Serverless Application Using Lambda and DynamoDB with Auto Scaling

Geetika Rathi
Master of Applied Computer Science
Concordia University
Montreal, Canada
geetika.rathi94@gmail.com
40206213

Harmanpreet Kaur
Master of Applied Computer Science
Concordia University
Montreal, Canada
harmanpreetk08@gmail.com
40198317

Shubhang Khattar
Master of Applied Computer Science
Concordia University
Montreal, Canada
khattarshubhang@gmail.com
40163063

Smit Parmar
Master of Applied Computer Science
Concordia University
Montreal, Canada
smitparmar011999@gmail.com
40163999

***Abstract*—Given the convergence of dense hardware and the global cloud, we set out to build a system that can function effectively at any scale, offering outstanding performance on a single multi-core system while scaling up elastically to Geo-distributed cloud deployment. In order to meet a variety of application needs, we also intended to offer broad architectural flexibility and broad consistency semantics.**

We used DynamoDB as a NO-SQdatabase, lambda for middleware operations, and react for the front end for this. Due to the use of lambda, serverless computing is possible because no dedicated server is required. Our software will obtain the product ID and display the top 100 customer reviews for a certain item. We preprocessed the dataset using a lambda function after uploading it to S3. We then performed an entire data migration using the same lambda code. Lambda can autoscale, has a 15-minute timeout, and will always be accessible because there are replicas running in several countries. Another form of scaling used by AWS is the creation of new instances to execute queries every 700–1000 RU. 2000 WRU (Read Write Units) or so were used throughout the migration. As soon as the data was uploaded, we created a web application using the React framework, which could take the product ID and call the AWS Lambda API to retrieve the results. Lambda can scale vertically, which means it could grow its resource capacity. AWS assigns a new instance/query engine for every 700–1000 request units, therefore we have assigned 3000 read request units in DynamoDB, which is referred to as horizontal scaling. Our main database is in the North Virginia region, and we have created one database replica in a different availability zone (Canada-Central).

I. Introduction

Innovations in cloud computing have changed how we create large-scale applications over the last ten years. Hardware virtualization technology is used by cloud service providers like Amazon Web Services (AWS), Google Cloud, and Microsoft Azure to give application developers access to virtual machines (VMs) with CPUs, RAM, and storage. The computing environment has been completely transformed as a result, enabling enterprises without prior access to big computer clusters to create and grow distributed applications. Serverless computing solutions have emerged as cloud technologies have advanced further, taking cloud computing to the next level.There are two main benefits to the new software design pattern known as serverless computing. It starts by providing a higher-level abstraction for cloud programming. Function-as-a-service (FaaS) platforms, like AWS Lambda, enable programmers to create application code locally and upload it to the platform without worrying about resource configurations or the environment in which the applications are run. The application code is then served via FaaS platforms in response to several trigger events. For instance, AWS Lambda supports triggers from API Gateway, S3, and DynamoDB events.The second major benefit of serverless is usage-based pricing; FaaS platforms like AWS Lambda charge for each function invocation based on computing time, with millisecond-level accuracy. Like this, serverless storage systems like AWS DynamoDB charge users according to how much space their datasets require. Both cloud service providers and developers are drawn to usage-based pricing.

II. Problem Statement

For serverless systems, achieving good performance at any scale is essential. To demonstrate distributed system features in the serverless application, such as autoscaling, replication, fault tolerance, data partitioning, and consistency we have developed a system such that we expected from the beginning that we would need to partition (shard) the key space across cores for high performance as well as between nodes at cloud scale to assure data scaling. Also we made use of multi-master replication to simultaneously serve puts and gets against a single key from many threads to enable workload scaling.For this, we used a NO-SQL database called DynamoDB, lambda for middleware operations, and react for the front end. Lambda is used to provide serverless computing since it eliminates the need for dedicated servers.

