An Experimental Key-Value Database Using Serverless Cloud Functions

Austin Patello
Rochester Institute of Technology
ajp1708@rit.edu

Beshani Weralupitiya
Rochester Institute of Technology
bw4052@rit.edu

Chung-An Huang
Rochester Institute of Technology
ch1949@rit.edu

1 ABSTRACT

This project explores the application of AWS Lambda functions to simulate a NoSQL database system in a cloud environment. The study aims to assess the feasibility of using AWS Lambda to construct a database system and compare its effectiveness with cloud databases. In this paper, we present a novel serverless architecture that is capable of serving simple Create, Read, Update, and Delete (CRUD) requests. Particularly, the create function features linear insertion and auto-increment ID generation, while the remaining three operations can carry out parallel data processing on table partitions. The project benchmarks the proposed system against DynamoDB, evaluating performance and efficiency. The results reveal that our system performed less than ideal given the challenges and limitations of serverless database systems and the AWS Free Tier setup for each of the CRUD operations.

2 INTRODUCTION

In today's technology-driven world, cloud computing has rapidly emerged in the applications of storing, managing, and accessing data. Most organizations are considering cloud services as promising alternatives, as traditional databases often struggle to meet the dynamic demands of modern applications that require rapid scalability, agility, and cost-effectiveness. The capacity of cloud function services like AWS Lambda or Azure Functions to execute precise, small-scale tasks positions them as an intuitive option for query processing [12].

This project mainly focuses on exploring the application of AWS Lambda functions to simulate a database system in a cloud environment. AWS Lambda, a fundamental component of Amazon Web Services (AWS), provides a serverless compute service that enables users to run code in response to various events without the need to provision or manage servers [8]. This study aims to assess the viability of utilizing AWS Lambda functions to construct a database system that utilizes the capabilities of cloud computing. A review of its effectiveness compared to cloud databases will also be performed. Moreover, this project specifically focuses on NoSQL databases due to their scalability, flexibility, schemaless nature, high performance, and suitability for high-availability, fault-tolerant cloud environments. This comparative analysis aims to unravel the strengths and limitations of AWS Lambda in meeting the demands posed by contemporary applications.

2.1 Objectives

This study mainly aims at an in-depth exploration of the feasibility of the AWS lambda function to build a serverless key-value database. Below are further objectives that this study aims to achieve throughout the course. Each will facilitate a comprehensive analysis of developing a serverless database system compared to NoSQL database systems.

- Assess the feasibility of a serverless key-value database running solely on event-driven functions on the cloud along with its strengths and weaknesses.
- Benchmarking the proposed system with DocumentDB [1] and DynamoDB [2] with regards to efficiency, performance.

2.2 Background

With the growing use of cloud technology, there has been a shift in the usage of traditional servers to those on cloud-based platforms. These provide the benefit of being able to eliminate up-front infrastructure costs, as well as the ability to scale resources based on the workloads being received by an application [9]. This shift to the cloud is especially prevalent for databases. Since cloud services are able to easily scale, distributed database systems are able to add more nodes depending on how much data is coming in, which, in the modern age of computing, is essential for accommodating the alarming growth in data generation.

Looking further into the cloud computing space, serverless architecture also has been a significant advancement. There are many ways this can be implemented, but as mentioned in the introduction, we will be using Lambda functions. One such project to utilize these functions is called Starling [12]. Starling is a query engine built to provide ad-hoc query processing, which is a model that allows users to only pay for resources the user actually uses. While Starling is a good first step, it was built with the mentality that it was going to be used for data analytics. Our proposed system is intended to be used as the primary database.

