Serverless

The AWS-related services

Serverless?

- Debunking the myths:

  Serverless does not mean there are no servers

- It means, the developer:
  - Doesn’t need to care about servers when coding.
  - Doesn't have to manage physical capacity.

Without Serverless ….

- What is the right size for my server?
- What capacity is left on my server?
- How many servers should I provision?
- How do I handle hardware failures?

- What OS should my server run?
- Whom should have access to my servers?
- How do I detect a compromised server?
- How do I keep my server patched?

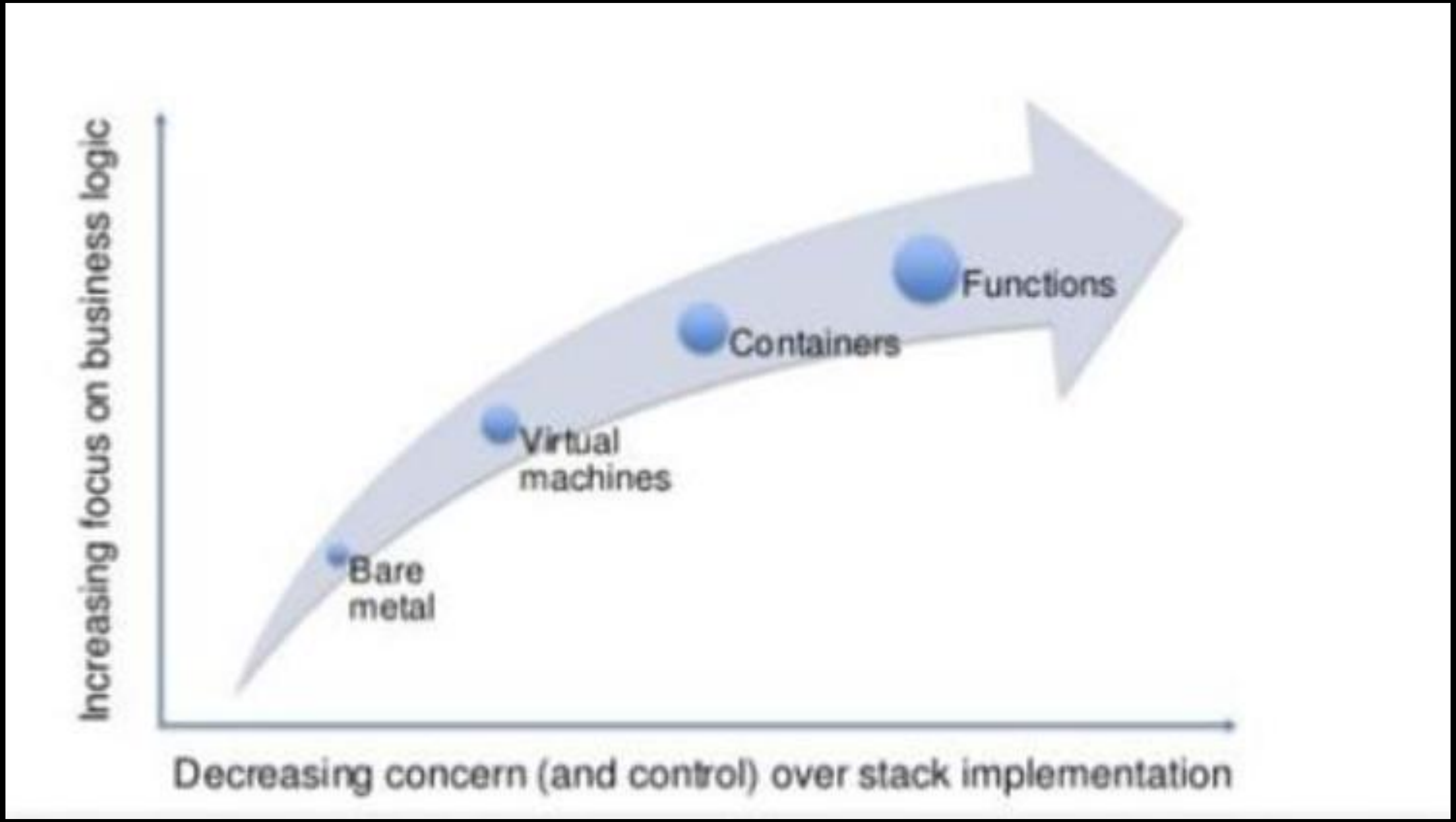
With Serverless

- No servers to provision or manage.
- Scales automatically based on demand.
- No up-front cost; Pay for how much you use.
- Availability and fault tolerance built-in.

