TCSS 462/562: (Software Engineering for) Cloud Computing
Fall 2025

School of Engineering and Technology
University of Washington – Tacoma

Tutorial 6 – Introduction to Lambda III: Serverless Databases

Disclaimer: Subject to updates as corrections are found – please report errors
Version 0.10
Scoring: 20 pts maximum

The purpose of this tutorial is to introduce the use of relational databases from AWS Lambda. This tutorial will demonstrate the use of SQLite, a file-based database that runs inside a Lambda function to provide a "temporary" relational database that lives for the lifetime of the Lambda function's runtime container. Secondly, the tutorial demonstrates the use of the Amazon Relational Database Service (RDS) to create a persistent relational database using Serverless Aurora MySQL 5.6 for data storage and query support for Lambda functions.

Goals of this tutorial include:

1. Introduce the Sqlite database using the command line "sqlite3" tool.
2. Deploy a Lambda Function that uses a file-based SQLite3 database in the "/tmp" directory of the Lambda container that persists between client function invocations
3. Compare the difference between using file-based and in-memory SQLite DBs on Lambda.
4. Create an Amazon RDS Aurora MySQL Serverless database
5. Launch an EC2 instance and install the mysql command line client to interface with the Aurora serverless database.
6. Deploy an AWS Lambda function that uses the MySQL Serverless database.

1. Using the SQLite Command Line

To begin, create a directory called "saaf_sqlite".
Then clone the git repository under the new directory:

```
git clone https://github.com/wlloyduw/saaf_sqlite3.git
```

If using Windows or Mac, download the "Precompiled binaries" as a zip file from:
https://www.sqlite.org/download.html

On Windows/Mac, unzip the zip file, and then run the **sqlite3** program.

On Ubuntu Linux, the package sqlite3 can be installed which is version ~3.37.2 on Ubuntu 22.04 LTS. Then launch the sqlite3 database client:

```
sudo apt update
sudo apt install sqlite3

# navigate to your java project directory first
cd {base directory where project was cloned}/saaf_sqlite3/java_template/
```

```
# run the sqlite3 database command-line console
sqlite3
```

Check out the version of the db using ".version". (the hash value will not match)

```
sqlite> .version
SQLite 3.45.1 2024-01-30 16:01:20 872ba256cbf61d9290b571c0e6d82a20c224ca3ad82971edc46b29818d5dalt1
zlib version 1.3
gcc-13.2.0 (64-bit)
```

Check out available commands using ".help".

Next create a new database file, and then exit the tool:

```
sqlite> .save new.db
sqlite> .quit
```

Then, check the size of an empty sqlite db file:

```
$ ls -l new.db
total 3848
-rw-r--r-- 1 wlloyd wlloyd    4096 Nov  1 19:05 new.db
```

It is only 4096 bytes, very small!

Next, work with data in the database:

```
$ sqlite3 new.db
SQLite version 3.45.1 2024-01-30 16:01:20
Enter ".help" for usage hints.
sqlite> .databases
main: /home/wlloyd/git/saaf_sqlite3/java_template/new.db r/w
sqlite> .tables
```

There are initially no tables.
Create a table and insert some data:

```
sqlite> create table newtable (name text, city text, state text);
sqlite> .tables
newtable
sqlite> insert into newtable values('Susan Smith','Tacoma','Washington');
sqlite> insert into newtable values('Bill Gates','Redmond','Washington');
sqlite> select * from newtable;
Susan Smith|Tacoma|Washington
Bill Gates|Redmond|Washington
```

Now check how the database file has grown after adding a table and a few rows:

sqlite> .quit
$ ls -l new.db

Question 1. After creating the table 'newtable' and loading data to sqlite, what is the size of the new.db database file?

The sqlite3 command line tool can be used to perform common **C**reate **R**ead **U**pdate and **D**elete queries on a sqlite database. This allows the database to be preloaded with data and bundled with a Lambda function for deployment to the cloud as needed.

If you're unfamiliar with SQL, and writing SQL queries, please consider completing the online tutorial:

SQLite Tutorial:
http://www.sqlitetutorial.net/

Follow the four steps for "Getting started with SQLite", and then complete the Basic SQLite tutorial to review performing different types of queries using the sample Chinook database with 11 tables downloaded from step 3.

2. Combining SQLite with AWS Lambda

SQLite can be leveraged directly from programming languages such as Java, Python, and Node.JS. SQLite provides an alternative to using CSV or text files to store data. SQLite provides a SQL-compatible file-based relational database. SQLite does not replace a full-fledged enterprise relational database management system (dbms) in terms of scalability, etc. But the small footprint of SQLite is ideal for deployment in a serverless function or on an Internet of Things (IoT) device.

Next, explore the saaf_sqlite project in Netbeans or another Java IDE (or text editor).

The project "saaf_sqlite3" provides a Java-based Lambda "Hello" function based on SAAF from Tutorial #4. Look at the code inside: **saaf_sqlite3/java_template/src/main/java/lambda/HelloSqlite.java**, ~ around line 60

```
setCurrentDirectory("/tmp");

pwd = System.getProperty("user.dir");
        logger.log("pwd=" + pwd);

try
{
       // Connection string an in-memory SQLite DB
       Connection con = DriverManager.getConnection("jdbc:sqlite:");

       // Connection string for a file-based SQlite DB
       // Connection con = DriverManager.getConnection("jdbc:sqlite:/tmp/mytest.db");
```

The first line of code (LOC) calls a helper function to set the working directory to "/tmp" inside the Lambda function.

