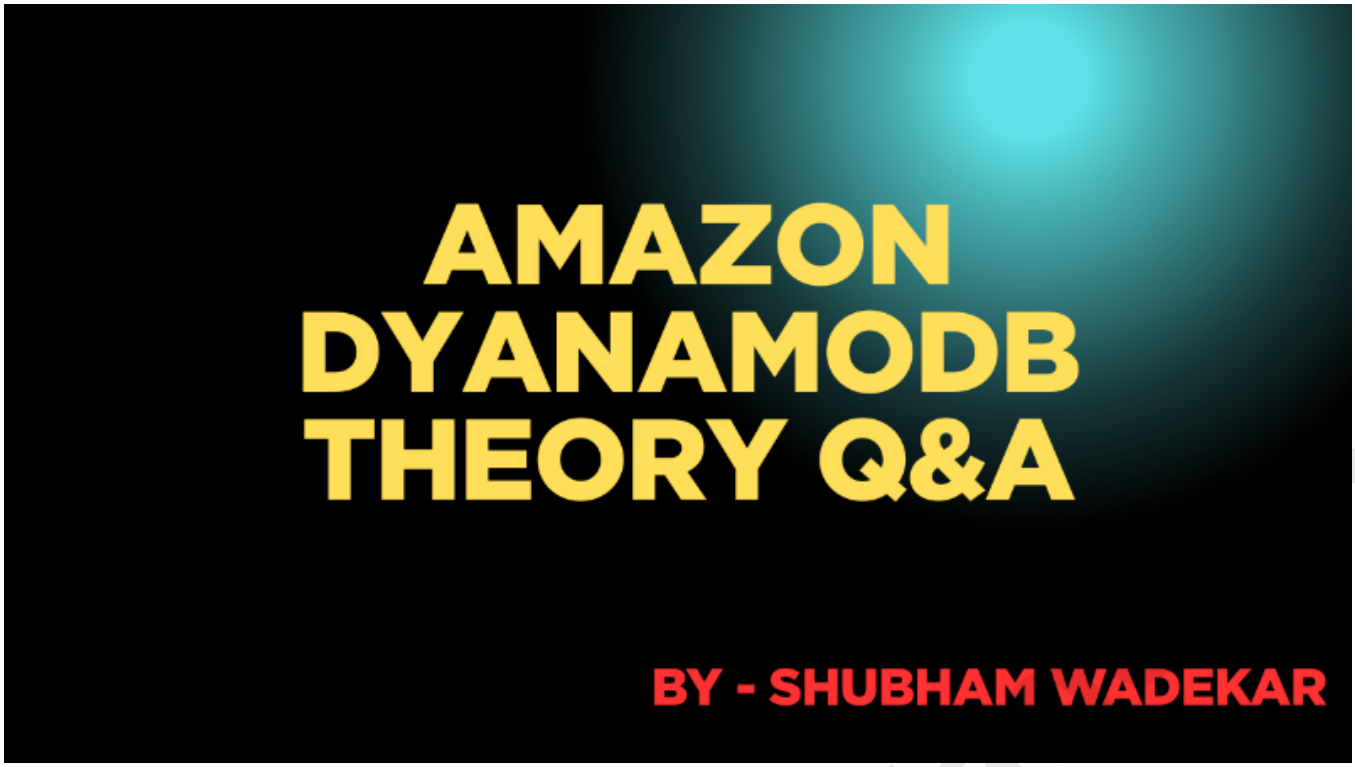

1. What is Amazon DynamoDB and how does it differ from traditional relational databases?

Amazon DynamoDB is a fully managed NoSQL database service on AWS. It's designed for applications that need very fast performance, typically in milliseconds, no matter how much data we store or how many requests are coming in. AWS automatically takes care of scaling, replication across multiple availability zones, backups, and patching, so as a developer or data engineer I don't worry about infrastructure.

The biggest difference from a traditional relational database is how the data is organized and queried. In relational databases, we have fixed schemas with tables, rows, and columns, and we can use SQL to run complex joins, aggregations, and transactions across tables. Scaling is usually vertical meaning we increase the server size when the data grows.

In DynamoDB, the approach is very different. It's schema-less, which means each item can have different attributes. Instead of relying on joins or complex queries, we design our tables based on access patterns we think about how the application will query the data, and then structure the tables to serve those queries quickly. It scales horizontally, meaning AWS spreads the data across multiple servers using partition keys, so performance stays fast even when data grows to terabytes or petabytes.

Another difference is cost. In relational databases, we usually pay for the server capacity regardless of how much we use it. In DynamoDB, we pay for what we consume: read/write capacity or on-demand requests plus storage.

2. Could you explain how DynamoDB tables look like (tables, items, and attributes)?

In DynamoDB, the structure is very simple but flexible:

- A table is the top-level container. For example, I can have a table called "Users".
- Inside the table, we have items. Each item is like a row in a relational database, but here it can be very flexible. For example, one item might represent user UserID=1, another UserID=2.
- Each item is made up of attributes, which are the data fields. For example, UserID, Name, Email.

The key difference from relational databases is that not all items need to have the same attributes. One user might have an Address attribute, while another user might not have it. This flexibility is why DynamoDB is schema-less.

Think of it like this:

- Table = folder
- Item = file inside the folder
- Attribute = fields inside the file

So the structure is very flat, and it gives me freedom to store data without worrying about schema changes like in relational databases.

3. What is a primary key in DynamoDB?

The primary key in DynamoDB is how we uniquely identify each item in a table. Without it, DynamoDB wouldn't know how to quickly look up or store data.

There are two types of primary keys:

- Simple primary key: This uses only one attribute, called the partition key. Example: In a "Users" table, the UserID can be the primary key. Every user must have a unique UserID.
- Composite primary key: This uses two attributes partition key + sort key. Example: In an "Orders" table, the partition key could be CustomerID and the sort key could be OrderID. This way, the same customer can have multiple orders, but the combination of CustomerID + OrderID is always unique.

The primary key is also critical for performance. DynamoDB internally uses the partition key to decide where to store the data across different servers (partitions). So designing the right primary key is one of the most important tasks when working with DynamoDB.

4. What is a sort key in DynamoDB?

A sort key is the second part of a composite primary key. It works together with the partition key to uniquely identify an item.

For example, in a table called "Orders":

- Partition key = CustomerID
- Sort key = OrderDate

This means all the orders from one customer are grouped under the same partition (because of CustomerID), and then ordered by the sort key (OrderDate).

The sort key also enables queries within a partition. For example, I can easily ask DynamoDB:

- "Get all orders for CustomerID=123 in the last 30 days."
- Or "Get the most recent order for CustomerID=123."

Without a sort key, I can only fetch items by a single partition key, which limits flexibility. With a sort key, I can model one-to-many relationships in DynamoDB.

