AMAZON AURORA SERVERLESS SCALING DRIVEN BY APPLICATION METRICS

Amazon CloudWatch | AWS Lambda | Serverless

beSharp | 18 October 2019

Nowadays cloud architectures are more **serverless** than ever before.

We deploy our products using **CloudFormation templates** and **deployment scripts**, and the ideal architecture has **no servers to maintain.** We leverage **serverless storage like Amazon S3**, **serverless computing power such as Amazon Lambda**, why not using a serverless RDBMS?

In August 2018 AWS released **Aurora Serverless**, a **serverless relational DBMS** to finally make applications that require a relational database fully serverless.

Aurora Serverless promises to make the database **autoscaling** possible and efficient, but is it really a silver bullet?

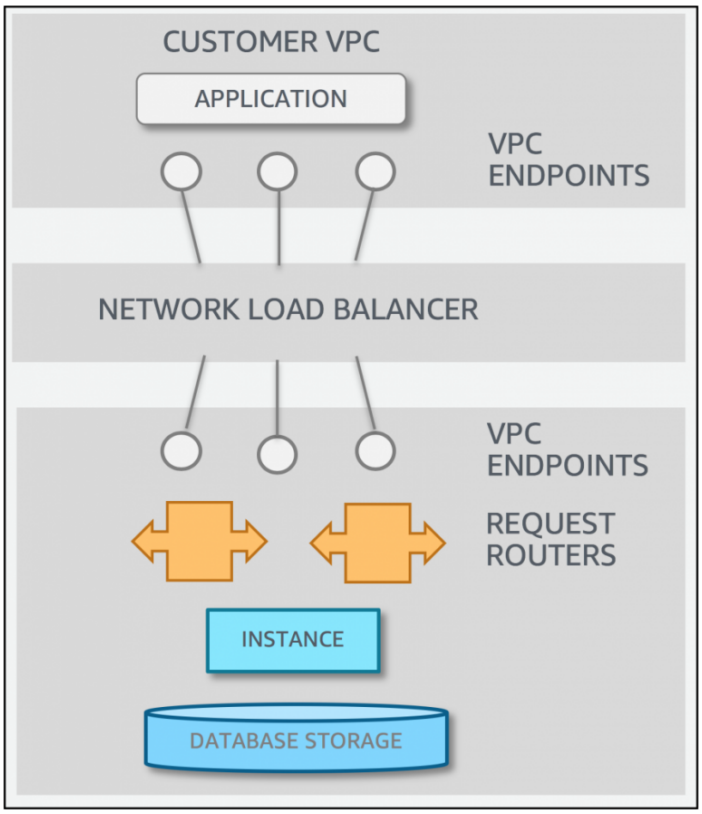
To answer this question we need to deep dive into the service operation.

Let's start with the basics.

What is Aurora Serverless?

Amazon Aurora Serverless is an **on-demand, auto-scaling configuration for Amazon Aurora** (MySQL-compatible and PostgreSQL-compatible editions), where the database will automatically start up, shut down, and scale capacity up or down based on your application's needs. It enables you to run your database in the cloud without managing any database instances. It's a simple, cost-effective option for **infrequent, intermittent, or unpredictable workloads.**

How does it scale automatically?



Aurora Serverless can scale automatically using **AWS provided metrics,** so you are not required to design and maintain a suitable **autoscaling policy**. The service creates an Aurora storage volume replicated across multiple AZs to provide the auto-scaling capability. It creates an **endpoint in your VPC** for the application to connect to and it configures a N**etwork Load Balancer** (invisible to the customer) behind that endpoint. Aurora Serverless also configures multi-tenant request routers to route database traffic to the underlying instances.

When the cluster needs to autoscale up or down or resume after a pause, Aurora Serverless grabs capacity from **a pool of already available nodes** and adds them to the request routers. **This process takes almost no time** and since the storage is shared between nodes Aurora can scale up or down in seconds for most workloads.

The cluster scales up when capacity constraints are seen in **CPU**, **connections, or memory**. It also scales up when it detects **performance issues** that can be resolved by scaling up.