- Benefits:
  1. Greater agility.
  2. Less overhead.
  3. Increased scale.
  4. Better focus.
  5. Faster time to market.

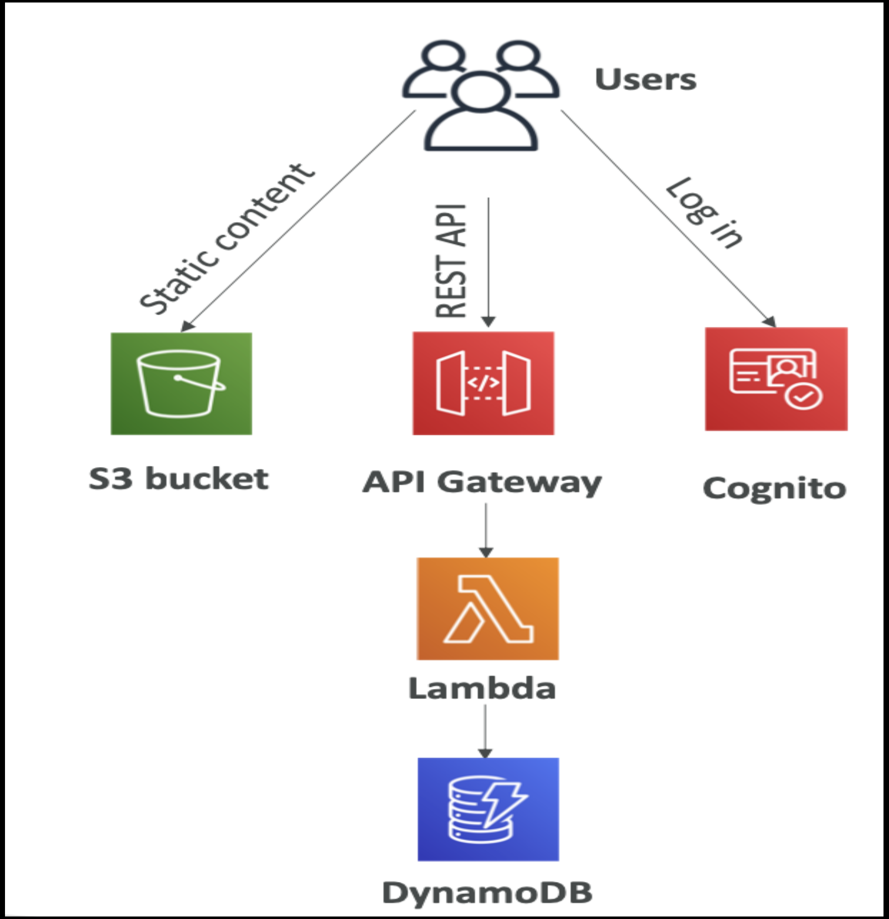
Developer’s focus.

- Developers should focus on the product, not infrastructure.


Increasing focus on business logic
Bare metal
Virtual machines
Containers
Functions
Decreasing concern (and control) over stack implementation

Serverless Services on AWS

- AWS Lambda. (Compute)
- DynamoDB. (NoSQL)
- AWS Cognito. (User Accounts Mgt.)
- API Gateway (HTTP/REST endpoints)
- S3 (Storage)
- SNS & SQS. (Messaging)
- AWS Kinesis Data Firehose
- Aurora Serverless (RDB)
- Step Functions (Orchestration)
- Fargate (Containers)
- And more …