5. What is the difference between partition keys and composite keys?

A partition key is just one attribute that uniquely identifies an item in the whole table. For example, in a "Users" table, the partition key could be UserID. Every UserID must be unique across the table.

A composite key is a combination of two attributes: partition key + sort key. Together they uniquely identify an item. For example, in an "Orders" table:

- Partition key = CustomerID
- Sort key = OrderID

This allows multiple items to share the same partition key (multiple orders from one customer), but the sort key ensures uniqueness within that partition.

The main difference is:

- Partition key alone = uniqueness across the table.
- Composite key = uniqueness within a group of items under the same partition.

Composite keys are very powerful when you want to model relationships like one customer → many orders, or one device → many events.

6. What are the advantages and disadvantages of using DynamoDB?

Advantages:

- Fully managed: I don't have to manage servers, replication, scaling, or backups AWS takes care of it.
- High performance: It provides single-digit millisecond latency even when data grows huge.
- Scalability: DynamoDB scales horizontally, so it can handle millions of requests per second without me changing the infrastructure.
- Flexibility: It's schema-less, meaning I can store different attributes for different items without worrying about schema changes.
- High availability: Data is automatically replicated across multiple availability zones.
- Integrated features: It supports global tables for multi-region replication, TTL (time-to-live) for expiring records automatically, and streams for real-time event-driven processing.

Disadvantages:

- Query limitations: Unlike relational databases, I cannot run complex joins, aggregations, or ad-hoc queries easily.
- Access pattern design: I must design tables carefully up front based on how the application will query the data. If access patterns change later, redesigning can be difficult.
- Cost predictability: If not optimized, read/write costs can become high, especially with bad partition key design or small, frequent queries.
- Limited transaction support: Transactions are supported, but they are not as powerful as in relational databases.

7. What are the common use cases of DynamoDB?

DynamoDB is used whenever we need high performance at scale with flexible schema. Common use cases include:

- E-commerce applications: storing product catalogs, customer profiles, shopping carts, and order history.
- Gaming: leaderboards, player sessions, game state, and player profiles.
- IoT: storing sensor data, device events, or telemetry data from millions of devices.
- Web/mobile apps: user authentication, session management, chat applications, and activity feeds.
- Financial services: fraud detection systems, transaction history, or real-time analytics.
- Serverless applications: DynamoDB integrates very well with Lambda and Step Functions for event-driven pipelines.

Basically, DynamoDB is best suited where predictable, fast lookups and writes are more important than running complex queries.

8. What is DynamoDBMapper, and in which language is it available?

DynamoDBMapper is a high-level, object persistence API provided by AWS. It helps developers interact with DynamoDB in an object-oriented way instead of writing low-level DynamoDB API calls.

It is available in Java as part of the AWS SDK. With DynamoDBMapper, I can map my Java objects (classes) directly to DynamoDB tables. For example, if I have a User class in Java, I can annotate it with DynamoDB annotations, and DynamoDBMapper can save, load, update, or delete objects directly in DynamoDB without me writing query code manually.

This makes it very convenient for developers who prefer working with objects rather than database commands.

9. What are NoSQL databases, and where does DynamoDB fit in?

NoSQL databases are non-relational databases that are designed to handle large volumes of unstructured or semi-structured data, with high scalability and flexibility. Unlike relational databases, they don't follow rigid schemas, and they avoid heavy joins and transactions.

There are different types of NoSQL databases:

- Key-Value stores (like Redis, DynamoDB)
- Document stores (like MongoDB)
- Column-family stores (like Cassandra)
- Graph databases (like Neo4j)

DynamoDB fits into the key-value and document store category. At its core, every item is stored and retrieved using a primary key, but it also supports storing JSON-like documents as attributes, so it can act as a document database too.

So DynamoDB is Amazon's managed NoSQL offering that combines the strengths of key-value and document models.

10. How does DynamoDB handle Read/Write Capacity?

DynamoDB handles read and write capacity in two main ways: Provisioned Mode and On-Demand Mode.

- Provisioned mode: Here I specify how many reads and writes per second my application needs (Read Capacity Units – RCUs, Write Capacity Units – WCUs). DynamoDB will reserve that capacity. If I exceed it, I may get throttled. This mode is cost-effective when I have predictable traffic.
- On-demand mode: Here I don't need to specify capacity. DynamoDB automatically adjusts to the traffic, and I pay per request. This is useful for unpredictable or spiky workloads.

Reads and writes are measured differently:

- 1 RCU = one strongly consistent read per second for an item up to 4 KB in size.
- 1 WCU = one write per second for an item up to 1 KB in size.

So if I'm designing a table, I calculate capacity based on item size and traffic patterns.

DynamoDB also has auto-scaling for provisioned mode, where it can adjust RCUs and WCUs up or down based on traffic. This gives flexibility to handle traffic changes without manual adjustments.

11. What are Read Capacity Units (RCU) and Write Capacity Units (WCU) in DynamoDB?

In DynamoDB, capacity is measured in units called RCUs and WCUs. They define how much read and write throughput a table can handle.

- **Read Capacity Unit (RCU):**
  1 RCU = 1 strongly consistent read per second for an item up to 4 KB in size.
  If I use eventually consistent reads, I get double the throughput (so 1 RCU = 2 eventually consistent reads per second).
- **Write Capacity Unit (WCU):**
  1 WCU = 1 write per second for an item up to 1 KB in size.

For example, if my items are 2 KB in size and I want 100 strongly consistent reads per second, I need:

- (2 KB ÷ 4 KB) = 1 RCU per read → 100 RCUs total.

If my items are 500 bytes and I want 50 writes per second, I need:

- (500 bytes ÷ 1 KB) = 1 WCU per write → 50 WCUs total.

So, RCUs and WCUs help me calculate and reserve the exact throughput I need.

12. What are the two Read/Write Capacity Modes (Provisioned vs On-Demand)?

DynamoDB offers two capacity modes:

1. **Provisioned Mode:**
    - I define how many RCUs and WCUs my application needs.
    - DynamoDB reserves that capacity, and I pay for it whether I use it fully or not.
    - If I exceed the capacity, requests may be throttled.
    - Best for steady, predictable workloads.
2. **On-Demand Mode:**
    - I don't have to specify RCUs or WCUs.
    - DynamoDB automatically scales up or down based on the traffic.
    - I pay per request (read/write).
    - Best for unpredictable or spiky workloads.

In short: provisioned = fixed reserved throughput, on-demand = pay-per-use with automatic scaling.

13. What is Provisioned Capacity in DynamoDB?

Provisioned capacity means I tell DynamoDB in advance how many reads and writes per second I need (in terms of RCUs and WCUs). DynamoDB then allocates that capacity for my table.

This is cost-effective when:

- My application has steady, predictable traffic.
- I know the peak traffic patterns (for example, an e-commerce app that always gets heavy traffic during daytime and lighter traffic at night).

