Before we learn about DynamoDB, we need to understand the difference between SQL and NoSQL Databases

SQL (Structured Query Language) and NoSQL (***Not Only or Non*** SQL) databases serve different needs and use cases.

Feature	SQL Databases	NoSQL Databases
Schema	Fixed, predefined schema	Dynamic, flexible schema
Data Model	Relational, table-based	Varied (document, key-value, wide-column, graph)
Transactions	ACID compliance	BASE properties
Query Language	SQL	Varies (e.g., JSON-like queries, custom APIs)
Scalability	Vertical scaling	Horizontal scaling
Consistency	Strong consistency	Eventual consistency (in many cases)
Use Cases	Complex queries, structured data, high integrity	Big data, unstructured data, high availability
Examples	MySQL, PostgreSQL, Oracle, SQL Server	MongoDB, Cassandra, Redis, Neo4j

Choose SQL if:

- You need complex queries and transactions.
- Data integrity and ACID compliance are critical.
- Your data structure is well-defined and unlikely to change.

ACID (Atomicity, Consistency, Isolation, Durability)

Choose NoSQL if:

- You need to handle large volumes of unstructured or semi-structured data.
- Your application requires high availability and horizontal scaling.
- The data model is flexible or likely to evolve over time.
- You are building distributed systems with low-latency requirements.

BASE (Basically Available, Soft state, Eventual consistency)

Both SQL and NoSQL databases have their strengths and are suitable for different types of applications. The choice depends on your specific use case, data requirements, and scalability needs.

NoSQL Databases

- They are non-relational databases and are distributed
- They do not support the query joins
- All the data that is needed for a query is present in a row
- They don't perform aggregation such as SUM, AVG
- Scaled horizontally

Now we know the SQL and NoSQL Differences ...! Let's get into our Dynamo DB

Amazon Dynamo DB

- Fully Managed, highly available with replication across multiple AZs
- NoSQL database – not a relational database
- Scales to massive workloads, distributed database
- Millions of requests per second, trillions for row, 100s of TB Storage
- Fast & Consistent in performance
- Integrated with IAM for security, authorization and administration
- Enables event driven programming with DynamoDB Streams
- Low cost and auto scaling capabilities
- Standard & Infrequent Access (IA) Table Class

How it looks like

- DynamoDB is made of Tables
- Each table has a Primary Key (must be added at the creation time)
- Each Table can have an infinite number of items (We can call it as Rows)
- Each item has attributes (Can be added over time – can be null)
- Maximum size of ***an item is 400 KB***
- Data types supported

1) **Scalar Types** – String, Number, Binary, Boolean & Null
2) **Document Types** – List, Map
3) **Set Types** - String set, Number set and Binary set

Primary Keys

Choosing one: Partition Key (HASH)

- Partition key must be unique for each item
- Partition Key must be diverse so that data is distributed
- Example: Student_ID for the student's table

Primary Key	Attributes		
Partition Key			
User_ID	**First_Name**	**Last_Name**	**Age**
7791a3d6-...	John	William	46
873e0634-...	Oliver		24
a80f73a1-...	Katie	Lucas	31

Choosing Two: Partition Key + Sort Key (HASH+RANGE)

- The combination must be unique for each item
- Data is grouped by partition key
- Example: students and sports, Student_ID for the partition key and sport_ID for the sort key

	Primary Key		Attributes	
	Partition Key	Sort Key		
	User_ID	**Game_ID**	**Score**	**Result**
	7791a3d6-...	4421	92	Win
Same partition key Different sort key	873e0634-...	1894	14	Lose
	873e0634-...	4521	77	Win

Partition key should be a column in the table that has highest cardinality to maximize the number of partitions (data distribution). Partition key should also be highly diverse to ensure that the data is distributed equally across partitions.

If the partition (hash) key is not highly diverse (only few unique values), add a suffix to the partition key to make the partition key diverse. The suffix can be generated either randomly or calculated using a hashing algorithm.

As a basic fundamental – now we know how DynamoDB looks like so, now we need to understand, what size we need and all other things related to the Dynamo DB

We need to learn few terms before we step into the Introvert Path of the DynamoDB

Capacity Planning

Capacity planning in Amazon DynamoDB refers to the process of estimating and provisioning the read and write capacity units (RCUs and WCUs, respectively) that your DynamoDB tables will require to handle the expected workload efficiently.

Throughput

In Amazon DynamoDB, throughput refers to the measure of the amount of read and write activity that a DynamoDB table can handle per unit of time. Throughput capacity in DynamoDB is provisioned and measured in terms of Read Capacity Units (RCUs) and Write Capacity Units (WCUs).

