AWS DynamoDB

Verified 2026-11-02

Model DynamoDB tables from their queries — keys, single-table design, GSIs and LSIs, capacity, consistency, transactions and hard limits.

Contents

Mental model 2

Keys and items 2

Query, Scan and pagination 3

Single-table design 5

Secondary indexes 6

Capacity and consistency 7

Conditional writes and transactions 8

Batches, TTL, streams and DAX 9

Further reading 10

Mental model

DynamoDB is a distributed hash map with a sorted list under every key: you read efficiently only by naming a partition key, optionally narrowed by a range of sort keys. There are no joins and no ad-hoc queries worth paying for, so you list the access patterns first and design keys and indexes to answer each one with a single `Query`. A table modeled like a relational schema works on day one and becomes a full-table `Scan` the first time someone asks a new question.

Keys and items

Term	Meaning
Partition key (PK)	Hashed to pick the storage partition
Sort key (SK)	Orders items that share a PK
Item collection	All items with the same PK
Item size	400 KB max, attribute names included

The primary key is either a PK alone or PK plus SK, and it is fixed at table creation. Attributes other than the key are schemaless: two items in one table can have completely different shapes. The SDK v3 `DynamoDBDocumentClient` converts plain JS values to and from DynamoDB's typed JSON for you.

```
import { DynamoDBClient }
  from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, PutCommand,
```

```
  GetCommand } from "@aws-sdk/lib-dynamodb";

const db = DynamoDBDocumentClient.from(
  new DynamoDBClient({}));
const Key = { PK: "USER#42", SK: "PROFILE" };

await db.send(new PutCommand({
  TableName: "App", Item: { ...Key, name: "Ana" },
}));
const { Item } = await db.send(
  new GetCommand({ TableName: "App", Key }));
```

> **Gotcha:** Attribute names count toward the 400 KB item limit and toward the bytes billed on every read and write. Long names such as `customerShippingAddressLine1` cost money on every request.

Query, Scan and pagination

Operation	Reads
`GetItem`	One item by full primary key
`Query`	One item collection, optional SK condition
`Scan`	Every item in the table or index
Page size	1 MB per call, before filtering

`Query` needs an equality on the PK and accepts `=`, `<`, `between` or `begins_with` on the SK. Each call returns at most 1 MB; if more remains,

the response carries `LastEvaluatedKey`, which you pass back as `ExclusiveStartKey`. Stop only when `LastEvaluatedKey` is absent.

```
import { QueryCommand }
  from "@aws-sdk/lib-dynamodb";
let key, items = [];
do {
  const page = await db.send(new QueryCommand({
    TableName: "App",
    KeyConditionExpression:
      "PK = :pk AND begins_with(SK, :p)",
    ExpressionAttributeValues: {
      ":pk": "USER#42", ":p": "ORDER#" },
    ExclusiveStartKey: key,
  }));
  items.push(...page.Items);
  key = page.LastEvaluatedKey;
} while (key);
```

> **Gotcha:** `FilterExpression` runs after the 1 MB read, so you pay for every item read and a page can come back empty while more pages exist. `Limit` also caps items evaluated, not items returned.

Single-table design

Access pattern	Key design
User profile	`PK=USER#42`, `SK=PROFILE`
User's orders, newest first	`PK=USER#42`, `SK=ORDER#<iso-date>`
Order by ID	GSI: `GSI1PK=ORDER#9`, `GSI1SK=ORDER#9`

Put entities that are read together under one PK so a single `Query` fetches the parent and its children in one round trip. Prefix keys with the entity type so one table holds many types without collisions. Sort keys that embed ISO timestamps sort chronologically; read newest first with `ScanIndexForward: false`.

```
[
  { "PK": "USER#42", "SK": "PROFILE",
    "name": "Ana" },
  { "PK": "USER#42", "SK": "ORDER#2026-11-01",
    "GSI1PK": "ORDER#9", "GSI1SK": "ORDER#9",
    "total": 30 }
]
```

> **Tip:** Write the access-pattern table before you create the table. Every row needs a key or index that answers it with one `Query`; a row that needs a `Scan` is a design bug.

