

Tutorial 6 – Introduction to Lambda III: Serverless Databases

Disclaimer: Subject to updates as corrections are found
Version 0.12
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/  
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 872ba256cbf61d9290b571c0e6d82a20c224ca3ad82971edc46b29818d5da1t1
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 wllloyd wllloyd 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/wllloyd/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**psert 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).

K

"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 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 is commented out - it is used when seeking to work with a SQLite database **saved to a disk file**.

The advantage of creating the database on disk is that data persists beyond the runtime of the Java code. On 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, if the file-based SQLite databases has not saved to S3 or other service, the data 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`

Then run the script. Below the API Gateway invocation code in BASH 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 additional server infrastructure.

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 String CONTAINER_ID = "/tmp/container-id";
    static Charset CHARSET = Charset.forName("US-ASCII");

    static int uses = 0;
```

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 is initialized above. The connection will stay open as long as the Lambda function's runtime container is preserved in the "warm" state.

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. 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. Note that serverless is in quotes. With version 2, the database, while serverless, no longer supports the ability to scale down to zero to suspend all charges. With version 2, the database can scale down to 0.5 Aurora Compute Units when not actively used. 0.5 Amazon Compute Units provides approximately 1 GB of memory and a fraction of a vCPU to host an always-on database at the prices of **6c/hour, \$1.44/day, or \$10.08/month** plus storage charges are additional. Serverless Aurora supports automatic horizontal scaling up of database servers from 0.5 ACUs to as many as 128 ACUs, where 1 ACU provides 2 GB of memory and relative number of vCPUs that may scale like the r6 family of ec2 instances. The exact value is not published, but we expect 1 ACU provides 0.25 r6.large-style vCPUs. (recent 10-nm Intel Xeon 8375C). Aurora additionally provides automatic database backups and replicas.

**** TO STOP CHARGES FOR THE "SERVERLESS" DATABASE IT IS NECESSARY TO PAUSE THE DB CLUSTER ! ****

To get started, create a serverless database!

**** This portion of the tutorial requires AWS credits to complete. **
Aurora serverless does not run in the FREE tier.**

Under Services, search for and go to "RDS".

DO NOT "Try the new Amazon RDS Multi-AZ deployment option"

This will be more expensive.

Instead, scroll down to **Create database**, and press the "Create database" button to launch the wizard:

Create database

Amazon Relational Database Service (RDS) makes it easy to set up, operate, and scale a relational database in the cloud.

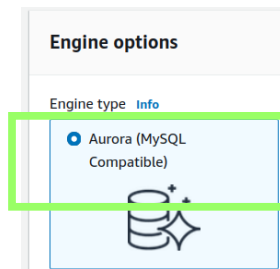
You can use a backup from Amazon S3 to restore and create a new Aurora MySQL and MySQL database.

[Create database](#) [Restore from S3](#)

Note: your DB instances will launch in the **US East (N. Virginia)** region

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

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



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

Engine version

Aurora MySQL 3.05.2 (compatible with MySQL 8.0.32) - default for major version 8.0

⚠ Parallel query is off by default. To enable it, use a DB instance parameter group with the `aurora_parallel_query` parameter enabled. [Learn more](#)

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

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

Then from the **Available versions** list, select the latest version:

Show versions that support Serverless v2
Offers instance scaling for even the most demanding workloads.

Available versions (10/20) [Info](#)

Aurora MySQL 3.05.0.1 (compatible with MySQL 8.0.32)

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

For **Settings**, specify a value for the ‘**DB cluster identifier**’, ‘**Master username**’, as well as the **Master password**.

Use the following values:

DB cluster identifier: tcss462-562
Master username: tcss462_562 (DO NOT USE A DASH – this is not accepted- underscore is okay)
Credentials management: select: “Self managed”
Master password: <provide a password, 8-char minimum>
Confirm password: <repeat the password>

*** Write this information down or memorize it. It is needed later. ***

The screenshot shows the 'Settings' section of the AWS RDS console. It includes the following fields and options:

- DB cluster identifier:** tcss462-562
- Master username:** tcss462_562
- Credentials management:** Self managed (selected)
- Auto generate password:** unchecked