DynamoDB – Read/Write Capacity Modes

Control how you manage your table's capacity (read/write throughput)

Provisioned Mode

- You specify the number of reads/writes per second
- You need to plan capacity beforehand
- Pay for provisioned read & write capacity units

On-Demand Mode

- Read/Writes automatically scales up/down with your workloads
- No Capacity planning needed
- Pay for what you use, more & more expensive

We can switch between modes once every 24 hours

Read/Write Capacity Modes – Provisioned

- Table must have provisioned read and write capacity units
- **Read Capacity Units (RCU) – throughput for reads**
- **Write Capacity Reads (WCU) – throughput for writes**
- Option to setup auto-scaling of throughput to meet demand
- Throughput can be exceeded temporarily using the burst capacity
- If Burst Capacity has been consumed, we will get a "ProvisionedThroughputExceededException"
- It's then advised to do an exceptional backoff retry

Before we get into Read and Write Capacity Units calculation, we might need to learn about few terms here – those are

Eventually Consistent Read: (Default)

An eventually consistent read may not reflect the latest write data but will eventually converge to the most up-to-date data. DynamoDB reaches consistency across all copies of data within typically a second.

- Low latency
- May get stale data (if the replication hasn't happened yet)

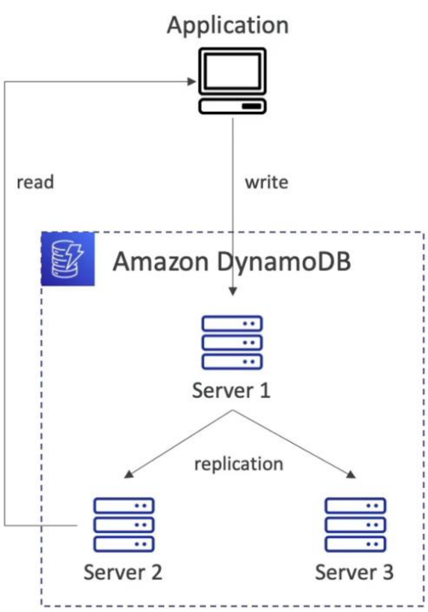
Strongly Consistent Read:

A strongly consistent read returns the most up-to-date data, reflecting all writes that were completed before the read.

- More latency
- Set "ConsistentRead" Parameter to True in API Calls

Transactional Read:

Transactional reads are part of DynamoDB transactions, ensuring that the read operations within a transaction see a consistent snapshot of the data.



Standard Write: (Default)

Standard write operations are atomic, meaning each write either fully succeeds or fails. However, they do not offer cross-item or cross-table atomicity guarantees.

Transactional Write:

Transactional write operations ensure all operations within a transaction (including writes across multiple items or tables) either fully succeed or fail together. This provides ACID (Atomicity, Consistency, Isolation, Durability) guarantees.

Read Capacity Units (RCUs)

One RCU represents the read throughput capacity for strongly consistent reads of items up to 4 KB in size, or two eventually consistent reads per second.

General Formula:

- **Eventually Consistent** Read: ***1 RCU = 2 reads per second*** for items up to 4 KB.
- **Strongly Consistent** Read: ***1 RCU = 1 read per second*** for items up to 4 KB.
- **Transactional** Read: ***1 RCU = 0.5 read per second*** for items up to 4 KB.

For items larger than 4 KB, RCUs are calculated by rounding up the item size to the nearest 4 KB block.

$$\text{RCUs} = \left\{ \frac{\textbf{Item Size}}{\textbf{4 KB}} \right\} \times \left\{ \frac{\textbf{Read Request Rate}}{\textbf{Consistency factor}} \right\}$$

- Consistency Factor = **2** for Eventually Consistent Reads
- Consistency Factor = **1** for Strongly Consistent and
- Consistency Factor = **0.5** for Transactional Reads

Example:

If you need to read 80 items per second, each 6 KB in size, using eventually consistent reads:

- Item size = 6 KB
- Read request rate = 80 reads/second
- Consistency factor = 2

$$\text{RCUs} = \left(\frac{6\,\text{KB}}{4\,\text{KB}}\right) \times \left(\frac{80}{2}\right) = 2 \times 40 = 80\ \text{RCUs}$$

Write Capacity Units (WCU)

One WCU represents the write throughput capacity for writes of items up to 1 KB in size.

General Formula:

- **Standard** Write: **1 WCU = 1 write** per second for items up to 1 KB.
- **Transactional** Write: **1 WCU = 0.5** write per second for items up to 1 KB.

For items larger than 1 KB, WCUs are calculated by rounding up the item size to the nearest 1 KB block.

