**DynamoDB Core Concept Interview Challenge**

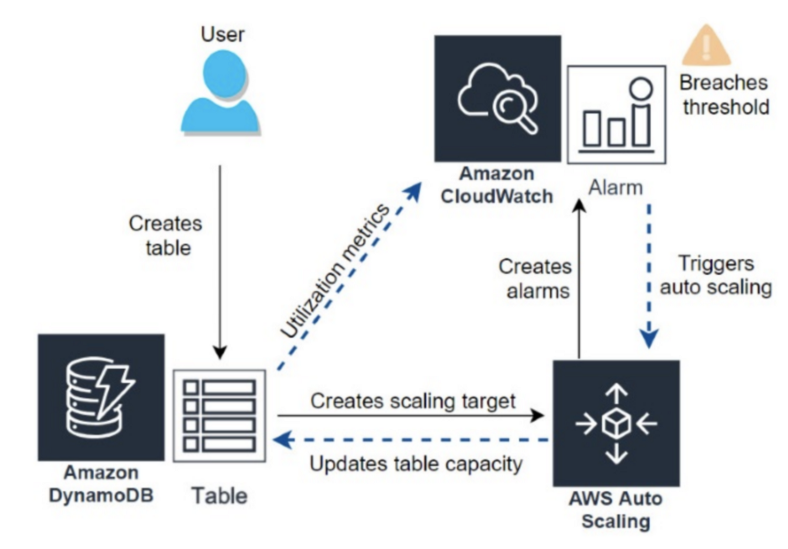
In this article we will go through basic knowledge and core concept help for new guy learn about serval DynamoDB features.

**Q1: What is DynamoDB?**



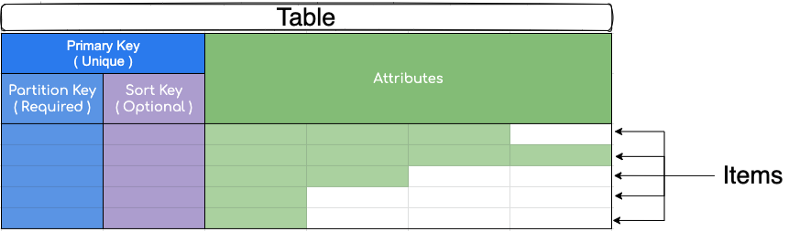
DynamoDB Table

* DynamoDB is a NoSQL (non-relational) database service designed for Online Transactional Processing (OLTP) workloads.
* DynamoDB is a serverless, fully managed
* Flexible Schema JSON document or key-value data structures
* Supports event-driven programming
* Availability, durability, and scalability built-in
* Scales horizontally



**Q2: What are Tables and Partitions in DynamoDB?**

* **Data is stored in tables.**
* **A table contains items with attributes. You can think of items as rows in a relational database and attributes as columns.**



* Item : An item is a collection of attributes and attribute has a name, data type, and value. Item can have any number of attributes ( Note: Items in a table can have different types of attributes )
* Partition Key : DynamoDB stores data in partitions and divides a table’s item into multiple partitions based on the partition key value.
* Sort Key: A sort key can be defined to store all of the items with the same partition key value physically close together and order them by sort key value in the partition. It represents a one-to-many relationship based on the partition key and enables querying on the sort key attribute.

**What is Partition and the relationship between Partition with the Table?**

* Partition is an allocation of storage for a table ( backed by Solid-State Drives SSDs) and automatically replicated across multiple Availability Zones within an AWS Region.
* Partition management is handled entirely by DynamoDB. The partition key of an item is also known as its hash attribute.
* As the data grows and throughput requirements are increased, the number of partitions are increased automatically. DynamoDB handles this process in the background.

What is limits of a partition of table and partition?

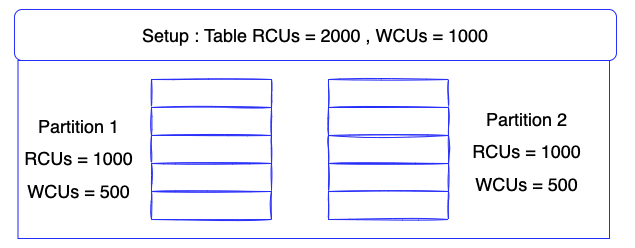
* Maximum storage of data/partition : 10 GB
* Maximum size of item : 400KB
* Maximum items one partition can hold : 10 GB/Size of item ( Ex: 10GB/400KB ~ 25000 items/partition )
* Maximum Read capacity Units ( RCUs ) : 3000
* Maximum Write Capacity Units ( WCUs ) : 1000