III. Literature Review

There are already enough solutions available through various vendors that enable us to achieve our goal. Firstly, we were challenged with determining the type of database for our application. There are broadly two categories of database systems available: SQL and NoSQL[7]. Although both databases are excellent choices, they have some significant distinctions. Structured Query Language (SQL) is used in SQL-based databases to define and manipulate the database. Especially for large complex queries, SQL is one of the most robust and widely used, making SQL-based databases a safe choice. On the other hand, it might be limiting. Before working with your data, SQL requires you to use predefined schemas to establish its structure. Also, all the data must have a consistent structure. It makes it challenging to do modifications to the existing database if required.

Whereas a No-SQL database implements a dynamic schema for unstructured data. Data can be stored in many ways, i.e., it can be document-oriented, column-oriented, graph-based, or organized as a Key-value store. Because of this, flexibility is provided to database, and documents can be written without a predetermined structure. Additionally, each document may have a different structure. Also, the fields can be added on the go. SQL databases are vertically scalable in nearly all circumstances. This means that you may increase the demand on a single server by boosting components like RAM, CPU, or SSD. NoSQL databases, however, are horizontally scalable. You can handle significant traffic by sharding or adding multiple servers to your NoSQL database. As a result, NoSQL can grow larger and more powerful, making these databases the best option for big or constantly changing data sets. Considering the scalability and distributed benefits of the No-SQL databases, we decided to go forward with it.

We have various No-SQL databases solutions available, the most common being MongoDB and DynamoDB[1]. MongoDB database came into light around the mid-2000s. MongoDB provides a cloud-based service called MongoDB Atlas; it complies with all regulatory requirements and security standards and can be installed on any significant cloud provider with guaranteed availability and scalability. Amazon's DynamoDB is a proprietary NoSQL database that handles key-value and document data made available through the AWS. This AWS offers a managed DynampDB database platform that is scalable, highly available, and secure for any application.

Both MongoDB and DynamoDB offer similar features and capabilities. However, there are a few significant differences that must be considered while choosing between the two. The most significant difference that affected our decision of choosing the desired database is that MongoDB is platform-agnostic whereas, DynamoDB is limited to AWS.

DynamoDB can be configured solely through AWS. As a native AWS service, DynamoDB provides much better integration with other services and tools provided by AWS, which will be beneficial for our project application. Although DynamoDB offers a downloadable version for testing and deployment, the deployment for production depends only on AWS. In contrast, MongoDB can be configured to run virtually anywhere, be it the local machine, container, or production deployment on any cloud provider, including AWS.

Because AWS will take care of all the scaling, availability, and updates for DynamoDB, users may begin utilizing the database immediately. With only a few clicks, it makes it possible to provision a multi-region, highly available database, significantly minimizing the requirement for specialized infrastructure management. Whereas for the MongoDB deployment, users must manage all the infrastructure and configurations. This gives users the most significant degree of control over the database but adds to the increased complexity. As for our project, we will be using the AWS ecosystem to deploy the application; DynamoDB inherently offers Compatibility, Ease of use, and more straightforward Integrations with AWS applications.

To implement the backend, we had two choices under the AWS ecosystem, i.e., EC2 (Amazon Web Services Elastic Compute Cloud) and lambda. AWS EC2 is a service that enables scalability and virtual machines in the cloud known as EC2 instances. We can alter the resource such as disk space, CPU performance, and memory on the go. While setting up the EC2 instance, we must provide a base image with the necessary pre-installed OS and the required applications. AWS provides the root access for the EC2 instances, enabling us to create additional users if required. We can fully control the EC2 instances, including rebooting and shutting down the instance if required.

AWS Lambda is a computing platform that allows you to run a piece of code written in one of the supported programming languages such as Java, JavaScript, node.js, python[2][3], and many more. A trigger is linked to an event that is when fired; the lambda function is executed. Unlike EC2 instances, we don't need to configure any virtual image and environment to run an application. Simply enter your program code into the AWS Lambda interface, link the Lambda function to the event, and run the application in the cloud as needed without worrying about server administration or environment setup. This way, the user just needs to focus on their application and not on the server management, making AWS lambda serverless.