- Consistency Factor = ***1*** for Standard Writes
- Consistency Factor = ***0.5*** for Transactional Writes

$$\text{WCUs} = \left\{ \frac{\textbf{Item Size}}{\textbf{1 KB}} \right\} \times \left\{ \frac{\textbf{Write Request Rate}}{\textbf{Consistency factor}} \right\}$$

Example:

If you need to write 50 items per second, each 2.5 KB in size:

- Item size = 2.5 KB
- Write request rate = 50 writes/second

$$\text{WCUs} = \left(\frac{2.5\ \text{KB}}{1\ \text{KB}}\right) \times 50 = 3 \times 50 = 150\ \text{WCUs}$$

Batches: When performing batch operations, sum the total size of the items in the batch and then apply the formula.

Indexes: Each secondary index will consume additional RCUs and WCUs based on the size and access patterns of the indexed attributes.

Just to practice

1. **Reading 100 items per second, each 8 KB, with strong consistency:**

   Item size = 8 KB

   $$\text{RCUs} = \left(\frac{8\text{ KB}}{4\text{ KB}}\right) \times 100 = 2 \times 100 = 200\text{ RCUs}$$

2. **Writing 200 items per second, each 0.5 KB:**

   Item size = 0.5 KB

   $$\text{WCUs} = \left(\frac{0.5\text{ KB}}{1\text{ KB}}\right) \times 200 = 1 \times 200 = 200\text{ WCUs}$$

3. **Reading 150 items per second, each 12 KB, with eventual consistency:**

   Item size = 12 KB

   $$\text{RCUs} = \left(\frac{12\text{ KB}}{4\text{ KB}}\right) \times \left(\frac{150}{2}\right) = 3 \times 75 = 225\text{ RCUs}$$

Throttling

- If RCU or WCU for any partition is exceeded, the request will throttle and we'll get **`ProvisionedThroughputExceededException`.**

Reasons for throttling:

- Hot Partitions (most of the data is read or written to only a few partitions)
- Large item size

Solutions to throttling:

- Exponential backoff (already in SDK)
- Evenly distributed partition keys (to avoid hot partitions)
- DynamoDB Accelerator (DAX)

Backups

Two types:

- On-demand
- Point-in-time recovery (PITR) - automatic continuous backups
- No performance impact during backups
- Backups are written to S3 under the hood but we cannot access these backup buckets

Query and Scan Operation Modes

In DynamoDB, **Query and Scan are two different operations used to retrieve data from a table.** Understanding the differences between these operations and their appropriate use cases is crucial for optimizing your database interactions.

Query Operation

Primary Use: Retrieve items based on primary key values.

Efficiency: More efficient and faster than Scan because it searches only the specified primary key values and uses indexes.

Targeted Read: A Query operation is more efficient because it allows you to specify a partition key and optionally a sort key to narrow down the results. DynamoDB retrieves only the items that match the specified keys.

Index Utilization: A Query operation uses the primary key (partition key and sort key) to locate items. This means it leverages the underlying index to quickly find the relevant items without scanning the entire table.

Filters will be applied before read

Usage:

- Requires a partition key value and optionally a sort key value or a range of sort key values.
- Can filter results using a FilterExpression, but filtering is done after the Query operation, which means all matching items are read and then filtered in-memory.
- Supports secondary indexes (Global Secondary Indexes and Local Secondary Indexes) for more flexibility.

```
response = table.query(
    KeyConditionExpression=Key('CustomerID').eq('12345')
)
```

Scan Operation

Primary Use: Retrieve all items in a table.

Efficiency: Less efficient and slower than Query because it reads every item in the table and filters the results based on conditions. This can result in high read throughput consumption and potential performance issues.

Usage:

- Can filter results using a FilterExpression, but filtering is done after the Scan operation, which means all items are read and then filtered in-memory.
- Typically used for small tables or infrequent operations where retrieving all data is necessary.
- Can use Parallel Scan to speed up the operation for larger tables.

Full Table Read: When you perform a Scan operation, DynamoDB reads every item in the table to find items that match the filter criteria. This means that the entire table is read regardless of any filters you apply.

Filters Applied After Read: Filters in a Scan operation is applied after the table is read. This means all items are read into memory and then filtered, which can be very inefficient and result in high read capacity unit (RCU) consumption.

```
response = table.scan(
    FilterExpression=Attr('StockValue1').gt(100)
)
```

Index

An index in a database is a data structure that improves the speed of data retrieval operations on a table at the cost of additional space and write time. **Indexes are used to quickly locate data without having to search every row in a database table every time a database table is accessed.**

Secondary Index meant for: Allow running queries on non-primary key attributes

So now, we understood – what is an Index is, we will see how many indexes that Dynamo DB Supports, DynamoDB supports two types of indexes

Local Secondary Index (LSI)

Global Secondary Index (GSI)

At one go – these are the definitions

LSI is used when you want to query within the same partition key but need different sorting options.