Secondary indexes

Property	GSI	LSI
Keys	Any PK and SK	Same PK, different SK
Created	Any time	Only with the table
Per table	20 (default quota)	5
Strong reads	Not supported	Supported

A global secondary index (GSI) is a separately partitioned copy of the projected attributes, kept in sync asynchronously. A local secondary index (LSI) re-sorts an existing item collection. Items missing the index key attributes are left out of the index, which makes a GSI on an optional attribute a cheap sparse index.

```
aws dynamodb query --table-name App \
  --index-name GSI1 \
  --key-condition-expression "GSI1PK = :o" \
  --expression-attribute-values \
    '{":o":{"S":"ORDER#9"}}'
```

> **Gotcha:** A table with an LSI caps every item collection, including LSI entries, at 10 GB, and writes beyond it fail. Tables without an LSI have no per-collection size limit.

Capacity and consistency

Unit	Covers
1 RCU	One strong read per second, up to 4 KB
Eventual read	Half the cost of a strong read
1 WCU	One write per second, up to 1 KB
Transactional op	Twice the units of a normal one

On-demand mode bills per request and scales automatically, instantly absorbing up to double the table's previous peak traffic. Provisioned mode bills per hour for the capacity you set, used or not, and can auto scale. You can switch from provisioned to on-demand up to four times in a rolling 24 hours. Reads are eventually consistent unless you set `ConsistentRead: true`; GSI reads and stream reads are always eventually consistent.

```
const { Item } = await db.send(new GetCommand({
  TableName: "App",
  Key: { PK: "USER#42", SK: "PROFILE" },
  ConsistentRead: true, // 1 RCU per 4 KB, not 0.5
}));
```

> **Gotcha:** Throughput is spread across partitions, and one partition serves at most 3,000 read units and 1,000 write units per second. A hot key throttles at that ceiling no matter how much table capacity you buy.

Conditional writes and transactions

Tool	Use
`ConditionExpression`	Write only if state matches
`attribute_not_exists(PK)`	Insert without overwriting
`TransactWriteItems`	All-or-nothing, up to 100 items, 4 MB
`TransactGetItems`	Consistent snapshot of up to 100 items

A condition turns a blind write into compare-and-set, which is how you build uniqueness, optimistic locking and idempotency without a lock. Transactions span tables in one account and Region, but cost twice the capacity of the same writes done singly.

```
import { TransactWriteCommand }
  from "@aws-sdk/lib-dynamodb";


await db.send(new TransactWriteCommand({
  TransactItems: [
    { Put: { TableName: "App",
      Item: { PK: "EMAIL#a@x.io", SK: "EMAIL" },
      ConditionExpression:
        "attribute_not_exists(PK)" } },
    { Put: { TableName: "App",
      Item: { PK: "USER#43", SK: "PROFILE" } } },
  ],
}));
```

> **Gotcha:** A failed condition on a single write throws `ConditionalCheckFailedException`, but inside a transaction it throws `TransactionCanceledException` with a `ConditionalCheckFailed` reason. A `catch` that tests only the first name misses the second.

Batches, TTL, streams and DAX

Feature	Rule
`BatchWriteItem`	25 puts or deletes, 16 MB
`BatchGetItem`	100 keys, 16 MB
TTL	Deletes expired items, typically within 2 days
Streams	Change records kept 24 hours
DAX	In-memory cache for eventually consistent reads

Batch calls are not atomic: whatever DynamoDB could not process comes back in `UnprocessedItems` or `UnprocessedKeys`, and you retry those with backoff. TTL reads a Number attribute holding epoch seconds and deletes without consuming write capacity. Streams feed Lambda with ordered per-item changes, TTL deletions included.

```
aws dynamodb update-time-to-live \
  --table-name App \
  --time-to-live-specification \
    "Enabled=true, AttributeName=expiresAt"
```

> **Gotcha:** TTL deletion is best-effort, so expired items stay readable until they are removed. Filter on `expiresAt > :now` in queries that must not see them.

Further reading

- DynamoDB Developer Guide
- DynamoDB quotas
- Data modeling building blocks
- Best practices for querying and scanning