- Master password:** [masked]
- Password strength:** Very strong
- Confirm master password:** [masked]

Next, select ‘Aurora Standard’ for the Cluster storage configuration.

Next, specify the ‘Instance configuration’ options.

To save cost, specify a very low range of Aurora Compute Units:

The screenshot shows the 'Instance configuration' section of the AWS RDS console. It includes the following options and fields:

- DB instance class:** Serverless v2 (selected)
- Capacity range:** Minimum ACUs: 0.5 (1 GiB), Maximum ACUs: 2 (4 GiB)

Note that running the server for 1 hour with 0.5 Aurora Capacity Unit and 1GB of memory costs 6¢. Aurora Serverless costs 12¢/per ACU/per hour billed to the nearest second. Databases are always-on. Databases scale up instantly to the maximum ACUs, and take approximately 3-minutes to scale to the minimum ACUs after a period of inactivity. To limit charges, set the minimum Aurora Capacity Units to the minimum setting of **0.5**. Set the maximum to a small number for development/test purposes. If there are performance issues the maximum can be increased later.

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 security group (firewall):

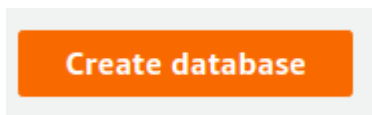
Select “Choose existing”, and then verify that “default” is specified as the “Existing VPC security groups”.

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.

| DB identifier | Role | Engine | Region & AZ | Size | Status | CPU | Current activity | Maintenance | VPC | Multi-AZ |
|------------------------|------------------|--------------|-------------|------------------------------|-----------|--------|------------------|-------------|--------------|----------|
| tcss462-562 | Regional cluster | Aurora MySQL | us-east-2 | 1 Instance | Available | - | - | none | - | - |
| tcss462-562-instance-1 | Writer instance | Aurora MySQL | us-east-2c | Serverless v2 (0.5 - 2 ACUs) | Available | 30.60% | - | none | vpc-4d30b824 | No |

5. Launch a t2.micro EC2 VM to connect to the Aurora DB MySQL database

It is not possible to directly connect to the Aurora MySQL Serverless database. This is because the database lives on a Virtual Private Cloud (VPC) that does not allow inbound traffic from the internet. This provides network isolation and security. Accessing the database requires launching an EC2 instance in the same VPC as the RDS database and associating a Public IP address with this EC2 instance. The RDS database itself does not have a public IP that is accessible from the outside. Some solutions to access the RDS database directly via the internet include: (1) setting up either a NAT Gateway (4.5¢/hour) on the VPC, (2) using an always-on VM to serve as a router/gateway between the public internet and database on the private network, (3) hosting a

proxy server such as haproxy on a publicly accessible VM which has access to the database on the private network. The proxy server can direct inbound traffic for MySQL to the RDS database, or (4) setting up an RDS proxy. For proxy solutions, database clients connect to the VM proxy or RDS proxy, not directly to your database, and the traffic is redirected. Fortunately, *for tutorial 6*, we only use AWS Lambda functions deployed in the same VPC as the RDS database, no special networking approach as described above is required to access the database. This saves cost and complexity !

Launch an EC2 as described in Tutorial 3. Refer to tutorial 3 to refresh how to do this.
Select the latest version of Ubuntu.

Specify a t2.micro, a free-tier instance.
The VM is being used as a database client.
A powerful VM is not required.
The t2.micro can also be launched as a spot-instance by specifying a spot request.
New AWS accounts receive 750 hours of free t2.micro time/month for the first year.

Provide the following settings:

Network: vpc (default) - select your default VPC -

Subnet: **** MUST match the Region and Availability Zone (AZ) of the database instance (for the screen capture above, the db instance shows as running in us-east-2c. Check your RDS console to learn the 'Region & AZ' of the serverless database.)**

Use an auto-assigned Public IP

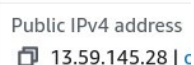
For the security group, select an existing security group, such as default, where you have already set up SSH permissions from your client computer (laptop/desktop).

Review settings and Launch

Choose an existing keypair from a prior tutorial if available.
Otherwise, create a new keypair.

In the EC2 console, select the VM. If you have not enabled SSH access from your network, select the new VM, and click on "default" for Security groups. Click the "Inbound" Tab, and hit the [Edit] button. Click [Add Rule] and add a "SSH" "TCP" "22" rule for "My IP". This should allow SSH access to the t2.micro instance.
Refer to tutorial 3 to review the procedure.

Next in the EC2 console copy the public IP:
Use the COPY icon on the left-hand side to copy the IP address:



Public IPv4 address
13.59.145.28 | c

Now, using the command line, navigate to the folder where the keypair is stored, and ssh into the newly created t2.micro VM. Paste the address and SSH:

```
$ ssh -i <your key file name> ubuntu@<Public IPv4 address of VM>
```

Now from the Ubuntu t2.micro instance launched in the same VPC as the RDS database, optionally, install the AWSCLI if wanting to work with AWS directly from the VM. **It is not required.**

```
# install the AWSCLI - THIS STEP IS OPTIONAL
sudo apt update
sudo apt install awscli
aws configure
# provide ACCESS_KEY and SECRET_KEY
# Find your credentials on your existing VM with: "cat ~/.aws/credentials"
```

Next, install the mysql client to support connecting to the new RDS database:

```
# Install mysql client

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 "tcss462-562-instance-1".

It is necessary to configure the security group to allow the t2.micro to connect to the database. In the Connectivity & security tab, look under "Security" on the right:

Security

VPC security groups

default (sg-48e2ad21)
(active)

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].

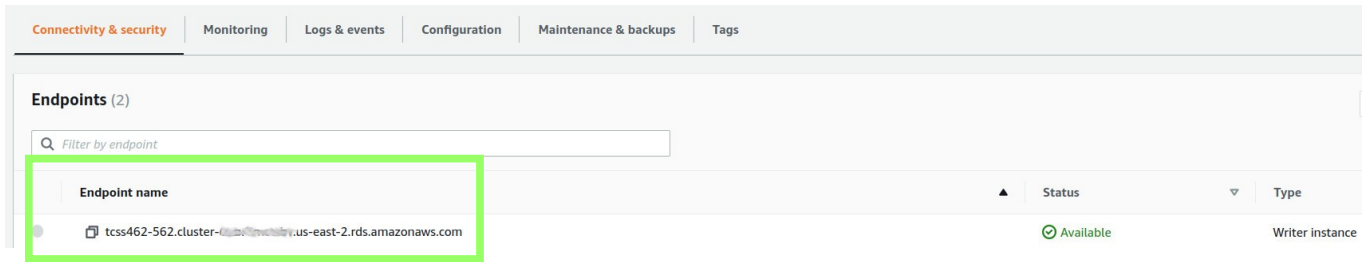
Now copy the database endpoint name which can be found by browsing the Aurora database in the Amazon RDS GUI.

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 customize the mysql command to connect to your RDS database.
Replace <Database endpoint> and <your database password>.

```
mysql --host=<Database endpoint> --port=3306 --enable-cleartext-plugin --  
user=tcss462_562 --password=<your database password>
```

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

```
mysql: [Warning] Using a password on the command line interface can be insecure.  
Welcome to the MySQL monitor.  Commands end with ; or \g.  
Your MySQL connection id is 100  
Server version: 8.0.32 Source distribution  
  
Copyright (c) 2000, 2024, Oracle and/or its affiliates.  
  
Oracle is a registered trademark of Oracle Corporation and/or its  
affiliates. Other names may be trademarks of their respective  
owners.  
  
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.  
mysql>
```