GSI is used when you need to query using completely different keys, providing more flexibility in query patterns.

Now we will look into deep

Local Secondary Index (LSI)

An LSI is an index that has the **same Partition Key as the base table but allows you to have a different Sort Key**. You can create up to 5 LSIs per table. LSIs allow you to query the table based on the alternative Sort Key, providing a different sort order.

When to Use LSI:

- When you need multiple query patterns on the same partition key with different sorting needs.
- When your application requires querying on different attributes but within the same partition key.
- When you need to query on non-primary key attributes and ensure strong consistency, as LSIs allow for strongly consistent reads.

Example Situation:

You have a table storing user activities with UserID as the Partition Key and Timestamp as the Sort Key. You might want an LSI to sort the activities by ActivityType within the same UserID.

Keep in mind:

- Alternative Sort Key for a table (uses the same partition key)
- Queries are made to a single partition
- The sort key of the **LSI must be a scalar attribute** (String, Number or Binary)
- **Max 5** LSI per table
- Must be **defined at table creation time**
- **Uses RCU and WCU of the main table** (works on the same table partition)
- Queries on LSI support **both eventual consistency and strong consistency**

Global Secondary Index (GSI)

A GSI is an index with a **completely different Partition Key and optionally a different Sort Key from the base table.** You can create up to 20 GSIs per table. GSIs allow for more flexible query patterns as they provide an entirely different key structure.

When to Use GSI:

- When you need to query your data using different attributes as the primary access patterns.
- When your application requires querying on attributes that are not part of the primary key.
- When you need more flexibility in your queries, especially when the query involves attributes that are not part of the base table's primary key.

Example Situation:

You have a table storing order information with OrderID as the primary key. You might want a GSI with CustomerID as the Partition Key and OrderDate as the Sort Key to quickly retrieve all orders placed by a specific customer sorted by date.

Keep in mind:

- **Alternative Primary Key** for a table (Hash or Hash + Sort) to speed up queries on non-key attributes.
- The hash key and sort key (optional) of the GSI **must be scalar attributes** (String, Number or Binary)
- Queries are made to the entire database
- **Must provision RCU and WCU** (supports auto-scaling) for GSI since it partitions the same table differently. **If the writes are throttled on the GSI, the main table will be throttled for writes as well.**
- **GSIs can be added or modified after table creation**
- **Queries on GSI support eventual consistency only**

DynamoDB APIs

In Amazon DynamoDB, APIs are the set of operations that allow you to interact programmatically with the DynamoDB service. They enable you to perform various actions such as creating tables, inserting items, querying data, and more. Using APIs, you can build applications that read from and write to DynamoDB tables, as well as manage your database resources.

Table Management APIs:

- CreateTable: Creates a new table.
- UpdateTable: Modifies the settings or structure of an existing table.
- DeleteTable: Deletes a table and all of its data.
- DescribeTable: Returns information about the table, such as its key schema, provisioned throughput settings, and indexes.

Data Management APIs:

- PutItem: Adds a new item or replaces an old item with a new one.
- GetItem: Retrieves a single item from a table by its primary key.
- UpdateItem: Modifies an existing item or adds a new item if it doesn't exist.
- DeleteItem: Deletes a single item from a table by its primary key.
- BatchWriteItem: Writes or deletes multiple items across one or more tables in a single API call.
- BatchGetItem: Retrieves multiple items from one or more tables in a single API call.

Query and Scan APIs:

- Query: Retrieves items based on primary key values. You can use Query with both partition key and sort key.
- Scan: Examines all items in a table and returns all data attributes by default. Scan can filter results and retrieve specific attributes.

Index Management APIs:

- CreateGlobalSecondaryIndex: Adds a new global secondary index to an existing table.
- DeleteGlobalSecondaryIndex: Deletes a global secondary index from a table.
- UpdateGlobalSecondaryIndex: Updates the settings of a global secondary index.

Stream Management APIs:

- DescribeStream: Returns information about a stream, such as its settings and shard structure.
- GetShardIterator: Provides a shard iterator for a specified shard.
- GetRecords: Retrieves the stream records from a given shard iterator.

Transactions APIs:

- TransactWriteItems: Allows coordinated, all-or-nothing changes to multiple items in one or more tables.
- TransactGetItems: Retrieves multiple items from one or more tables in a single, atomic operation.

