# AWS® Certified Developer Official Study Guide

Associate (DVA-C01) Exam

Now that you know the purpose of these database services and what they can do, review the type of applications that can be used for each AWS database service. Refer to the application type mappings shown in Table 4.2.

**TABLE 4.2** Application Mapping to AWS Database Service

| Applications | Product |
|---|---|
| Transactional applications, such as ERP, CRM, and ecommerce to log transactions and store structured data. | Aurora or Amazon RDS |
| Internet-scale applications, such as hospitality, dating, and ride sharing, to serve content and store structured and unstructured data. | DynamoDB or Amazon DocumentDB |
| Analytic applications for operational reporting and querying terabyte- to exabyte-scale data. | Amazon Redshift |
| Real-time application use cases that require submillisecond latency such as gaming leaderboards, chat, messaging, streaming, and Internet of Things (IoT). | ElastiCache |
| Applications with use cases that require navigation of highly connected data such as social news feeds, recommendations, and fraud detection. | Neptune |
| Applications that collect data at millions of inserts per second in a time-series fashion, for example clickstream data and IoT devices. | Timestream |
| Applications that require an accurate history of their application data; for example, tracking the history of credits and debits in banking transactions or verifying the audit trails created in relational databases. | Amazon QLDB |

## Relational Databases

Many developers have had to interact with relational databases in their applications. This section describes first what a relational database is. Then, it covers how you can run a relational database in the AWS Cloud with Amazon RDS or on Amazon EC2.

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. Tables store information about the *objects* to be represented in the database. Each *column* in a table holds certain 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. Each row in a table contains a unique identifier called a *primary key*, and rows among multiple tables can be linked by using *foreign keys*. You can access data in many different ways without reorganizing the database tables.

## Characteristics of Relational Databases

Relational databases include four important characteristics: Structured Query Language, data integrity, transactions, and atomic, consistent, isolated, and durable compliance.

### Structured Query Language

*Structured query language (SQL)* is the primary interface that you use to communicate with relational databases. The standard American National Standards Institute (ANSI) SQL is supported by all popular relational database engines. Some of these engines have extensions to ANSI SQL to support functionality that is specific to that engine. You use SQL to add, update, or delete data rows; to retrieve subsets of data for transaction processing and analytics applications; and to manage all aspects of the database.

### Data Integrity

*Data integrity* is the overall completeness, accuracy, and consistency of data. Relational databases use a set of constraints to enforce data integrity in the database. These include primary keys, foreign keys, NOT NULL constraints, unique constraint, default constraints, and check constraints. These integrity constraints help enforce business rules in the tables to ensure the accuracy and reliability of the data. In addition, most relational databases enable you to embed custom code triggers that execute based on an action on the database.

### Transactions

A database *transaction* is one or more SQL statements that execute as a sequence of operations to form a single logical unit of work. Transactions provide an all-or-nothing proposition, meaning that the entire transaction must complete as a single unit and be written to the database, or none of the individual components of the transaction will continue. In relational database terminology, a transaction results in a `COMMIT` or a `ROLLBACK`. Each transaction is treated in a coherent and reliable way, independent of other transactions.

### ACID Compliance

All database transactions must be *atomic, consistent, isolated, and durable (ACID)*–compliant or be atomic, consistent, isolated, and durable to ensure data integrity.

**Atomicity** *Atomicity* requires that the transaction as a whole executes successfully, or if a part of the transaction fails, then the entire transaction is invalid.

**Consistency** *Consistency* mandates that the data written to the database as part of the transaction must adhere to all defined rules and restrictions, including constraints, cascades, and triggers.

**Isolation** *Isolation* is critical to achieving concurrency control, and it makes sure that each transaction is independent unto itself.

**Durability** *Durability* requires that all of the changes made to the database be permanent when a transaction is successfully completed.

## Managed vs. Unmanaged Databases

Managed database services on AWS, such as Amazon RDS, enable you to offload the administrative burdens of operating and scaling distributed databases to AWS so that you don't have to worry about the following tasks:

- Hardware provisioning
- Setup and configuration
- Throughput capacity planning
- Replication
- Software patching
- Cluster scaling

AWS provides a number of database alternatives for developers. As a managed database, Amazon RDS enables you to run a fully featured relational database while offloading database administration. By contrast, you can run unmanaged databases on Amazon EC2, which gives you more flexibility on the types of databases that you can deploy and configure; however, you are responsible for the administration of the unmanaged databases.

### Amazon Relational Database Service

With *Amazon Relational Database Service (Amazon RDS)*, you can set up, operate, and scale a relational database in the AWS Cloud. It provides cost-efficient, resizable capacity for open-standard relational database engines. Amazon RDS is easy to administer, and you do not need to install the database software. Amazon RDS manages time-consuming database administration tasks, which frees you up to focus on your applications and business. For example, Amazon RDS automatically patches the database software and backs up your database. The Amazon RDS managed relational database service works with the popular database engines depicted in Figure 4.1.

**FIGURE 4.1** Amazon RDS database engines

Amazon RDS assumes many of the difficult or tedious management tasks of a relational database:

## Procurement, configuration, and backup tasks

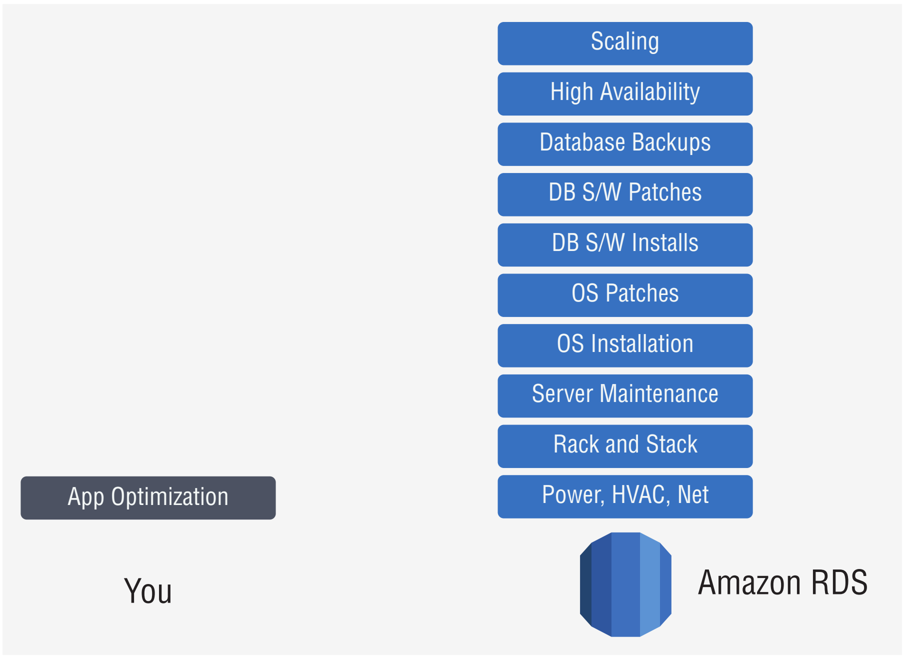
- When you buy a server, you get a central processing unit (CPU), memory, storage, and input/output operations per second (IOPS) all bundled together. With Amazon RDS, these are split apart so that you can scale them independently and allocate your resources as you need them.
- Amazon RDS manages backups, software patches, automatic failure detection, and recovery.
- You can configure automated backups or manually create your own backup snapshot and use these backups to restore a database. The Amazon RDS restore process works reliably and efficiently.
- You can use familiar database products: MySQL, MariaDB, PostgreSQL, Oracle, Microsoft SQL Server, and the MySQL- and PostgreSQL-compatible Amazon Aurora DB engine.

## Security and availability

- You can enable the encryption option for your Amazon RDS DB instance.
- You can get high availability with a primary instance and a synchronous secondary instance that you can fail over to when problems occur. You can also use MySQL, MariaDB, or PostgreSQL read replicas to increase read scaling.
- In addition to the security in your database package, you can use AWS Identity and Access Management (IAM) to define users, and permissions help control who can access your Amazon RDS databases. You can also help protect your databases by storing them in a virtual private cloud (VPC).
- To deliver a managed service experience, Amazon RDS does not provide shell access to DB instances, and it restricts access to certain system procedures and tables that require advanced permissions.

When you host databases on Amazon RDS, AWS is responsible for the items in Figure 4.2.

**FIGURE 4.2** Amazon RDS host responsibilities



## Relational Database Engines on Amazon RDS

Amazon RDS provides six familiar database engines: Amazon Aurora, Oracle, Microsoft SQL Server, PostgreSQL, MySQL, and MariaDB. Because Amazon RDS is a managed service, you gain a number of benefits and features built right into the Amazon RDS service. These features include, but are not limited to, the following:

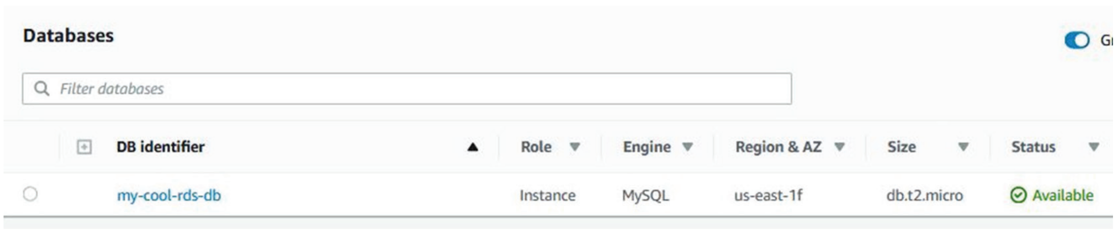
- Automatic software patching
- Easy vertical scaling
- Easy storage scaling
- Read replicas
- Automatic backups
- Database snapshots
- Multi-AZ deployments
- Encryption
- IAM DB authentication
- Monitoring and metrics with Amazon CloudWatch

To create an Amazon RDS instance, you can run the following command from the AWS CLI:

```
aws rds create-db-instance \
--db-instance-class db.t2.micro \
--allocated-storage 30 \
--db-instance-identifier my-cool-rds-db --engine mysql \
--master-username masteruser --master-user-password masterpassword1!
```

Depending on the configurations chosen, the database can take several minutes before it is active and ready for use. You can monitor the Amazon RDS Databases console to view the status. When the status states `Available`, it is ready to be used, as shown in Figure 4.3.

**FIGURE 4.3** Amazon RDS Databases console



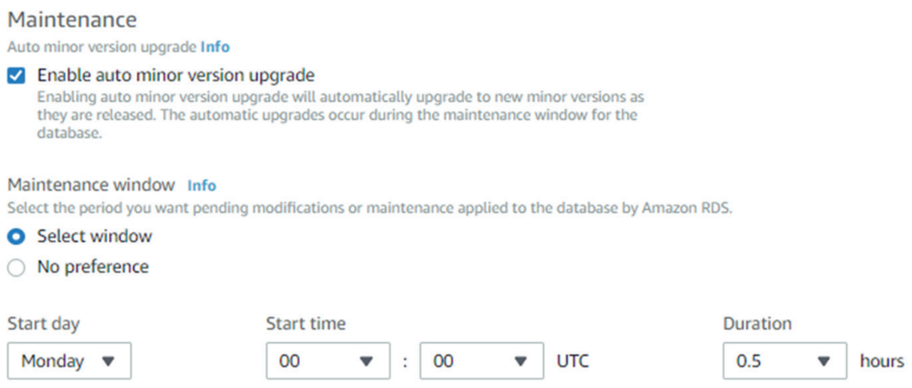
| | DB identifier | Role | Engine | Region & AZ | Size | Status |
|---|---|---|---|---|---|---|
| ○ | my-cool-rds-db | Instance | MySQL | us-east-1f | db.t2.micro | Available |

## Automatic Software Patching

Periodically, Amazon RDS performs maintenance on Amazon RDS resources. Maintenance mostly involves patching the Amazon RDS database underlying operating system (OS) or database engine version. Because this is a managed service, Amazon RDS handles the patching for you.