"/tmp" provides a read/write 512MB ephemeral filesystem on Lambda.
As a security precaution, functions deployed to AWS Lambda can only write to the /tmp filesystem. /tmp is enabled for read/write.

The next code is debugging output to print the present working directory to the Lambda function's log.
This allows you to verify that changing the working directory was successful.

Next, SQLite can work with databases entirely in memory, or on disk. The first database connection string in the code creates a connection to an **in-memory** SQLite database. The second database connection string, *which is commented out*, can be used to work with a SQLite database **stored in a disk file** on /t mp.

The advantage of creating the database on disk is that data **persists** beyond the runtime of the Java code. On AWS Lambda, this means as long as the original function's runtime container (function instance) is preserved, the data is preserved. If function instances are kept WARM (via continual usage or through provisioned concurrency), they can last up to 2 to 4 hours. **After 2-4 hours, the file-based SQLite database stored under /tmp inside a function instance, if not saved to S3 or another service, will be lost !**

NOW, change the code to use the file-based sqlite database.
Comment out the first line of code that initializes the database connection to use the in-memory DB, and **uncomment the second line** of code that initializes the database connection to use the file-based DB.

Perform a clean build of the saaf_sqlite project to create a jar file.
Following instructions from tutorial #4, deploy a new lambda function called "helloSqlite".
Be sure to set the function's handler in the AWS Lambda GUI.
Choose one method (AWS CLI or CURL) for invoking "helloSqlite" from **callservice.sh**.
If wanting to use a HTTP/REST URL, configure the API Gateway to provide a URL for access via Curl. Otherwise use the "helloSqlite" Lambda function name and the AWS Lambda CLI.

Under your new project, modify the callservice.sh script to invoke your newly deployed Sqlite Lambda function:
saaf_sqlite/java_template/test/callservice.sh

Be sure to update callservice.sh to use the proper AWS region (us-east-2).

Now run the script. The API Gateway invocation code in BASH from github has been commented out using a "#" in front of each line.

```
$ ./callservice.sh
Invoking Lambda function using AWS CLI
real 0m11.985s
user 0m0.288s
sys 0m0.064s

AWS CLI RESULT:
{
  ….. // some attributes removed from brevity...
  "uuid": "8c321d18-d16e-4cd8-acac-cbc8d65fe138",
  "error": "",
  "vmuptime": 1541129227,
  "newcontainer": 1,
  "value": "Hello Susan Smith",
  "names": [
    "Susan Smith"
  ]
}
```

Using a file, each time the service is called and the same runtime container is used, a name is appended to the temporary file-based SQLite database. We see the "names" array in the JSON grow with each subsequent call.

Try running the ./callservice.sh script now several times (10x) to watch the names array grow.

Now, try out what happens when two clients call the Lambda function at the same time.

Inspect the simple calltwice.sh script:

```
cat calltwice.sh
```

Now, try running calltwice.sh:

```
./calltwice.sh
```

If you do not see the command prompt after awhile, press [ENTER].

Invoking a Lambda with two clients in parallel forces Lambda to create an second function instance runtime environment. (this is a separate microVM !)

Question 2. When the second client calls the helloSqlite Lambda function, how is the data different in the second container environment compared to the initial/first container?

Now, try out a memory-only SQLite database. Restore your Lambda code to the original state to use an in-memory Sqlite database. Comment out the file-based database connection initializer:

```
setCurrentDirectory("/tmp");
try
{

  // Connection string an in-memory SQLite DB
  Connection con = DriverManager.getConnection("jdbc:sqlite:");

  // Connection string for a file-based SQlite DB
  // Connection con = DriverManager.getConnection("jdbc:sqlite:/tmp/mytest.db");
```

Build a new JAR file, and redeploy it to Lambda for the helloSqlite Lambda function.

Using callservice.sh, try calling the Lambda several times in succession.

Question 3. For Lambda calls that execute in the same runtime container identified by the UUID returned in JSON, does the data persist between client Lambda calls with an in-memory DB? (YES or NO)

Next, let's modify the code for helloSqlite to add a static int counter that tracks the total number of calls to the container.

Define a static int at the start of public class HelloSqlite:

```
public class HelloSqlite implements RequestHandler<Request, HashMap<String,
Object>> {
    static int uses = 0;
/**
  * Lambda Function Handler
```

Then modify the definition of String hello near the bottom of the Lambda function to report the uses count:

```
        // ***************************************************************
        // Set hello message here
        // ***************************************************************
        uses = uses + 1;
        String hello = "Hello " + request.getName() + " calls=" + uses;
```

Build a new JAR file, and redeploy it to Lambda for the helloSqlite Lambda function.

Using callservice.sh, try calling the Lambda with the static uses counter several times in succession:

```
./callservice.sh
./callservice.sh
./callservice.sh
```

Question 4. Does the value of the static int persist for Lambda calls that execute in the same runtime container identified by the UUID returned in JSON? (YES or NO)

Now, try running with calltwice.sh.

Question 5. How is the value of the static int different across different runtime containers identified by the UUID returned in JSON?

Next, inspect the SQL code for the helloSqlite Lambda function:

```
// Detect if the table 'mytable' exists in the database
PreparedStatement ps = con.prepareStatement("SELECT name FROM sqlite_master WHERE
type='table' AND name='mytable'");
ResultSet rs = ps.executeQuery();
if (!rs.next())
{
       // 'mytable' does not exist, and should be created
       logger.log("trying to create table 'mytable'");
       ps = con.prepareStatement("CREATE TABLE mytable ( name text, col2 text, col3
text);");
       ps.execute();
}
rs.close();

// Insert row into mytable
ps = con.prepareStatement("insert into mytable values('" + request.getName() +
"','b','c');");
ps.execute();

// Query mytable to obtain full resultset
ps = con.prepareStatement("select * from mytable;");
rs = ps.executeQuery();
```

The approach of our helloSqlite Lambda is to create a new file (or memory) database each time.