Now we will look into few APIs - which we need to pay more attention regarding few specifics

- **PutItem** - **fully update** the item based on the primary key or create a new item if it does not exist
- **UpdateItem** - **partially update** the item based on the primary key or create a new item if it does not exist
- **GetItem** - **read an item** using the primary key (hash / hash + range)
  - Eventually consistent read by default, strongly consistent read optional
  - ProjectionExpression can be specified to retrieve a subset of attributes
- **Query** - **read multiple items based on a query** from a table, LSI or GSI
  - KeyConditionExpression
    - Partition key (=) - required
    - Sort key (=, <, ≤, >, ≥, between, begins with) - optional
  - FilterExpression
    - Client-side filtering on non-key attributes
    - Does not allow partition key or sort key attributes
  - Returns a list of items where number of items = limit in the query or up to 1 MB of data. To get more data, use pagination.
- **Scan** - **scan the entire table** (every partition) and return all the data
  - Consumes a lot of RCU
  - Recommended to use limit in the scan operation
  - Returns up to 1 MB of data (use pagination to get more data)

    - Use FilterExpression to filter items and ProjectionExpression to retrieve a subset of attributes (client-side)
    - **Not recommended** unless you need to read the entire table (Eg. Analytics)
    - Use **Parallel Scan** for faster scans
        - Multiple workers
        - Increases throughput and RCU consumed
        - Recommended to use limit in the parallel scan operation
- DeleteItem - **delete an item**
    - Optional conditional deletes (Eg. Delete this item only if age < 0)
- DeleteTable - **delete the whole table** and its items
    - Much faster than calling DeleteItem on all the items
- BatchWriteItem - **write items in a batch of operations** (processed in parallel)
    - **Max 25 operations** or max 16 MB of data written in a single API call
    - Supports PutItem and DeleteItem (does not support UpdateItem)
    - Can write to multiple tables in the same batch API call
    - UnprocessedItems (failed writes) can be retried with exponential backoff
    - If the batch write exceeds WCU limits, add more WCU
- BatchGetItem - **read items in a batch of operations** (processed in parallel)
    - **Max 100 items** or max 16 MB of data can be read in a single API call
    - Can read from multiple tables in the same batch API call
    - UnprocessedKeys (failed reads) can be retried with exponential backoff
    - If the batch read exceeds RCU limits, add more RCU

Conditional Writes

- Option to specify a Conditional Expression to apply the write operation only to the items that satisfy the condition.
- Supported by PutItem, UpdateItem, DeleteItem and BatchWriteItem APIs
- Using attribute_not_exists(partition_key) and attribute_not_exists(sort_key) in our conditional expressions, we can ensure that a data is never overwritten by write commands.
- Optimistic Locking - strategy to ensure an item hasn't changed after you read it and before you update it
- Optimistic Locking - strategy to ensure an item hasn't changed after you read it and before you update it
  - Each item has an attribute Version which is incremented every time the item is updated
  - The idea is to read the item and send a write (update or delete) request to DynamoDB with a conditional expression that the version of the item is what you read. If the version gets incremented before updating, the write operation will fail.

Filter Expressions are for read operations, whereas Conditional Expressions are for write operations.

Batch Operations

In DynamoDB, Batch Operations refer to a feature that allows you to efficiently perform multiple read or write operations on multiple items within a single API call. This capability is designed to optimize throughput, reduce request latency, and lower operational costs by minimizing the number of API calls needed to process large amounts of data.

DynamoDB supports two types of Batch Operations:

BatchGetItem: This operation allows you to retrieve multiple items from one or more DynamoDB tables in a single request. You specify the primary keys of the items you want to retrieve, and DynamoDB returns the requested items. BatchGetItem can retrieve up to 16 MB of data or up to 100 items per call.

BatchWriteItem: This operation allows you to put, delete, or update multiple items in one or more DynamoDB tables. You specify a list of write requests (PutRequest, DeleteRequest, or UpdateRequest), each containing a primary key and the attributes to modify. DynamoDB processes the write requests in parallel, subject to the usual transactional and capacity constraints.

PartiQL

- SQL-compatible query language for DynamoDB
- Can run queries across multiple DynamoDB tables
- Supports some (not all) SQL statements: `INSERT`, `UPDATE`, `SELECT`, `DELETE` (does not support `JOIN` operation)
- -Supports batch operations

DynamoDB Fine-Grained Access Control (FGAC)

DynamoDB Fine-Grained Access Control (FGAC) allows you to control access to individual items or attributes within a DynamoDB table using AWS Identity and Access Management (IAM) policies. This feature is particularly useful in scenarios where you need to enforce granular access permissions based on specific conditions or attributes within your data.

DynamoDB Fine-Grained Access Control (FGAC) is used to enforce granular access permissions to individual items or attributes within a DynamoDB table using IAM policies. By leveraging FGAC, you can ensure data security, compliance with regulations, and efficient data access management tailored to your application's specific requirements and use cases.