The risk is that if traffic suddenly spikes beyond what I provisioned, requests may get throttled unless I enable auto scaling.

So provisioned mode gives control over costs, but I need to plan capacity carefully.

14. What is DynamoDB Auto Scaling and how does it work?

DynamoDB Auto Scaling is a feature that automatically adjusts the provisioned RCUs and WCUs for a table based on traffic patterns.

How it works:

- I define a target utilization (for example, 70%).
- Auto scaling monitors my table's traffic through CloudWatch metrics.
- If usage goes above the target, it increases capacity.
- If usage goes below the target for some time, it reduces capacity.

Example: If I provision 100 RCUs with 70% utilization, auto scaling will try to keep usage around 70 RCUs. If traffic spikes and my table needs 200 RCUs, auto scaling can increase capacity up to the max limit I set.

This way, I don't need to constantly monitor and manually adjust capacity, and I can balance performance with cost.

15. What is throttling in DynamoDB and how can you handle it?

Throttling happens when the application tries to read or write more than the provisioned (or available) capacity of the table. DynamoDB rejects the extra requests with a ProvisionedThroughputExceededException.

For example, if my table is provisioned for 100 WCUs but my app suddenly tries to write 200 items per second, 100 of those writes may get throttled.

Ways to handle throttling:

- **Use auto scaling:** So capacity adjusts automatically during spikes.
- **Use on-demand mode:** Removes throttling issues since DynamoDB adjusts capacity automatically.
- **Exponential backoff and retries:** When throttled, retry after a short, increasing delay.
- **Distribute traffic better:** Sometimes throttling happens because of "hot partitions" (too many requests going to the same partition key). Choosing a better partition key design can spread traffic more evenly.
- **Batch operations:** Combine multiple items in a single request (BatchWrite or BatchGet) to use capacity more efficiently.

Throttling isn't always bad it's DynamoDB's way of protecting performance. The key is to design the table and capacity mode to handle traffic patterns effectively.

16. Explain the difference between eventual consistency and strong consistency in DynamoDB.

When I read data from DynamoDB, I can choose the type of consistency:

- **Strongly Consistent Reads:**
  The read always returns the latest data that was successfully written. It guarantees that once a write is confirmed, any strongly consistent read will see it immediately.
  Example: If a customer updates their shipping address and I do a strong read right after, I'll see the updated address for sure.

- **Eventually Consistent Reads:**
  The read might not immediately reflect the latest write because data is replicated across multiple servers in the background. After a short time (usually milliseconds), all copies will catch up.
  Example: If the same customer updates their address and I do an eventual read immediately, I might still see the old address for a brief moment.

Strong consistency gives accuracy but costs more in capacity (1 RCU = 4KB per read). Eventual consistency is cheaper (double the reads per RCU) and faster for distributed scaling.

17. How does DynamoDB prevent data loss?

DynamoDB is designed for durability and high availability. It prevents data loss in several ways:

- Multi-AZ replication: Every piece of data is automatically replicated across at least 3 availability zones in a region. So if one zone fails, data is still safe.
- Synchronous replication: When I write an item, DynamoDB confirms success only after it is safely replicated to multiple storage nodes.
- Backups: DynamoDB provides on-demand backups and point-in-time recovery (PITR), which allows me to restore data to any second in the past 35 days.
- Global Tables: For cross-region resilience, DynamoDB can replicate data to multiple AWS regions, so even if one region goes down, another can serve the data.
- Streams: DynamoDB Streams capture every change (insert, update, delete), so I can use them for auditing or replaying events if needed.

So data durability is built into DynamoDB by default, without me having to configure complex replication setups.

18. What are Global Secondary Indexes (GSI) and Local Secondary Indexes (LSI) in DynamoDB?

Indexes in DynamoDB are like alternate ways to query the data apart from the main primary key.

- **Global Secondary Index (GSI):**
  A GSI lets me query on a different partition key and/or sort key than the base table. It works across all partitions.
  Example: If my base table's primary key is UserID, but I want to also query by Email, I can create a GSI on Email.
- **Local Secondary Index (LSI):**
  An LSI lets me define an alternate sort key for the same partition key of the base table.
  Example: If my base table's key is CustomerID (partition) + OrderID (sort), I can create an LSI with CustomerID (same partition) but a different sort key, like OrderDate.

So GSIs give more flexibility since I can query using a completely different partition key, while LSIs only work within the same partition key but let me sort/filter differently.

19. What is the difference between LSI and GSI?

The differences can be summarized like this:

- **Partition key:**
  LSI must use the same partition key as the base table.
  GSI can have a completely different partition key.
- **Creation:**
  LSIs must be defined at table creation time.
  GSIs can be created any time after the table is created.
- **Number allowed:**
  A table can have up to 5 LSIs.
  A table can have up to 20 GSIs.
- **Size limits:**
  LSI shares the 10GB size limit per partition with the base table.
  GSI has its own throughput and storage, separate from the base table.
- **Performance:**
  LSIs always provide strongly consistent reads.
  GSIs only provide eventually consistent reads by default (strong consistency is not supported).

So LSIs are more restrictive but useful for alternate sorting, while GSIs are more flexible and powerful for new access patterns.

20. What are DynamoDB Projections, and why are they useful?

When I create an index (GSI or LSI), I can choose which attributes from the base table should be copied into that index. This selection is called a projection.

There are three types of projections:

- KEYS_ONLY: Only the partition and sort keys are projected.
- INCLUDE: Keys plus a selected list of non-key attributes.
- ALL: Every attribute from the base table is copied into the index.

Why useful:

- Projections help optimize cost and performance. If I only need a few attributes for queries, I don't have to project all of them, which saves storage and reduces read costs.
- For example, in an "Orders" table, if my index is on CustomerID, and I only need OrderDate and OrderStatus, I can project just those attributes instead of the whole order record.

So projections are a way to design indexes efficiently by including only the attributes needed for queries.

21. How do you perform a Query operation in DynamoDB? Provide an example.

A Query operation in DynamoDB is used to find items based on their primary key (partition key, and optionally sort key conditions). It is efficient because DynamoDB looks up items directly using the key values.

Steps for a Query:

- I must provide the partition key value.
- I can optionally provide sort key conditions (like equals, greater than, begins_with, between, etc.).
- I can also use filters, but filters are applied after fetching results, so they don't reduce read cost.

Example:
Suppose I have an "Orders" table with:

- Partition key: CustomerID
- Sort key: OrderDate

If I want to get all orders for customer C123 in January 2025, I would run a query like:

- Partition key = C123
- Sort key between 2025-01-01 and 2025-01-31

DynamoDB will return only those items. This is fast because it uses indexes directly rather than scanning the whole table.

22. How does the Scan operation differ from Query?

The Scan operation goes through every item in the table (or index) and returns items that match optional filter conditions. It does not use keys for direct lookup.