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
```

It may be useful to “stop” and “start” your t2.micro ec2 instance that has command-line access to the Amazon RDS database to support working with mysql. If no longer planning to use the ec2instance, **terminate it completely**. Note that “stopped” instances incur storage charges. New AWS accounts receive 30GB of EBS disk space for 1 year for free. After 1 year, the charge is 10¢/GB/month (gp2) or 8¢/GB/month (gp3). The Ubuntu t2.micro requires 8GB of storage. The annual storage cost after the free introductory year is ~\$9.60/year for a stopped t2.micro instance with 1 x 8GB EBS volume. After 1 year, a running t2.micro instance is no longer FREE, but the cost rises to 1.16¢/hour, ~27.8¢/day, ~\$8.35/month, or ~\$101.62/year for GP2, and slightly less for GP3.

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.

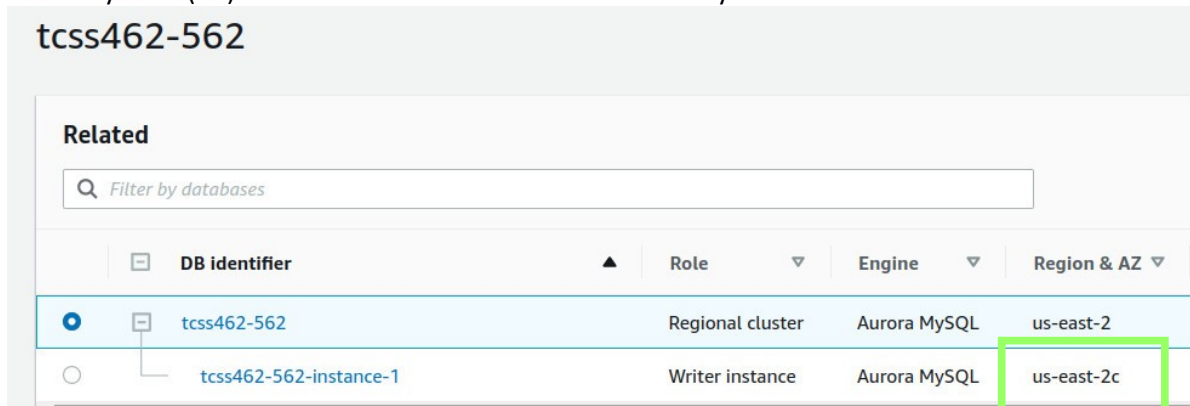
Match the availability zone of your database with your serverless function.

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.

The screenshot shows the 'Basic settings' tab for an AWS Lambda function. It includes a 'Description - optional' field, a 'Memory' field set to 512 MB, and a 'Timeout' field set to 3 minutes and 0 seconds. The 'Timeout' field is highlighted with a green border.

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).

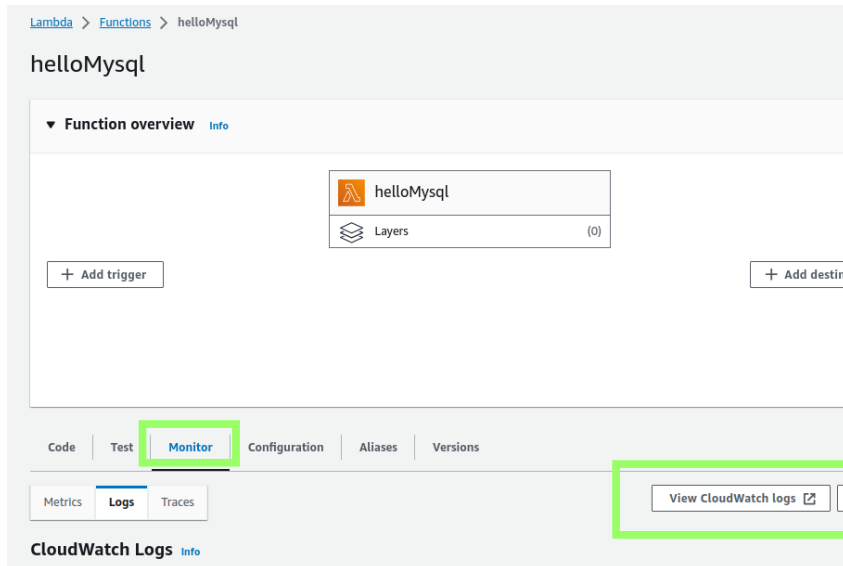
Each call to `HelloMySQL` will append a row to the table with the provided name.

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.

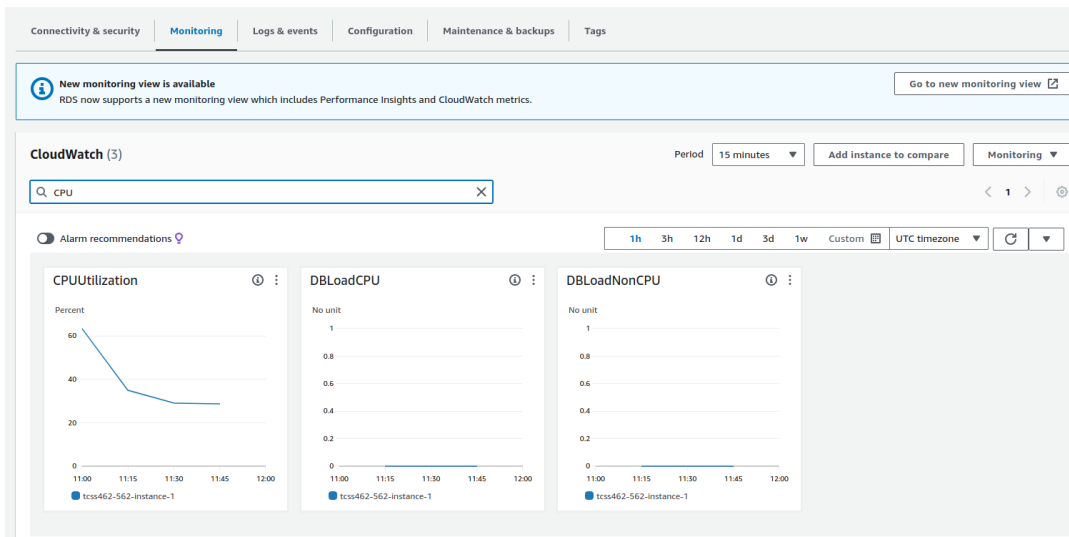
Now, using callservice.sh, run the service. Capture the complete output from the script using a terminal and provide this output as the answer for #8.

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:

| Recent events (4) | |
|--|---------------------------------------|
| <input type="text" value="Filter by db events"/> | |
| Time | System notes |
| November 03, 2022, 02:55 (UTC-07:00) | DB instance created |
| November 03, 2022, 02:57 (UTC-07:00) | Monitoring Interval changed to 60 |
| November 03, 2022, 02:57 (UTC-07:00) | Performance Insights has been enabled |
| November 03, 2022, 02:58 (UTC-07:00) | Finished updating DB parameter group |

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

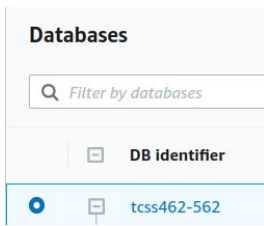


7. Stopping the database for up to 7 days

