MODULE 5 · STUDY NOTES

Developing Flexible NoSQL Solutions

Amazon DynamoDB — tables, items, and keys; partitions and secondary indexes; read/write capacity and consistency; streams, global tables, backups; and developing with the AWS SDKs

The café's website is going to store its product inventory in a database, and Sofía chooses **Amazon DynamoDB** — a fully managed NoSQL service — to hold it, then builds proof-of-concept scripts that read every inventory item and that retrieve a single product by name. This module builds the DynamoDB knowledge a developer needs to make that choice well and then work with the data. It starts by placing DynamoDB among the AWS database options (relational vs. nonrelational), then drills into the concepts that define a DynamoDB table: items and attributes, the all-important **primary key**, and how DynamoDB spreads data across **partitions**. From there it covers **secondary indexes** for querying on non-key attributes, the **throughput** and **consistency** models that govern performance and cost, **streams** and **global tables** for change capture and multi-Region replication, **backup and restore**, and finally the **SDK operations** — the same create, read, update, delete, batch, and transaction calls the lab uses.

LEARNING OBJECTIVES

- Recognize different database options on Amazon Web Services (AWS)
- Identify the features of Amazon DynamoDB
- Describe the components of DynamoDB
- Explain how DynamoDB uses partitions
- Indicate how indexes are used with DynamoDB
- Describe how DynamoDB keeps data consistent
- Recognize when streaming and global tables are used
- Explain the backup and restore process
- Develop with DynamoDB by using the AWS software development kits (SDKs)

5.1 AWS database options: relational and nonrelational

Databases are excellent solutions for **persistent storage**, and they come in two major types. A **relational** database is a persistent store where data has relationships to other data; those relationships mean you must enforce a **schema** — the definitions of the data stored in each table — and enforce how tables relate to one another, and changing that schema to meet new requirements can be a challenge. A **nonrelational** (or **NoSQL**) database is *schemaless*: items are not required to conform to a fixed set of rules, and it is common to see items in the same table with different attributes.

CHOOSE RELATIONAL (SQL) WHEN	CHOOSE NONRELATIONAL (NOSQL) WHEN
strict, moderate-throughput data ▪ You must enforce strict **schema** rules and data quality ▪ The database doesn't need *extreme* read/write capacity ▪ The dataset is relational and needs no extreme performance	*scale and flexible shape* ▪ You need to scale **horizontally** (add servers on demand) ▪ Data doesn't fit a traditional schema — key-value pairs fit ▪ Read/write rates exceed what SQL economically supports

Relational databases can deliver fast performance, but for *extreme* performance — latency in the millisecond or microsecond range — a nonrelational database is often a better fit. Nonrelational databases scale by **adding compute**: servers come online as customer demand rises and shut down as it falls to reduce cost, and for the price they can exceed the performance that traditional SQL databases economically support.

TABLE 5.1 Comparing structured data storage

FEATURE	RELATIONAL (SQL)	NONRELATIONAL (NOSQL)
Data storage	Rows and columns	Key-value, documents, and graphs
Schemas	Fixed	Dynamic
Querying	SQL-based queries	Focused on a collection of documents
Scalability	Vertical	Horizontal

AWS offers managed services for both families (these are common examples, not the full catalog). On the **relational** side, **Amazon RDS** provides six database engines — Amazon Aurora, PostgreSQL, MySQL, MariaDB, Oracle Database, and Microsoft SQL Server — with optimizations you can choose for memory, performance, and I/O; **Amazon Aurora** is built for the cloud, is compatible with MySQL and PostgreSQL, can run up to **five times** faster than standard MySQL and **three times** faster than standard PostgreSQL, and provides security, availability, and reliability at about **one-tenth** the cost of commercial databases. On the **nonrelational** side, **Amazon DynamoDB** is a key-value and document database, **Amazon ElastiCache** is an in-memory data store that boosts database performance (caching, session stores, gaming, geospatial services, real-time analytics, and queueing), and **Amazon Neptune** is a graph database optimized for billions of relationships (recommendation engines, fraud detection, drug discovery, network security).