Various events can trigger the lambda functions, such as uploading the file to the Amazon S3 bucket, manipulating the DynamoDB tables, getting an HTTP request to the API gateway service, and many others. Once the event occurs, the lambda function will be executed automatically. In the AWS lambda function, your application runs in a container that is seamless. Code and libraries are contained in the container. Amazon supplies resources in accordance with the requirements of the application, and scaling is seamless and automatic. The life cycle of each container is transient. The Lambda function doesn't store the state, and if the user wants to store some results, EC2 should be used. As AWS Lambda provides a serverless architecture, enables us to focus on code only, and is scalable and inexpensive, we decided to use AWS Lambda for our backend instead of an EC2 instance.

IV. Dataset

For dataset, we have used Amazon US Customers Review Dataset[5] for Electronics which is approximately 2 GBs in size and contains more than 3 millions records. This dataset includes the data from year 1995 to 2015 which contains over a hundred million reviews by m illions of Amazon customers describing their experiences and opinions regarding products on the website. We have selected this dataset as it is a rich source of information that can be applied to many different areas, including Natural Language Processing (NLP), Information Retrieval (IR), and Machine Learning (ML), etc. This dataset is used to represent a customer evaluations and their opinions, variations in how a product is perceived across different geographic areas, and if reviews are biased or motivated by promotion. The name of the dataset is $amazon\_reviews\_us\_Electronics\_v1\_00.tsv$[5] which is a TSV (Tab Separated Values) File which consists of the following columns :

1) marketplace - It is a two letter country code where the review was written.
2) $customer\_id$ - Identifier that can be used to represent reviews written by a single user.
3) $review\_id$ - The unique ID of the review.
4) $product\_id$ - The unique Product ID the review pertains to.
5) $product\_parent$ - Identifier that can be used to gather reviews for the same product.
6) $product_title$ - Title of the product.
7) $product\_category$ - For our dataset, this value will be set to 'Electronics'
8) $star\_rating$ - Rating of the review from 1 to 5
9) $helpful\_votes$ - Number of helpful votes.
10) $total\_votes$ - Total number of votes of the review
11) $review\_headline$ - The title of the review.
12) $review\_body$ - The detailed description of the review given by the user.
13) $review\_date$ - The date on which the review was written.

V. Technology

A. Amazon DynamoDB

Amazon DynamoDB is a NoSQL[7] database service with different type of keys where primary key is hashed and store indexes. DynamoDB provides very fast performance with features such as auto-scaling. DynamoDB Streams auto-scales the number of shards based on traffic. It also handles the administrative tasks of scaling and operating the distributed systems automatically such as provisioning of hardware, replication, set up and configuration, software patching and cluster scaling.
DynamoDB helps in creating the database tables which can store and retrieve large amount of data as well as serve huge levels of request traffic. The tables' throughput capacity can be scaled up or scaled down without any downtime or degrading the performance. Moreover, it uses Amazon Web Services Management Console for monitoring the resource utilization. It handles the throughput, fast performance and storage requirements by automatically spreading the data and traffic for tables over multiple servers. Solid State Disks (SSDs) are used to store the data, which is then automatically replicated among a number of Availability Zones in the Amazon Web Services region to provide high data availability and durability.
With DynamoDB, as there are no servers so there is no requirement of provisioning , patching, managing or installing any softwares. DynamoDB ensures performance with zero administration while automatically scaling tables to account for capacity.
In order to process reads and writes on the tables, Amazon DynamoDB offers two read/write capacity modes, On-demand and Provisioned. The read/write capacity mode manages capacity and how to charge for read and write throughput. Other important features include :

1) Auto-Scaling and Performance
2) Access to Control Rules
3) Automatic Data Management
4) Storage of inconsistent schema items

B. AWS Lambda

AWS Lambda is a serverless computing service which is provided by Amazon Web Services (AWS). Users build functions, self-contained applications written in any supported languages and runtimes, which are then uploaded to AWS Lambda, that executes those functions in an adaptable and effective manner.

Any type of computing operations can be carried out via Lambda functions, including serving web pages, processing data streams, calling APIs[2], and interacting with other AWS services.