After scaling up, **the cooldown period** for scaling down is 15 minutes, and after scaling down, the cooldown period for scaling down again is 310 seconds.

This is a completely automated process managed by AWS itself, you can only set **a range of ACUs** (minimum and maximum values) as boundaries during the autoscaling action.

By default, the autoscaling mechanism described above works only when a **scaling point** is available.

A scaling point is a **point in time at which the database can safely initiate the scaling operation.** Under specific conditions, Aurora Serverless might not be able to find a scaling point. For example

when Long-running queries or transactions are in progress, or when temporary tables or table locks are in use.

In these circumstances, Aurora Serverless continues to try to find a scaling point so that it can initiate the scaling operation. It does this for as long as it determines that the DB cluster should be scaled.

You can also choose to apply capacity changes even when a scaling point is not found. **If you opt to forcibly apply capacity changes, active connections to your database may get dropped.**

This configuration could be used to more readily scale the capacity of your Aurora Serverless clusters if your application is resilient to connection drops.

However, while the default behavior may cover a lot of use cases, there are specific scenarios in which Aurora Serverless will not be able to scale properly using only the AWS managed auto-scaling.

In those situations, the DevOps can aid the autoscaling process by leveraging insights from the application.

How to aid the autoscaling of Aurora Serverless

If your application operates in a corner case, and the autoscaling isn't performing good enough, you may want to consider some way to help the mechanism.

In practice, you can **increase the cluster "desired ACUs" parameter.** The autoscaling will then continue to operate normally, scaling the cluster up or down as needed.

An example of a quite common corner case is when an application needs to face sudden and sharp traffic spikes; in this case, especially if the allocated ACUs are very low, the autoscaling will not be able to react fast enough causing some of the traffic to experience errors.

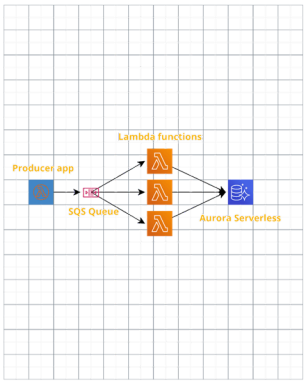
If the spikes are predictable, the Aurora Serverless cluster can be prewarmed to achieve optimal performances and avoid errors; the scaling operation may be initiated manually or in an automated/scheduled manner.

In other situations, the application may be aware of an incoming traffic spike, and so, it can take action to change the "Desired ACUs" of the DB cluster.

If the application has access to **metrics** that can forecast the need for a scaling operation, then you can **collect and push them out as CloudWatch custom metrics**. You can then use the metrics to setup **CloudWatch alarms** to trigger a **Lambda function.** In the Lambda function, you can implement your **custom scaling logic.**

A real-world scenario

We would like to end the article with a real-world example.



In this project, we had to make **a system to elaborate and store data in an RDBMS.** The starting point of the data flow is a producer app.

We leveraged **Amazon SQS** to enqueue the raw data and trigger a fleet of Lambda function used to elaborate it and then save the results on Aurora Serverless.

Since the process is human triggered, the cluster would remain almost with no traffic until someone starts a new analysis on the producer application. When the process started a huge number of **Lambda functions** reading the raw data and connecting to the cluster to save the result were started. Anyway, the sudden peak caused Aurora Serverless to scale up, but in the meantime, a lot of lambdas failed due to connection errors.

To solve the problem, we used **the producer application to scale the DB.** We also configured the **SQS visibility delay** of each message to allow the cluster to scale before flooding it with connections.

In the end, the solution turned out to be very performing: the autoscaling scales up properly after the external intervention and it scales down to the minimum at the end of the task.

With Aurora Serverless AWS succeded in covering the last gap remained in creating Serverless application: it finally provided us with a Serverless RDMS.

Now it’s time to have a try!
Let us know what do you think about this service!

See you, guys!

beSharp

Dal 2011 beSharp guida le aziende italiane sul Cloud. Dalla piccola impresa alla grande multinazionale, dal manifatturiero al terziario avanzato, aiutiamo le realtà più all'avanguardia a realizzare progetti innovativi in campo IT.

Get in touch

beSharp.it
proud2becloud@besharp.it