Question 6. Before inserting rows into 'mytable', what has to be done and why in the Java code above?

An in-memory SQLite DB can be preserved between calls. The Lambda function could use a static Connection object that is initialized similar to how the "uses" variable was initialized. **The problem with our code is that we do not use a static Connection object, and the connection gets closed. See the line of code with the 'con.close();' call. This is around line 101. As a BONUS ACTIVITY:** make the Connection object a static variable. Initialize it only one time, and do not close the 'conn' object at the end of the function. If these changes are made correctly, then the connection object to the in-memory database will stay open and the data is persisted as long as the Lambda function's runtime container is preserved in the "warm" state.

A similar optimization for the file-based database, it is not necessary to re-establish the database connection for every call. The Connection object can be made static, and the connection can remain open between calls. This effectively "caches" the database connection between calls and is a **best-practice** for better performance.

3. Optional Exercise: Persisting SQLite database files to S3

Leveraging concepts from tutorial #5, modify the file-based version of helloSqlite to always save the database file in /tmp to S3 at the end of the function's code. Since we have no idea when the last function call in the warm-state will occur, we have to persist the database to S3 ***FOR EVERY CALL!!***. Add a key to the request.java to allow the user to specify a database filename.

At the beginning of the function handler, using the user provided database filename from the request, check if the specified file exists in /tmp. If it does not exist, then download the file from S3. Then change the SQLite connection string to open the user provided database name. This way a user could request a specific database from S3 for their Lambda function call. As an added feature, if the file does not exist in S3, then create an initial version in /tmp and upload this to S3.

4. Create an Amazon Aurora MySQL Serverless Database v2

The AWS Relational Database Service now offers the Amazon Aurora Serverless MySQL database. As of November 20, 2024, Aurora Serverless version 2 now supports the ability to scale to ZERO capacity, which suspends always-on charges when you're not actively using the database. This is effectively a serverless relational database:
https://aws.amazon.com/blogs/database/introducing-scaling-to-0-capacity-with-amazon-aurora-serverless-v2/

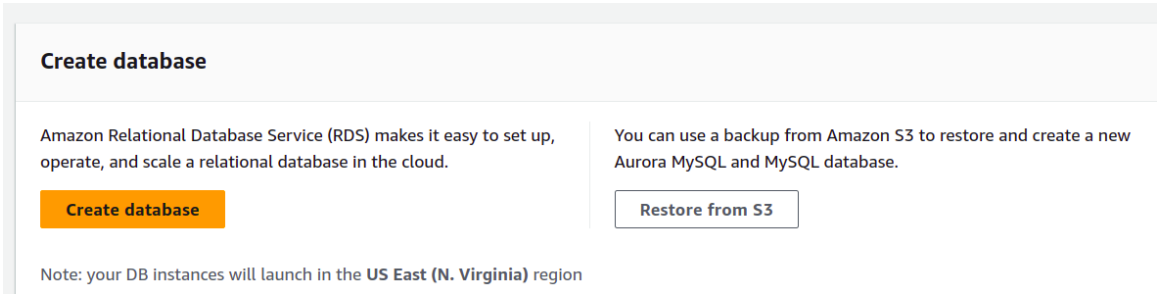
Amazon Aurora charges based on the number of Aurora Compute Units used per hour. 1 Amazon Compute Unit (ACU) provides approximately 2 GB of memory and 1 vCPU to host a database for 1-hour. Prices for 1 ACU are: 12c/hour, $2.88/day, or $20.16/month plus additional storage charges to persist data to disk.

Serverless Aurora supports automatic horizontal scaling of database servers from 0 ACUs to as many as 256 ACUs in 0.5 increments. Aurora supports automatic database backups and replicas.

To get started, create a serverless database!

**** This portion of the tutorial requires a standard AWS account to complete. ****
Aurora serverless is not supported using an AWS Free Plan account.

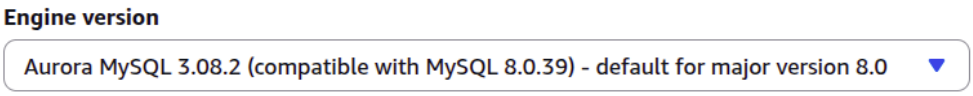
Under Services, search for and go to “RDS”.
Scroll down to **Create database**, and press the “**Create database**” button to launch the wizard:



First use the “**Standard create”** database creation method.

Then specify the “Aurora (MySQL Compatible)” engine:

For the **“Engine version”**, keep the default setting.



For “**Templates**”, select **Dev/Test**.

Now scroll down to “Settings”.