"Serverless" computing refers to the idea of not needing to manage your own servers in order to perform certain tasks. AWS Lambda service will take care of all the infrastructure needs. Therefore, "serverless" does not imply that there are no servers involved; rather, it simply denotes that the infrastructure, including the servers, operating systems, network layer, and other components, has already been taken care of, allowing you to concentrate on developing application code.

AWS Lambda can execute multiple instances of same functions or different functions concurrently. It is simple to integrate with data sources like Amazon DynamoDB and trigger Lambda functions for particular types of data events. Other important features of AWS Lambda includes :

1) Load Balancing
2) Auto Scaling
3) Handle Failures
4) Security Isolation
5) OS management

VI. Implementation

The project was executed using the AWS ecosystem. To migrate the data to DynamoDB from the provided TSV file, we have implemented a streamlined approach to this problem by leveraging the AWS Lambda services. To achieve the same, we first push TSV data into an S3 bucket. Then we pushed the data to S3 bucket, which can be done either programmatically or via your AWS account console; we preferred the latter. We created an IAM Role, having the proper permissions to access your DynamoDB tables and S3 bucket. With the right IAM policy applied, the Lambda function will programmatically access the contents of your bucket.

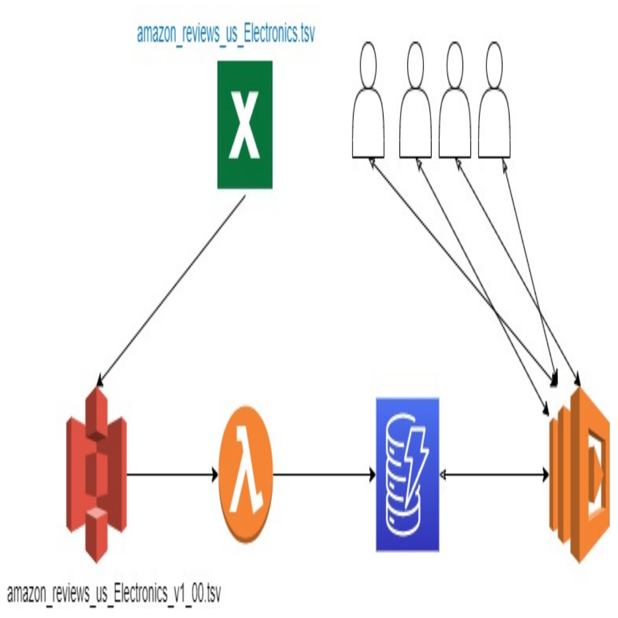
amazon_reviews_us_Electronics.tsv
amazon_reviews_us_Electronics_v1_00.tsv

Fig. 1. Workflow

A. *S3 To DyanmoDB with The Help Of Lambda*

We can now develop a Lambda function to fetch data from S3 bucket and ingest it into our DynamoDB table as our data set and required permissions are prepared. Attaching the IAM role we defined in the previous stage to our Lambda function in the permissions section would be the crucial step in this process. This would enable our lambda function code's programmatic access to S3 and DynamoDB. With this, we are done with the required configurations for migrating data to DynamoDB. Now, for the code part, we have implemented the program in Python, utilizing libraries such as pandas in the Lambda function. We created a boto3 session and used it to launch an S3 client in our Python code. Next, we implemented code to read the TSV file from our S3 bucket and return the pandas data frame created from it. We call get-object on our S3 client, passing the arguments bucket and TSV filename. The TSV file contents would be fetched into the body-field of the response, which is afterward read into a pandas dataframe. Next, we can begin creating the DynamoDB table into which we will be ingesting the data. Providing parameters such as TableName and KeySchema information are crucial when creating the DynamoDB table. For our project, we have used KeySchema-AttributeName 'ReviewID'. After initializing and configuring our S3 and DynamoDB clients, the next step would be to begin to read the dataframe row by row and write it into the table. Each row of the dataframe is initialized as an item and ingested into the DynamoDB table using the initialized batch writer on the table object. With this approach, we successfully migrated all the data to DynamoDB. For Lambda timeout of 15 minutes, it can autoscale and will be available as there will be replicas across different regions. We have used approx 2000 WRU (Read Write Units) while migrating.[6]