2.3 Related Work

This project is inspired by the efforts of various past research work, with the most notable being the Starling system built on top of AWS Lambda [12]. While this project initiative shares conceptual ground with Starling, nuanced differentiations exist, establishing our project's unique contributions in the domain of serverless database systems. Mainly, the Starling system is targeted for analytics workloads, whereas the primary goal of this project is to experiment with the feasibility of a general-purpose database built on the cloud. Furthermore, our system does not aim to serve complex queries. Our goal of this experimental attempt is to maximize performance for basic queries and deliberately exclude the support for any form of nested query structures to be executed against the query resolver. Distinguishing our project further, we opt not to employ compression or columnar formats in contrast to the design choices made by Starling. This will reduce the run-time at function executions and minimize the conversion costs as much as possible.

An aspect not covered in the Starling literature is the benchmarking against popular cloud-based key-value stores such as DynamoDB and DocumentDB. In our project, we intend to bridge this gap by providing comprehensive benchmark results comparing the performance, efficiency, and cost implications of our serverless database system against these widely utilized options.

Despite these differences, the work done by Starling has laid the foundation for several components of our architecture. Beyond that, other similar papers have offered us valuable insights into the different approaches to building database engines on the cloud. The earlier attempt from MIT CSAIL [10] shows the viability of constructing B+ Tree Indexes on S3. In addition, the checkpointing and logging techniques suggest the possibility of preserving transaction-level consistency guarantees in a completely stateless cloud setting. The researchers from ETH Zurich have implemented a similar query engine [11] that outperforms existing query-as-a-service systems. The paper points to the potential performance gains that could yield from optimizing inter-function communications and input data size. Further, it examines the effects of incorporating columnar formats into the selection and projection query plans. These works will not only facilitate the development of our system greatly but also guide our research as we continue to add new features to the design.

3 METHODOLOGY

3.1 Setting up the Cloud

For the entirety of this project, we implement our database system on Amazon Web Services (AWS). Since our work is distributed among three members, we have set up different access levels for each of our member to ensure the security of resources. In AWS, this functionality is managed through a service named Identity and Access Management (IAM) [3]. The IAM allows developers to attach fine-grained permission policies to users and service roles. The following describes the permissions we have applied to users based on the distribution of work. One member of our team is responsible for administering the billing information and activation of services on AWS. As such, this person is granted root access to the entire AWS platform. Another person in our team primarily interacts with DynamoDB in the web console and on their local machine. Thus, they are given full access to DynamoDB along with a set of security credentials to make API invocations from the AWS Command Line Interface (AWS CLI). As they may need to occasionally retrieve objects for testing purposes, full access to Amazon S3 [4] has also been attached to their account. Finally, since the third person focuses on DocumentDB and requires S3 for data restoration and backup, they have been assigned full read/write access to both services. Table 1 is a summary of the key IAM policies we have granted to each user.

User ID	IAM Permission Granted
User A	root access
User B	AmazonS3FullAccess
User B	AmazonDynamoDBFullAccess
User C	AmazonS3FullAccess
User C	AmazonDocDBFullAccess

Table 1: IAM Permissions Granted to Each User

3.2 System Design

The high-level design of our system (see Figure 1) consists of three main components: a resolver, a dispatcher, and multiple workers. Each of these components runs as independent Lambda functions in AWS except the dispatcher, which is implemented in AWS Step Functions. When a query is sent to the system, the resolver first validates the query parameters and determine whether to proceed or abort the request. If the query is determined valid, the resolver will then pass the request down to the dispatcher for processing. Once the dispatcher receives the request from the upstream resolver, it distributes the workload according to the type of operation requested. Finally, once the dispatcher has finished distributing the workload, one or more workers will be invoked and the corresponding query operation will be performed on the data stored on S3.

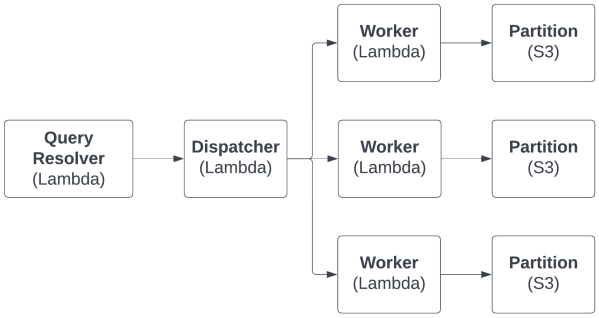

Figure 1: The high-level design of the serverless system.

