**EDR Assignment**

**Team:**

* **Oana Sabău**
* **Teodora Boboc**
* **Cosmin Badea**
* **Valentin Bănică**

**Task 1:**

**Research & Documentation for DynamoDB:**

**Denormalization**:

* You can store related data from multiple tables within a single DynamoDB item;
* Easier to scale and faster to develop applications based on microservice architecture;
* Store and process small items with few attributes. For items up to 4 KB in size, one RCU can perform two eventually consistent or 1 strongly consistent read requests per second. For items up to 1 KB in size, one WCU can perform one write request per second.
* The application need to read and write data in a high-traffic environment, without being concerned about synchronization of data across multiple tables

**Composite Keys:**

* A combination of the partition key and sort key;
* All Get, Update, and Delete item commands must include both the partition key and sort key
* The partition key is no longer considered unique, but the composite key is considered unique, and therefore you need both elements to identify the specific item you are referring to
* This allows for more granular and efficient data retrieval, especially in large datasets, like in our case.

**Query Across Tables**

* Effective for certain use cases;
* DynamoDB doesn’t automatically sync the keys across multiple tables;
* It isn’t recommended for complex read queries;

**Conclusion**

The most efficient technique for *our assignment* is **Composite Keys**, which we have implemented in our solution because they enable efficient querying and logical data grouping, allowing us to perform range queries and target specific subsets of data. This approach supports multiple access patterns and enhances scalability by distributing data evenly across partitions. Composite keys also provide flexibility in sorting and organizing hierarchical data, making them ideal for our application’s requirements.

**Task 2:**

**A screenshot of a computer

Description automatically generated**

Fig. 1. Our designed schema

**Task 3 & Task 4:**

Rough approximations on performance of implemented functions:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Duration (***seconds***)** | | **Notes** |
| **Query** | **Scan** |
| Get\_events | 0.588 | 0.635 | 1 vehicle – 1 event – 5 incidents |
| Generate\_vehicle\_data | 1.455 | | 1 vehicle – 1 event – 5 incidents |
| 8.371 | | 2 vehicles – 2 events – 8 incidents |
| Delete\_data | 1.163 | 1.241 | 2 vehicles – 2 events – 8 incidents |

For our examined templates the measurements show how the scan is slightly slower than the querying methods.

**Output example**:

A screen shot of a computer

Description automatically generated

Fig. 2. Example of found accident



Fig. 3. Example of false positive

**Task 5:**

**Global Secondary Indexes (GSI)**

* A GSI allows you to query data across the entire table using a different partition key and sort key from the base table.
* GSIs are highly flexible as they can span all the data in the base table, across all partitions.
* Ideal for queries that need to access data across multiple partitions or when you need to query on attributes that are not part of the primary key2.

**Local Secondary Indexes (LSI)**

* An LSI uses the same partition key as the base table but allows a different sort key.
* LSIs are scoped to a base table partition, meaning they are limited to the partition key value1.
* The total size of indexed items for any one partition key value cannot exceed 10 GB1.

**Composite Keys**

* Composite keys in DynamoDB consist of a partition key and a sort key. This combination allows for more complex query patterns.
* They enable you to store multiple items with the same partition key but different sort keys, allowing for efficient querying and sorting within a partition.
* Ideal for scenarios where you need to store and query related items together.

**Comparison**

* GSIs offer the most flexibility as they can query across all partitions, while LSIs are limited to a single partition.
* LSIs have a size limitation of 10 GB per partition key, while GSIs do not have this restriction.
* GSIs are better for global queries across the table, while LSIs are suited for more localized queries within a partition.

**Development** **notes**

* During development it has been noticed that choosing both partition and sort keys are vital when it comes to proper querying and filtering.
  + Using the scan functionality is slower but allows easier retrieval of data with a less complex structure.
  + Using the query and filter allows faster retrieval of data within a complex structure.
* It has been noticed that having lots of LSIs provides a better way of filtering larger datasets.