Specify the DB cluster identifier:
DB cluster identifier: **tcss462-562**

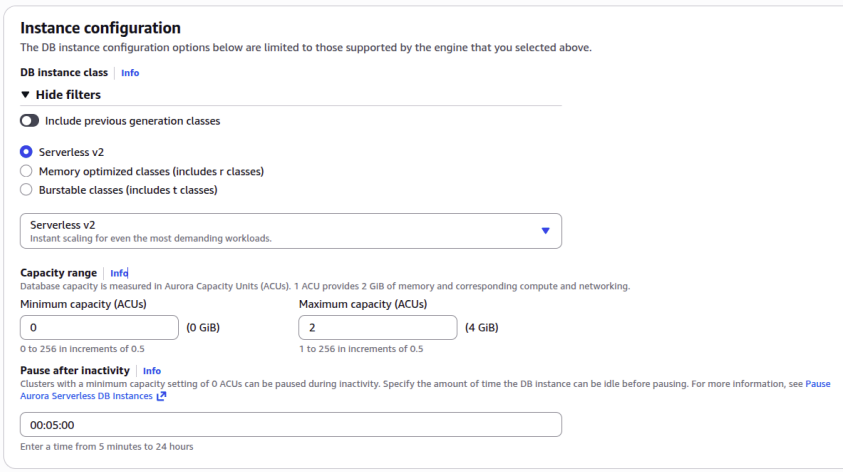
Scroll down to “Credentials Settings”, “Credentials management” and select “**Self managed**”.
Next, specify a Master username, a Master password, and then confirm the master password.
The default Master username is “admin”. Keep this, or change it if desired.
An example password is “tcss462562”.

***** Write down the Master username and Master password or memorize it. This is needed later. *****

Keep ‘**Aurora Standard**’ as the Cluster storage configuration.

Next, scroll down to “**Instance Configuration”** and choose the following option:
“**Serverless v2”**:

The instance configuration dialog is as follows:



Specify the '**Instance configuration'** options.
To save cost, specify a very low range of Aurora Compute Units:

Minimum capacity (ACUs):	0	(0 GB, 0 vCPUs)
Maximum capacity (ACUs):	2	(4 GB, 2 vCPUs)
Pause after inactivity:	00:05:00	(pause after 5 minutes of inactivity, the minimum setting)

Note that running the server for 1 hour with 0.5 Aurora Capacity Units costs 6¢.
Aurora Serverless database cost 12¢/per ACU/hour billed to the nearest second.
Your database will scale up to the maximum defined ACUs, and then take approximately 3-minutes to scale down to 0.5 ACUs after a period of inactivity. After 5 minutes, the database hibernates and goes to sleep. Waking the database up can take several seconds.

For Availability & durability, select: '**Don't create an Aurora Replica**'.

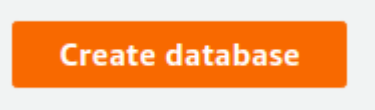
For **Connectivity** settings specify:

Virtual Private Cloud (VPC):	**Default VPC (vpc-xxxxxx)**	note: *the vpc-id will be visible*
DB subnet group:	**Select "default-vpc-xxxxxxxx"**	
Public access:	**No**	

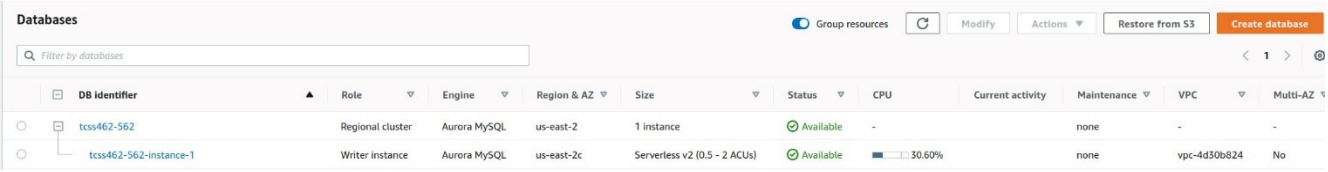
VPC security group (firewall):
Select "Choose existing", and then verify that "**default**" is specified as the "Existing VPC security groups".

Availability Zone: Can be left as-is, "**No Preference**".
Optionally, the availability zone can be set to FORCE the database to be created in a specific zone.

For the remaining settings, the defaults can be used.
The press:



The RDS databases list will appear.
The state of creation can be monitored. Database creation may take ten minutes or more.
While the database is being created, go to the next step in the tutorial.

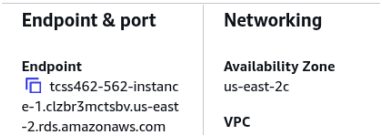


5. Use the FREE CloudShell environment inside your web browser to connect to the Aurora database

It is not possible to directly connect to the Aurora MySQL Serverless database from the command line of your laptop/desktop Ubuntu environment unless the database is configured to have a public IP address which costs extra. Our database deployment lives on a Virtual Private Cloud (VPC) that does not allow inbound traffic from the internet. This provides network isolation and security.

To access the database for free, we can use the "CloudShell" from inside the AWS Management Console in your web browser. First, you'll need to learn which availability zone your database was created in.

Select your DB "instance". This is the row below the first row in the "DB identifier" column.
Inspect the DB instance details, and check the "Availability Zone":



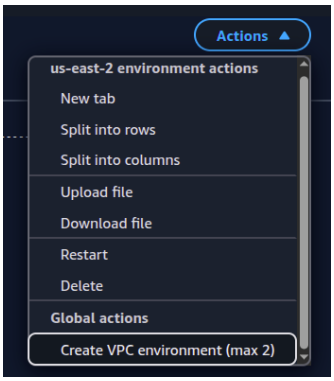
The example here shows "us-east-2c".

<u>Discussion for directly connecting to the database from your desktop/laptop:</u>
To save money and enhance security, we created an Aurora database without a public IP address, To connect to this database which lives on the private network inside your Amazon Virtual Private Cloud (VPC), solutions include: (1-NAT Gateway) setting up a NAT Gateway (4.5¢/hour) for your the VPC, (2-ec2 VM as a gateway) using an ec2 instance with a public IP and a private IP on the same subnet as your database to serve as a router/gateway between the public internet and database on the private network, (3-proxy server) hosting a proxy server such as haproxy on an ec2 instance in the same availability zone with a private IP on the same subnet as your database to route traffic to your database through the proxy server, or (4-RDS proxy) setting up an RDS proxy. Setting up any of these solutions is outside the scope of Tutorial 6. ***<u>For tutorial 6</u>***, **we will use AWS Lambda functions deployed in the same VPC as the Aurora database**. This way no special networking is required to access the database. This saves cost and complexity!