When you create an Amazon RDS database instance, you can define a maintenance window. A *maintenance window* is where you can define a period of time when you want to apply any updates or downtime to your database instance. You also can enable the automatic minor version upgrade feature, which automatically applies any new minor versions of the database as they are released (see Figure 4.4).

**FIGURE 4.4** Maintenance window



Maintenance

Auto minor version upgrade Info

☑ Enable auto minor version upgrade
Enabling auto minor version upgrade will automatically upgrade to new minor versions as they are released. The automatic upgrades occur during the maintenance window for the database.

Maintenance window Info
Select the period you want pending modifications or maintenance applied to the database by Amazon RDS.

● Select window
○ No preference

Start day: Monday
Start time: 00 : 00 UTC
Duration: 0.5 hours

You can select a maintenance window by using the AWS Management Console, AWS CLI, or Amazon RDS API. After selecting the maintenance window, the Amazon RDS instance is upgraded (if upgrades are available) during that time window. You can also modify the maintenance window by running the following AWS CLI command:

```
aws rds modify-db-instance --db-instance-identifer your-db-instance-identifer
--preferred-maintenance-window Mon:07:00-Mon:07:30
```

### Vertical Scaling

If your database needs to handle a bigger load, you can vertically scale your Amazon RDS instance. At the time of this writing, there are 40 available DB instance classes, which enable you to choose the number of virtual CPUs and memory available. This gives you flexibility over the performance and cost of your Amazon RDS database. To scale the Amazon RDS instance, you can use the console, AWS CLI, or AWS SDK.

If you are in a Single-AZ configuration for your Amazon RDS instance, the database is unavailable during the scaling operation. However, if you are in a Multi-AZ configuration, the standby database is upgraded first and then a failover occurs to the newly configured database. You can also apply the change during the next maintenance window. This way, your upgrade can occur during your normal outage windows.

To scale the Amazon RDS database by using the AWS CLI, run the following command:

```
aws rds modify-db-instance --db-instance-identifer your-db-instance-identifer
--db-instance-class db.t2.medium
```

### Easy Storage Scaling

Storage is a critical component for any database. Amazon RDS has the following three storage types:

**General Purpose SSD (gp2)** This storage type is for cost-effective storage that is ideal for a broad range of workloads. Gp2 volumes deliver single-digit millisecond latencies and the ability to burst to 3,000 IOPS for extended periods of time. The volume's size determines the performance of gp2 volumes.

**Provisioned IOPS (io1)** This storage type is for input/output-intensive workloads that require low input/output (I/O) latency and consistent I/O throughput.

**Magnetic Storage** This storage type is designed for backward compatibility, and AWS recommends that you use General Purpose SSD or Provisioned IOPS for any new Amazon RDS workloads.

To scale your storage, you must modify the Amazon RDS DB instance by executing the following AWS CLI command:

```
aws rds modify-db-instance --db-instance-identifer your-db-instance-identifer
--allocated-storage 50 --storage-type io1 --iops 3000
```

This command modifies your storage to 50 GB in size, with a Provisioned IOPS storage drive and a dedicated IOPS of 3000. While modifying the Amazon RDS DB instance, consider the potential downtime.

### Read Replicas (Horizontal Scaling)

There are two ways to scale your database tier with Amazon RDS: vertical scaling and horizontal scaling. Vertical scaling takes the primary database and increases the amount of memory and vCPUs allocated for the primary database. Alternatively, use horizontal scaling (add another server) to your database tier to improve the performance of applications that are read-heavy as opposed to write-heavy.

Read replicas create read-only copies of your master database, which allow you to offload any reads (or SQL SELECT statements) to the read replica. The replication from the master database to the read replica is asynchronous. As a result, the data queried from the read replica is not the latest data. If your application requires strongly consistent reads, consider an alternative option.

At the time of this writing, Amazon RDS MySQL, PostgreSQL, and MariaDB support up to five read replicas, and Amazon Aurora supports up to 15 read replicas. Microsoft SQL Server and Oracle do not support read replicas.

To create a read replica by using AWS CLI, run the following command:

```
aws rds create-db-instance-read-replica --db-instance-identifier your-db-
instance-identifier --source-db-instance-identifier your-source-db
```

## Backing Up Data with Amazon RDS

Amazon RDS has two different ways of backing up data of your database instance: automated backups and database snapshots (DB snapshots).

### Automated Backups (Point-in-Time)

With Amazon RDS, automated backups offer a point-in-time recovery of your database. When enabled, Amazon RDS performs a full daily snapshot of your data that is taken during your preferred backup window. After the initial backup is taken (each day), then Amazon RDS captures transaction logs as changes are made to the database.

After you initiate a point-in-time recovery, to restore your database instance, the transaction logs are applied to the most appropriate daily backup. You can perform a restore up to the specific second, as long as it's within your retention period. The default retention period is seven days, but it can be a maximum of up to 35 days.

To perform a restore, you must choose the `Latest Restorable Time`, which is typically within the last 5 minutes. For example, suppose that the current date is February 14 at 10 p.m., and you would like to do a point-in-time restore of February 14 at 9 p.m. This restore would succeed because the `Latest Restorable Time` is a maximum of February 14 at 9:55 p.m. (which is the last 5-minute window). However, a point-in-time restore of February 14 at 9:58 p.m. would fail, because it is within the 5-minute window.

Automated backups are kept until the source database is deleted. After the source Amazon RDS instance is removed, the automated backups are also removed.

### Database Snapshots (Manual)

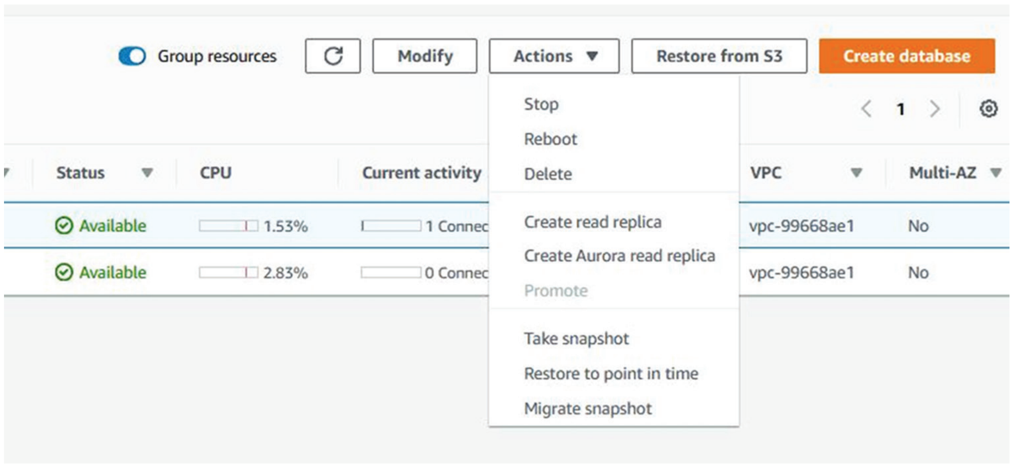
Unlike automated backups, database snapshots with Amazon RDS are user-initiated and enable you to back up your database instance in a known state at any time. You can also restore to that specific snapshot at any time.

Similar to the other Amazon RDS features, you can create the snapshots through the AWS Management Console, with the `CreateDBSnapshot` API, or with the AWS CLI.

With DB snapshots, the backups are kept until you explicitly delete them; therefore, before removing any Amazon RDS instance, take a final snapshot before removing it. Regardless of the backup taken, storage I/O may be briefly suspended while the backup process initializes (typically a few seconds), and you may experience a brief period of elevated latency. A way to avoid these types of suspensions is to deploy in a Multi-AZ configuration. With such a deployment, the backup is taken from the standby instead of the primary database.

To create a snapshot of the database, from the Amazon RDS Databases console, under Actions, select the Take snapshot option (see Figure 4.5). After a snapshot is taken, you can view all of your snaps from the Snapshots console.

**FIGURE 4.5** Taking an Amazon RDS snapshot



| Status | CPU | Current activity | VPC | Multi-AZ |
|---|---|---|---|---|
| Available | 1.53% | 1 Connec | vpc-99668ae1 | No |
| Available | 2.83% | 0 Connec | vpc-99668ae1 | No |

## Multi-AZ Deployments

By using Amazon RDS, you can run in a Multi-AZ configuration. In a Multi-AZ configuration, you have a primary and a standby DB instance. Updates to the primary database replicate synchronously to the standby replica in a different Availability Zone. The primary benefit of Multi-AZ is realized during certain types of planned maintenance, or in the unlikely event of a DB instance failure or an Availability Zone failure. Amazon RDS automatically fails over to the standby so that you can resume your workload as soon as the standby is promoted to the primary. This means that you can reduce your downtime in the event of a failure.

Because Amazon RDS is a managed service, Amazon RDS handles the fail to the standby. When there is a DB instance failure, Amazon RDS automatically promotes the standby to the primary—you will not interact with the standby directly. In other words, you will receive one endpoint connection for the Amazon RDS cluster, and Amazon RDS handles the failover.

Amazon RDS Multi-AZ configuration provides the following benefits:

- Automatic failover; no administration required
- Increased durability in the unlikely event of component failure
- Increased availability in the unlikely event of an Availability Zone failure
- Increased availability for planned maintenance (automated backups; I/O activity is no longer suspended)

To create an Amazon RDS instance in a Multi-AZ configuration, you must specify a subnet group that has two different Availability Zones specified. You can specify a Multi-AZ configuration by using AWS CLI by adding the `--multi-az` flag to the AWS CLI command, as follows:

```
aws rds create-db-instance \
--db-instance-class db.t2.micro \
--allocated-storage 30 \
--db-instance-identifier multi-az-rds-db --engine mysql \
--master-username masteruser \
--master-user-password masterpassword1! \
--multi-az
```

## Encryption

For encryption at rest, Amazon RDS uses the AWS Key Management Service (AWS KMS) for AES-256 encryption. You can use a default master key or specify your own for the Amazon RDS DB instance. Encryption is one of the few options that must be configured when the DB instance is created. You cannot modify an Amazon RDS database to enable encryption. You can, however, create a DB snapshot and then restore to an encrypted DB instance or cluster.

Amazon RDS supports using the Transparent Data Encryption (TDE) for Oracle and SQL Server. For more information on TDE with Oracle and Microsoft SQL Server, see the following:

- Microsoft SQL Server Transparent Data Encryption Support at:

  `https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Appendix.SQLServer.Options.TDE.html`

- Options for Oracle DB Instances:

  `https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Appendix.Oracle.Options.html#Appendix.Oracle.Options.AdvSecurity`

At the time of this writing, the following Amazon RDS DB instance types are not supported for encryption at rest:

- Db.m1.small
- Db.m1.medium
- Db.m1.large
- Db.m1.xlarge
- Db.m2.xlarge
- Db.m2.2xlarge
- Db.m2.4xlarge
- Db.t2.micro

For encryption in transit, Amazon RDS generates an SSL certificate for each database instance that can be used to connect your application and the Amazon RDS instance. However, encryption is a compute-intensive operation that increases the latency of your database connection. For more information, see the documentation for the specific database engine.

## IAM DB Authentication

You can authenticate to your DB instance by using IAM. By using IAM, you can manage access to your database resources centrally instead of storing the user credentials in each database. The IAM feature also encrypts network traffic to and from the database by using SSL.

IAM DB authentication is supported only for MySQL and PostgreSQL. At the time of this writing, the following MySQL versions are supported:

- MySQL 5.6.34 or later
- MySQL 5.7.16 or later

There's no support for the following:

- IAM DB Authentication for MySQL 5.5 or MySQL 8.0
- db.t2.small and db.m1.small instances

The following PostgreSQL versions are supported:

- PostgreSQL versions 10.6 or later
- PostgreSQL 9.6.11 or later
- PostgreSQL 9.5.15 or later

To enable IAM DB authentication for your Amazon RDS instance, run the following command:

```
aws rds modify-db-instance --db-instance-identifier my-rds-db --enable-iam-
database-authentication --apply-immediately
```

Because downtime is associated with this action, you can enable this feature during the next maintenance window. You can do so by changing the last parameter to `--no-apply-immediately`.