> **DYNAMODB AT A GLANCE**
>
> - A key-value **and** document database
> - Multi-Region, multi-active, and durable, with **single-digit millisecond** performance
> - Works well for internet-scale applications
> - Supports **ACID** (atomicity, consistency, isolation, durability) transactions, so critical business applications can rely on it for persistent storage

5.2 Key concepts for DynamoDB

DynamoDB organizes data into four basic components. A **table** stores data and contains items. An **item** is a group of attributes that is uniquely identifiable among all other items, and each table holds zero or more items. An **attribute** is a fundamental data element that doesn't need to be broken down any further. A **primary key** uniquely identifies each

item, and no two items can have the same key. Compared with a relational table, **items are analogous to rows and attributes to columns** — but unlike a relational database, DynamoDB is not constrained by a predefined schema, so different items in the same table can have a different number of attributes.

DynamoDB supports two types of primary keys, and the choice shapes how you can query the table. The primary-key value (partition key, plus sort key if present) is the *key*, and the remaining attributes are the *value* it maps to.

SIMPLE PRIMARY KEY	COMPOSITE PRIMARY KEY
partition key only	*partition key + sort key*
▪ One attribute — the **partition key**	▪ Two attributes — **partition key** + **sort key**
▪ DynamoDB builds an **unordered** hash index	▪ Unordered partition index; **sorted** sort-key index
▪ Each item's partition-key value **must be unique**	▪ Same partition key OK if **sort keys differ**
▪ `SensorLocation`: `SensorId` → one location	▪ `SensorReadings`: `SensorId` + `Time`

Because there is no predefined schema, an item can have any number of attributes with different value types, and each attribute has a name and a value. The size of an item is the **sum of the lengths of its attribute names and values**, and an item can be up to **400 KB**.

TABLE 5.2 Attribute value types

CATEGORY	TYPES
Scalar	Number, string, binary, Boolean, null
Set (multi-value)	String set, number set, binary set
Document	List, map

5.3 Partitions and data distribution

DynamoDB stores table data in **partitions**. A partition is an allocation of storage for a table, backed by **solid state drives (SSDs)** and automatically replicated across multiple **Availability Zones** within an AWS Region; partition management is handled entirely by DynamoDB. Data is distributed based on the **partition key** — DynamoDB applies a **hash function** to the partition-key value, and the output determines which partition holds the item. For this reason the partition key is also called the item's **hash attribute**.

With a **simple** primary key, DynamoDB stores and retrieves each item by its partition-key value alone, and items are **not** kept in sorted order. With a **composite** primary key, DynamoDB stores all items that share a partition-key value **physically close together** and orders them by **sort key** (the *range attribute*). For a `Pets` table keyed on `AnimalType` + `Name`, adding an item with partition key `Dog` and sort key `Rover` places it among the other `Dog` items in ascending sort-key order — Fido, Rover, Spot.

> **HOW PARTITIONING WORKS**
>
> - A partition is SSD-backed storage, auto-replicated across AZs and fully managed by DynamoDB
> - The **hash of the partition key** selects the partition
> - Simple key → items are not stored in sorted order
> - Composite key → items sharing a partition key are ordered by sort key

5.4 Secondary indexes

A **secondary index** lets you query data based on attributes that are **not** part of the table's primary key: the index defines an **alternate key**, and you query it much as you query the table itself. Every secondary index contains the alternate-key attributes, the base table's primary-key attributes, and an optional subset of other attributes that you choose to **project** (copy) from the base table. At a minimum, DynamoDB projects the key attributes into the index. DynamoDB supports two types of secondary indexes.