**Q4. When a partition created?**

**1 >**The first time when we create a table ( You have a step to setup ReadCapacityUnits and WriteCapacityUnits for the table )

Example : ReadCapacityUnits = 2000 & WriteCapacityUnits = 1000

Num of Partitions = Math.Round((RCUs / 3000) + (WCUs / 1000)))  
-> Math.Round(2000/3000 + 1000/1000) = 2 Partitions

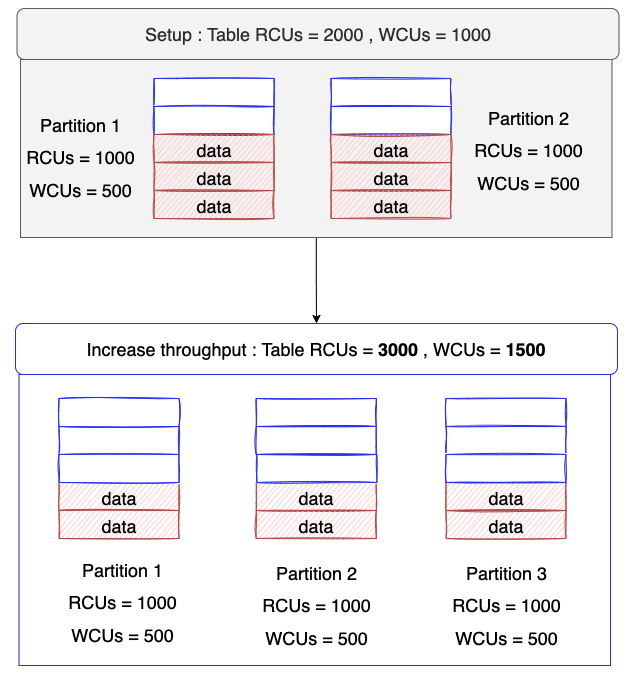


**2 >** Manual scale provisioned ( Example : Some performance issues and you need to increase your RCUs and WCUs to resolve the problem )

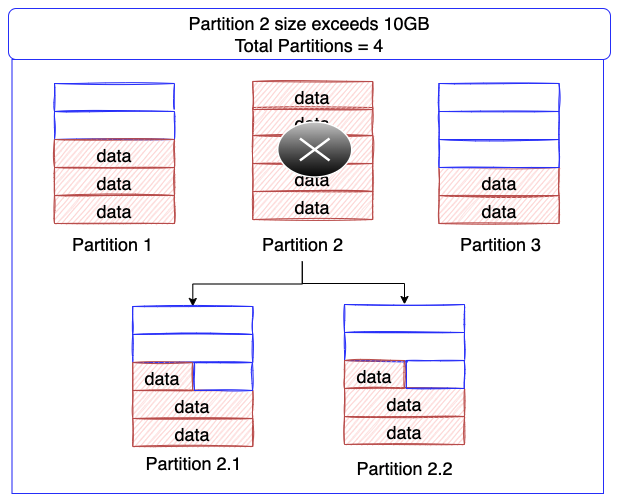
Example

* Original Current (RCUs = 2000 — WCUs 1000)
* Your plan (RCUs = 3000, WCUs 1500)

Num of Partitions = Math.Round((RCUs / 3000) + (WCUs / 1000)))  
-> Math.round(3000/3000 + 1500/1000) = 3 Partitions



3 > Partition size exceeds storage limit (10GB/partition)



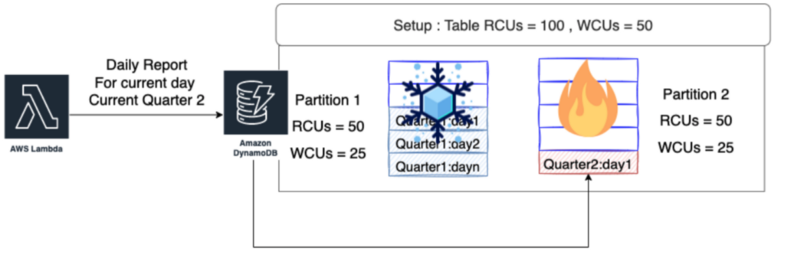
**Q5: How to we know items saved on the same partitions?**

* Same partition key are always -> the same partition ( However different partition keys also can the same partition Example because you have only 1 partition) It means partition and partition keys are not mapped on a one-to-one
* A range key ensures that items with the same partition key are stored in order.
* Based on the Hash Function ( Internally hash function of DynamoDB )

**Q6: What is the Hot Key Problem?**

Because the total throughput provisioned for a table is divided equally across partitions. So If you choose a partition key for which some values get much more traffic than most others, you will have a “hot” key, and the partition where it lives will also run “hot”.