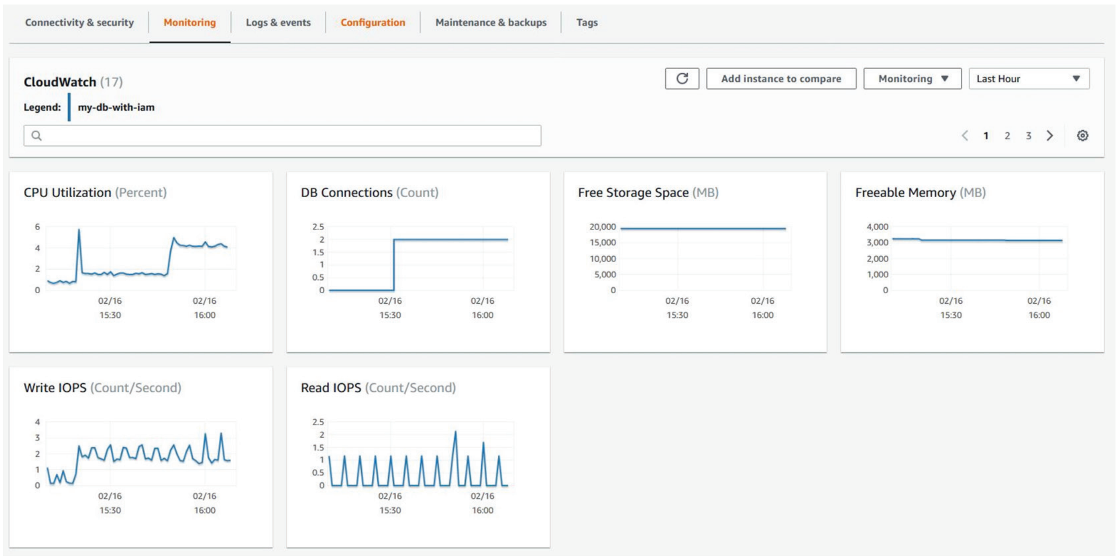
## Monitoring with Amazon CloudWatch

Use Amazon CloudWatch to monitor your database tier. You can create alarms to notify database administrators when there is a failure.

By default, CloudWatch provides some built-in metrics for Amazon RDS with a granularity of 5 minutes (600 seconds). If you want to gather metrics in a smaller window of granularity, such as 1 second, enable enhanced monitoring, which is similar to how you enable these features in Amazon EC2.

To view all the Amazon RDS metrics that are provided through CloudWatch, select the Monitoring tab from the Amazon RDS console (see Figure 4.6).

**FIGURE 4.6** Amazon RDS with CloudWatch metrics

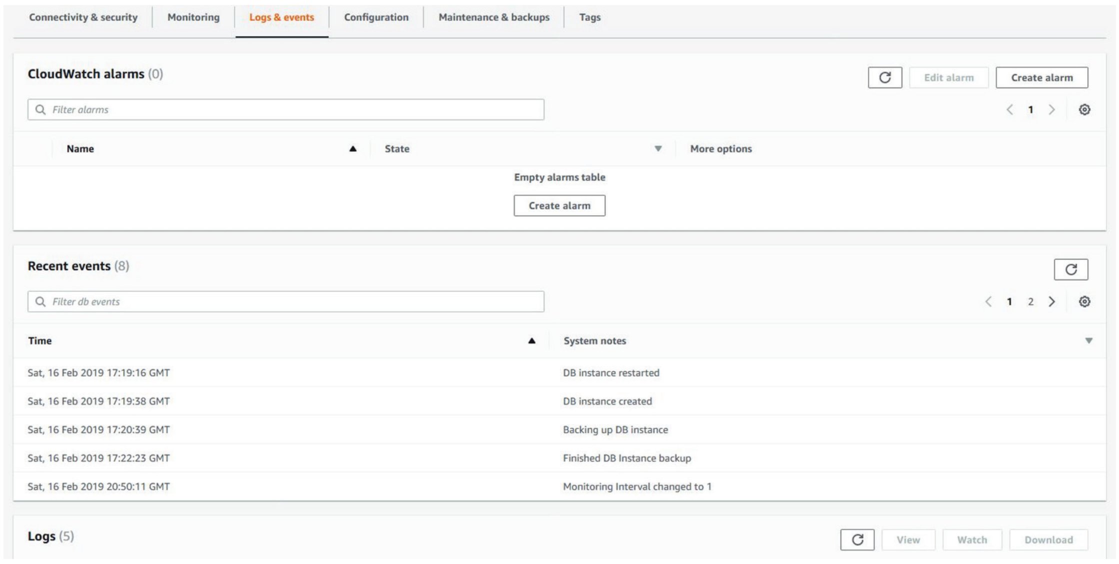


Amazon RDS integrates with CloudWatch to send it the following database logs:

- Audit log
- Error log
- General log
- Slow query log

From the Amazon RDS console, select the Logs & events tab to view and download the specified logs, as shown in Figure 4.7.

**FIGURE 4.7** Amazon RDS with CloudWatch Logs



CloudWatch alarms (0)

| Name | State | More options |
|---|---|---|

Empty alarms table

Create alarm

Recent events (8)

| Time | System notes |
|---|---|
| Sat, 16 Feb 2019 17:19:16 GMT | DB instance restarted |
| Sat, 16 Feb 2019 17:19:38 GMT | DB instance created |
| Sat, 16 Feb 2019 17:20:39 GMT | Backing up DB instance |
| Sat, 16 Feb 2019 17:22:23 GMT | Finished DB Instance backup |
| Sat, 16 Feb 2019 20:50:11 GMT | Monitoring Interval changed to 1 |

Logs (5)

For more information on CloudWatch and its capabilities across other AWS services, see Chapter 15, "Monitoring and Troubleshooting."

## Amazon Aurora

*Amazon Aurora* is a MySQL- and PostgreSQL-compatible relational database engine that combines the speed and availability of high-end commercial databases with the simplicity and cost-effectiveness of open source databases.

Aurora is part of the managed database service Amazon RDS.

### Amazon Aurora DB Clusters

Aurora is a drop-in replacement for MySQL and PostgreSQL relational databases. It is built for the cloud, and it combines the performance and availability of high-end commercial databases with the simplicity and cost-effectiveness of open source databases. You can use the code, tools, and applications that you use today with your existing MySQL and PostgreSQL databases with Aurora.

The integration of Aurora with Amazon RDS means that time-consuming administration tasks, such as hardware provisioning, database setup, patching, and backups, are automated.

Aurora features a distributed, fault-tolerant, self-healing storage system that automatically scales up to 64 TiB per database instance. (In comparison, other Amazon RDS options allow for a maximum of 32 TiB.) Aurora delivers high performance and availability with up to 15 low-latency read replicas, point-in-time recovery, continuous backup to Amazon Simple Storage Service (Amazon S3), and replication across three Availability Zones. When you create an Aurora instance, you create a DB cluster. A *DB cluster* consists of one or more DB instances and a cluster volume that manages the data for those instances. An Aurora *cluster volume* is a virtual database storage volume that spans multiple Availability Zones, and each Availability Zone has a copy of the DB cluster data.

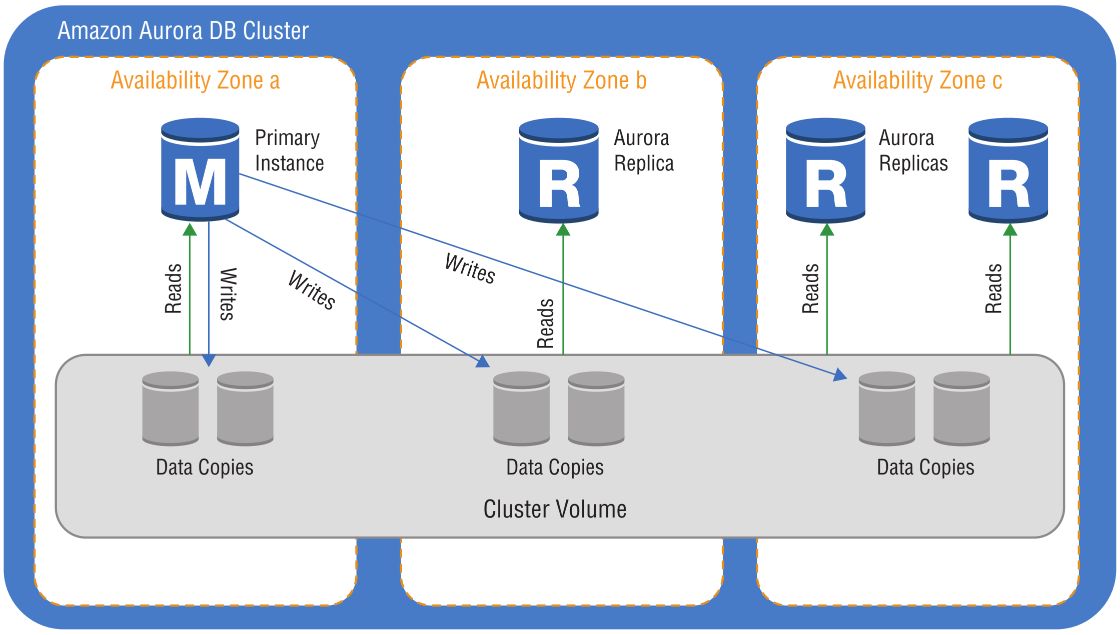
An Aurora DB cluster has two types of DB instances:

**Primary Instance** Supports read and write operations and performs all of the data modifications to the cluster volume. Each Aurora DB cluster has one primary instance.

**Amazon Aurora Replica** Supports read-only operations. Each Aurora DB cluster can have up to 15 *Amazon Aurora Replicas* in addition to the primary instance. Multiple Aurora Replicas distribute the read workload, and if you locate Aurora Replicas in separate Availability Zones, you can also increase database availability.

Figure 4.8 illustrates the relationship between the cluster volume, the primary instance, and Aurora Replicas in an Aurora DB cluster.

**FIGURE 4.8** Amazon Aurora DB cluster



As you can see from Figure 4.8, this architecture is vastly different from the other Amazon RDS databases. Aurora is engineered and architected for the cloud. The primary difference is that there is a separate storage layer, called the *cluster volume*, which is spread across multiple Availability Zones in a single AWS Region. This means that the durability of your data is increased.

Additionally, Aurora has one primary instance that writes across the cluster volume. This means that Aurora replicas can be spun up quickly, because they don't have to copy and store their own storage layer; they connect to it. Because the cluster volume is separated in this architecture, the cluster volume can grow automatically as your data increases. This is in contrast to how other Amazon RDS databases are built, whereby you must define the allocated storage in advance.

### Amazon Aurora Global Databases

With Aurora, you can also create a multiregional deployment for your database tier. In this configuration, the primary AWS Region is where your data is written (you may also do reads from the primary AWS Region). Any application performing writes must write to the primary AWS Region where the cluster is operating.

The secondary AWS Region is used for *reading* data only. Aurora replicates the data to the secondary AWS Region with typical latency of less than a second. Furthermore, you can use the secondary AWS Region for disaster recovery purposes. You can promote the secondary cluster and make it available as the primary typically in less than a minute. At the time of this writing, Aurora global databases are available in the following AWS Regions only:

- US East (N. Virginia)
- US East (Ohio)
- US West (Oregon)
- EU (Ireland)

Additionally, at the time of this writing, Aurora global databases are available only for MySQL 5.6.

### Amazon Aurora Serverless

Aurora Serverless is an on-demand, automatic scaling configuration for Aurora. (It is available only for MySQL at the time of this writing.) With Aurora Serverless, the database will *automatically* start up, shut down, and scale capacity up or down based on your application's needs. This means that, as a developer, you can run your database in the AWS Cloud and not worry about managing any database instances.