GLOBAL SECONDARY INDEX (GSI)	LOCAL SECONDARY INDEX (LSI)
different keys, its own throughput	*same partition key, shared throughput*
▪ Partition and sort key **both differ** from the base table ▪ **Global**: queries can span all base-table data, across all partitions ▪ **No size limit**; its own provisioned throughput for reads/writes ▪ Up to **20** per table (default limit) ▪ Example: index `Music` on `Genre` (partition) + `AlbumTitle` (sort)	▪ **Same partition key** as the base table; a **different sort key** ▪ **Local**: each index partition is scoped to a base-table partition with the same partition-key value ▪ ≤ **10 GB** indexed items per partition-key value; **shares** table throughput ▪ Up to **5** per table ▪ Example: index `Music` on `Artist` (partition) + `AlbumTitle` (sort)

Suppose a `Music` table lets you query by `Artist`, or by `Artist` and `SongTitle`. To query by `Genre` and `AlbumTitle` instead, you create a **global** secondary index on those attributes — note that a genre can hold many albums, so the combination need not be unique. To query by `Artist` and `AlbumTitle`, you create a **local** secondary index that keeps `Artist` as the partition key but uses `AlbumTitle` as the sort key; an LSI's sort key can be any scalar attribute. One caveat matters on the exam: **strongly consistent reads are not supported on global secondary indexes**.

> **COMMON PITFALL**
>
> Do not confuse the two indexes. A GSI is *global* because it has different keys, its own throughput, no size limit, and its queries span the whole table. An LSI is *local* because it reuses the table's partition key, shares the table's throughput, and is scoped to a single partition-key value — with a 10 GB cap per partition-key value. When the scenario needs a new partition key, it is a GSI.

5.5 Read/write throughput and consistency

DynamoDB keeps multiple copies of your data across Availability Zones for durability, and all copies are usually consistent within **one second** of a write. When you read, you choose the consistency level. An **eventually consistent** read might return slightly **stale** data if it runs immediately after a write; repeat it a moment later and it returns the latest data (fine for blog posts). A **strongly consistent** read returns the **most up-to-date** data, reflecting all prior successful writes (needed for scoreboards, booking systems, and financial algorithms), but it might be unavailable during a network delay or outage — and it is not supported on global secondary indexes.

EVENTUALLY CONSISTENT READ	STRONGLY CONSISTENT READ
default · may be stale	*most up-to-date*
▪ May not reflect a recently completed write ▪ Repeat after a short time to get the latest data ▪ Cheaper — uses **half** a strong read's capacity ▪ Use cases: blog posts, content feeds	▪ Reflects all prior successful writes ▪ May be unavailable in a network delay/outage ▪ **Not** supported on global secondary indexes ▪ Use cases: scoreboards, booking, financial apps

DynamoDB **transactions** simplify making coordinated, **all-or-nothing** changes (inserts, updates, or deletes) to multiple items both within and across tables, providing **ACID** guarantees that help maintain data correctness. Typical use cases include processing financial transactions, fulfilling and managing orders, building multiplayer game engines, and coordinating actions across distributed components and services.

Throughput is the maximum capacity an application can consume from a table or index, and DynamoDB divides it **evenly among partitions**; exceeding it causes request **throttling**. You pick one of two capacity modes. With **provisioned** throughput you specify capacity in **read capacity units (RCUs)** and **write capacity units (WCUs)**. With **on-demand** throughput you pay per request and DynamoDB adapts instantly as traffic scales up or down to any previously reached level — a good fit for new tables with unknown workloads, unpredictable traffic, or when you prefer paying only for what you use. You can switch a table between modes without changing application code.

TABLE 5.3 Provisioned capacity units

UNIT	DEFINITION
Read capacity unit (RCU)	One strongly consistent read per second for an item up to 4 KB. Eventually consistent reads use half the capacity — one RCU = two eventually consistent reads per second up to 4 KB.
Write capacity unit (WCU)	One write per second for an item up to 1 KB.