We will connect with the CloudShell from the Browser. Next, open the CloudShell environment.
You can search for "CloudShell" in the Search bar, or click on the icon on the upper right toolbar:

The CloudShell that you launch by clicking on the icon, is not created within a VPC.
It is not able to connect to your database.

We must open a CloudShell in the same VPC and availability zone as your database.
On the right, from the “Actions” drop down menu, select “Create VPC environment (max 2)”.



Configure your new CloudShell to use a “VPC environment”.
Set the following parameters:

Name: Give your session a unique name, such as “us-east-2c” (the AZ of your database)
VPC: Select the same VPC as your database
Subnet: Choose a subnet associated with the availability zone of the database.
You should be able to see this subnet name when inspecting the database networking properties.
Security Group: Select the same security group as your database. (this is probably the ‘Default’).

The CloudShell uses Amazon Linux, which uses Yum package repositories.
The CloudShell already has the AWS CLI installed as well as the mysql database client.

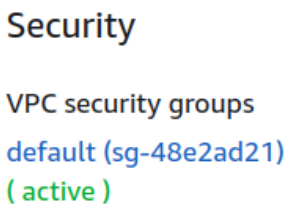
If you ever need to install the mysql client in a Ubuntu environment, it can be installed as follows:

```
# Install mysql client – not required using Amazon CloudShell

sudo apt update

# sudo apt install mysql-client-core-5.7   # For old-versions of Ubuntu < 20.04
sudo apt install mysql-client-core-8.0     # for Ubuntu >= 20.04
```

Now customize the following command to point at your RDS database.

Navigate back into RDS in the AWS management console.
On the left hand-side select “Databases”, then select your database instance.

Security Group Configuration – Not Required with CloudShell
If you were going to connect to your database using an ec2 instance, or personal laptop/desktop, it is necessary to configure the security group to open port 3306 to allow clients to connect to the database. In the Connectivity & security tab, look under “Security” on the right:

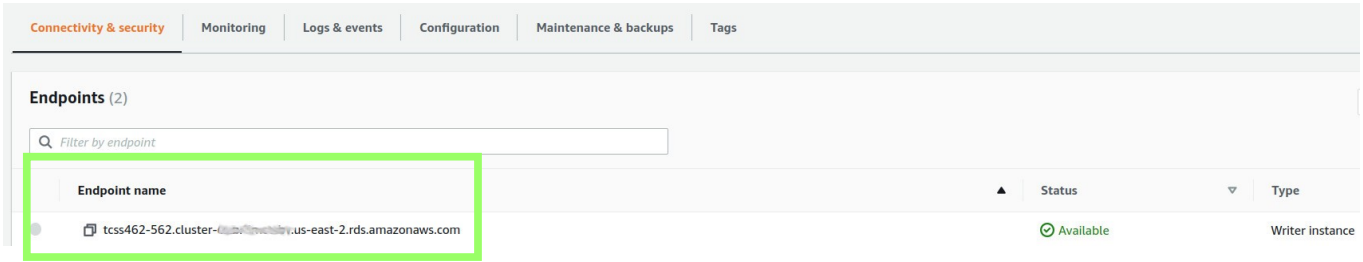


Click on the blue security group label to jump to the EC2 dashboard to edit network security settings.

In the Security editor, click the [Inbound rules] tab.
Click [Edit Inbound rules].
Click [Add Rule].
Select “MYSQL/Aurora”.
For the source, select “Anywhere-IPv4” to obtain the address range of “0.0.0.0/0”.
This enables any VM within the private VPC network to be able to connect to the database.
Hit [Save].

Now copy the database endpoint name which can be found by browsing the Aurora database in the Amazon RDS GUI.

Now navigate back to the **RDS service**.
On the left hand-side select “Databases”, then select the “tcss462-562” DB.
Look under the “Connectivity & security” tab,
Copy and paste the Endpoint name of the “Writer Instance” using the copy icon:



Now let’s use the mysql client from CloudShell (which is already installed) to connect to your RDS database.
Replace <DatabaseEndpoint> and <YourPassword>. Note all parameters require double-dashes.

```
mysql --host=<DatabaseEndpoint> --port=3306 --user=tcss462_562 --password=<YourPassword>
```

Your Mysql client program should connect to the backend database.
The client program provides a command-line interface for working with the database server.

```
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MySQL connection id is 740
Server version: 8.0.39 8bc99e28
```

```
Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MySQL [(none)]>
```

Try out the following commands.
MySQL can support multiple databases within a single server.
Display the databases on your RDS database:

```
show databases;
```

Now, create a new database:

```
create database TEST;
```

And check the list again:

```
show databases;
```

It is necessary to issue a “use” command for mysql to direct SQL queries to the database:

```
use TEST;
```

Next, create “mytable” to store data:

```
CREATE TABLE mytable (name VARCHAR(40), col2 VARCHAR(40), col3 VARCHAR(40));
```

Then display the list of known tables in the database:

```
show tables;
```

And describe the structure of the table:

```
describe mytable;
```

Now, try adding some data:

```
insert into mytable values ('fred','testcol2','testcol3');
```

And then check if it was inserted:

```
select * from mytable;
```

Help is available with the “help” command:

```
help
```

Exit mysql with:

```
\q
```

6. Accessing Aurora Serverless Database from AWS Lambda

Next, on your development computer, create a directory called "saaf_rds".
Then under the new directory, clone the git repository:

```
git clone https://github.com/wlloyduw/saaf_rds_serverless.git
```

This project, provides a Lambda function that will interact with your Amazon RDS database. It requires "mytable" to have been created under the "TEST" database.
Optionally, it should be possible to create the database and table programmatically from Java if necessary.