Example: You have a 2 partitions with partition key is “quarter::${n}” and the init setup RCUs = 100, WCUs = 50 ( Today is begin the Quarter 2 )



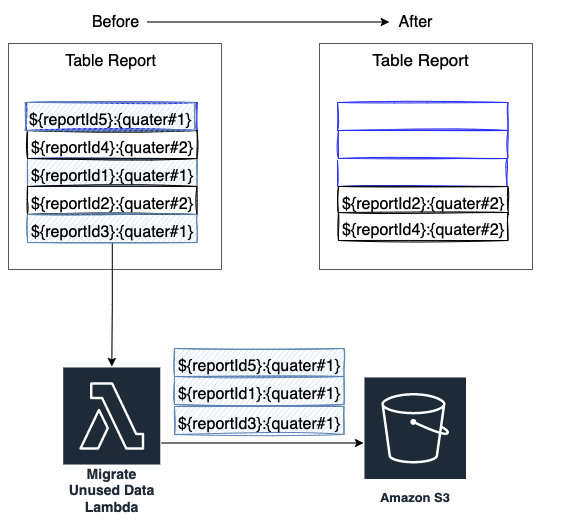
The problem all requests from (04/2022 ~ 06/2022) only access on the partition 2. the HOT partition happening on the partition 2 and The throughput allocated to partition 1 remains**unused.**

**Most importantly**you need to choose partition key which will result in an even distribution of item data and traffic across the hash space. A good partition key has high “cardinality” — that means there are lots of unique values. ( Example The report\_id would be a much better choice pk = ${report\_id} uuid instead pk ={quarter#n} )

**Q7: What is the Cold Key(data) Problem?**

**Once the data are no longer required,** you can delete them (and not have to pay for a lot of deletes with your WCUs). Better yet, consider a cold tier which is stored to S3.

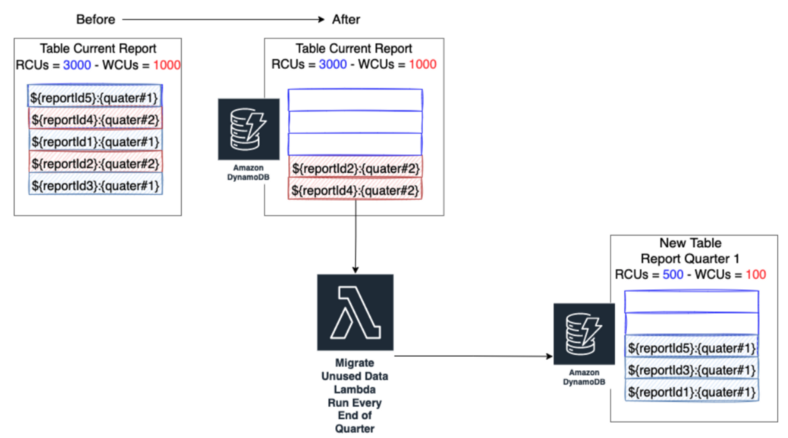
***> > > >Solution 1***: Backup If you want so far in the future we can use it



***> > > > Solution 2*: Apply TTL ( Time to live ) on the record and DynamoDB will remove it.**

**It might rarely access data about very old orders,** consider breaking the data into separate tables. Store the frequently accessed “hot” data in a separate table with higher throughput. Store rarely accessed “cold” data in tables with lower throughput.

Example : Currently we always get data in the Quarter2 and the Quarter1 rarely to access



**Q8: How to resolve problem with large attributes ( > 400 KB)**

Items in DynamoDB are limited to 400kB in size. Reading or writing such large objects results in hot activity localized to a single partition.

>>> Solution 1: Ideally, you want to keep item sizes small (between 1KB and 4KB is optimal). To achieve this, first consider storing your large objects in S3, and keep only a reference to that object in your DynamoDB item as metadata — you can consider this model with DynamoDB essentially serving as an index for your S3 objects.

>>> Solution 2: Consider compression for large objects — store them as a binary attribute, and keep smaller metadata attributes separately.

>>> Solution 3: Break up large attribute values across multiple items. If you really do have a need to work with large objects in DynamoDB and you also need high performance and low latency for those items, break the object into smaller chunks and keep them in a number of separate items.