CALCULATING RCUS

Read 20 items of 11 KB each per second, with eventual consistency. How many RCUs? Round the item size up to the next multiple of 4 KB: 11 → **12**. Divide by 4 KB per RCU: **3** (strongly consistent RCUs per item). Multiply by items per second: 3 × 20 = **60**. Divide by 2 for eventual consistency: **30 RCUs**.

CALCULATING WCUS

Write 120 items of 7 KB each per minute. How many WCUs? Each 7-KB item needs **7** WCUs (one WCU per 1 KB). Convert the rate to per second: 120 per minute = **2** items per second. Multiply WCUs per item by items per second: 7 × 2 = **14 WCUs**.

COMMON PITFALL

Reversing the two consistency levels is the classic error. **Eventually consistent** is the cheaper default that may briefly return stale data; **strongly consistent** returns the latest data but costs twice as much (1 RCU = 1 strong read or 2 eventual reads per second, ≤ 4 KB) and cannot be used on a GSI.

5.6 Streams and global tables

DynamoDB Streams is an optional feature that captures a **time-ordered sequence** of item-level modifications in a table and stores it for up to **24 hours**. When an application creates, updates, or deletes an item, DynamoDB Streams writes a **stream record** — recording at least the primary-key attributes of the changed item, and optionally the item's **before and after images**. Each record gets a **sequence number** reflecting the order it was published. Records are grouped into **shards**, which act as containers that applications iterate through; records in a shard are automatically removed after 24 hours. Streams can be consumed by other AWS services, such as **AWS Lambda**, to respond to changes in near-real time — for example, an ad server reacting to updated user preferences.

Global tables provide a fully managed solution for a **multi-Region, multi-active** database without building and maintaining your own replication. You specify the AWS Regions where the table should be available, and DynamoDB creates identical **replica tables** there and propagates ongoing data changes to all of them. Each replica has the same table name and primary-key schema and stores the **same complete set** of items — DynamoDB does not support partial replication — and a newly written item is usually propagated to all replicas within seconds. A `CustomerProfiles` table replicated across North America, Europe, and Asia lets customers update their profile in any Region and keeps the data available even if one Region becomes temporarily unavailable.

HOW GLOBAL TABLES REPLICATE AND RECONCILE

- Global tables use **DynamoDB Streams** to propagate changes: a change in one replica is recorded in a stream and propagated to the other replicas
- Concurrent updates to the same item in different Regions are resolved by **last writer wins**; all replicas then converge on identical data
- An application can read and write **any** replica, but a global table has only **one replica per Region**
- **Strongly consistent reads do not work across Regions** — do strongly consistent reads and writes in the same Region as the client; a cross-Region read may be stale
- Transactions are enabled on single-Region tables but **disabled on global tables by default** (available by request; cross-Region replication stays asynchronous and eventually consistent, so partial transactions may be observed during replication)

5.7 Backup and restore

DynamoDB provides two ways to protect table data, both usable from the **AWS Management Console**, the **AWS CLI**, or the **DynamoDB API**, and both running with little impact on table performance or availability.

ON-DEMAND BACKUPS VS. POINT-IN-TIME RECOVERY

- **On-demand backups** create full backups for long-term retention and archiving. A time marker is cataloged and the backup is created **asynchronously** — processed near instantaneously and available for restore within minutes.
- Each on-demand backup captures the **entire** table, there is **no limit** on the number of backups, and backups **consume no provisioned throughput**.
- **Point-in-time recovery (PITR)** protects against accidental write or delete operations by maintaining **incremental** backups automatically.
- With PITR you can restore a table to **any point in time during the last 35 days**, without creating, maintaining, or scheduling backups yourself.

5.8 Basic operations for DynamoDB tables (SDK)

You invoke DynamoDB through its API — directly or through an AWS **SDK** such as the SDK for Python (**Boto3**), which the lab uses. Operations fall into four categories.