Note the version of the SAAF framework in this project may not be up-to-date. For the term project, it is recommended to use this project only for reference purposes, and then to create a new project with the proper dependencies by cloning SAAF directly.

Once acquiring the project files, it is necessary to create a file called "db.properties".
There is a template provided. Copy this template to be named "db.properties" and edit this file to specify how to connect to your RDS database:

Find and edit this file:

```
cd saaf_rds_serverless/java_template/src/main/resources/
cp db.properties.empty db.properties
gedit db.properties
```

The **URL** should be specified as follows:

```
jdbc:mysql://<your database endpoint>:3306/TEST
```

Replace "<your database endpoint>" with the RDS database endpoint used to connect with mysql above. Be sure to add values for **password**, and **username** as well based on how your RDS database was initially configured.

Next, using NetBeans, perform a Clean Build of the Maven project to create the function's jar file for deployment.

Now, create a new Lambda function - - *this could be called **'helloMySQL'**.*
Refer to tutorial 4 for detailed instructions of creating Lambda functions.
In the Create a Function Wizard, for **Permissions**, initially create the function using default permission settings.

Be sure to upload the **code source** under the **Code** tab as the newly created jar file.
Be sure to specify the **handler** under the **Code** tab and under **Runtime settings**.
The function handler should be set to use the HelloMySQL class:
lambda.HelloMySQL::handleRequest

Next, adjust the security permissions.
Under the function's **Configuration** tab, select **Permissions** from the left.

Click on the blue **Role name** link.
This will open the function's security role in the IAM role editor.

On the right, select the **Add permissions** drop-down list, and select **Attach policies**:

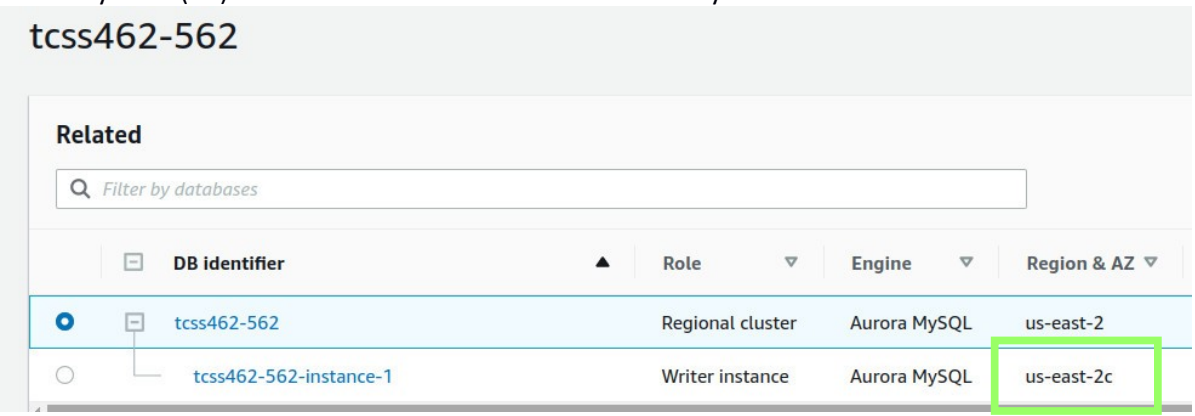
Then and attach the following policies one at a time:
AmazonRDSFullAccess
AWSLambdaVPCAccessExecutionRole

Then close the IAM role editor and go back to the AWS Lambda GUI.

Next, configure this Lambda function to run inside the same VPC and subnet as your RDS database. If not, there will be no connectivity between Lambda and the RDS database.

Under the "**Configuration**" tab, select **VPC** on the left. Now, click **Edit** to change the VPC configuration.
From the dropdown list, select the VPC that is labeled as "Default". Next, specify the function's subnet(s).
Every subnet has an availability zone listed on the far right.
Match the availability zone of your database with your serverless function.
For example, if the database is in us-east-1d, the function's subnet must match: us-east-1d.

To check which subnet (availability zone) your RDS serverless database is using, navigate to RDS.
On the left hand-side select "Databases", then select your database "tcss462-562".
The availability zone (AZ) of the database instance is shown clearly:



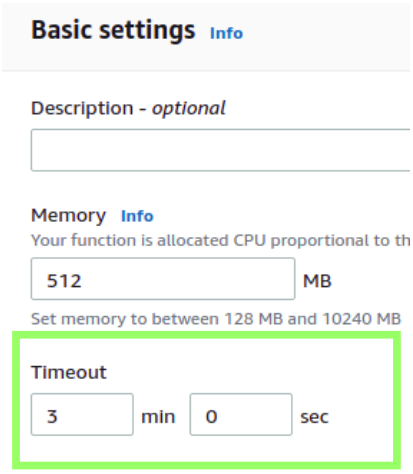
IMPORTANT: BE sure the subnet for your Lambda function matches the database instance.
Select **at least one subnet** for your function that is shared with the RDS database. If using the Virginia region, expect subnets to be us-east-1a, us-east-1b, us-east-1c, us-east-1d, us-east-1e, and us-east-1f.