Now we need to learn about the few terms before we go to DynamoDB Accelerator (DAX)

Cache:

A cache is a temporary storage area that holds a subset of data, typically to accelerate retrieval times for frequently accessed or computationally expensive data. The primary purpose of a cache is to reduce access time and improve performance by storing this data in a faster, more accessible medium, such as RAM, compared to the original storage location, like a hard drive or a remote server.

Distributed Cache:

A cache that spans multiple machines or nodes, allowing it to handle larger data sets and higher loads.

In-memory distributed cache:

An in-memory distributed cache refers to a caching mechanism where data from an application or database is stored temporarily in the main memory (RAM) of multiple servers or nodes within a distributed computing environment. This approach aims to improve data access speed and scalability by reducing the latency associated with fetching data from disk-based storage systems like databases or file systems.

Now we will look into the DAX – what it does

DynamoDB Accelerator (DAX)

- In-memory distributed cache for DynamoDB
- Caches reads, queries and individual objects automatically
- Microseconds latency for cached items
- Doesn't require application code changes (managed on DB side)
- Solves the Hot Key Problem (too many reads from a single partition)
- Default 5 min TTL (Time to Live)
- Encryption at rest using KMS
- DAX is good for automatic caching of reads and queries made to a DynamoDB table. To cache aggregation results, ElastiCache needs to be used.
- Enable DAX and provision nodes for the DAX cluster
- Multi-AZ recommended (min 3 nodes) in production

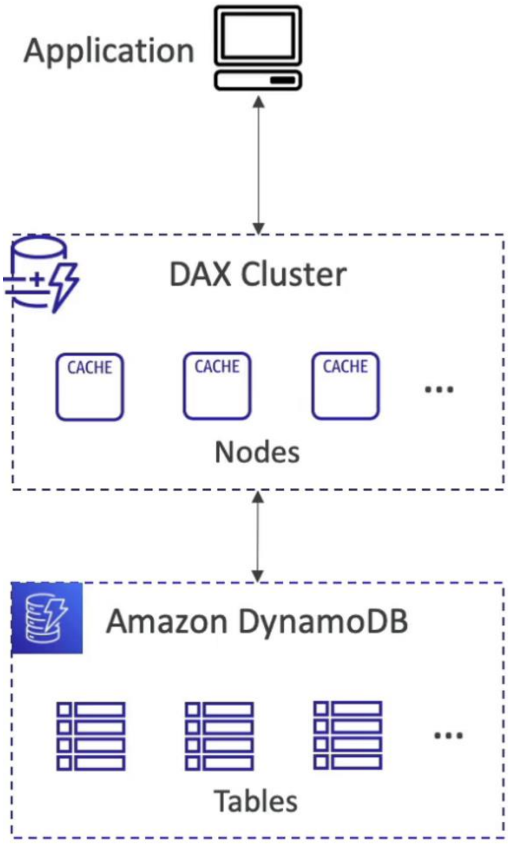
Max 10 nodes in a clusters

Node types:

- r-type: fixed resource with always-capacity (recommended for higher throughput)

- t-type: baseline performance with burst capability (recommended for lower throughput)

- DAX needs IAM permissions to access DynamoDB table (created automatically)
- DAX is expensive (should not be used in low-cost applications)


Application
DAX Cluster
CACHE
CACHE
CACHE
...
Nodes
Amazon DynamoDB
...
Tables

In DynamoDB, ***"burst capability"*** refers to the ability of a table to handle temporary spikes in read or write traffic beyond its provisioned capacity. This is made possible by the use of "burst credits," which allow DynamoDB tables to temporarily exceed their provisioned throughput limits.

Burst Capacity in DynamoDB

DynamoDB tables are provisioned with a certain number of read and write capacity units, which determine the maximum number of reads and writes per second the table can handle. However, workloads can be unpredictable, and there can be sudden spikes in demand. To accommodate these temporary increases in traffic, DynamoDB provides burst capacity.

How Burst Capacity Works

Burst Credits: When your DynamoDB table operates below its provisioned capacity, it accumulates burst credits. These credits can be thought of as a buffer that you can use to handle sudden increases in traffic. Each DynamoDB partition accumulates up to 300 seconds of unused capacity.

Using Burst Credits: When a spike in traffic occurs, the table can use these accumulated burst credits to handle the excess load. This allows the table to temporarily exceed its provisioned capacity, ensuring that your application remains responsive during traffic spikes.