The following sub-sections will further explain the specific features and implementation details of each component in the system.

3.2.1 Lambda. The functionality of the experimental database has been built solely using serverless functions on AWS Lambda. Below highlights the implementation details we have completed for each component of the database, as well as the permission policies we have added to each Lambda function. The functions have been developed in Python using Boto3 SDK from AWS. All functions have been configured with a run-time environment of Python 3.11 that runs on the arm64 architecture with 128MB of memory and 512MB of ephemeral storage. The use of arm64 reduces the cost associated with every function invocation.

3.2.2 S3. S3 is the file system of our database. It is the destination of the actual data files which come in the format of JSON. At the top level of S3, we have created a single bucket that serves as the root path for all tables of the database. Within the bucket, a list of user-created tables can be found. While these entries do not represent objects nor provide direct access to the tables, they help identify the partitioned data at the next path level with a unified name, abstracting the collection of JSON files into a virtual table. At the lowest level of the storage lies the physical data files. These files are not organized in a particular order at present, however, the

```
{
  "type": "delete",
  "table": "users",
  "search_key": "username",
  "search_value": "user81"
}
```

Figure 2: A sample delete query sent to the resolver.

name of the file includes a numerical count of records stored in it. This design decision will help the worker threads execute tasks in a more efficient manner which will be discussed in 3.2. Note that the table and partition paths are not coded manually. Rather, they are dynamically generated based on the user input and the size of the partition.

3.2.3 Query Resolver. The query resolver is the Lambda function that interfaces with the end user. Its main purpose is to serve as the front-end servlet to answer query requests from users. As of this phase, we have been able to support four operations: insert, find, update, and delete. This Lambda function takes as an input a single-line query statement in the form of JSON. When a query arrives at the query resolver, the program first checks if the syntax is valid in the Lambda event object. If it is valid, the resolver will then forward the request to the dispatcher for parallel processing. Otherwise, no operation will be performed, and an HTTP status code with 400 will be returned. Figure 2 is a sample JSON input to the resolver which performs a delete operation on records with their username fields equal to "user81."

3.2.4 Dispatcher (AWS Step Functions). The dispatcher's job is to parallelize workloads among workers. It is implemented in AWS Step Functions' state machines, a service that enables step-by-step workflow orchestration across multiple AWS services [5]. For insertions, we do not currently support parallel execution. When an insertion is requested, it bypasses the dispatcher stage and invokes the insert Lambda worker directly. In the future, as we extend the batch insertion feature to the insertion worker, it is expected that parallel processing will be supported to streamline the performance. For the remaining three types of workers, they follow a three-step process to achieve concurrent execution. First, the state machine in AWS Step Functions examines the input JSON to determine the type of operation to perform. Second, it uses the S3 ListObjectsV2 API to obtain a list of partitions of the target table and passes the available partitions (JSON file names) to the next stage for further processing. Finally, using the special Map state in Step Functions, each partition is issued a Lambda function to process its designated task. We have configured all dispatcher invocations to run in synchronous mode. As such, the result will only be returned to the upstream query resolver if all dispatched workers have successfully completed. The designed workflow can be found in Figure 3.

3.2.5 Worker (Insert). The insert worker appends a new record to the database given a valid table name. It accepts as an input the