Don't worry about this message if you receive it:

⚠ We recommend that you choose at least 2 subnets for Lambda to run your functions in high availability mode.

Select the default security group for the VPC settings and then SAVE the VPC settings.

Next, under the "**Configuration**" tab, under "**General configuration**" on the left:
Set the Timeout to be greater than 3 minutes. Working with RDS for the term project, some queries may take several minutes. For tutorial 6, a ***VERY*** long timeout is not required.



Your Lambda function should be ready to use.

Next configure callservice.sh under 'saaf_rds_serverless/java_template/test' to use the name of your newly deployed Lambda function.

Use the AWS CLI to invoke the function directly. **Using the AWS CLI to invoke Lambda directly is recommended because of the potential for long timeouts when executing long queries with RDS Aurora Serverless.**

Once complete, test the service by running callservice.sh to invoke your new Lambda function to write names to your Aurora MySQL database specified in the db.properties file: database=TEST table=mytable

7. After modifying callservice.sh, test your function with the time command as follows:

```
time ./callservice.sh
```

a) What is the "real" time reported for your Lambda function **in seconds** for the **FIRST CALL**?

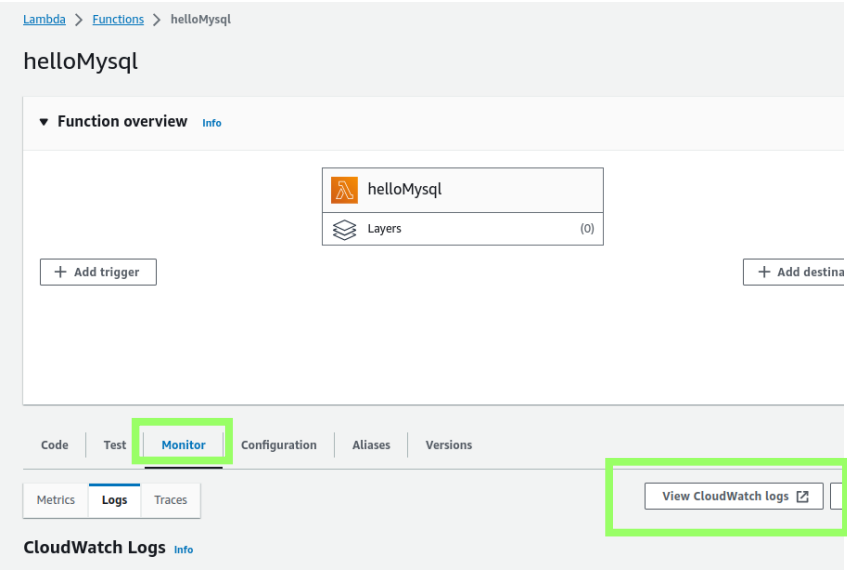
b) What is the value of the newcontainer attribute?
Does this indicate your Lambda function is warm? (YES/NO)
A “1” indicates a **COLD** container.

Run your function again with the time command.
c) How long does a second call take **in seconds**?

Now rerun the script several more times (3x-5x).

Each call to HelloMySQL will append a row to the table with the provided name.
The function queries the names stored in the table using a select SQL query.
The names are output in JSON using the “names” array.
INSPECT THE JAVA SOURCE CODE to check how this is done in the HelloMySQL.java class in the handleRequest() method.

It may be necessary to troubleshoot your Lambda function’s connectivity to RDS.
From the Monitoring tab of Lambda, use the [View logs in CloudWatch] button:



8. For question #8, modify the Lambda service to return the MySQL version as a response object parameter. Add a getter/setter to Response.java for “mysqlversion”. Then, add an additional SQL query to obtain the version of MySQL. Use the following SQL query:

```
select version() as version;
```

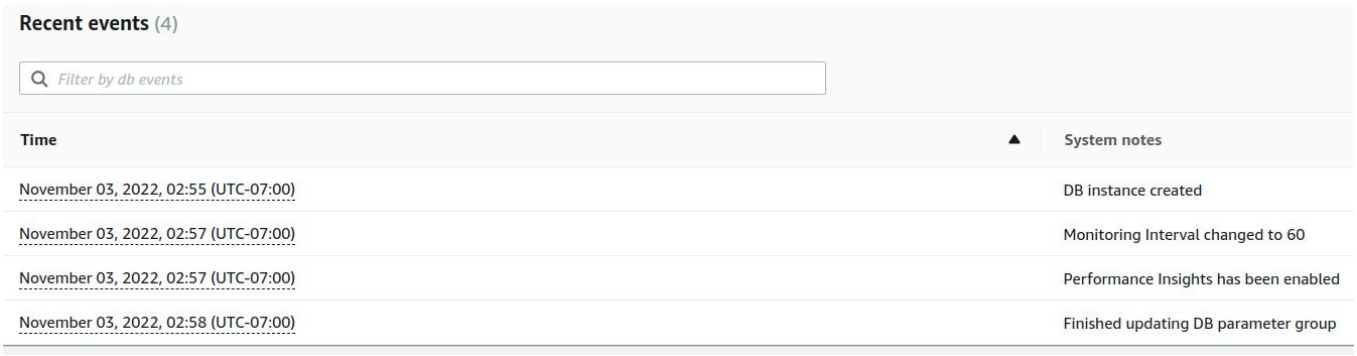
With a result set, read the value from the column, and add it to the Response object.

Your code should now have 3 total SQL queries. 1 insert, and 2 selects.