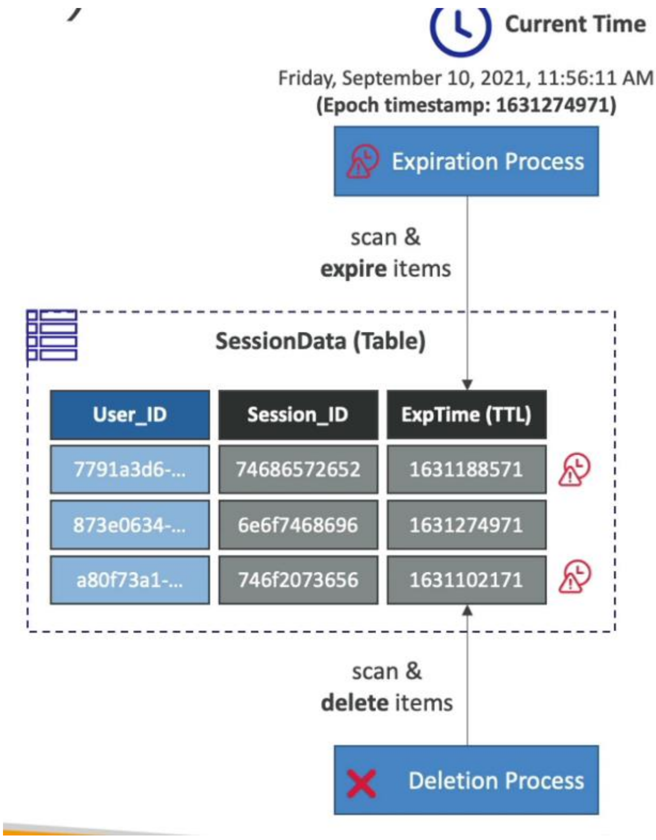
Limitations: Burst capacity is not unlimited. The amount of burst credits available depends on the partition's previous usage. If the table has been using its full provisioned capacity consistently, it will have fewer burst credits available. Once the burst credits are exhausted, any additional requests that exceed the provisioned capacity will be throttled.

“*If your table is not using its full provisioned capacity, the unused capacity units are converted into burst credits*. For instance, if your table is provisioned for 100 WCUs but is only using 50 WCUs, the remaining 50 WCUs per second are considered unused.”

Time To Live (TTL)

In DynamoDB, Time to Live (TTL) is a mechanism that allows you to set a specific timestamp for each item in your table, indicating when that item should be considered expired and eligible for deletion.

- Option to delete items after their expiry timestamp
- The TTL attribute must be `Number` type with Unix Epoch timestamp value
- Expired items are deleted within 48 hours (might appear in queries after expiry)
- Deletion through TTL does not consume WCU (no extra cost)
- Expired items are deleted from secondary indexes
- DynamoDB stream can be used to recover deleted items
- Simple Terms: A process within DynamoDB performs a scan on the table regularly to check the TTL timestamps and mark the items as expired. Another process performs a scan to find the expired items and deletes them.



Time To Live (TTL) Diagram

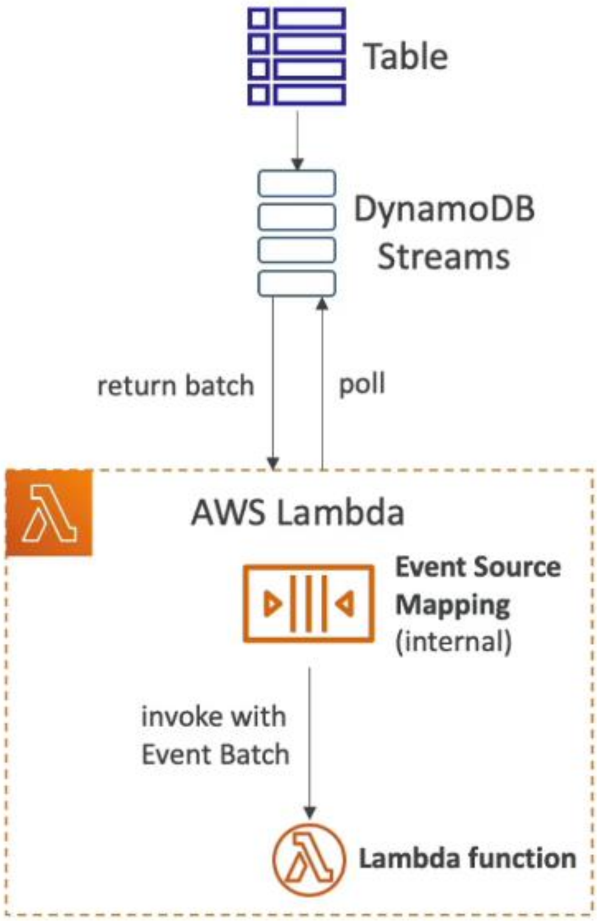
DynamoDB Streams

DynamoDB Streams capture a time-ordered sequence of item-level changes (insert, update, delete) in a DynamoDB table and provide a reliable way to process and react to these changes in real-time.