TABLE 5.4 DynamoDB API operation categories

CATEGORY	WHAT IT DOES	EXAMPLES
Control	Create and manage tables (and dependent indexes and streams)	`CreateTable`
Data	Create, read, update, and delete items; can also read from a secondary index	`PutItem`, `UpdateItem`, `DeleteItem`, `GetItem`, `Query`, `Scan`
Batch	Run multiple get/write operations in one request for higher throughput	`BatchGetItem`, `BatchWriteItem`
Transaction	Coordinated, all-or-nothing changes to multiple items within and across tables	`TransactWriteItems`, `TransactGetItems`

`CreateTable` is asynchronous: it returns immediately with a `TableStatus` of `CREATING` and moves to `ACTIVE` when ready — you can read and write only an `ACTIVE` table. It takes a `TableName`, a `KeySchema` (the primary-key attributes, with key type `HASH` for the partition key and `RANGE` for the sort key), `AttributeDefinitions` (data types such as `S` for string), and, in provisioned mode, `ProvisionedThroughput` (for example, 5 RCUs and 5 WCUs). For items: **`PutItem`** creates a new item **or replaces (overwrites)** an existing item with the same primary key — only the primary-key attributes are required. **`UpdateItem`** edits an existing item's attributes (or adds the item if it doesn't exist) using an `UpdateExpression`. **`DeleteItem`** removes a single item by its primary key.

For the `PutItem`, `UpdateItem`, and `DeleteItem` operations you can attach a **condition expression** — the operation proceeds only if the expression evaluates to **true**, otherwise it fails. For example, an `accountLocked` attribute may be set to `N` only if the last failed login attempt was more than 24 hours earlier. Condition expressions make writes **conditional** and are the basis of safe concurrent updates.

To read items: **`GetItem`** retrieves a single item by its full primary key (partition key, plus sort key if present); by default it returns all attributes and uses an eventually consistent read, but you can request a strongly consistent read or a projection expression. **`Query`** reads only the items that match a `KeyConditionExpression` from a table or secondary index, optionally refined by a `FilterExpression`. **`Scan`** reads **every** item from a table or index and can be refined with a filter — because it reads everything it is expensive, so a `Query` with an appropriate key condition is **more efficient**; when a scan is unavoidable, use a **parallel scan**.

> **OPTIMIZING PERFORMANCE AND COST**
>
> - **1-MB pagination limit**: a single `Query` or `Scan` returns at most 1 MB of data; process results page by page using `LastEvaluatedKey` and `ExclusiveStartKey`
> - **`Limit` parameter**: caps the maximum number of items a `Query` or `Scan` returns
> - Prefer `Query` over `Scan`; use a parallel scan only if a scan is required

BATCH AND TRANSACTION LIMITS

- **`BatchGetItem`**: read up to **16 MB** of data — as many as **100 items** — from multiple tables
- **`BatchWriteItem`**: write up to **16 MB** — as many as **25** PUT or DELETE requests — across tables (individual items up to 400 KB)
- In a **batch**, if one request fails the whole operation does **not** fail — retry with the failed keys and data the operation returns
- In a **transaction** (`TransactWriteItems` / `TransactGetItems`), if one operation fails the **entire transaction fails**

SAMPLE EXAM QUESTION, WORKED

A developer needs every update to a DynamoDB table sent to AWS Lambda, which will notify the Operations team when certain values appear. How should the developer modify the table? Key phrases: *Amazon DynamoDB* and *anytime the table is updated, the changes must be sent*. A global or local secondary index only speeds up **queries**, so both are wrong. The answer is to **enable DynamoDB Streams** on the table: the stream can emit *keys only*, the *new image*, the *old image*, or *new and old images* of each changed item, and it can trigger a Lambda function that inspects the change and decides whether to notify Operations.