Now, we create a Lambda function for the back-end of our API. This Lambda Function will allow our clients to read amazon item reviews for a given product id from DynamoDB. The Lambda function uses events from API Gateway to interact with DynamoDB to read data. We have implemented a single lambda function for GET operation capable of scaling horizontally as the number of clients increases. Lambda function can also scale vertically and increases the resources if required. The back-end lambda function is implemented in the node.js programming language. At the back-end Lambda function to get the required product-id information, ddbDocumentClient object is created on which the call is made to the database by providing the required parameters such as TableName, FilterExpression, ExpressionAttributeNames, ExpressionAttributeValues. To optimize the performance of our project, if a product has multiple reviews, the query will execute for only a maximum of 100 reviews per product. Also, if there are no reviews for a product-id instead of searching for the whole database, the program will execute for only 1000 WRU. The reviews collected are packed to the return request body for the front end.

B. *Reading Data With Serverless Application*

Next, we added an API Gateway Trigger to our AWS Lambda Function. As our clients will be retreiving the data, we need to create a GET method for API Gateway configuration. From this, we get a URL, which will be used for the 'axios' GET request to retrieve data from DynamoDB. Now for the front-end, we implemented create-react-app. In the front-end code, we specify API Gateway we configured earlier. The front end makes the GET call to the lambda endpoint with the product-id in the get request body. The front-end application uses 'axios' package to make the call and wait for the result. In addition, we have made one replica of the database in different availability zone(canada-central) and the main database in the North Virginia region.[4]

VII. TeamWork

We divided tasks based on pair programming. That means we divided out team in 2 parts. Harmanpreet Kaur and Geetika Rathi handled data migration part. They have accomplished this task by using s3 bucket and lambda function written in python. They faced some difficulties with timeout which

resolved later by referring to AWS documentation for lambda and DyanamoDB.
Second part, which we referred as data reading with serverless application is done by Shubhang Khattar and Smit Parmar. In this, both tried to implement frontend using react and developer lambda API endpoint in node.js. There were certain problems as request size limit and all. But we used pagination to overcome this problem

VIII. Conclusion

During this project, we explored the distributed properties of several serverless functionalities of AWS such as AWS DynamoDB and lambda. We went through several challenges.The first challenge we faced was data migration to dynamo DB. For that, we tried AWS lambda and configured Read and Write capacity units accordingly. Thus, the first half was about writing data and next second half was about reading data. In that part, by using Lambda API endpoint, we implemented a react base serverless web application that can read the dynamo DB data just not from key but any column in the best possible manner. Here, dynamo DB was making several partition based on hashed key value and it will increase the speed of scanning.

IX. Future Work

We are planning to make it more secure by using API gateway and restricted endpoint. Additionally, we are planning to try out distributed functionality of DynamoDB such as fault tolerance and coordination with across several regions.

References

[1] bmc.com. mongodb vs dynamodb. https://www.serverless.com/aws-lambda.

[2] AWS developer Guide for Node.js. Building lambda functions with node.js. https://docs.aws.amazon.com/lambda/latest/dg/lambda-nodejs.html.

[3] AWS developer Guide for Python. Building lambda functions with python. https://docs.aws.amazon.com/lambda/latest/dg/lambda-python.html.

[4] Guli Kholmatova. Building a serverless react app using aws lambda, dynamodb, and an api gateway. https://medium.com/@gulikholmatova/building-a-serverless-react-app-using-aws-lambda-dynamodb-and-an-api-gateway-f846696f34cd.

[5] Cynthia Rempel. Amazon us customer reviews dataset. https://www.kaggle.com/datasets/cynthiarempel/amazon-us-customer-reviews-dataset?select=amazon$_reviews_us_Books_v1_02.tsv$.

[6] Sathvik Sanagavarapu. Bulk data ingestion from s3 into dynamodb via aws lambda. https://medium.com/analytics-vidhya/bulk-data-ingestion-from-s3-into-dynamodb-via-aws-lambda-b5bdc30bd5cd.

[7] Aayush Sharma. Difference between sql and nosql. https://www.geeksforgeeks.org/difference-between-sql-and-nosql/.