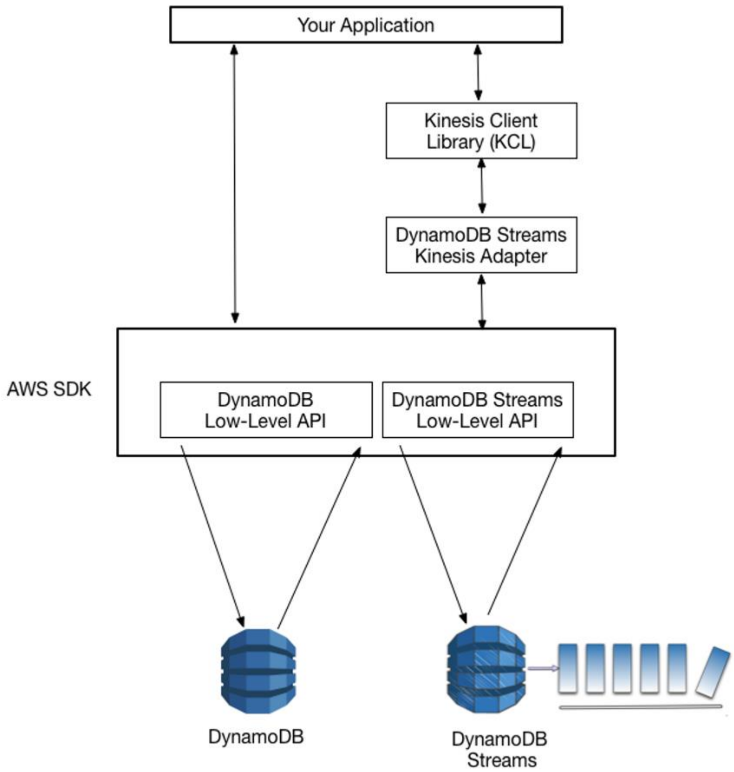
- Ordered stream of item-level modifications (create / update / delete)
- Data retention: **1 day** (data retention refers to the period during which the stream records are kept and available for processing after being written to the stream. When the data retention is set to 1 day, it means that the stream records ***will be retained for 24 hours from the time they are added to the stream***.)

Stream records can be:

➔ Read directly by AWS Lambda (using Event Source Mapping)



➔ Read by KCL as a stream (requires Kinesis Adapter)
The Kinesis Adapter is the recommended way to consume streams from DynamoDB for real-time processing. The KCL simplifies coding by providing useful abstractions above the low-level Kinesis Streams API.



Stream record contents:

- `KEYS_ONLY` - key attributes of the modified item
- `NEW_IMAGE` - modified item
- `OLD_IMAGE` - old item
- `NEW_AND_OLD_IMAGES` - both new and old items

Here's how DynamoDB Streams can help:

Enable DynamoDB Streams on the relevant tables (users, orders, products).

- Create Lambda Functions Replicate changes to another DynamoDB table in a different region.
- Update the Elasticsearch index with new or changed data.
- Send notifications using Amazon SNS or another messaging service.

Use Cases for DynamoDB Streams

Data Replication:

- **Cross-Region Replication:** Synchronize data across multiple AWS regions to provide high availability and disaster recovery.
- **Cross-Account Replication:** Share data across different AWS accounts for collaboration or backup purposes.

Real-Time Analytics:

- **Real-Time Data Processing:** Perform analytics on data changes as they happen. For example, updating real-time dashboards, monitoring application metrics, or detecting trends.
- **Event-Driven Architectures:** Trigger downstream processing workflows in real-time, such as updating search indices, sending notifications, or performing computations.

Auditing and Logging:

- **Change Tracking:** Maintain a history of changes for auditing purposes. Track who made changes, what changes were made, and when they occurred.
- **Compliance:** Ensure compliance with regulatory requirements by maintaining a log of data changes.

Practice and Read

Numerical Questions for the Dynamo DB

https://lisireddy.medium.com/aws-dynamodb-numerical-questions-e4ea3d08cb0d

Scenario based Questions for the Dynamo DB

https://lisireddy.medium.com/aws-dynamodb-scenario-based-questions-1534d5fe812b

Use case-based Questions for the Dynamo DB

https://lisireddy.medium.com/aws-dynamodb-use-case-based-questions-ea7943ff2c81

Wish you the best …! Happy Learning ..!

Yours' Love (@lisireddy across all the platforms)