Note: **This way you can read and write in parallel for best performance, and the traffic is well distributed across your partition space.**

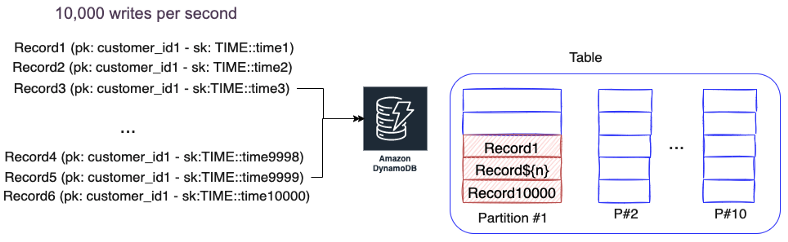
**Q9: How to create Partition Key in a right way?**

This partition key determines the partition on which that item will store and each partition maximum RCUs = 3,000 and WCUs = 1,000 . If a “hot” partition happen, We could got a performance issue.

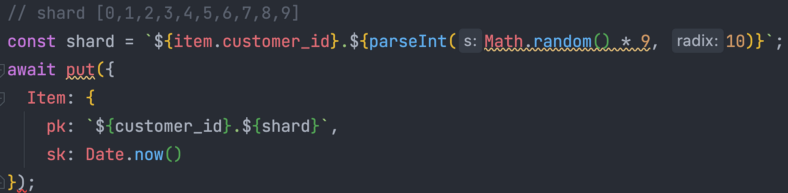
**The Random Partition Key ( ex: uuid )** : Each request essentially gets its own partition, and therefore we can achieve 3000 individual request item reads per second (assuming the item is 4KB or less). This is extremely high read throughput for an entity like an api request, and serves our use case well.

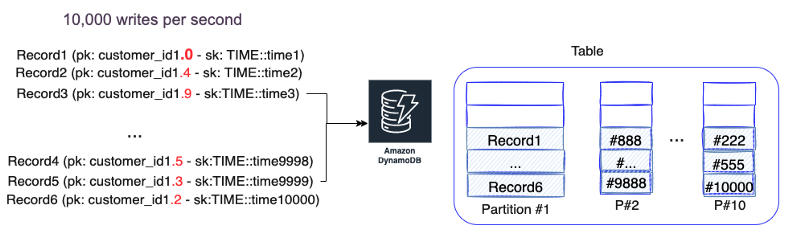
>>> What problem with this design: **Impossible to fetch all items for a given** ( Expect Scan action ) The use cases where items do not require many updates, a random partition key provides essentially unlimited write throughput and very acceptable read throughput.

**The Sharded Partition Key:**I want to retrieve all items for a given by ${customer\_id} you need to split a tenants partition into multiple smaller partitions or shards and distribute their data evenly across those shards.



Apply a post-fix shard key to the partition key





**Q9: What are LSIs, GSIs and the different between them?**

Use indexes sparingly — remember that each index will result in additional writes that you will need to pay for with WCUs. Remember that LSIs will limit collection size for a partition key to approximately 10GB of data.

Project attributes selectively, and take advantage of sparse indexes to optimize your throughput consumption. And recall that LSIs have the same lifetime as the base table.

***Don’t build indexes you don’t need — they will consume capacity***

**Local Secondary Indexes (LSIs)**

* Local secondary indexes consume storage and the table’s provisioned throughput. Keep the size of the index as small as possible.
* Choose projections carefully.
* Project only those attributes that you request frequently.
* Take advantage of sparse indexes.

Note : To create one or more local secondary indexes on a table, use the LocalSecondaryIndexes parameter of the CreateTable operation. Local secondary indexes on a table are created **when the table is created**. When you delete a table, any local secondary indexes on that table are also deleted.

**Global Secondary Indexes (GSIs)**

GSIs do not support strongly consistent reads, but are otherwise much more flexible than LSIs. Any LSI can also be modeled as a GSI.

GSI acts like any other table — choose a partition key and sort key that will distribute reads across multiple partitions.

* Take advantage of sparse indexes.
* Create a global secondary index with a subset of table’s attributes for quick lookups.
* Use as an eventually consistent read replica.

**Q10: How to managing concurrency with the same write?**

This would be important where you only want to update an item if it has not changed since you last read it.

Use optimistic locking with a version number to make sure an item has not changed since the last time you read it.

Use optimistic locking with a version number as described below to make sure that an item has not changed since the last time you read it. This approach is also known as the read-modify-write design pattern or optimistic concurrency control.

<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/WorkingWithItems.html#WorkingWithItems.ConditionalUpdate>