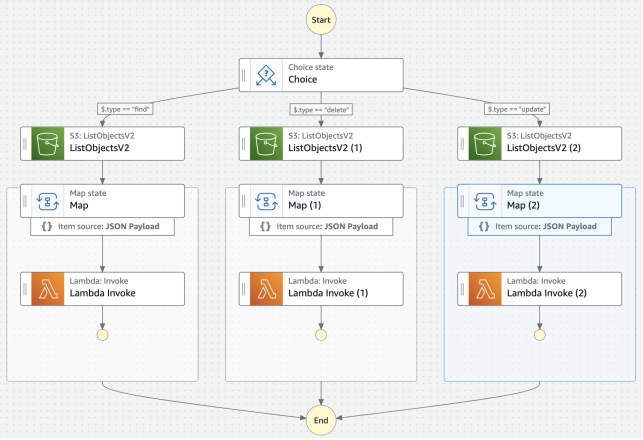

Figure 3: The workflow of the dispatcher stage.

```
{
  "table": "users",
  "data": {
    "username": "user26",
    "password": "262626262626"
  }
}
```

Figure 4: A sample input JSON to the insert worker.

table name to which the record will be inserted and the resolved query in the form of JSON key-value pairs. The worker function then looks up the available partitions in the target table with a list_objects function call to s3. It then adopts a first-fit approach to insert the input data into the first partition that has not yet reached the defined MAX_RECORD_LIMIT. To facilitate future indexing needs, we have developed an automatically incremented object ID field that is attached to each new record. This is done by storing a system file named current_object_id.json to keep track of the most recently added object ID. On each insertion, the system file is opened, updated, and written back to the S3 storage space. Figure 4 provides a sample input to the insert worker which appends a new record with username "user26" and password "262626262626" to the table "users."

3.2.6 Worker (Find, Update, Delete). The find, update and delete workers all perform their operations in a similar procedure except that they apply their respective actions to records. Based on the partition value passed into the event object, they will perform the requested query action by reading the JSON file contents and return a status code to the dispatcher once complete. In the case of an empty object resulting from a deletion, the worker will treat it as a delete_object request and subsequently remove the object. In the case of an update, the resulting JSON object is written back to the S3 storage. Figure 5 illustrates a sample input to the update worker

```
{
  "type": "update",
  "table": "users",
  "search_key": "username",
  "search_value": "user21",
  "update_key": "password",
  "update_value": "121212"
}
```

Figure 5: A sample input JSON to the update worker.

which searches for records with username "user21" and changes their password field to "121212."

3.3 DynamoDB

One of the databases we're going to use to compare our system to is Amazon's DynamoDB [2]. This is because DynamoDB is one of the main NoSQL solutions that is provided by AWS, and is one of the most widely used cloud data solutions with a lot of built-in features that make it much simpler to work with in the AWS architecture. Performing tests with DynamoDB will allow us to have a baseline for the performance of one of the leading cloud-based database systems, while also allowing us to quickly set up some tests, which can then be applied to our serverless, ad-hoc system based on Lambda functions.

In order to be able to properly test DynamoDB, the tables had to be set up and all of the different settings configured. To stay within the costs of the project it also was important that there would not end up being any charges for through AWS for using the way it would be used. Once all of this was decided, the default settings were chosen since they were the best for our use case. Permissions also had to be set so that all of the team members would have access to the database.

After setting up the table, the first step was setting up the local machines to be able to populate them. This was done through the utilization of an SDK with local code. The Boto3 SDK was used since the Lambda architecture was already using it. After this setup was completed, initial testing was performed to ensure that everything was configured properly when it was time to perform the evaluation. This consisted of generating random user data with a username and password and then inserting the data. This was done for about 1000 users. It was noted that during some of the operations there was a noticeable slowdown caused by AWS if too many requests came in at once.

3.4 DocumentDB

The initial database system suggested in the proposal for comparison was "MongoDB Atlas". Unfortunately, it is a pay-as-you-go service hosted on Amazon Web Services (AWS) and is not part of the AWS free-tier services. Given our budgetary limitations, we have made the informed choice to exclude "MongoDB Atlas" from our comparative analysis. After careful exploration of various other options, we have decided that AWS Document DB is the most suitable alternative for "MongoDB Atlas". The AWS DocumentDB, which has MongoDB compatibility, is a fully managed native JSON document database that makes it easy and cost-effective to operate critical document workloads at virtually any scale without managing infrastructure [1]. AWS DocumentDB proves to be a great comparison tool for our serverless, ad-hoc system testing based on Lambda function, as its compatibility with MongoDB drivers allows for seamless integration with our Lambda-based architecture, enabling efficient data management within the dynamic serverless environment. DocumentDB's scalability, high performance, reliability, and data durability support carried out our ad-hoc tests and system analysis with speed and precision.