## Best Practices for Running Databases on AWS

The following are best practices for working with Amazon RDS:

**Follow Amazon RDS basic operational guidelines.** The Amazon RDS Service Level Agreement requires that you follow these guidelines:

- Monitor your memory, CPU, and storage usage. Amazon CloudWatch can notify you when usage patterns change or when you approach the capacity of your deployment so that you can maintain system performance and availability.
- Scale up your DB instance when you approach storage capacity limits. Have some buffer in storage and memory to accommodate unforeseen increases in demand from your applications.
- Enable automatic backups, and set the backup window to occur during the daily low in write IOPS.

- If your database workload requires more I/O than you have provisioned, recovery after a failover or database failure will be slow. To increase the I/O capacity of a DB instance, do any or all of the following:
  - Migrate to a DB instance class with high I/O capacity.
  - Convert from standard storage either to General Purpose or Provisioned IOPS storage, depending on how much of an increase you need. If you convert to Provisioned IOPS storage, make sure that you also use a DB instance class that is optimized for Provisioned IOPS.
  - If you are already using Provisioned IOPS storage, provision additional throughput capacity.
- If your client application is caching the Domain Name Service (DNS) data of your DB instances, set a time-to-live (TTL) value of less than 30 seconds. Because the underlying IP address of a DB instance can change after a failover, caching the DNS data for an extended time can lead to connection failures if your application tries to connect to an IP address that no longer is in service.
- Test failover for your DB instance to understand how long the process takes for your use case and to ensure that the application that accesses your DB instance can automatically connect to the new DB instance after failover.

**Allocate sufficient RAM to the DB instance.** An Amazon RDS performance best practice is to allocate enough RAM so that your working set resides almost completely in memory. Check the `ReadIOPS` metric by using CloudWatch while the DB instance is under load to view the working set. The value of `ReadIOPS` should be small and stable. Scale up the DB instance class until `ReadIOPS` no longer drops dramatically after a scaling operation or when `ReadIOPS` is reduced to a small amount.

**Implement Amazon RDS security.** Use IAM accounts to control access to Amazon RDS API actions, especially actions that create, modify, or delete Amazon RDS resources, such as DB instances, security groups, option groups, or parameter groups, and actions that perform common administrative actions, such as backing up and restoring DB instances, or configuring Provisioned IOPS storage.

- Assign an individual IAM account to each person who manages Amazon RDS resources. Do not use an AWS account user to manage Amazon RDS resources; create an IAM user for everyone, including yourself.
- Grant each user the minimum set of permissions required to perform his or her duties.
- Use IAM groups to manage permissions effectively for multiple users.
- Rotate your IAM credentials regularly.

Use the AWS Management Console, the AWS CLI, or the Amazon RDS API to change the password for your master user. If you use another tool, such as a SQL client, to change the master user password, it might result in permissions being revoked for the user unintentionally.

**Use enhanced monitoring to identify OS issues.** Amazon RDS provides metrics in real time for the OS on which your DB instance runs. You can view the metrics for your DB instance by using the console or consume the Enhanced Monitoring JSON output from CloudWatch Logs in a monitoring system of your choice. Enhanced Monitoring is available for the following database engines:

- MariaDB
- Microsoft SQL Server
- MySQL version 5.5 or later
- Oracle
- PostgreSQL

Enhanced Monitoring is available for all DB instance classes except for `db.m1.small`. Enhanced Monitoring is available in all regions except for AWS GovCloud (US).

**Use metrics to identify performance issues.** To identify performance issues caused by insufficient resources and other common bottlenecks, you can monitor the metrics available for your Amazon RDS DB instance.

Monitor performance metrics regularly to see the average, maximum, and minimum values for a variety of time ranges. If you do so, you can identify when performance is degraded. You can also set CloudWatch alarms for particular metric thresholds.

To troubleshoot performance issues, it's important to understand the baseline performance of the system. When you set up a new DB instance and get it running with a typical workload, you should capture the average, maximum, and minimum values of all the performance metrics at a number of different intervals (for example, 1 hour, 24 hours, 1 week, or 2 weeks) to get an idea of what is normal. It helps to get comparisons for both peak and off-peak hours of operation. You can then use this information to identify when performance is dropping below standard levels.

**Tune queries.** One of the best ways to improve DB instance performance is to tune your most commonly used and most resource-intensive queries to make them less expensive to run.

A common aspect of query tuning is creating effective indexes. You can use the Database Engine Tuning Advisor to get potential index improvements for your DB instance.

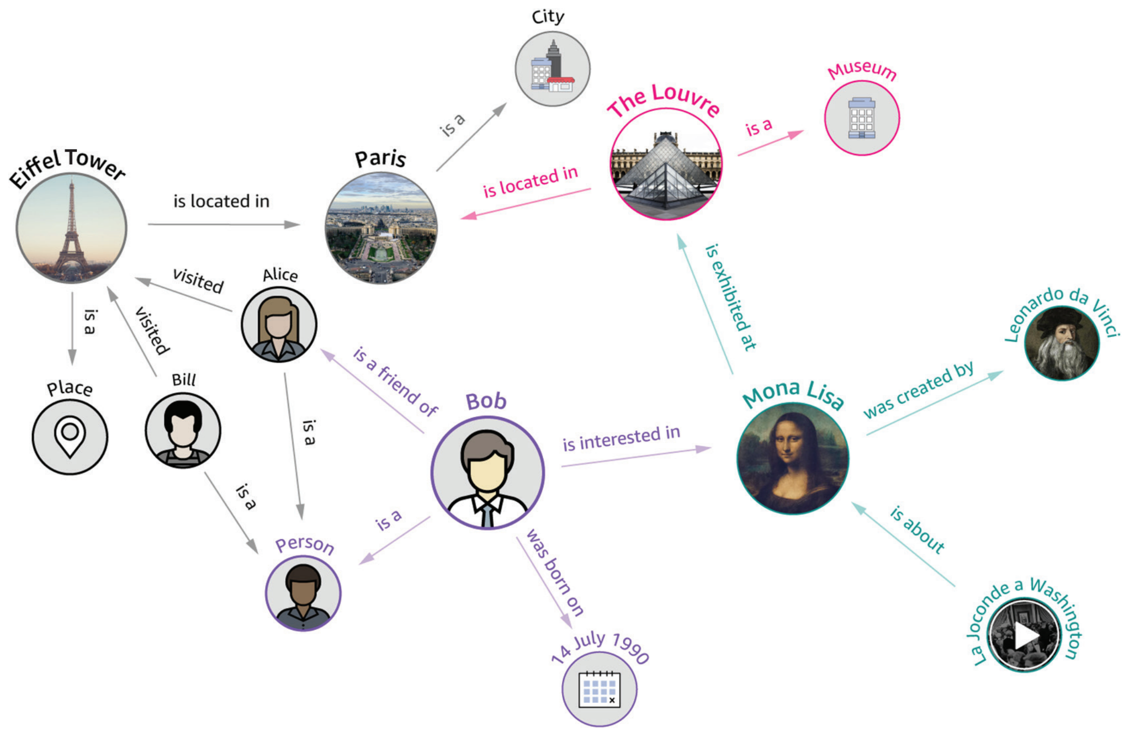
**Use DB parameter groups.** AWS recommends that you apply changes to the DB parameter group on a test DB instance before you apply parameter group changes to your production DB instances. Improperly setting DB engine parameters in a DB parameter group can have unintended adverse effects, including degraded performance and system instability. Always exercise caution when modifying DB engine parameters, and back up your DB instance before modifying a DB parameter group.

**Use read replicas.** Use read replicas to relieve pressure on your master node with additional read capacity. You can bring your data closer to applications in different regions and promote a read replica to a master for faster recovery in the event of a disaster.

TIP

You can use the AWS Database Migration Service (AWS DMS) to migrate or replicate your existing databases easily to Amazon RDS.

**FIGURE 4.22** Example of a graph database architecture running on Amazon Neptune



Neptune supports the popular graph models Property Graph and W3C's RDS and their respective query languages Apache TinkerPop Gremlin and SPARQL. With these models, you can easily build queries that efficiently navigate highly connected datasets. Neptune graph databases include the following use cases:

- Recommendation engines
- Fraud detection
- Knowledge graphs
- Drug discovery
- Network security

## Cloud Database Migration

Data is the cornerstone of successful cloud application deployments. Your evaluation and planning process may highlight the physical limitations inherent to migrating data from on-premises locations into the cloud. Amazon offers a suite of tools to help you move data via networks, roads, and technology partners.

This chapter focuses on the AWS Database Migration Service (AWS DMS) and the AWS Schema Conversion Tool (AWS SCT). Customers also use other AWS services and features that are discussed in Chapter 3, "Hello, Storage," for cloud data migration, including the following:

- AWS Direct Connect (DX)
- AWS Snowball
- AWS Snowball Edge
- AWS Snowmobile
- AWS Import/Export Disk
- AWS Storage Gateway
- Amazon Kinesis Data Firehose
- Amazon S3 Transfer Acceleration
- Virtual private network (VPN) connections

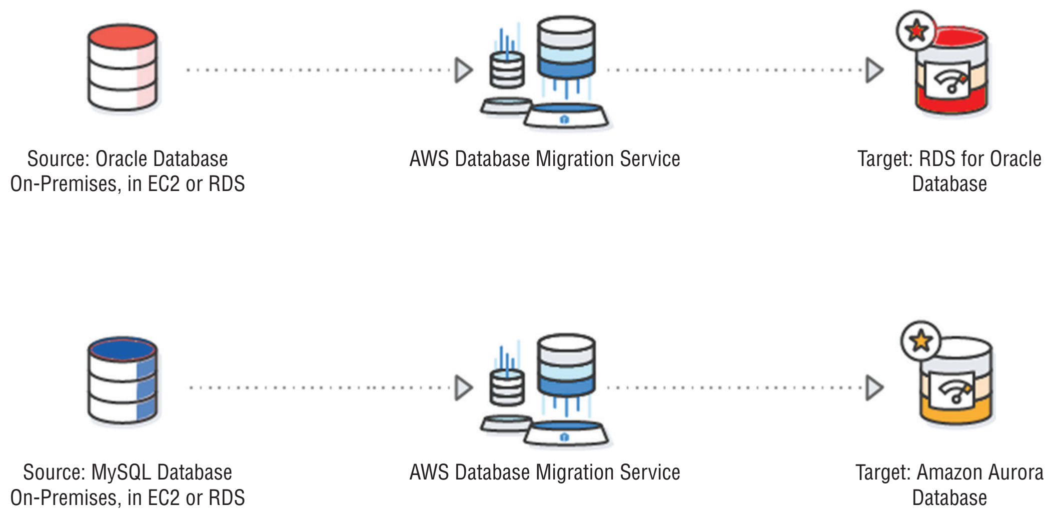
## AWS Database Migration Service

*AWS Database Migration Service* (AWS DMS) helps you migrate databases to AWS quickly and securely. The source database remains fully operational during the migration, minimizing downtime to applications that rely on the database. AWS DMS can migrate your data to and from the most widely used commercial and open-source databases.

The service supports *homogenous database migrations*, such as Oracle to Oracle, in addition to *heterogeneous migrations* between different database platforms, such as Oracle to Amazon Aurora or Microsoft SQL Server to MySQL. You can also stream data to Amazon Redshift, Amazon DynamoDB, and Amazon S3 from any of the supported sources, such as Amazon Aurora, PostgreSQL, MySQL, MariaDB, Oracle Database, SAP ASE, SQL Server, IBM DB2 LUW, and MongoDB, enabling consolidation and easy analysis of data in a petabyte-scale data warehouse. You can also use AWS DMS for continuous data replication with high availability.

Figure 4.23 shows an example of both heterogeneous and homogenous database migrations.