Key differences:

- Query requires a partition key and is efficient.
- Scan does not require a partition key, but it reads the entire table, making it much slower and more expensive for large datasets.

Example:
If I scan the "Orders" table to find all orders with OrderStatus = Shipped, DynamoDB has to look at every item, check its status, and then return matches.

So in practice:

- Use Query whenever possible.
- Scan should be avoided on large tables, or used carefully with pagination, filters, or parallel scans.

23. List 5 ways to fetch data from DynamoDB.

There are multiple ways to read or fetch data:

1. **GetItem** – Retrieve a single item by its primary key. Example: Get user profile by UserID.
2. **BatchGetItem** – Retrieve multiple items from one or more tables in a single request (up to 100 items).
3. **Query** – Fetch items efficiently using partition key and optional sort key conditions.
4. **Scan** – Go through the entire table (less efficient, should be avoided for large tables).
5. **Using Secondary Indexes (GSI or LSI)** – Fetch items using alternate keys (like querying by Email if the base table key is UserID).

These options allow flexibility depending on whether I know the primary key or I need to use other attributes.

24. What is the concept of item collections in DynamoDB?

In DynamoDB, an item collection is the group of all items that share the same partition key value.

Example:
In an "Orders" table with partition key CustomerID and sort key OrderDate:

- All orders belonging to CustomerID = C123 form one item collection.
- That collection might have 5 items (if the customer made 5 orders).

Why important:

- LSIs (Local Secondary Indexes) work only within an item collection.
- DynamoDB has a 10 GB size limit per item collection (per partition key). If one partition key has too many items, it can hit that limit.

So item collections represent logical groups of items under the same partition key.

25. How do secondary indexes improve query performance?

Secondary indexes allow me to query the data using alternate keys apart from the base table's primary key. Without them, I would be forced to rely only on partition key + sort key, or use expensive scans.

Ways they improve performance:

- I can query using different attributes (like Email or OrderStatus) instead of only UserID.
- Queries become more flexible and still efficient, since indexes use a similar key-based lookup internally.
- They reduce the need for scans, saving cost and improving speed.
- GSIs can even be provisioned with their own capacity, so heavy query workloads can be shifted away from the base table.

Example:
If my base table key is UserID, but I often need to fetch users by Email, creating a GSI on Email allows me to query efficiently. Without the GSI, I would have to scan the entire table, which is very slow.

26. What are Indexes and Secondary Indexes in DynamoDB?

In DynamoDB, an index is an additional data structure that lets me query the table using keys other than the base table's primary key. It works just like an alternate "view" of the table with different keys.

There are two types of secondary indexes:

- **Local Secondary Index (LSI):** Same partition key as the base table but a different sort key. Useful for querying items in the same partition with alternate sorting or filtering. Must be created at table creation time.
- **Global Secondary Index (GSI):** Has its own partition key and sort key, independent of the base table. Can be created anytime after the table is created. Useful for querying completely different attributes like Email or Status.

Indexes don't duplicate the full table, but they keep the projected attributes you choose. This allows efficient queries without scanning the base table.

27. What are DynamoDB Streams?

DynamoDB Streams is a feature that captures a time-ordered sequence of changes (insert, update, delete) made to items in a DynamoDB table.

Think of it like a changelog: whenever data changes, a record of that change is written to the stream. The stream can then be consumed by other services like AWS Lambda, Kinesis, or custom apps for real-time processing.

Example use cases:

- Sending a notification when an order status changes.
- Maintaining an audit log of all changes.
- Replicating data to another system in near real time.

28. How do DynamoDB Streams work internally (records, shards, images)?

Internally, DynamoDB Streams is built on a model similar to Amazon Kinesis:

- **Stream Records:** Each record represents a change (insert/update/delete) on an item. The record contains metadata like table name, event type, and optionally the item data (before and/or after the change).
- **Shards:** Streams are divided into shards for scaling. Each shard contains a sequence of records, and multiple shards allow parallel processing of large volumes of changes.
- **Images:** Images represent what the item looked like before and/or after the change. Depending on the stream setting, I can capture only keys, or both old and new images.

Consumers like Lambda read from shards in order and process these records. Records remain in the stream for 24 hours by default.

29. What are the different stream view types in DynamoDB Streams?

When enabling a stream on a table, I can choose what kind of information (image) is written into the stream. The options are:

1. **KEYS_ONLY** – Only the primary key attributes of the item that changed.
   Example: Just UserID if a user item was updated.
2. **NEW_IMAGE** – The entire item as it looked after the change.
   Example: The updated user record with all new values.
3. **OLD_IMAGE** – The entire item as it looked before the change.
   Example: The user record before an update or delete.
4. **NEW_AND_OLD_IMAGES** – Both the new and old item images.
   Example: Useful for auditing or comparing old vs new values.

So depending on my use case (auditing, replication, notifications), I choose the view type.

30. What are the key features of DynamoDB Streams?

Some important features are:

- **Time-ordered:** Stream records are guaranteed to appear in the order changes happened.
- **Durability:** Records are stored for 24 hours, giving enough time for consumers to process.
- **Integration with Lambda:** I can trigger AWS Lambda functions automatically on stream events, making real-time event-driven pipelines easy to build.
- **Flexible views:** I can choose to store only keys, old image, new image, or both.
- **Scalable:** Streams use shards to handle high change volumes and allow parallel consumption.
- **Cross-region replication:** Streams + Lambda (or Kinesis) can be used to replicate DynamoDB tables across regions.
- **Near real-time processing:** Changes are usually available in the stream within seconds.

In short, Streams turn DynamoDB from just a storage system into a real-time event-driven system.

31. What are the real-world applications of DynamoDB Streams?

DynamoDB Streams are used in many real-world applications where changes to data need to trigger further actions in near real time. Some examples:

- Audit logging: Every insert, update, or delete can be captured in a stream and written to an audit log table or S3 for compliance.
- Notifications: When an order status changes (e.g., "Shipped"), a Lambda triggered by the stream can send an email or push notification.
- Data replication: Changes in a DynamoDB table in one region can be streamed and applied to another region for disaster recovery or multi-region setups.
- Analytics pipelines: Streams can feed changes into services like Kinesis, S3, or Redshift for reporting or machine learning.
- Cache invalidation: If I'm caching DynamoDB data (e.g., in Redis), Streams can notify me when data changes so I can invalidate or refresh the cache.

32. How can DynamoDB Streams be used in Traditional Architecture?

In a traditional (non-serverless) architecture, DynamoDB Streams can still integrate with backend systems:

- Message bus integration: Streams can publish data changes into systems like Kafka or SQS so that legacy services can consume them.
- ETL pipelines: Streams can be connected to Lambda or Kinesis Firehose to move data into warehouses like Redshift or data lakes on S3.
- Microservices communication: If one service updates DynamoDB, Streams can notify other services to keep them in sync without direct coupling.
- Legacy application sync: A traditional monolithic app may still rely on SQL databases. Streams can replicate DynamoDB changes into RDS so both systems stay consistent.