The DocumentDB test cluster was set up using engine version = 5.0.0, the number of instances = 1, and a custom parameter group with tsl parameter disabled for the initial testing. Also, note that the free tier version of DocumentDB has several restrictions that limit us to set up only db.t3.medium (2 vCPUs, 4GiB RAM) cluster [7].

During our effort to set up DocumentDB, we encountered significant challenges, especially due to its inherent nature as a virtual private cloud (VPC)-only service, devoid of support for public endpoints. Hence, the attempt to establish a local benchmark on our machine failed due to this VPC restriction. As per the AWS official documentation, DocumentDB is commonly integrated with AWS EC2 instances, a configuration marked by complexity and associated costs [6]. In our quest for a more scalable and cost-effective solution, we explored the use of AWS Lambda functions to connect with DocumentDB. However, this avenue posed its own set of challenges, primarily related to intricate permission issues and the integration of Python dependencies, specifically the PyMongo library, into the AWS Lambda function. Unfortunately, these complexities hindered our successful completion of the DocumentDB setup for the intended evaluation.

4 EVALUATION

4.1 Experimental setup

To test and compare the Lambda, the instance of DynamoDB on AWS had to be configured properly. A lot of this was discussed in the DynamoDB subsection in the last section. The setup for DynamoDB consisted of a table with default settings and a partition key under the product name attribute, and the configuring of local environments to run the test from for DynamoDB. For the Lambda system, the tests were set up as different JSON requests that would be sent to the system.

4.2 Testing

To perform the tests, queries were created that would simulate a basic online store's various operations. This consisted of basic insert, update, delete, and read operations along with setting up the database tables to contain items with 5 different attributes: product name, stock, price, sale, and category. The insert and delete operations simply dealt with inserting and deleting products into the database. Update operations were created to simulate the removing and adding items to the stock of the website, along with the

toggling of the sale status of a product. The basic read operations consisted of a getting the price of a product, which would be modified depending on whether an item was on sale or not. There also were larger read operations that filtered based on category and sale. All of these operations were chosen randomly on a database with 1000 records, with the various different chances shown below:

- Insert product: 2%
- Delete product: 1%
- Change stock: 12%
- Get price: 50%
- Toggle sale: 5%
- Get items on sale: 10%
- Get items in category: 20%

Due to running out of time for the project, the tests were not able to be properly performed for the Lambda system, but the tests that were run gave us sufficient information to deduce what the results likely would have been.

4.3 Discussion and Results

The results of running these tests on DynamoDB are shown in Table 2.

Operation	Average latency
Insert product	21ms
Get price	140ms
Toggle sale	392ms
Stock change	337ms
Delete product	21ms
Get items on sale	144ms
Get items in category	106ms

Table 2: DynamoDB Test Results

When testing basic inserts, reads, updates, and deletes on the Lambda system there was a significant variance in the results. For the inserts it ranged from taking anywhere from 1 second all the way to 5 seconds. For reading, updating, and deleting, the range was anywhere from 1 second to 4 seconds. The variance was likely due to how AWS scheduled the lambda function. All of these simple operations were performed on a small dataset of less than 10 partitions of 10 records each, so it was concluded that the system would perform significantly worse than DynamoDB even if the tests developed were adapted for the system.

So alluded to in the last paragraph, the results show that the Lambda system performed significantly worse than DynamoDB in every operations. While it was expected that the Lambda system would perform a bit worse, this significant and variable difference in performance shows the system would not be viable for general purpose usage. The latency of requests is just way to great to be able to utilize properly, even for small datasets, which was why it was decided that there was not a need to evaluate it on much larger datasets.