AWS Lambda Service

Serverless Compute

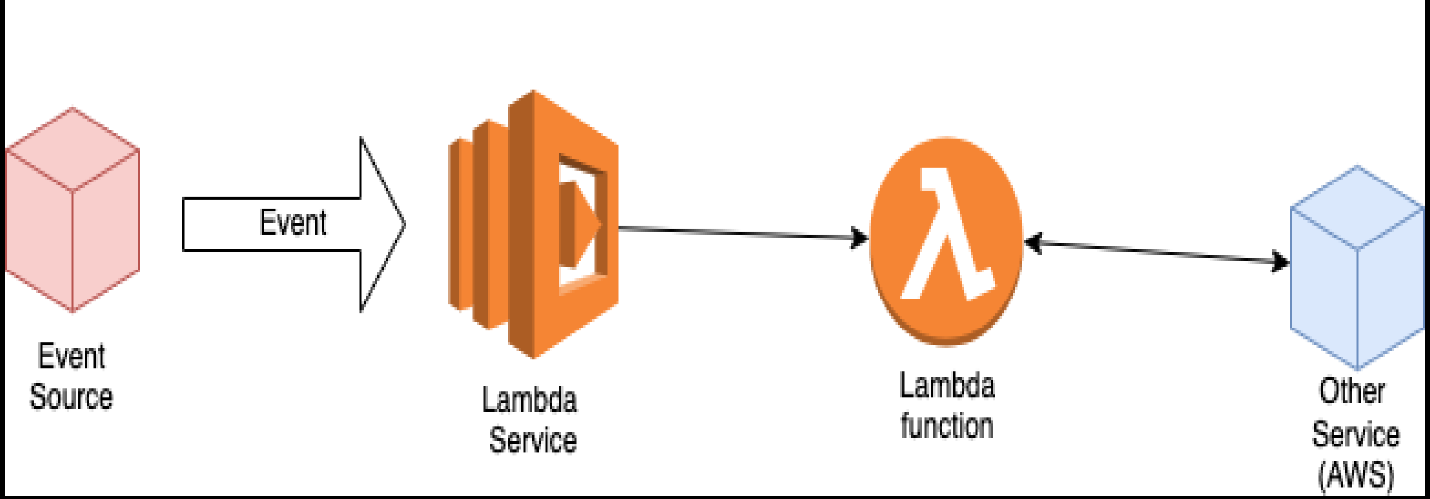
AWS Lambda

- “Lambda is an event-driven, serverless computing platform provided by AWS. It runs code (a function) in response to events and automatically manages the compute resources (CPU, memory, networking) required by that code. It was introduced on November 13, 2014.” Wikipedia

- “Custom code that runs in an ephemeral container” Mike Robins

- FaaS (Functions as a Service)
  - IaaS, PaaS, SaaS

Lambda runtime model.



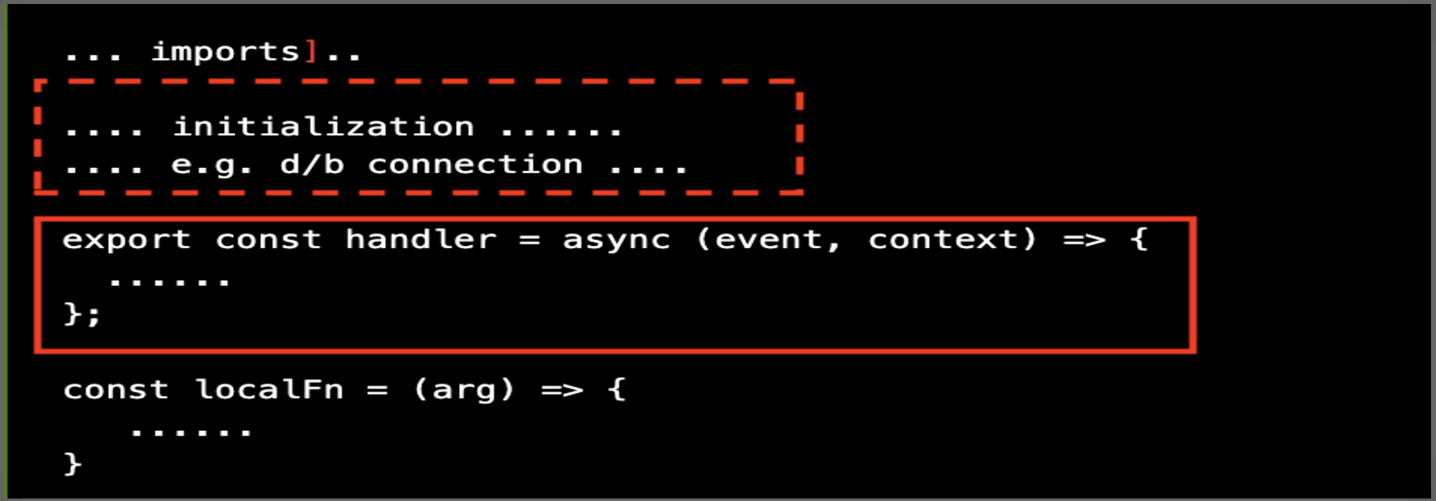
- Event Source (Trigger), e.g. HTTP request; Change in data store, e.g. database, S3 upload; Change in resource state, e.g. EC2 instance.
- Lambda function: Python, Node, Java, Go, C#, etc.

