In Amazon DynamoDB, secondary indexes provide additional querying capabilities by enabling queries on attributes other than the primary key. DynamoDB supports two types of secondary indexes: Global Secondary Indexes (GSI) and Local Secondary Indexes (LSI). These indexes help improve flexibility and performance when querying data based on different access patterns. Here's an overview of both types:

1. Global Secondary Index (GSI)

- **Definition**: A Global Secondary Index allows querying on any attribute as the partition key or sort key, independent of the base table's primary key. A GSI can be created on attributes that are not part of the primary key, providing flexibility for additional query patterns.
- **Structure**: A GSI consists of a partition key and an optional sort key. It can have different attributes as its partition key and sort key compared to the base table.
- **Global Scope**: Since GSIs are independent of the base table's primary key, they offer a global view across all partitions of the table. This allows querying across the entire dataset based on different attributes.
- **Provisioned Throughput**: GSIs have separate provisioned read and write capacity units (RCUs and WCUs) from the base table, which can be adjusted based on the expected query load. This helps manage performance and costs independently of the main table.
- **Eventually Consistent Reads**: By default, reads on a GSI are eventually consistent. However, strongly consistent reads are not supported on GSIs, so some latency may exist before changes to the base table are reflected in the GSI.
- **Use Cases**: GSIs are ideal for querying data across multiple partitions using attributes other than the primary key, such as querying by status, category, or timestamp when these attributes are not part of the main table's primary key.

2. Local Secondary Index (LSI)

- **Definition**: A Local Secondary Index allows querying within a partition, using an alternate sort key. The partition key of an LSI must match the partition key of the base table, but the LSI can have a different sort key, enabling additional sorting and querying within the same partition.
- **Structure**: An LSI consists of the same partition key as the base table but uses a different sort key. This enables sorting and querying based on multiple sort keys within the same partition.
- **Local Scope**: Since LSIs share the partition key with the base table, they provide a local view within each partition. This allows for fine-grained access to items that share the same partition key, sorted by different attributes.
- **Shared Provisioned Throughput**: LSIs share the same provisioned throughput as the base table. This means that read and write operations on an LSI consume capacity units

from the base table's provisioned throughput, which can affect performance if the index is heavily used.
- **Strongly Consistent Reads**: Unlike GSIs, LSIs support strongly consistent reads. This means that queries on an LSI can return the most recent data for a given partition, providing real-time consistency within the same partition.
- **Use Cases**: LSIs are useful for querying related items within the same partition based on alternative sort criteria, such as querying user activity by different attributes (e.g., by date or activity type) within the same user partition.

3. Key Differences Between GSI and LSI

- **Partition Key Requirements**: GSIs can have a different partition key from the base table, allowing global queries across partitions. LSIs, on the other hand, must have the same partition key as the base table, restricting queries to individual partitions.
- **Provisioned Throughput**: GSIs have independent provisioned throughput, which can be scaled separately from the base table. LSIs share the base table's provisioned throughput, which can impact overall capacity planning.
- **Consistency**: GSIs support only eventually consistent reads, while LSIs support both eventually and strongly consistent reads. This makes LSIs more suitable for applications that require real-time consistency within a partition.
- **Capacity and Cost Considerations**: Since LSIs share the base table's throughput, they can be more cost-effective for queries that do not require separate throughput. However, if the query load is high, GSIs provide better scalability by allowing independent throughput allocation.

4. Best Practices

- **Use GSIs for Flexibility and Scalability**: GSIs provide greater flexibility for querying on any attribute and are well-suited for applications with diverse access patterns across large datasets. They also allow for independent scaling, making them ideal for high-traffic queries.
- **Use LSIs for Fine-Grained, Consistent Queries**: LSIs are useful when you need consistent queries and alternative sort keys within the same partition. They are suitable for applications with predictable access patterns within partitions, such as querying different aspects of data for a single user or item.
- **Plan for Capacity and Cost Management**: Since GSIs and LSIs impact the provisioned throughput and storage costs, it's essential to consider their usage patterns. For GSIs, monitor throughput usage and adjust capacity as needed. For LSIs, be mindful of their shared impact on the base table's capacity.

5. Use Case Scenarios

- **GSI Example**: In a table of e-commerce orders, the main table may have an `OrderID` as the partition key. A GSI could be created with a `CustomerID` as the partition key and

`OrderDate` as the sort key, allowing queries by customer and date without affecting the main table's structure.

- **LSI Example**: In a user activity log table where `UserID` is the partition key, the main table might use `ActivityDate` as the sort key. An LSI could use `ActivityType` as a sort key, allowing the retrieval of all activities of a particular type for a user while maintaining strong consistency.