Amazon RDS Aurora Serverless v2 databases have **always-on charges !!!**
 Databases configured to have 0.5 ACUs will cost 6c/hour, \$1.44/day, \$10.08/week, or \$43.80/month if kept running with 0.5 ACUs. **** THIS IS VERY EXPENSIVE AND WILL DRAIN YOUR CREDITS FAST ** !!!**

“Serverless” databases can be paused for up to 7 days to STOP always-on charges. This allows the database to remain inactive in the account where only data storage charges apply. **WARNING !!! ** After 7 days, the database will automatically restart and begin create charges !!!**

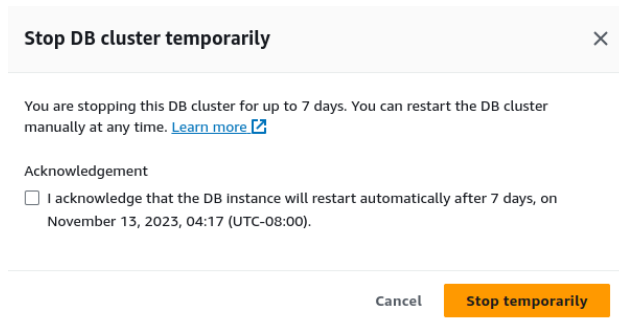
Now, practice temporarily stopping the database cluster. From, RDS, select your database cluster, not the instance, but the whole cluster.



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, however, is to **DELETE THE DATABASE**, and recreate the database later after a week. An alternative, when working with large databases that must be pre-loaded with data, database backups can be used to more quickly restore the database from scratch with schemas and data.

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 starts charging the user up to \$43.80/month at 0.5 ACUs ?

b. What is the best practice if an Amazon Aurora serverless database must be stopped for a very long time greater than Amazon’s allowed temporary stop duration?

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 Serverless v2 database:

<https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-serverless-v2.html>

Blog Article: No, AWS, Aurora Serverless v2 Is Not Serverless:

<https://www.lastweekinaws.com/blog/no-aws-aurora-serverless-v2-is-not-serverless/>

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>

Note

Currently, Aurora Serverless v2 writers and readers don't scale all the way down to zero ACUs. Idle Aurora Serverless v2 writers and readers can scale down to the minimum ACU value that you specified for the cluster.

That behavior is different than Aurora Serverless v1, which can pause after a period of idleness, but then takes some time to resume when you open a new connection. When your DB cluster with Aurora Serverless v2 capacity isn't needed for some time, you can stop and start clusters as with provisioned DB clusters. For details about stopping and starting clusters, see [Stopping and starting an Amazon Aurora DB cluster](#).

Is scaling to zero a feature on the Aurora Serverless v2 roadmap ???

We are still waiting...