So even in non-serverless setups, Streams act as a bridge between DynamoDB and other systems.

33. What are the use cases for DynamoDB Streams in real-time pipelines?

Streams are a natural fit for real-time, event-driven pipelines. Use cases include:

- **Real-time analytics:** Capturing clickstream or IoT device data in DynamoDB, and using Streams to send updates into Kinesis → Redshift/S3 for dashboards.
- **Fraud detection:** As transactions are written into DynamoDB, Streams can send them to a real-time ML model for anomaly detection.
- **Search indexing:** When new content is added or updated, Streams can trigger Lambda to update an Elasticsearch (OpenSearch) index.
- **Monitoring and alerting:** Streams can feed CloudWatch metrics or trigger alerts when unusual patterns (like too many failed logins) are detected.
- **Event-driven workflows:** For example, when a payment is recorded in DynamoDB, Streams can trigger downstream processes like invoice generation or shipment scheduling.

In short, Streams connect DynamoDB with other real-time systems.

34. What is DynamoDB Accelerator (DAX) and how does it improve performance?

DAX is an in-memory caching layer for DynamoDB provided by AWS. It sits between the application and DynamoDB, caching frequently accessed items.

How it improves performance:

- It reduces read latency from milliseconds (standard DynamoDB) to microseconds (because data is served from memory).
- It's fully managed and highly available, so I don't need to build my own Redis or Memcached layer.
- It supports the same DynamoDB API, so applications don't need much code change I just point my SDK to DAX instead of DynamoDB.

Example: If my application is repeatedly reading the same product catalog items, instead of hitting DynamoDB every time, DAX will return results from its memory cache instantly. This improves user experience and reduces DynamoDB read costs.

35. What are DynamoDB Global Tables and how do they enable multi-region replication?

DynamoDB Global Tables allow me to create a single table that is replicated automatically across multiple AWS regions.

Key points:

- It provides multi-master replication I can read and write to the table in any region, and DynamoDB keeps them in sync.
- It enables low-latency access for global applications because users can connect to the region nearest to them.
- It improves disaster recovery since data exists in multiple regions.
- It is built on top of DynamoDB Streams, which capture changes in one region and apply them to other regions.

Example: For a global e-commerce site, customers in the US can write to a table in us-east-1, while customers in Europe can write to the same table in eu-west-1. Both regions stay consistent automatically.

Global Tables are useful for multi-region apps, DR setups, and worldwide services that need both resilience and fast local access.

36. What is the DynamoDB Local, and when would you use it?

DynamoDB Local is a downloadable version of DynamoDB that runs on your local machine. It behaves just like the cloud DynamoDB service but stores data locally.

When to use it:

- **Development and testing:** I can test my applications without incurring AWS costs or needing internet connectivity.
- **Offline prototyping:** I can design and try out table structures and queries before deploying to AWS.
- **CI/CD pipelines:** It can be used in automated testing environments so tests don't depend on the live DynamoDB service.

It's important to remember that DynamoDB Local is for development only it doesn't provide the same scale, replication, or durability guarantees as the real service.

37. What is the maximum item size in Amazon DynamoDB?

The maximum size of a single item in DynamoDB is 400 KB, including all attribute names and values.

This limit applies whether the data is stored in base tables or secondary indexes. If my data is larger than 400 KB, I need to break it down into multiple items or store large blobs in S3 and just store references (like S3 object keys) in DynamoDB.

38. What are the DynamoDB pricing tiers (On-Demand vs Provisioned)?

DynamoDB pricing depends on which capacity mode I choose:

- **On-Demand Pricing:**
    - Pay-per-request model.
    - I don't specify capacity; DynamoDB scales automatically.
    - Best for unpredictable or spiky workloads.
    - Cost is calculated per million read/write request units.
- **Provisioned Pricing:**
    - I specify how many RCUs and WCUs my table should have.
    - I pay for that capacity whether I use it or not.
    - Auto Scaling can adjust capacity if traffic changes.
    - Best for steady and predictable workloads where I want to control cost.

In both modes, I also pay for storage, backups, streams, and global table replication.

39. What is Encryption at Rest in DynamoDB, and how does it work with KMS?

Encryption at Rest in DynamoDB means all the data stored in tables (and backups, streams, indexes) is automatically encrypted on disk. This protects data from unauthorized access to the physical storage.

How it works:

- By default, DynamoDB encrypts data at rst using AWS-managed keys in KMS (AWS Key Management Service).
- I can also choose to use a customer-managed KMS key if I need more control (for example, rotating keys, disabling them, or setting access policies).
- The encryption/decryption process is transparent to the application I don't need to change my code.

This ensures compliance with security standards and gives me flexibility to manage encryption keys the way my organization requires.

40. How do you implement transactions in DynamoDB?

DynamoDB supports ACID transactions, which means I can group multiple read and write operations so they succeed or fail together.

How it works:

- I use the TransactWriteItems API for multiple writes.
- I use the TransactGetItems API for multiple reads.
- Up to 25 items or 4 MB of data can be part of a single transaction.
- Transactions are strongly consistent and provide all-or-nothing behavior.

Example: In a banking application, if I need to transfer money from Account A to Account B:

1. Deduct amount from Account A.
2. Add amount to Account B.
   If either operation fails, both are rolled back, so balances remain correct.

Transactions are useful for critical workflows like financial systems, order processing, or inventory management.

41. How do you set up backups and Point-in-Time Recovery (PITR) in DynamoDB?

There are two kinds of backups: on-demand backups and continuous backups via PITR.

On-demand backup (good for full snapshots before big changes):

- Open the DynamoDB console, select the table, Backups, Create backup, give it a name, Create.
- It's a full backup of the table and indexes; it doesn't slow down traffic.
- To restore, choose the backup → Restore → pick a new table name. DynamoDB creates a new table from that snapshot. You can later swap traffic to the restored table.

Point-in-Time Recovery (good for "oops, roll back to 10:17:32 AM"):

- Open the table → Backups → Enable point-in-time recovery.
- DynamoDB then keeps a rolling 35-day history. You can restore to any second in that window.
- To restore, choose Restore to point in time, set the exact timestamp, provide a new table name, and confirm.
- Restores always create a new table; you don't overwrite the existing one. After checks, cut over your app to the new table.

Notes:

- Costs: you pay for stored backups and for restored table storage. PITR adds continuous backup charges.
- Restores bring back data and indexes; you may need to reconfigure features like auto scaling settings, alarms, and stream consumers on the new table.
- For very large tables, plan the cutover (read replicas, dual-write, or brief maintenance window).