Module summary

- DynamoDB is a fully managed key-value and document (NoSQL) database — schemaless, horizontally scaling, single-digit-millisecond, ACID-capable — chosen when you need scale and flexible data shape rather than a strict relational schema; AWS also offers RDS and Aurora (relational) and ElastiCache and Neptune (nonrelational).
- A table holds items (rows) made of attributes (columns), up to 400 KB per item; a primary key uniquely identifies each item as either a simple key (partition key only) or a composite key (partition key + sort key).
- DynamoDB hashes the partition key to place items in SSD-backed, AZ-replicated partitions; with a composite key, items sharing a partition key are stored together and ordered by sort key (two such items must have different sort keys).
- Secondary indexes query non-key attributes: GSIs use different keys with their own throughput and no size limit (up to 20 per table); LSIs keep the partition key with a new sort key, share table throughput, and cap at 10 GB per partition-key value (up to 5 per table).
- RCUs and WCUs meter provisioned capacity — 1 RCU = one strongly consistent (or two eventually consistent) reads/sec of ≤ 4 KB, 1 WCU = one write/sec of ≤ 1 KB; on-demand mode pays per request and scales automatically.
- Reads are eventually consistent (possibly stale, cheaper, default) or strongly consistent (latest data, not on GSIs); transactions add ACID, all-or-nothing changes across items and tables.
- DynamoDB Streams captures time-ordered item changes in shards for 24 hours and can trigger services like Lambda; global tables use streams to replicate multi-Region, multi-active data with last-writer-wins reconciliation (no strongly consistent reads across Regions).
- On-demand backups take unlimited full snapshots with no throughput cost; point-in-time recovery restores a table to any moment in the last 35 days.

- The SDK exposes control, data, batch, and transaction operations — CreateTable, PutItem/UpdateItem/DeleteItem, GetItem/Query/Scan (Query beats Scan), condition expressions for conditional writes, 1-MB paginated results, and 16-MB/100-item or 16-MB/25-request batches.

Review questions

1. When would you choose a nonrelational database over a relational one?

Answer: When you need to scale horizontally, your data doesn't fit a traditional fixed schema (key-value pairs fit better), or your read/write rates exceed what SQL databases economically support. Relational databases suit strict-schema data that doesn't require extreme performance.

2. Name DynamoDB's four basic components, and give the relational equivalent of two of them.

Answer: Table, item, attribute, and primary key. Items are analogous to rows and attributes to columns; unlike a relational table, different items can have different numbers of attributes.

3. What is the difference between a simple and a composite primary key, and when can two items share a partition-key value?

Answer: A simple key is a partition key only; a composite key is a partition key plus a sort key. With a composite key, two items may share a partition-key value as long as their sort-key values differ. With a simple key, the partition-key value must be unique.

4. How does DynamoDB decide which partition stores an item, and how are composite-key items ordered?

Answer: It applies a hash function to the partition-key value; the output selects the partition. Items that share a partition-key value are stored physically together and ordered by their sort key.

5. Contrast a global secondary index with a local secondary index.

Answer: A GSI uses a different partition and sort key, has its own throughput and no size limit, spans all partitions, and allows up to 20 per table. An LSI uses the same partition key with a different sort key, shares the table's throughput, is limited to 10 GB per partition-key value, and allows up to 5 per table. Strongly consistent reads are not supported on GSIs.

6. You must support 20 eventually consistent reads per second of 11-KB items. How many RCUs, and how do you get there?

Answer: 30 RCUs. Round 11 KB up to 12 (next multiple of 4), divide by 4 = 3 RCUs per item, multiply by 20 items/sec = 60, then halve for eventual consistency = 30.

7. How do global tables replicate data and resolve conflicting concurrent updates across Regions?

Answer: They use DynamoDB Streams to propagate each change to the other replica tables. Concurrent updates to the same item in different Regions are reconciled with last writer wins, so all replicas converge on identical data. Strongly consistent reads do not work across Regions.

8. Why is Query preferred over Scan, and what two mechanisms limit the data a Query or Scan returns?

Answer: Scan reads every item in the table or index, which is expensive; Query reads only items matching the key condition, so it is more efficient. Returned data is limited by the 1-MB pagination limit (paged with LastEvaluatedKey/ExclusiveStartKey) and by the Limit parameter.