**FIGURE 4.23** Homogenous database migrations using AWS DMS


Source: Oracle Database On-Premises, in EC2 or RDS
AWS Database Migration Service
Target: RDS for Oracle Database
Source: MySQL Database On-Premises, in EC2 or RDS
AWS Database Migration Service
Target: Amazon Aurora Database

To perform a database migration, AWS DMS connects to the source data store, reads the source data, and formats the data for consumption by the target data store. It then loads the data into the target data store. Most of this processing happens in memory, though large transactions might require some buffering to disk. Cached transactions and log files are also written to disk.

At a high level, when you're using AWS DMS, complete the following tasks:

- Create a replication server.
- Create source and target endpoints that have connection information about your data stores.
- Create one or more tasks to migrate data between the source and target data stores.

A task can consist of three major phases:

- The full load of existing data
- The application of cached changes
- Ongoing replication

## AWS Schema Conversion Tool

For heterogeneous database migrations, AWS DMS uses the *AWS Schema Conversion Tool* (AWS SCT). AWS SCT makes heterogeneous database migrations predictable by automatically converting the source database schema and a majority of the database code objects, including views, stored procedures, and functions, to a format compatible with the target database. Any objects that cannot be automatically converted are clearly marked so that they can be manually converted to complete the migration.

AWS SCT can also scan your application source code for embedded SQL statements and convert them as part of a database schema conversion project. During this process, AWS SCT performs cloud-native code optimization by converting legacy Oracle and SQL Server functions to their equivalent AWS service, thus helping you modernize the applications at the same time as database migration.

Figure 4.24 is snapshot of the Action Items tab in the AWS SCT report, which shows the items that the tool could not convert automatically. These are the items that you would need to evaluate and adjust manually as needed. The report helps you to determine how much work you would need to do to complete a conversion.

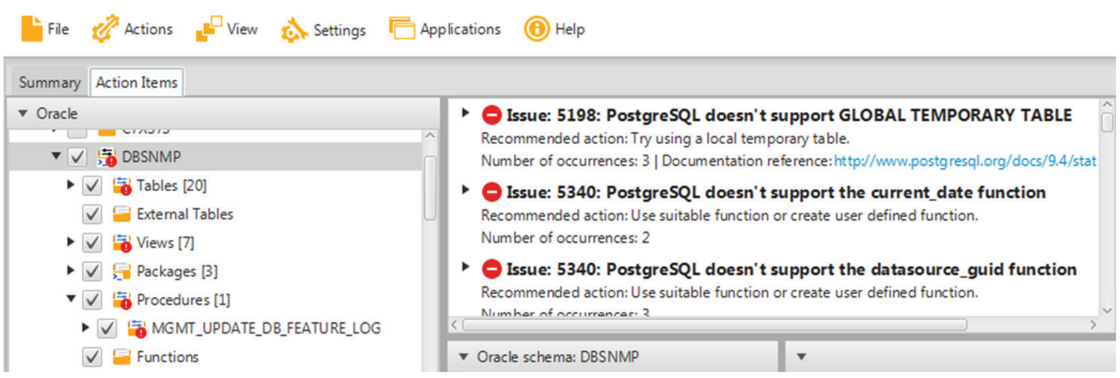


**FIGURE 4.24** AWS SCT action items

**Issue: 5198: PostgreSQL doesn't support GLOBAL TEMPORARY TABLE**
Recommended action: Try using a local temporary table.
Number of occurrences: 3 | Documentation reference: http://www.postgresql.org/docs/9.4/stat

**Issue: 5340: PostgreSQL doesn't support the current_date function**
Recommended action: Use suitable function or create user defined function.
Number of occurrences: 2

**Issue: 5340: PostgreSQL doesn't support the datasource_guid function**
Recommended action: Use suitable function or create user defined function.
Number of occurrences: 3

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

Your source database can be on-premises, in Amazon RDS, or in Amazon EC2, and the target database can be in either Amazon RDS or Amazon EC2. AWS SCT supports a number of different heterogeneous conversions. Table 4.9 lists the source and target databases that are supported at the time of this writing.

**TABLE 4.9** Source and Target Databases Supported by AWS SCT

| Source Database | Target Database on Amazon RDS |
|---|---|
| Oracle Database | Amazon Aurora, MySQL, PostgreSQL, Oracle |
| Oracle Data Warehouse | Amazon Redshift |
| Azure SQL | Amazon Aurora, MySQL, PostgreSQL |
| Microsoft SQL Server | Amazon Aurora, Amazon Redshift, MySQL, PostgreSQL |
| Teradata | Amazon Redshift |
| IBM Netezza | Amazon Redshift |
| IBM DB2 LUW | Amazon Aurora, MySQL, PostgreSQL |
| HPE Vertica | Amazon Redshift |
| MySQL and MariaDB | PostgreSQL |
| PostgreSQL | Amazon Aurora, MySQL |
| Amazon Aurora | PostgreSQL |
| Greenplum | Amazon Redshift |
| Apache Cassandra | Amazon DynamoDB |

## Running Your Own Database on Amazon Elastic Compute Cloud

This chapter focused heavily on the AWS services that are available from a managed database perspective. However, it is important to know that you can also run your own unmanaged database on Amazon EC2, not only for the exam but for managing projects in the real world. For example, if you want to run MongoDB on Amazon EC2, this is perfectly within the realm of possibility. However, by doing so, you lose the many benefits of using a managed database service.

## Compliance and Security

AWS includes various methods to provide security for your databases and meet the strictest of compliance standards. You can use the following:

- Network isolation through virtual private cloud (VPC)
- Security groups
- AWS resource-level permission controls that are IAM-based.
- Encryption at rest by using AWS KMS or Oracle/Microsoft Transparent Data Encryption (TDE)
- Secure Sockets Layer (SSL) protection for data in transit
- Assurance programs for finance, healthcare, government, and more

### AWS Identity and Access Management

You can use *Identity and Access Management* (IAM) to perform governed access to control who can perform actions with Amazon Aurora MySQL and Amazon RDS for MySQL. Here's an example:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowCreateDBInstanceOnly",
            "Effect": "Allow",
            "Action": [
                "rds:CreateDBInstance"
            ],
            "Resource": [
                "arn:aws:rds:*:123456789012:db:test*",
                "arn:aws:rds: * : 123456789012:og:default*",
                "arn:aws:rds:*:123456789012:pg:default*",
                "arn:aws:rds: * : 1234 56789012 :subgrp: default"
            ],
                "Condition": {
                    "StringEquals": {
                    "rds:DatabaseEngine": "mysql",
                    "rds:DatabaseClass": "db.t2.micro"
                    }
                }
            }
        }
    }
```

## Summary

In this chapter, you learned the basic concepts of different types of databases, including relational, nonrelational, data warehouse, in-memory, and graph databases. From there, you learned about the various managed database services available on AWS. These included Amazon RDS, Amazon DynamoDB, Amazon Redshift, Amazon ElastiCache, and Amazon Neptune. You also saw how you can run your own database on Amazon EC2. Finally, you looked at how to perform homogenous database migrations using the AWS Database Migration Service (AWS DMS). For heterogeneous database migrations, you learned that AWS DMS can use the AWS Schema Conversion Tool (AWS SCT).

## Exam Essentials

**Know what a relational database is.** A relational database consists of one or more tables. Communication to and from relational databases usually involves simple SQL queries, such as "Add a new record" or "What is the cost of product x?" These simple queries are often referred to as online transaction processing (OLTP).

**Know what a nonrelational database is.** Nonrelational databases do not have a hard-defined data schema. They can use a variety of models for data management, such as in-memory key-value stores, graph data models, and document stores. These databases are optimized for applications that have a large data volume, require low latency, and have flexible data models. In nonrelational databases, there is no concept of foreign keys.

**Understand the database options available on AWS.** You can run all types of databases on AWS. You should understand that there are managed and unmanaged options available, in addition to relational, nonrelational, caching, graph, and data warehouses.

**Understand which databases Amazon RDS supports.** Amazon RDS currently supports six relational database engines:

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

**Understand the operational benefits of using Amazon RDS.** Amazon RDS is an AWS managed service. AWS is responsible for patching, antivirus, and the management of the underlying guest OS for Amazon RDS. Amazon RDS greatly simplifies the process of setting a secondary slave with replication for failover and setting up read replicas to offload queries.

**Remember that you cannot access the underlying OS for Amazon RDS DB instances.** You cannot use Remote Desktop Protocol (RDP) or SSH to connect to the underlying OS. If you need to access the OS, install custom software or agents. If you want to use a database engine that Amazon RDS does not support, consider running your database on an Amazon EC2 instance instead.

**Understand that Amazon RDS handles Multi-AZ failover for you.** If your primary Amazon RDS instance becomes unavailable, AWS fails over to your secondary instance in another Availability Zone automatically. This failover is done by pointing your existing database endpoint to a new IP address. You do not have to change the connection string manually; AWS handles the DNS changes automatically.

**Remember that Amazon RDS read replicas are used for scaling out and increased performance.** This replication feature makes it easy to scale out your read-intensive databases. Read replicas are currently supported in Amazon RDS for MySQL, PostgreSQL, and Amazon Aurora. You can create one or more replicas of a database within a single AWS Region or across multiple AWS Regions. Amazon RDS uses native replication to propagate changes made to a source DB instance to any associated read replicas. Amazon RDS also supports cross-region read replicas to replicate changes asynchronously to another geography or AWS Region.

**Know how to calculate throughput for Amazon DynamoDB.** Remember that one read capacity unit (RCU) represents one strongly consistent read per second or two eventually consistent reads per second for an item up to 4 KB in size. For writing data, know that one write capacity unit (WCU) represents one write per second for an item up to 1 KB in size. Be comfortable performing calculations to determine the appropriate setting for the RCU and WCU for a table.

**Know that DynamoDB spreads RCUs and WCUs across partitions evenly.** Recall that when you allocate your total RCUs and WCUs to a table, DynamoDB spreads these across your partitions evenly. For example, if you have 1,000 RCUs and you have 10 partitions, then you have 100 RCUs allocated to each partition.

**Know the differences between a local secondary index and a global secondary index.** Remember that you can create local secondary indexes only when you initially create the table; additionally, know that local secondary indexes must share the same partition key as the parent or source table. Conversely, you can create global secondary indexes at any time, with different partitions keys or sort keys.

**Know the difference between eventually consistent and strongly consistent reads.** Know that with eventually consistent reads, your application may retrieve data that is stale; but with strongly consistent reads, the data is always up-to-date.

**Understand the purpose of caching data and which related services are available.** Know why caching is important for your database tier and how it helps to improve your application performance. Additionally, understand the differences between the caching methods (lazy loading and write-through) and the corresponding AWS services (Amazon DynamoDB Accelerator (DAX), ElastiCache for Redis, and ElastiCache for Memcached).

## Resources to Review

What Is a Relational Database?

`https://aws.amazon.com/relational-database/`

AWS Databases:

`https://aws.amazon.com/products/databases/`

AWS Database Blog:

`https://aws.amazon.com/blogs/database/`

A One Size Fits All Database Doesn't Fit Anyone:

`https://www.allthingsdistributed.com/2018/06/purpose-built-databases-in-aws.html`

Amazon Relational Database Service (Amazon RDS) User Guide:

`https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html`

Amazon RDS FAQs:

`https://aws.amazon.com/rds/faqs/`

Development and Test on Amazon Web Services:

`https://d1.awsstatic.com/whitepapers/aws-development-test-environments.pdf`

Amazon Redshift Snapshots:

`https://docs.aws.amazon.com/redshift/latest/mgmt/working-with-snapshots.html`

Amazon Aurora:

`https://aws.amazon.com/rds/aurora/`

Amazon Aurora Overview:

`https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/CHAP_AuroraOverview.html`

Amazon RDS Resources:

`https://aws.amazon.com/rds/developer-resources/`

Best Practices for Amazon RDS:

`https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_BestPractices.html`

What Is a Document Database?

`https://aws.amazon.com/nosql/document/`

What Is a Columnar Database?

`https://aws.amazon.com/nosql/columnar/`

What Is NoSQL?

`https://aws.amazon.com/nosql/`

Amazon DynamoDB:

`https://aws.amazon.com/dynamodb/`

What Is Amazon DynamoDB?

`https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html`

Amazon DynamoDB Core Components:

`https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.CoreComponents.html`

Amazon DynamoDB Developer Guide:

`http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/`

GSI Attribute Projections:

`https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html#GSI.Projections`

Data Warehouse Concepts:

`https://aws.amazon.com/data-warehouse/`

Getting Started with Amazon Redshift:

`http://docs.aws.amazon.com/redshift/latest/gsg/`

Amazon Redshift Database Developer Guide:

`http://docs.aws.amazon.com/redshift/latest/dg/`

Using Amazon Redshift Spectrum to Query External Data:

`https://docs.aws.amazon.com/redshift/latest/dg/c-using-spectrum.html`

What Is a Key-Value Database?

`https://aws.amazon.com/nosql/key-value/`

Amazon ElasticCache for Redis User Guide:

`https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/WhatIs.html`

Amazon ElastiCache for Memcached User Guide:

`https://docs.aws.amazon.com/AmazonElastiCache/latest/mem-ug/WhatIs.html`

In-Memory Processing in the Cloud with Amazon ElastiCache (Whitepaper):

`https://d1.awsstatic.com/elasticache/elasticache_in_memory_processing_intel.pdf`

Performance at Scale with Amazon ElastiCache (Whitepaper):

`https://d1.awsstatic.com/whitepapers/performance-at-scale-with-amazon-elasticache.pdf`

Amazon DynamoDB Accelerator (DAX):

`https://aws.amazon.com/dynamodb/dax/`

Amazon DynamoDB Accelerator (DAX): A Read-Through/Write-Through Cache for DynamoDB:

`https://aws.amazon.com/blogs/database/amazon-dynamodb-accelerator-dax-a-read-throughwrite-through-cache-for-dynamodb/`

What Is a Graph Database?

`https://aws.amazon.com/nosql/graph/`

Amazon Neptune User Guide:

`https://docs.aws.amazon.com/neptune/latest/userguide/intro.html`

AWS Database Migration Service Documentation:

`https://docs.aws.amazon.com/dms/index.html`

Cloud Data Migration:

`https://aws.amazon.com/cloud-data-migration/`

AWS Database Migration Service User Guide:

`http://docs.aws.amazon.com/dms/latest/userguide/`

AWS Database Migration Service Step-by-Step Walkthroughs:

`http://docs.aws.amazon.com/dms/latest/sbs/DMS-SBS-Welcome.html`

AWS Schema Conversion Tool User Guide:

`https://docs.aws.amazon.com/SchemaConversionTool/latest/userguide/CHAP_Welcome.html`

## Exercises

In the following exercises, you will launch two types of databases: the first database is an SQL database on Amazon RDS, and the second is Amazon DynamoDB (NoSQL). For these sets of exercises, you will use the Python 3 SDK. You can download the Python 3 SDK at `https://aws.amazon.com/sdk-for-python/`.

**EXERCISE 4.1**

### Create a Security Group for the Database Tier on Amazon RDS

Before you can create your first Amazon RDS database, you must create a security group so that you can allow traffic from your development server to communicate with the database tier. To do this, you must use an Amazon EC2 client to create the security group. Security groups are a component of the Amazon EC2 service, even though you can use them as part of Amazon RDS to secure your database tier.

To create the security group, run the following script:

```
# Excercise 4.1
import boto3
import json
import datetime

# Let's create some variables we'll use throughout these Excercises in Chapter 4
# NOTE: Here we are using a CIDR range for incoming traffic. We have set it to
   0.0.0.0/0 which means
# ANYONE on the internet can access your database if they have the username and
   the password
# If possible, specify you're own CIDR range. You can figure out your CIDR range
   by visiting the following link
# https://www.google.com/search?q=what+is+my+ip
# In the variable don't forget to add /32!
# If you aren't sure, leave it open to the world

# Variables
sg_name = 'rds-sg-dev-demo'
sg_description = 'RDS Security Group for AWS Dev Study Guide'
my_ip_cidr = '0.0.0.0/0'


# Create the EC2 Client to create the Security Group for your Database
ec2_client = boto3.client('ec2')

# First we need to create a security group
```

```
response = ec2_client.create_security_group(
    Description=sg_description,
    GroupName=sg_name)
print(json.dumps(response, indent=2, sort_keys=True))
# Now add a rule for the security group
response = ec2_client.authorize_security_group_ingress(
    CidrIp=my_ip_cidr,
    FromPort=3306,
    GroupName=sg_name,
    ToPort=3306,
    IpProtocol='tcp'
    )
print("Security Group should be created! Verify this in the AWS Console.")
```

After running the Python code, verify that the security group was created successfully from the AWS Management Console. You can find this confirmation under the VPC or Amazon EC2 service.

**EXERCISE 4.2**

### Spin Up the MariaDB Database Instance

Use the Python SDK to spin up your MariaDB database hosted on Amazon RDS.

To spin up the MariaDB database, run the following script and update the `Variables` section to meet your needs:

```
# Excercise 4.2
import boto3
import json
import datetime


# Just a quick helper function for date time conversions, in case you want to
   print the raw JSON
def date_time_converter(o):
    if isinstance(o, datetime.datetime):
        return o.__str__()

# Variables
sg_name = 'rds-sg-dev-demo'
rds_identifier = 'my-rds-db'
db_name = 'mytestdb'
```

*(continued)*

**EXERCISE 4.2 *(continued)***

```
user_name = 'masteruser'
user_password = 'mymasterpassw0rd1!'
admin_email = 'myemail@myemail.com'
sg_id_number = ''
rds_endpoint = ''

# We need to get the Security Group ID Number to use in the creation of the RDS
   Instance
ec2_client = boto3.client('ec2')
response = ec2_client.describe_security_groups(
    GroupNames=[
        sg_name
    ])

sg_id_number = json.dumps(response['SecurityGroups'][0]['GroupId'])
sg_id_number = sg_id_number.replace('"','')

#  Create the client for Amazon RDS
rds_client = boto3.client('rds')

# This will create our MariaDB Database
# NOTE: Here we are hardcoding passwords for simplicity and testing purposes
   only! In production
# you should never hardcode passwords in configuration files/code!
# NOTE: This will create an MariaDB Database. Be sure to remove it when you are
   done.
response = rds_client.create_db_instance(
    DBInstanceIdentifier=rds_identifier,
    DBName=db_name,
    DBInstanceClass='db.t2.micro',
    Engine='mariadb',
    MasterUsername='masteruser',
    MasterUserPassword='mymasterpassw0rd1!',
    VpcSecurityGroupIds=[
        sg_id_number
    ],
    AllocatedStorage=20,
    Tags=[
        {
            'Key': 'POC-Email',
            'Value': admin_email
        },
        {
            'Key': 'Purpose',
            'Value': 'AWS Developer Study Guide Demo'
        }
    ]
)

# We need to wait until the DB Cluster is up!
print('Creating the RDS instance. This may take several minutes...')
waiter = rds_client.get_waiter('db_instance_available')
waiter.wait(DBInstanceIdentifier=rds_identifier)

print('Okay! The Amazon RDS Database is up!')
```

After the script has executed, the following message is displayed:

```
Creating the RDS instance. This may take several minutes.
```

After the Amazon RDS database instance has been created successfully, the following confirmation is displayed:

```
Okay! The Amazon RDS Database is up!
```

You can also view these messages from the Amazon RDS console.

## EXERCISE 4.3

### Obtain the Endpoint Value for the Amazon RDS Instance

Before you can start using the Amazon RDS instance, you must first specify your endpoint. In this exercise, you will use the Python SDK to obtain the value.

To obtain the Amazon RDS endpoint, run the following script:

```
# Exercise 4.3
import boto3
import json
import datetime


# Just a quick helper function for date time conversions, in case you want to
   print the raw JSON
```

(continued)

**EXERCISE 4.3 *(continued)***

```
def date_time_converter(o):
    if isinstance(o, datetime.datetime):
        return o.__str__()

# Variables
rds_identifier = 'my-rds-db'

#  Create the client for Amazon RDS
rds_client = boto3.client('rds')

print("Fetching the RDS endpoint...")
response = rds_client.describe_db_instances(
    DBInstanceIdentifier=rds_identifier
)

rds_endpoint = json.dumps(response['DBInstances'][0]['Endpoint']['Address'])
rds_endpoint = rds_endpoint.replace('"','')
print('RDS Endpoint: ' + rds_endpoint)
```

After running the Python code, the following status is displayed:

```
Fetching the RDS endpoint.. RDS Endpoint:<endpoint_name>
```

If the endpoint is not returned, from the AWS Management Console, under the RDS service, verify that your Amazon RDS database instance was created.

**EXERCISE 4.4**

### Create a SQL Table and Add Records to It

You now have all the necessary information to create your first SQL table by using Amazon RDS. In this exercise, you will create a SQL table and add a couple of records. Remember to update the variables for your specific environment.

To update the variables, run the following script:

```
# Exercise 4.4
import boto3
import json
import datetime
import pymysql as mariadb

# Variables
rds_identifier = 'my-rds-db'
```

```
db_name = 'mytestdb'
user_name = 'masteruser'
user_password = 'mymasterpassw0rd1!'
rds_endpoint = 'my-rds-db.****.us-east-1.rds.amazonaws.com'

# Step 1 - Connect to the database to create the table
db_connection = mariadb.connect(host=rds_endpoint, user=user_name,
   password=user_password, database=db_name)
cursor = db_connection.cursor()
try:
    cursor.execute("CREATE TABLE Users (user_id INT NOT NULL AUTO_INCREMENT,
    user_fname VARCHAR(100) NOT NULL, user_lname VARCHAR(150) NOT NULL, user_
    email VARCHAR(175) NOT NULL, PRIMARY KEY (`user_id`))")
    print('Table Created!')
except mariadb.Error as e:
    print('Error: {}'.format(e))
finally:
    db_connection.close()

# Step 2 - Connect to the database to add users to the table
db_connection = mariadb.connect(host=rds_endpoint, user=user_name,
password=user_password, database=db_name)
cursor = db_connection.cursor()
try:
    sql = "INSERT INTO `Users` (`user_fname`, `user_lname`, `user_email`) VALUES
    (%s, %s, %s)"
    cursor.execute(sql, ('CJ', 'Smith', 'casey.smith@somewhere.com'))
    cursor.execute(sql, ('Casey', 'Smith', 'sam.smith@somewhere.com'))
    cursor.execute(sql, ('No', 'One', 'no.one@somewhere.com'))
# No data is saved unless we commit the transaction!
    db_connection.commit()
    print('Inserted Data to Database!')
except mariadb.Error as e:
    print('Error: {}'.format(e))
    print('Sorry, something has gone wrong!')
finally:
    db_connection.close()
```

After running the Python code, the following confirmation is displayed:

```
Table Created! Inserted Data to the Database!
```

Your Amazon RDS database now has some data stored in it.

*(continued)*

**EXERCISE 4.4 *(continued)***

NOTE

In this exercise, you are hardcoding a password into your application code for demonstration purposes only. In a production environment, refrain from hard-coding application passwords. Instead, use services such as AWS Secrets Manager to keep your secrets secure.

**EXERCISE 4.5**

## Query the Items in the SQL Table

After adding data to your SQL database, in this exercise you will be able to read or query the items in the *Users* table.

To read the items in the SQL table, run the following script:

```
# Exercise 4.5
import boto3
import json
import datetime
import pymysql as mariadb

# Variables
rds_identifier = 'my-rds-db'
db_name = 'mytestdb'
user_name = 'masteruser'
user_password = 'mymasterpassw0rd1!'
rds_endpoint = 'my-rds-db.*****.us-east-1.rds.amazonaws.com'

db_connection = mariadb.connect(host=rds_endpoint, user=user_name,
password=user_password, database=db_name)
cursor = db_connection.cursor()
try:
    sql = "SELECT * FROM `Users`"
    cursor.execute(sql)
    query_result = cursor.fetchall()
    print('Querying the Users Table...')
    print(query_result)
except mariadb.Error as e:
    print('Error: {}'.format(e))
    print('Sorry, something has gone wrong!')
```