42. How do you create a DynamoDB table using the AWS Management Console?

Simple step-by-step:

1. Go to DynamoDB console → Tables → Create table.
2. Enter Table name.
3. Define the Primary key:
    - Partition key is required.
    - Add a Sort key if you need a composite key and range queries later.
4. Choose Capacity mode:
    - On-demand for unpredictable traffic.
    - Provisioned if you know read/write needs; optionally enable auto scaling.
5. (Optional) Add secondary indexes:
    - LSI (must share partition key) or GSI (different key), and choose projection (Keys only, Include, All).

6. Turn on TTL if you want items to auto-expire (set the attribute name).
7. Turn on Point-in-Time Recovery if you want continuous backups.
8. Leave encryption (KMS) as default or pick a customer-managed key.
9. Create table. Wait until status is Active before using it.

43. Write a simple Python (Boto3) script to insert an item into a DynamoDB table.

```
import boto3
from botocore.exceptions import ClientError

dynamodb = boto3.resource("dynamodb", region_name="ap-south-1")
table = dynamodb.Table("Users")

item = {
  "UserID": "U12345",       # partition key
  "Email": "alex@example.com",
  "Name": "Alex Kumar",
  "Plan": "Pro",
  "CreatedAt": "2025-08-25T18:30:00Z"
}

try:
  # Optional: ensure we don't overwrite an existing item with same key
  resp = table.put_item(
    Item=item,
    ConditionExpression="attribute_not_exists(UserID)"
  )
  print("PutItem succeeded:", resp.get("ResponseMetadata", {}).get("HTTPStatusCode"))
except ClientError as e:
  if e.response["Error"]["Code"] == "ConditionalCheckFailedException":
    print("Item already exists.")
  else:
    raise
```

44. Write a code snippet to update an item in DynamoDB using Boto3.

```
import boto3
from botocore.exceptions import ClientError

dynamodb = boto3.resource("dynamodb", region_name="ap-south-1")
table = dynamodb.Table("Users")

key = {"UserID": "U12345"}  # existing item's key

try:
    resp = table.update_item(
        Key=key,
        UpdateExpression="""
            SET Email = :email,
                Plan = :plan,
                UpdatedAt = :ts
        """,
        ExpressionAttributeValues={
            ":email": "alex.new@example.com",
            ":plan": "Enterprise",
            ":ts": "2025-08-25T18:45:00Z"
        },
        ConditionExpression="attribute_exists(UserID)",  # don't create if missing
        ReturnValues="ALL_NEW"
    )
    print("Updated item:", resp["Attributes"])
except ClientError as e:
    if e.response["Error"]["Code"] == "ConditionalCheckFailedException":
        print("Item not found.")
    else:
        raise
```

45. Write a code snippet to delete an item from DynamoDB using Boto3.

```
import boto3
from botocore.exceptions import ClientError

dynamodb = boto3.resource("dynamodb", region_name="ap-south-1")
table = dynamodb.Table("Users")

key = {"UserID": "U12345"}

try:
  # Optional: protect against deleting if on a certain plan
  resp = table.delete_item(
    Key=key,
    ConditionExpression="attribute_not_exists(Plan) OR Plan <> :blocked",
    ExpressionAttributeValues={":blocked": "Enterprise"},
    ReturnValues="ALL_OLD"
  )
  old = resp.get("Attributes")
  if old:
    print("Deleted:", old)
  else:
    print("No item to delete.")
except ClientError as e:
  if e.response["Error"]["Code"] == "ConditionalCheckFailedException":
    print("Delete blocked by condition.")
  else:
    raise
```

46. Write a code example to scan a DynamoDB table and filter results.

Scan reads every partition; filters are applied after read. Always paginate to avoid timeouts and high RCUs.

```
import boto3
from boto3.dynamodb.conditions import Attr

dynamodb = boto3.resource("dynamodb", region_name="ap-south-1")
table = dynamodb.Table("Orders")

# Example: find ACTIVE orders created between two timestamps
filters = (
  Attr("Status").eq("ACTIVE") &
  Attr("CreatedAt").between("2025-08-01T00:00:00Z", "2025-08-25T23:59:59Z")
)

items, last_evaluated_key = [], None

while True:
  params = {
    "FilterExpression": filters,
    "ProjectionExpression": "OrderId, CustomerId, Status, CreatedAt",  # optional
    "Limit": 200  # page size
  }
  if last_evaluated_key:
    params["ExclusiveStartKey"] = last_evaluated_key

  resp = table.scan(**params)
  items.extend(resp.get("Items", []))
  last_evaluated_key = resp.get("LastEvaluatedKey")
  if not last_evaluated_key:
    break

print(f"Fetched {len(items)} items")
```

Tip: prefer Query over Scan when you can, because Scan reads the whole table or index.

47. Write a code snippet to batch write items into DynamoDB.

BatchWriter handles automatic batching, retries, and unprocessed items. It supports put and delete (not conditional writes).

```
import boto3

from datetime import datetime


dynamodb = boto3.resource("dynamodb", region_name="ap-south-1")

table = dynamodb.Table("Users")


users = [

  {"UserID": "U1001", "Email": "a@x.com", "CreatedAt": datetime.utcnow().isoformat() + "Z"},

  {"UserID": "U1002", "Email": "b@x.com", "CreatedAt": datetime.utcnow().isoformat() + "Z"},

  {"UserID": "U1003", "Email": "c@x.com", "CreatedAt": datetime.utcnow().isoformat() + "Z"},

]


with table.batch_writer(overwrite_by_pkeys=["UserID"]) as batch:

  for u in users:

    batch.put_item(Item=u)


# To delete in bulk:

# with table.batch_writer() as batch:

#   for k in keys_to_delete:

#     batch.delete_item(Key={"UserID": k})
```

Note: overwrite_by_pkeys upserts; if you must prevent overwrites, do single PutItem with a condition.

48. Write a code snippet to use a Global Secondary Index (GSI) to query DynamoDB.

You must specify IndexName and use KeyConditionExpression with the GSI's keys.

```
import boto3
from boto3.dynamodb.conditions import Key

dynamodb = boto3.resource("dynamodb", region_name="ap-south-1")
table = dynamodb.Table("Users")

# Assume a GSI named "EmailIndex" with partition key "Email" and sort key "CreatedAt"
resp = table.query(
  IndexName="EmailIndex",
  KeyConditionExpression=Key("Email").eq("alex@example.com") &
            Key("CreatedAt").between("2025-08-01T00:00:00Z", "2025-08-25T23:59:59Z"),
  ProjectionExpression="UserID, Email, CreatedAt",
  Limit=50,
  ScanIndexForward=False  # newest first if sort key is time
)

for item in resp.get("Items", []):
  print(item)
```

Note: GSIs return eventually consistent reads; design projections to include only fields you need.