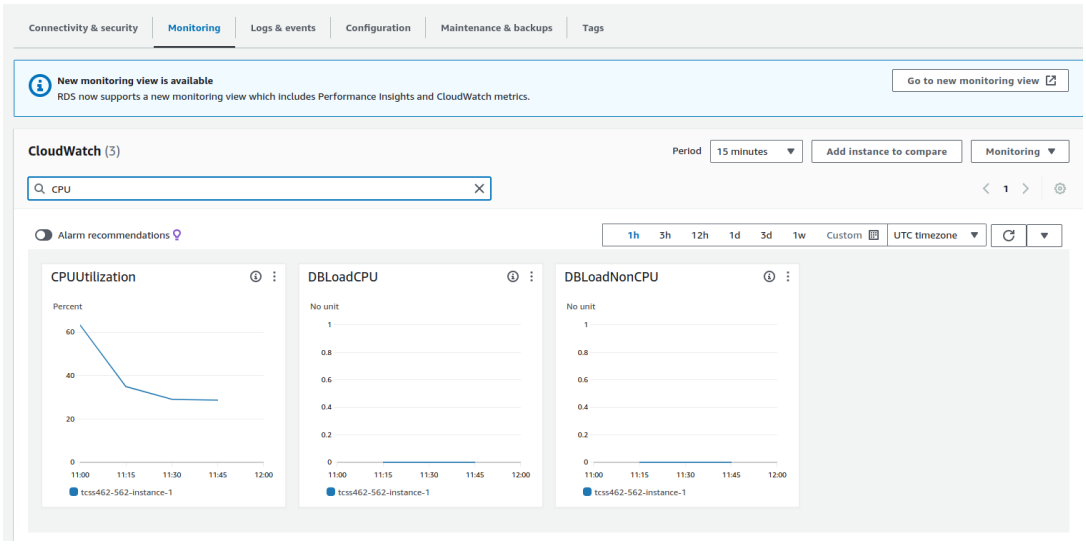
Now, using callservice.sh, run the service. Capture the complete output from the script using a terminal and provide this output as the answer for #8.

Aurora Serverless v2 scales down to the minimum number of ACUs specified for the database after ~3 minutes of inactivity.

Database events are reported under the “Logs & events” tab on the RDS database page. From the log state changes can be monitored:



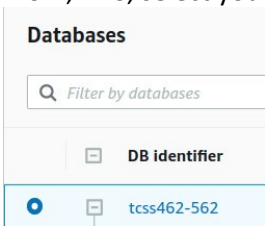
In RDS, under monitoring, CloudWatch graphs show RDS database resource utilization.
If you do not specify a filter, there are about 97 metrics.



7. Stopping the database for up to 7 days

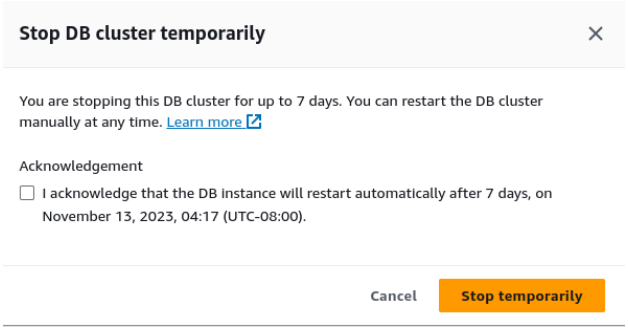
Amazon RDS Aurora Serverless v2 databases can be stopped for up to 7 days.
When stopped, they will not respond to any queries.
This allows the database to remain inactive in the account where only data storage charges apply.
WARNING !!! ** After 7 days, the database will automatically reactive, but there should be no charge if minimum ACUs have been configured to 0 (ZERO).

Now, practice temporarily stopping the database cluster.
From, RDS, select your database cluster, not the instance, but the whole cluster.



Then, select “Stop temporarily” from the Actions button drop-down.

A dialog box will now warn you how the database can only be suspended for 7 days.
It will then resume and start charging your account:



If needing to stop the database for more than a week, it will be necessary to re-stop it.
You could send a calendar remind with alarms on your smart phone, to remind yourself to re-stop the database each time.
The recommended best practice after completing the tutorial is to **DELETE THE DATABASE**, and recreate the database later if needed.

Please review the documentation for more information on database backup and recovery:
https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/BackupRestoreAurora.html

9. Question 9

a. How long can an Amazon Aurora serverless database be temporarily stopped before it automatically restarts and is able to respond to SQL queries ?

b. What is the best practice for your Amazon Aurora serverless database after completing tutorial 6 or any other assignment using a database?

Submitting Tutorial #6

Create a PDF file using Google Docs, MS Word, or OpenOffice. Capture answers to questions 1-9 and submit the PDF on Canvas.

After completing the tutorial, be sure to terminate EC2 instances, and **DELETE** your Aurora RDS database.

DON'T JUST STOP THE DATABASE - - - DELETE IT ENTIRELY.
USE THE "DELETE" ACTION TO DO SO.

Related Articles providing additional background:

Article describing use cases for when to use the SQLite database:
https://www.sqlite.org/whentouse.html

Using Aurora Severless v2 database:
https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-serverless-v2.html

Research paper on AWS Aurora – Cloud Native relational database with built in read replication up to 15-nodes:
https://media.amazonwebservices.com/blog/2017/aurora-design-considerations-paper.pdf

Key Aurora Serverless v2 limitation from:
https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-serverless-v2.how-it-works.html#aurora-serverless-v2.how-it-works.scaling

Scaling to Zero ACUs with automatic pause and resume for Aurora Serverless v2
https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-serverless-v2-auto-pause.html

Change History

Version	Date	Change
0.1	11/08/2025	Original Version