```
finally:
    db_connection.close()
```

After running the Python code, you will see the three records that you inserted in the previous exercise.

**EXERCISE 4.6**

### Remove Amazon RDS Database and Security Group

You've created an Amazon RDS DB instance and added data to it. In this exercise, you will remove a few resources from your account. Remove the Amazon RDS instance first.

To remove the Amazon RDS instance and the security group, run the following script:

```
# Exercise 4.6
import boto3
import json
import datetime

# Variables
rds_identifier = 'my-rds-db'
sg_name = 'rds-sg-dev-demo'
sg_id_number = ''

# Create the client for Amazon RDS
rds_client = boto3.client('rds')

# Delete the RDS Instance
response = rds_client.delete_db_instance(
    DBInstanceIdentifier=rds_identifier,
    SkipFinalSnapshot=True)

print('RDS Instance is being terminated...This may take several minutes.')

waiter = rds_client.get_waiter('db_instance_deleted')
waiter.wait(DBInstanceIdentifier=rds_identifier)

# We must wait to remove the security groups until the RDS database has been
   deleted, this is a dependency.
print('The Amazon RDS database has been deleted. Removing Security Groups')

# Create the client for Amazon EC2 SG
```

*(continued)*

**EXERCISE 4.6 *(continued)***

```
ec2_client = boto3.client('ec2')

# Get the Security Group ID Number
response = ec2_client.describe_security_groups(
    GroupNames=[
        sg_name
    ])
sg_id_number = json.dumps(response['SecurityGroups'][0]['GroupId'])
sg_id_number = sg_id_number.replace('"','')

# Delete the Security Group!
response = ec2_client.delete_security_group(
    GroupId=sg_id_number
    )

print('Cleanup is complete!')
```

After running the Python code, the following message is displayed:

```
Cleanup is complete!
```

The Amazon RDS database and the security group are removed. You can verify this from the AWS Management Console.

**EXERCISE 4.7**

### Create an Amazon DynamoDB Table

Amazon DynamoDB is a managed NoSQL database. One major difference between DynamoDB and Amazon RDS is that DynamoDB doesn't require a server that is running in your VPC, and you don't need to specify an instance type. Instead, create a table.

To create the table, run the following script:

```
# Exercise 4.7
import boto3
import json
import datetime
```

```
dynamodb_resource = boto3.resource('dynamodb')

table = dynamodb_resource.create_table(
    TableName='Users',
    KeySchema=[
        {
            'AttributeName': 'user_id',
            'KeyType': 'HASH'
        },
        {
            'AttributeName': 'user_email',
            'KeyType': 'RANGE'
        }
    ],
    AttributeDefinitions=[
        {
            'AttributeName': 'user_id',
            'AttributeType': 'S'
        },
        {
            'AttributeName': 'user_email',
            'AttributeType': 'S'
        }
    ],
    ProvisionedThroughput={
        'ReadCapacityUnits': 5,
        'WriteCapacityUnits': 5
    }
)

print("The DynamoDB Table is being created, this may take a few minutes...")
table.meta.client.get_waiter('table_exists').wait(TableName='Users')
print("Table is ready!")
```

After running the Python code, the following message is displayed:

```
Table is ready!
```

From the AWS Management Console, under DynamoDB, verify that the table was created.

**EXERCISE 4.8**

### Add Users to the Amazon DynamoDB Table

With DynamoDB, there are fewer components to set up than there are for Amazon RDS. In this exercise, you'll add users to your table. Experiment with updating and changing some of the code to add multiple items to the database.

To add users to the DynamoDB table, run the following script:

```
# Exercise 4.8
import boto3
import json
import datetime
# In this example we are not using uuid; however, you could use this to
autogenerate your user IDs.
# i.e. str(uuid.uuid4())
import uuid

# Create a DynamoDB Resource
dynamodb_resource = boto3.resource('dynamodb')
table = dynamodb_resource.Table('Users')

# Write a record to DynamoDB
response = table.put_item(
    Item={
        'user_id': '1234-5678',
        'user_email': 'someone@somewhere.com',
        'user_fname': 'Sam',
        'user_lname': 'Samuels'
    }
)