49. Write a code example to implement a conditional write in DynamoDB.

Example: decrement inventory only if enough stock exists (prevents oversell).

```
import boto3
from botocore.exceptions import ClientError

dynamodb = boto3.resource("dynamodb", region_name="ap-south-1")
table = dynamodb.Table("Inventory")

sku, qty = "SKU-123", 2

try:
  resp = table.update_item(
    Key={"SKU": sku},
    UpdateExpression="SET Stock = Stock - :q, LastUpdatedAt = :ts",
    ConditionExpression="attribute_exists(Stock) AND Stock >= :q",
    ExpressionAttributeValues={
      ":q": qty,
      ":ts": "2025-08-25T18:30:00Z"
    },
    ReturnValues="UPDATED_NEW"
  )
  print("New stock:", resp["Attributes"]["Stock"])
except ClientError as e:
  if e.response["Error"]["Code"] == "ConditionalCheckFailedException":
    print("Not enough stock or item missing.")
  else:
    raise
```

Another common conditional write is an “insert only if not exists” using PutItem with attribute_not_exists(PK).

50. Write a code snippet to handle throttling in DynamoDB using retries or exponential backoff.

Use retries on ProvisionedThroughputExceededException or ThrottlingException with exponential backoff and jitter. Also consider enabling SDK-level retries.

```
import time, random
import boto3
from botocore.config import Config
from botocore.exceptions import ClientError

# Enable SDK retries (good baseline)
cfg = Config(retries={"max_attempts": 10, "mode": "standard"})
dynamodb = boto3.resource("dynamodb", region_name="ap-south-1", config=cfg)
table = dynamodb.Table("Orders")

def put_with_backoff(item, max_retries=8, base_delay=0.05):
  attempt = 0
  while True:
    try:
      return table.put_item(Item=item)
    except ClientError as e:
      code = e.response["Error"]["Code"]
      if code in ("ProvisionedThroughputExceededException", "ThrottlingException"):
        if attempt >= max_retries:
          raise
        sleep_for = (2 ** attempt) * base_delay + random.uniform(0, base_delay)
        time.sleep(sleep_for)
        attempt += 1
        continue
      else:
        raise

resp = put_with_backoff({"OrderId": "O-1001", "CustomerId": "C-9", "Amount": 499})
print("PutItem status:", resp["ResponseMetadata"]["HTTPStatusCode"])
```

Hints:

- Spread hot keys by choosing a good partition key to avoid hot partitions.
- For provisioned mode, enable auto scaling or move to on-demand for unpredictable spikes.
- Batch operations (BatchWrite/BatchGet) are efficient, but still implement backoff for unprocessed items.

51. What are the best practices for designing a DynamoDB data model?

Start from access patterns, not tables. List every query your app must do (who reads what, by which key, in what order), then design keys and indexes to serve those queries in O(1) lookups or single-partition range scans. Prefer one "single-table" design per bounded context instead of many small tables.

Practical guidelines:

- Choose a composite primary key (partition + sort) that naturally groups related items and lets you do range queries (e.g., all orders for a customer, latest events for a device).
- Denormalize deliberately. Store the data where you read it. Create "materialized views" as separate items/indexes to avoid joins.
- Use GSIs/LSIs for alternate query paths; keep projections minimal (only attributes you read via that index).
- Model one-to-many with a shared partition key and sorted prefixes in the sort key. Use hierarchical prefixes like TYPE#ID to co-locate related entities.
- Prefer ISO-8601 timestamps or epoch numbers in the sort key for time ordering.
- Keep items small (max 400 KB). Put large blobs in S3 and store the S3 key in DynamoDB.
- Add TTL to auto-expire ephemeral data (sessions, carts, events).
- Plan capacity early: on-demand for spiky/unknown traffic, provisioned + auto scaling for predictable workloads.
- Use conditional writes for correctness (optimistic locking with a version attribute).
- Secure by default: least-privilege IAM, at-rest encryption (KMS), and fine-grained access with IAM conditions if exposing directly to clients.

52. How do you model a many-to-many relationship in DynamoDB?
Create a mapping (junction) entity and store it in a way that supports both directions of the relationship.

Pattern 1: Adjacency list in a single table

- Put items like:
    - PK = USER#<UserId>, SK = PROJECT#<ProjectId>
    - PK = PROJECT#<ProjectId>, SK = USER#<UserId>
- Now you can query "all projects for a user" by PK=USER#... and "all users for a project" by PK=PROJECT#...

Pattern 2: One write, two query paths via a GSI

- Base item: PK = USER#<UserId>, SK = PROJECT#<ProjectId>
- GSI1 with PK = PROJECT#<ProjectId>, SK = USER#<UserId>
- Write one item; query by user through base table, and by project via the GSI. This reduces duplication.

Good practices:

- Use consistent prefixes (USER#, PROJECT#) for clear grouping and filtering.
- If the list can be very large, paginate and consider soft limits per partition (e.g., split by time buckets).
- Use conditional writes to avoid duplicate links and to maintain counts (e.g., a membershipCount on the project).

53. How do you efficiently handle time-series data in DynamoDB?
Partition by entity, sort by time, and bucket to avoid hot keys and 10-GB item-collection limits.

Core pattern:

- PK = DEVICE#<DeviceId>, SK = <YYYYMMDD>#<epoch or seq>
- Query recent data with SK BETWEEN, or ScanIndexForward=False to get "latest first".

Bucketing to distribute writes:

- Add a day (or hour) bucket into the sort key or into the partition key:
    - PK = DEVICE#<DeviceId>#D#<YYYYMMDD>, SK = <epoch or seq>
- This keeps partitions smaller and improves write distribution.

Operational tips:

- Use TTL to expire old data automatically.
- Stream to S3 (via Streams + Lambda/Firehose) for long-term, cheap storage and Athena analytics.
- Project only needed attributes into GSIs for dashboards (e.g., status, metric aggregates).
- For "latest state" lookups, maintain a compact "snapshot" item per device you overwrite on each event, so reads don't scan the full series.

54. How do you handle large datasets in DynamoDB? (Pagination, queries)
Always page through results and favor Query over Scan.

Efficient reads:

- Use Query with a precise partition key and a tight sort-key range. Avoid Scan for large tables; if you must scan, use ProjectionExpression to cut payload, and paginate.
- Implement pagination with LastEvaluatedKey / ExclusiveStartKey. Never assume you'll get all items in one call.
- Limit page size (Limit) to keep latency predictable in UIs/APIs.
- For bulk reads by primary key, use BatchGetItem (remember per-call item limits and handle UnprocessedKeys with retries).
- For analytics or full-table operations, consider exporting to S3 (DynamoDB → S3 export) and use Athena/Glue instead of scanning live tables.