While the idea for the system was interesting in practice, it was not viable. The different challenges that come with creating and setting up the Lambda system also give it a lot more initial overhead. Since the system was run ad-hoc, it was very much limited by the AWS scheduler performance wise. It also did not have the ability to handle concurrency. While this has been done before in the building a database on S3 paper [10], it presents more that has to be done by the developers. Ready-made solutions like DynamoDB will be able to offer better performance and storage guarantees with much less developer effort.

5 CONCLUSION

This project embarked on a fundamental exploration of serverless computing, specifically focusing on the innovative application of AWS Lambda functions to simulate a NoSQL database in a cloud environment. Drawing inspiration from past works, particularly the Starling system, the experimental serverless architecture aimed to provide insights into the viability of such a system. However, benchmarking against DynamoDB yielded results that underscored significant performance disparities, particularly in basic CRUD operations. Notably, DynamoDB outperformed our system in terms of latency, rendering it unsuitable for general-purpose usage. However, it is still worth exploring how the AWS Lambda function works given that the system was built in the ad-hoc nature of our system, coupled with the challenges of AWS Lambda's scheduling.

The limitations, including constraints associated with the AWS free tier, support only for equality conditions, and the absence of fine-grained security policies, further imply that ready-made solutions offer better performance and storage guarantees with minimal developer effort. While the initial exploration revealed valuable insights into the potential of serverless computing for database simulations, the identified limitations underscore the need for careful consideration when venturing into unconventional architectural paradigms. In future endeavors, a balance between innovation and practical feasibility will be crucial for steering the development of serverless databases toward practical and efficient real-world applications.

6 FUTURE WORK

This study was carried out as a basic exploration of serverless computing specifically for the application of AWS Lambda functions to simulate a database system in a cloud environment within a limited time frame. This research endeavor aimed to establish the initial viability of the proposed database system. In order to enhance the value of both the system and the research work, future efforts can concentrate on several improvements and refinements. The preliminary testing of the system was constrained by the limitations of the AWS free tier, resulting in a small-scale assessment. Subsequent phases of research should involve comprehensive testing with larger datasets to provide a more nuanced understanding of the system's scalability and performance under varying loads. Additionally, the study can be extended to incorporate advanced mechanisms such as concurrency control, recovery management, and in-depth exploration of security aspects. These enhancements will contribute to a more robust and comprehensive evaluation of the system's capabilities, furthering the applicability of serverless computing in the context of database simulations.

REFERENCES

[1] Amazon. [n. d.]. Amazon DocumentDB. https://aws.amazon.com/pm/documentdb/

[2] Amazon. [n. d.]. Amazon DynamoDB. https://aws.amazon.com/dynamodb/

[3] Amazon. [n. d.]. Amazon IAM. https://aws.amazon.com/iam/

[4] Amazon. [n. d.]. Amazon S3. https://aws.amazon.com/s3/

[5] Amazon. [n. d.]. AWS Step Functions. https://aws.amazon.com/step-functions/

[6] Amazon. [n. d.]. Connect using Amazon EC2. https://docs.aws.amazon.com/documentdb/latest/developerguide/connect-ec2.html

[7] Amazon. [n. d.]. Get Started with Amazon DocumentDB. https://docs.aws.amazon.com/documentdb/latest/developerguide/get-started-guide.html

[8] Amazon. [n. d.]. Serverless Computing - AWS Lambda - Amazon Web Services. https://aws.amazon.com/lambda/

[9] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. 2010. A view of cloud computing. *Commun. ACM* 53, 4 (2010), 50–58.

[10] Matthias Brantner, Daniela Florescu, David A. Graf, Donald Kossmann, and Tim Kraska. 2008. Building a database on S3. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 251–264. https://doi.org/10.1145/1376616.1376645

[11] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 115–130. https://doi.org/10.1145/3318464.3389758

[12] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 131–141. https://doi.org/10.1145/3318464.3380609