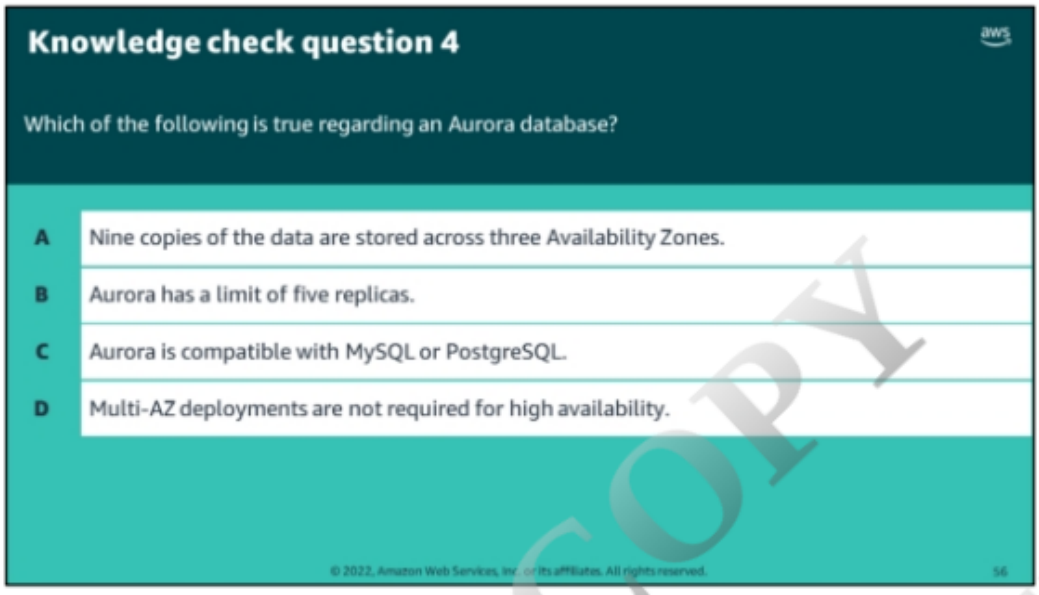


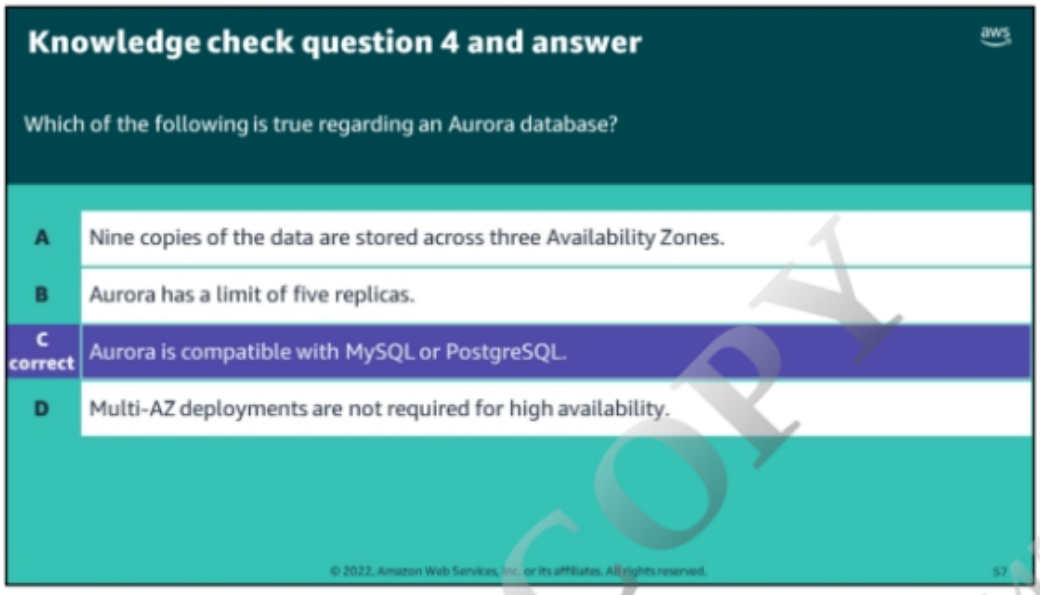


The correct answer is D, tables can be in different AWS Regions.

When you create a global table, you specify the Regions where you want the table to be available. DynamoDB performs all the necessary tasks to create identical tables in these Regions and propagate ongoing data changes to all of them.

For more information about Amazon DynamoDB global tables, see "Amazon DynamoDB global tables" (https://aws.amazon.com/dynamodb/global-tables/).

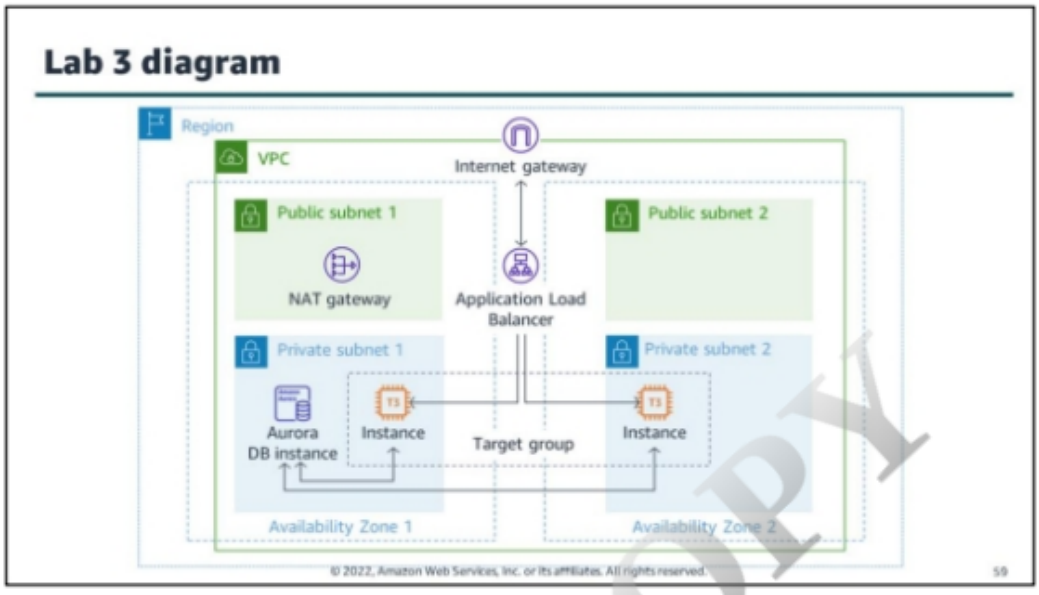




The correct answer is C, Aurora is compatible with MySQL or PostgreSQL.

Aurora is an enterprise-class relational database. It is compatible with MySQL or PostgreSQL relational databases. It is up to five times faster than standard MySQL databases and up to three times faster than standard PostgreSQL databases.

For more information about Amazon Aurora storage, see "Amazon Aurora storage and reliability" (https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Aurora.Overview.StorageReliability.html).



Lab 3 uses a VPC across two Availability Zones with a private and public subnet in each. Each private subnet contains a t3 EC2 instance. One private subnet contains an Aurora DB instance, which connects to a target group consisting of the two EC2 instances. Inbound traffic routes through an internet gateway into an Application Load Balancer. The load balancer distributes traffic between the EC2 instances.