Write/throughput tips:

- Use on-demand for unpredictable traffic, or provisioned + auto scaling for steady loads.
- Parallel scans only when truly necessary; choose a sensible segment count and still expect uneven partitions.

55. How do you prevent hot partitions in DynamoDB?
Design keys so traffic spreads evenly; avoid too many requests landing on the same partition key at the same time.

Techniques:

- High-cardinality partition keys: don't use a small set of values (e.g., "US", "EU"); include a user/device/order id or another naturally diverse attribute.
- Write sharding: append a small random/hash suffix to the partition key for heavy-write entities:
    - Instead of PK = ORDER#<Id>, use PK = ORDER#<Id>#SHARD#<00..09>
    - On write: pick a shard by hash or random. On read: query all shards in parallel if you need the full set; for "point reads," include the shard deterministically (hash of Id).
- Time bucketing: for time-series spikes, include the bucket (day/hour) in the key so concurrent writes fan out to multiple partitions.
- Even access via GSIs: sometimes moving a hot read pattern to a GSI with a better-distributed key helps.
- Avoid large, bursty BatchWrite to a single key. Spread batches across keys, and use retries with backoff.
- Monitor with CloudWatch (Consumed/Provisioned capacity, ThrottledRequests, HotPartition metrics via Contributor Insights) and fix patterns early.

56. What are the key differences between Amazon DynamoDB and Amazon Aurora?

- **Type of database:**
  DynamoDB is a NoSQL, key-value and document database. Aurora is a relational database (compatible with MySQL and PostgreSQL).
- **Schema:**
  DynamoDB is schema-less items can have flexible attributes. Aurora requires predefined schemas with tables, columns, and types.
- **Query model:**
  DynamoDB queries are based on primary keys and indexes; no joins or complex SQL. Aurora supports full SQL with joins, aggregations, and transactions.
- **Scaling:**
  DynamoDB scales horizontally (automatic partitioning). Aurora scales vertically for writes and horizontally for reads (read replicas).
- **Consistency:**
  DynamoDB offers eventual and strong consistency options per read. Aurora uses relational database consistency (ACID across transactions).
- **Latency:**
  DynamoDB is designed for consistent single-digit millisecond latency. Aurora is low-latency but higher than DynamoDB for simple lookups.
- **Use cases:**
  DynamoDB → gaming, IoT, e-commerce carts, user profiles (high-scale key/value). Aurora → financial systems, ERP, OLTP, reporting (structured data with relationships).

57. How do you monitor and troubleshoot performance issues in DynamoDB?

Monitoring tools and approaches:

- CloudWatch Metrics: Track Read/Write capacity usage, ThrottledRequests, ConsumedCapacity, Latency, and Error metrics.
- Contributor Insights: Identify hot keys or partitions that cause uneven traffic.
- CloudTrail Logs: Track API calls for debugging misuse or unexpected traffic.
- DynamoDB Streams: Audit changes and detect unusual write patterns.

Troubleshooting approach:

1. Check if requests are being throttled (look for ProvisionedThroughputExceededException).
2. Verify partition key design hot partitions often cause bottlenecks.
3. Review query vs scan usage scans are slower and costlier.
4. Confirm if capacity mode is sufficient (provisioned too low vs on-demand).
5. Look for large items approaching 400 KB; break them up or move blobs to S3.
6. Use retries with exponential backoff for throttled requests.

58. What security features are available in DynamoDB (IAM roles, policies, KMS)?

- **IAM Roles and Policies:**
  Control who can access tables and what actions they can perform (e.g., only GetItem, only from certain VPCs). Fine-grained access can restrict access at item/attribute level.
- **KMS Encryption:**
  DynamoDB encrypts all data at rest. It supports:
    - AWS-managed KMS keys (default).
    - Customer-managed KMS keys (more control: rotation, disabling, key policies).
- **Encryption in Transit:**
  All requests between app and DynamoDB use TLS (HTTPS).
- **VPC Endpoints:**
  Private connectivity from VPC to DynamoDB without going over the internet.
- **CloudTrail Logging:**
  Track all API requests for auditing and compliance.

Together, IAM + KMS + VPC endpoints + logging provide strong, enterprise-grade security.

59. What does BatchGetItem do in DynamoDB?

BatchGetItem allows me to retrieve multiple items from one or more DynamoDB tables in a single API call.

Key points:

- Up to 100 items or 16 MB of data per request.
- Uses primary keys (partition + sort key) to fetch items.
- More efficient than calling GetItem multiple times individually.
- Items not returned (due to throttling or capacity) appear in UnprocessedKeys; I should retry those.

Example use case: Fetching 50 user profiles by ID in one call instead of 50 separate calls.

60. Explain the purpose of provisioned throughput in DynamoDB.

Provisioned throughput is how I pre-allocate read and write capacity for a DynamoDB table. It ensures my application has guaranteed performance for a known workload.

- Read Capacity Units (RCUs): 1 strongly consistent read per second for items up to 4 KB.
- Write Capacity Units (WCUs): 1 write per second for items up to 1 KB.

Benefits:

- Predictable cost since I pay for the provisioned capacity whether I use it or not.
- Protects against overuse requests beyond provisioned limits are throttled (unless I use auto scaling).
- Best for steady workloads where traffic is predictable.

Provisioned throughput is DynamoDB's way of reserving resources to guarantee performance for my tables.

61. How many Global Secondary Indexes can be created on a single table?

In DynamoDB, I can create up to 20 Global Secondary Indexes (GSIs) per table.

Some key points about GSIs:

- They can be added anytime after the base table is created (unlike LSIs, which must be defined at creation).
- Each GSI has its own partition key and optional sort key, independent of the base table.
- Each GSI has its own read/write capacity (in provisioned mode) and storage, so heavy workloads on the GSI don't impact the base table as much.
- They only support eventually consistent reads (strongly consistent reads are not available on GSIs).

So while the default limit is 20 GSIs per table, if I need more, I'd usually reconsider my data model or denormalization strategy.

62. How does DynamoDB support in-place atomic updates?

DynamoDB supports atomic updates using the UpdateItem operation. This allows me to modify values directly in the database without first reading the item, updating it in my code, and then writing it back.

How it works:

- I can use UpdateExpression to increment, decrement, append to lists, or set new attribute values in a single operation.
- Updates are atomic at the item level DynamoDB ensures that if multiple requests try to update the same item at the same time, each update is applied correctly without conflicts.
- I can also add a ConditionExpression to make the update conditional (e.g., only update if stock ≥ 1).

Example: If I want to decrement stock by 1:

```
resp = table.update_item(
   Key={"SKU": "123"},
   UpdateExpression="SET Stock = Stock - :val",
   ExpressionAttributeValues={":val": 1},
   ConditionExpression="Stock >= :val",
   ReturnValues="UPDATED_NEW"
)
```

This guarantees that two customers buying at the same time won't oversell, because DynamoDB applies the updates atomically.