# Just printing the raw JSON response, you should see a 200 status code
print(json.dumps(response, indent=2, sort_keys=True))
```

After running the Python code, you receive a 200 HTTP Status Code from AWS. This means that the user record has been added.

From the AWS Management Console, under DynamoDB, review the table to verify that the user record was added.

is cached and routed through the closest edge location serving you. After you deploy your application on Elastic Beanstalk, use the Amazon CloudFront content delivery network (CDN) to cache static content from your application. To identify the source of your content in Amazon CloudFront, you can use URL path patterns to cache your content and then retrieve it from the cache. This approach serves your content more rapidly and offloads requests directly sourced from your application.

## AWS Config

With *AWS Config*, you can visualize configuration history and how configurations evolve over time. Tracking changes helps you to fulfill compliance obligations and meet auditing requirements. You can integrate AWS Config directly with your application and its versions or your Elastic Beanstalk environment. *You can customize AWS Config to record changes per resource, per region, or globally*. In the AWS Config console, you can select Elastic Beanstalk resource types to record specific applications and environment resources. You can view the recorded information in the AWS Config dashboard under Resource Inventory.

## Amazon RDS

Various options are available for creating databases for your environment, such as Amazon Relational Database Service (Amazon RDS) for SQL databases and Amazon DynamoDB for NoSQL databases. Elastic Beanstalk can create a database and store and retrieve data for any of your environments. Each service has its own features to handle scaling, capacity, performance, and availability.

To store, read, or write to your data records, you can set up an Amazon RDS database instance or an Amazon DynamoDB table by using the same configuration files for your other service option settings. You must create connections to the database, which require you to set up password management in Elastic Beanstalk. Your configurations are saved in the `ebextensions` directory. You can also create direct connections, within your application code or application configuration files, to both internal and external databases. When using Amazon RDS, avoid accidentally deleting and re-creating databases without a properly installed backup. To reduce the risk of losing data, take a manual snapshot of the master Amazon RDS database immediately before deleting.

NOTE If you create periodic tasks with a worker environment, Elastic Beanstalk automatically creates an Amazon DynamoDB table to perform leader election and stores task information.

## Amazon ElastiCache

For caching capabilities, you can integrate Amazon ElastiCache service clusters with the Elastic Beanstalk environment. If you use a nonlegacy container, you can set your configuration files to use the supported container and then offload requests to the cache cluster.

"aware" of each other. For example, if a layer in your stack installs and configures `haproxy` for load balancing, any instances in the same layer will need to update to include the new node in `/etc/hosts` (or remove the node that went offline).

### Deploy

After an instance has come online and completes the initial `Setup` and `Configure` events, a `Deploy` event deploys any apps that you configure for the layer. This step can copy application code from a repository, start or refresh services, and perform other tasks to bring your application(s) online.

After an instance has run `Deploy` for the first time, it will never do so again automatically. This prevents untested changes from reaching production instances. After you test a feature change or bug fix, you must manually run the `Deploy` event with the AWS OpsWorks Stacks console or AWS CLI.

### Undeploy

The `Undeploy` lifecycle event runs when you delete or remove an app from a layer. You use this to perform tasks such as when you want to remove an application's configuration or other cleanup tasks when you remove an app.

### Shutdown

Before the actual shutdown command issues to an instance, the `Shutdown` lifecycle event gives you the opportunity to perform tasks such as taking snapshots and copying log files to Amazon S3 for later use. If the instance's layer also includes a load balancer, the instance deregisters after the configured connection draining time.

## Resource Management

AWS OpsWorks Stacks allows for management of other resources in your account as part of your stack, and it includes elastic IP addresses, Amazon EBS volumes, and Amazon RDS instances. You register the resources with the stack to make them available to assign them to instances or layers. If you attach resources to instances in the stack and you delete the instance, the resource remains registered with the stack until it is manually deregistered. Deregistering resources does not automatically delete them. You must delete the resource itself with the respective service console or AWS CLI command.

### Amazon EBS Volumes

Amazon EBS volumes that are not currently attached to any instances can register with a stack, and you can assign them to instances if the volume uses XFS formatting. You cannot attach volumes to running instances. To attach a volume to a running instance, you must stop it. You can move a volume between instances that are both offline.

TIP You cannot attach Amazon EBS volumes to Windows stacks.

#### Elastic IP Addresses

As with Amazon EBS volumes, elastic IP addresses that are not associated with resources in your account may be registered with the stack. You can assign an elastic IP address to an instance regardless of whether it is running or not. After an Elastic IP address is disassociated from an instance, a configure lifecycle event updates instances in the stack with the instance's new IP address.

### Amazon RDS Instances

You can register Amazon EBS instances to only one stack at a time. However, you can register a single Amazon RDS instance with multiple apps in the same stack.

### Chef 11 and Chef 12

Both Chef 11 and Chef 12 provide unique functionality differences that are important to note before you use AWS OpsWorks Stacks. Each of the major differences is outlined in this section.

The differences in this section are with respect to AWS OpsWorks Stacks as a service, and they do not include differences between Chef versions 11.10 and 12.0. Since version 11.10 has been deprecated by Chef, community support will not be as strong as for later versions.

#### Separate Chef Runs

In Chef 11.10 stacks, AWS-provided cookbooks were run in the same Chef run as any custom cookbooks. The AWS cookbooks performed various tasks such as mounting Amazon EBS volumes that had been attached to the instance in the AWS OpsWorks console. However, this could result in situations where custom cookbooks had naming conflicts with those provided by AWS. You had to split this into two separate Chef runs on the instance to eliminate any potential namespace conflicts.

#### Community Support

Since the deprecation of Chef 11.10, community support has gradually decreased. Any open source cookbooks on the Chef Supermarket, for example, will likely make use of Chef 12.0 functionality, removing backward compatibility for Chef 11.10 stacks.

#### Built-in Layers

Chef 12.0 stacks no longer include the built-in layers as in Chef 11.10 stacks, such as the Rails layer. To implement these layers in Chef 12.0 stacks, you can still copy the built-in cookbooks from a Chef 11.10 stack and update them to be compatible with Chef 12.0. Chef 12.0 stacks still support built-in layer types from Chef 11.10 stacks.

- Amazon RDS instance layers
- Amazon ECS cluster layers

#### Berkshelf

Berkshelf is no longer available for the automatic installation on Chef 12.0 instances. Instead, install Berkshelf with a custom cookbook.

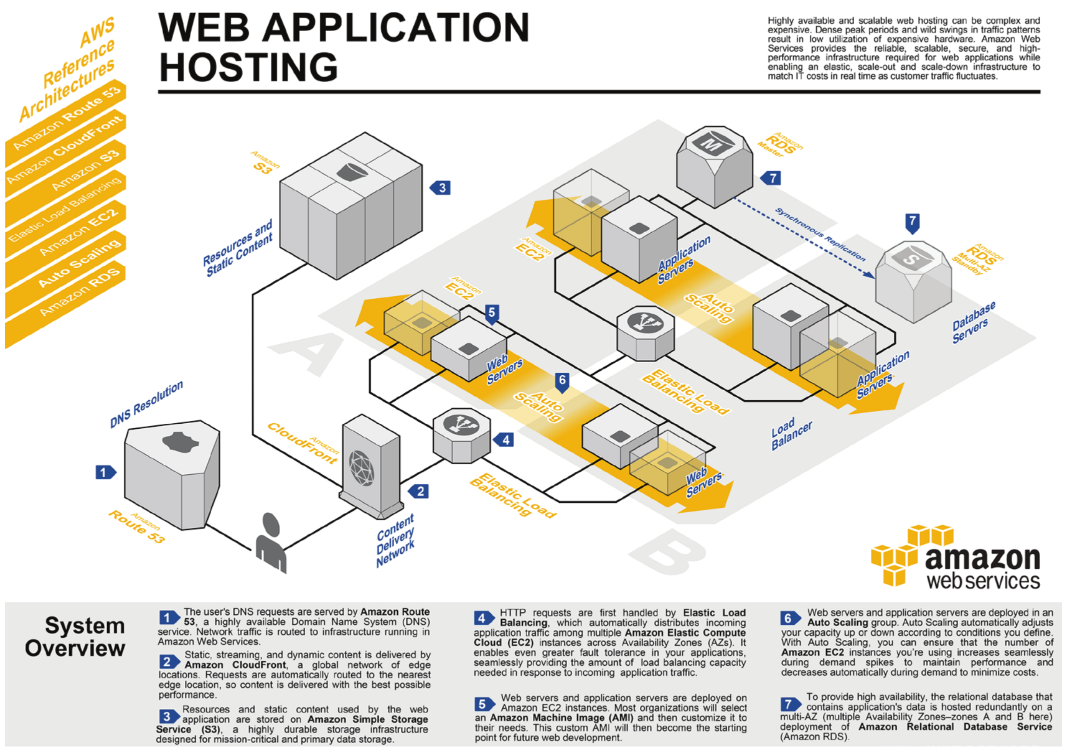
In addition to using the higher-level mobile and JavaScript SDKs, you can also use the lower-level APIs available via the following AWS SDKs to integrate all Amazon Cognito functionality in your applications:

- Java SDK
- .NET SDK
- Node.js SDK
- Python SDK
- PHP SDK
- Ruby SDK

## Standard Three-Tier vs. the Serverless Stack

This chapter has introduced serverless services and their benefits. Now that you know about some of the serverless services that are available in AWS, let's compare a traditional three-tier application against a serverless application architecture. Figure 13.5 shows a typical three-tier web application.

**FIGURE 13.5** Standard three-tier web infrastructure architecture



WEB APPLICATION HOSTING

Highly available and scalable web hosting can be complex and expensive. Dense peak periods and wild swings in traffic patterns result in low utilization of expensive hardware. Amazon Web Services provides the reliable, scalable, secure, and high-performance infrastructure required for web applications while enabling an elastic, scale-out and scale-down infrastructure to match IT costs in real time as customer traffic fluctuates.

System Overview

1 The user's DNS requests are served by **Amazon Route 53**, a highly available Domain Name System (DNS) service. Network traffic is routed to infrastructure running in Amazon Web Services.

2 Static, streaming, and dynamic content is delivered by **Amazon CloudFront**, a global network of edge locations. Requests are automatically routed to the nearest edge location, so content is delivered with the best possible performance.

3 Resources and static content used by the web application are stored on **Amazon Simple Storage Service (S3)**, a highly durable storage infrastructure designed for mission-critical and primary data storage.

4 HTTP requests are first handled by **Elastic Load Balancing**, which automatically distributes incoming application traffic among multiple **Amazon Elastic Compute Cloud (EC2)** instances across Availability Zones (AZs). It enables even greater fault tolerance in your applications, seamlessly providing the amount of load balancing capacity needed in response to incoming application traffic.

5 Web servers and application servers are deployed on Amazon EC2 instances. Most organizations will select an **Amazon Machine Image (AMI)** and then customize it to their needs. This custom AMI will then become the starting point for future web development.

6 Web servers and application servers are deployed in an **Auto Scaling** group. Auto Scaling automatically adjusts your capacity up or down according to conditions you define. With Auto Scaling, you can ensure that the number of **Amazon EC2** instances you're using increases seamlessly during demand spikes to maintain performance and decreases automatically during demand to minimize costs.

7 To provide high availability, the relational database that contains application's data is hosted redundantly on a multi-AZ (multiple Availability Zones–zones A and B here) deployment of **Amazon Relational Database Service** (Amazon RDS).

*Source*: `https://media.amazonwebservices.com/architecturecenter/AWS_ac_ra_web_01.pdf`

This architecture uses the following components and services:

**Routing:** Amazon Route 53

**Content distribution network (CDN):** Amazon CloudFront

**Static data:** Amazon S3

**High availability/decoupling:** Application load balancers

**Web servers:** Amazon EC2 with Auto Scaling

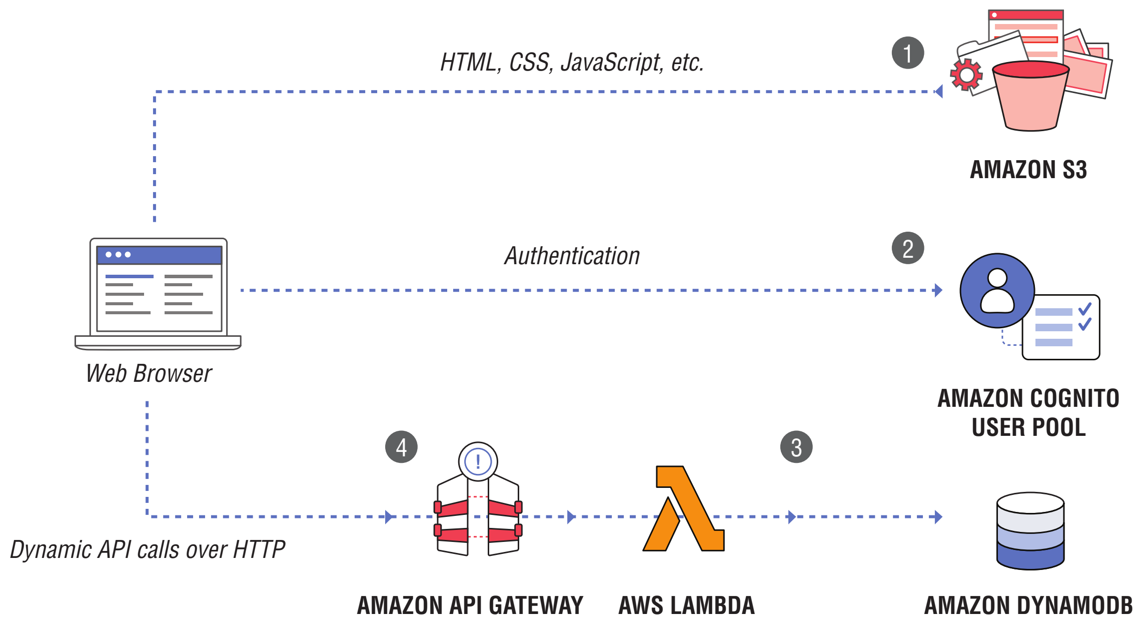
**App servers:** Amazon EC2 with Auto Scaling

**Database:** Amazon RDS in a multi-AZ configuration

Amazon Route 53 provides a DNS service that allows you to take domain names such as `examplecompany.com` and translate them to an IP address that points to running servers.

The CDN shown in Figure 13.6 is the Amazon CloudFront service, which improves your site performance with the use of its global content delivery network.

**FIGURE 13.6** Serverless web application architecture



*Source*: `https://aws.amazon.com/getting-started/projects/build-serverless-web-app-lambda-apigateway-s3-dynamodb-cognito/`

Amazon S3 stores your static files such as photos or movie files.

*Application load balancers* are responsible for distributing load across Availability Zones to your Amazon EC2 servers, which run your web application with a service such as Apache or NGINX.

Application servers are responsible for performing business logic prior to storing the data in your database servers that are run by Amazon RDS.

Amazon RDS is the managed database server, and it can run an Amazon Aurora, Microsoft SQL Server, Oracle SQL Server, MySQL, PostgreSQL, or MariaDB database server.

While this architecture is a robust and highly available service, there are several downsides, including the fact that you have to manage servers. You are responsible for patching those servers, preventing downtime associated with those patches, and proper server scaling.

In a typical serverless web application architecture, you also run a web application, but you have zero servers that run inside your AWS account, as shown in Figure 13.6.

Serverless web application architecture services include the following:

**Routing:** Amazon Route 53

**Web servers/static data:** Amazon S3

**User authentication:** Amazon Cognito user pools

**App servers:** Amazon API Gateway and AWS Lambda

**Database:** Amazon DynamoDB

Amazon Route 53 is your DNS, and you can use Amazon CloudFront for your CDN.

You can also use Amazon S3 for your web servers. In this architecture, you use Amazon S3 to host your entire static website. You use JavaScript to make API calls to the Amazon API Gateway service.

For your business or application servers, you use Amazon API Gateway in conjunction with AWS Lambda. This allows you to retrieve and save data dynamically.

You use Amazon DynamoDB as a serverless database service, and you do not provision any Amazon EC2s inside of your Amazon VPC account. Amazon DynamoDB is also a great database service for storing session state for stateful applications. You can use Amazon RDS instead if you need a relational database. However, it would not then be a fully serverless stack. There is a new service released called *Amazon Aurora Serverless*, which is a full RDS MySQL 5.6–compatible service that is completely serverless. This would allow you to run a traditional SQL database, but one that has the benefit of being serverless. Amazon Aurora Serverless is discussed in the next section.

You use Amazon Cognito user pools for user authentication, which provides a secure user directory that can scale to hundreds of millions of users. Amazon Cognito User Pools is a fully managed service with no servers for you to manage. While user authentication was not shown in Figure 13.6, you can use your web server tier to talk to a user directory, such as *Lightweight Directory Access Protocol* (LDAP), for user authentication.

As you can see, while some of the components are the same, you may use them in slightly different ways. By taking advantage of the AWS global network, you can develop fully scalable, highly available web applications—all without having to worry about maintaining or patching servers.

## Amazon Aurora Serverless

Amazon Aurora Serverless is an on-demand, auto-scaling configuration for the Aurora MySQL-compatible edition, where the database automatically starts, shuts down, and scales up or down as needed by your application. This allows you to run a traditional SQL database in the cloud without needing to manage any infrastructure or instances.

With Amazon Aurora Serverless, you also get the same high availability as traditional Amazon Aurora, which means that you get six-way replication across three Availability Zones inside of a region in order to prevent against data loss.

Amazon Aurora Serverless is great for infrequently used applications, new applications, variable workloads, unpredictable workloads, development and test databases, and multitenant applications. This is because you can scale automatically when you need to and scale down when application demand is not high. This can help cut costs and save you the heartache of managing your own database infrastructure.

Amazon Aurora Serverless is easy to set up, either through the console or directly with the CLI. To create an Amazon Aurora Serverless cluster with the CLI, you can run the following command:

```
aws rds create-db-cluster --db-cluster-identifier sample-cluster --engine aurora
--engine-version 5.6.10a \
--engine-mode serverless --scaling-configuration
MinCapacity=4,MaxCapacity=32,SecondsUntilAutoPause=1000,AutoPause=true \
--master-username user-name --master-user-password password \
--db-subnet-group-name mysubnetgroup --vpc-security-group-ids sg-c7e5b0d2
-region us-east-1
```

Amazon Aurora Serverless gives you many of the similar benefits as other serverless technologies, such as AWS Lambda, but from a database perspective. Managing databases is hard work, and with Amazon Aurora Serverless, you can utilize a database that automatically scales and you don't have to manage any of the underlying infrastructure.

## AWS Serverless Application Model

The *AWS Serverless Application Model* (AWS SAM) allows you to create and manage resources in your serverless application with *AWS CloudFormation* to define your serverless application infrastructure as a SAM template. A *SAM template* is a JSON or YAML configuration file that describes the AWS Lambda functions, API endpoints, tables, and other resources in your application. With simple commands, you upload this template to AWS CloudFormation, which creates the individual resources and groups them into an *AWS CloudFormation stack* for ease of management. When you update your AWS SAM template, you re-deploy the changes to this stack. AWS CloudFormation updates the individual resources for you.

AWS SAM is an extension of AWS CloudFormation. You can define resources by using the AWS CloudFormation in your AWS SAM template. This is a powerful feature, as you can use AWS SAM to create a template of your serverless infrastructure, which you can then build into a DevOps pipeline. For example, examine the following:

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: 'Example of Multiple-Origin CORS using API Gateway and Lambda'
```