AWS Lambda service

- The Lambda service manages:
  1. Auto scaling (horizontal)
  2. Load balancing.
  3. OS maintenance.
  4. Security isolation.
  5. Utilization (Memory, CPU, Networking)

- Characteristics:
  1. Function as a unit of scale.
  2. Function as a unit of deployment.
  3. Stateless nature.
  4. Limited by time - short executions.
  5. Run on-demand.
  6. Pay per execution and compute time – generous free tier.
  7. Do not pay for idle time.

Anatomy of a Lambda function



```
... imports]..

.... initialization ......
.... e.g. d/b connection ....

export const handler = async (event, context) => {
  ......
};

const localFn = (arg) => {
   ......
}
```

- handler() – function to execute on invocation.
- Event object – the payload and metadata provided by the event source.
- Context object – access to runtime information.
- Initialization code executes before the handler for cold starts only.

Lambda function configuration

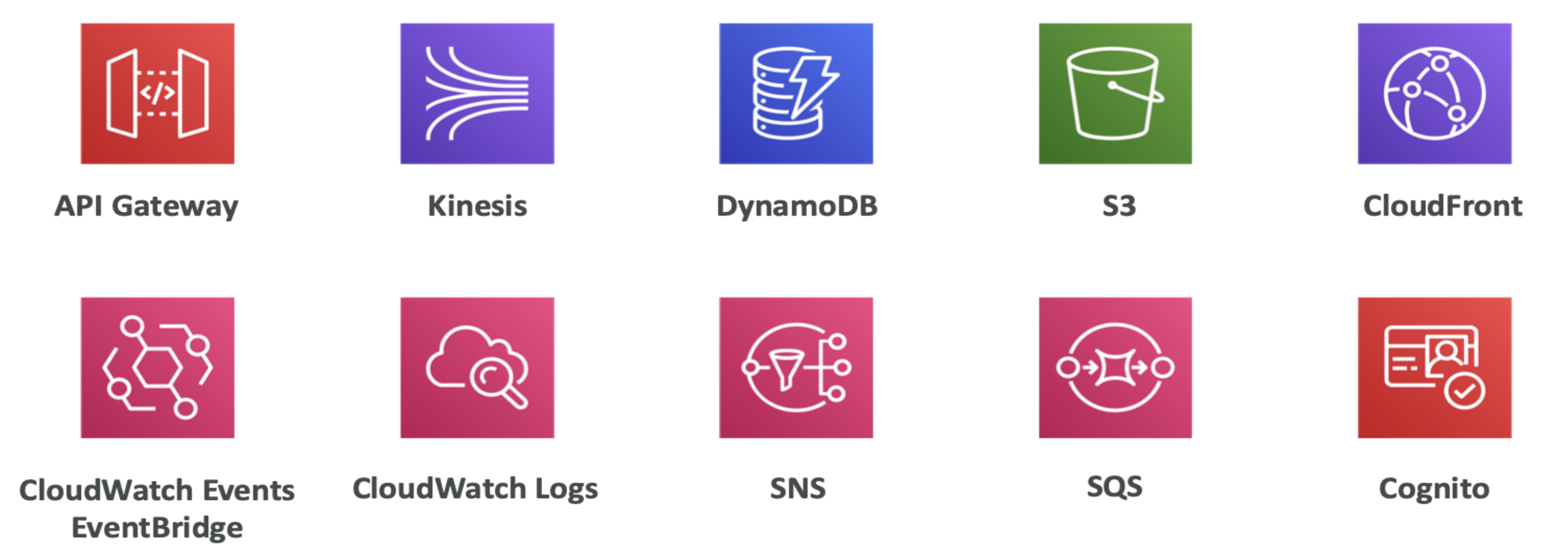
- Lambda service provides a memory control to configure a function’s compute power requirements.
    - The % of CPU core and network capacity are computed in proportion to its RAM.
- RAM:
    - From 128MB to 3,008 MB in 64 MB increments
    - The more RAM you add, the more vCPU credits you get.
        - 1,792 MB RAM allocation → one full vCPU reserved.
        - After 1,792 MB → more than one CPU assigned → should use multi-threading to fully utilize.
- For CPU-bound processing, increase the RAM allocation.
- Timeout setting: Max. runtime allowed.
    - Default 3 seconds; maximum 900 seconds (15 minutes).

Demo

- Objective:
    - Use the CDK to provision a ‘Hello World’ (Typescript/Node) lambda function.
    - Invok it (trigger it) using the AWS CLI.
    - Check console.log() statement output in the CloudWatch Logs.

Lambda integration

- What services can trigger a lambda function?

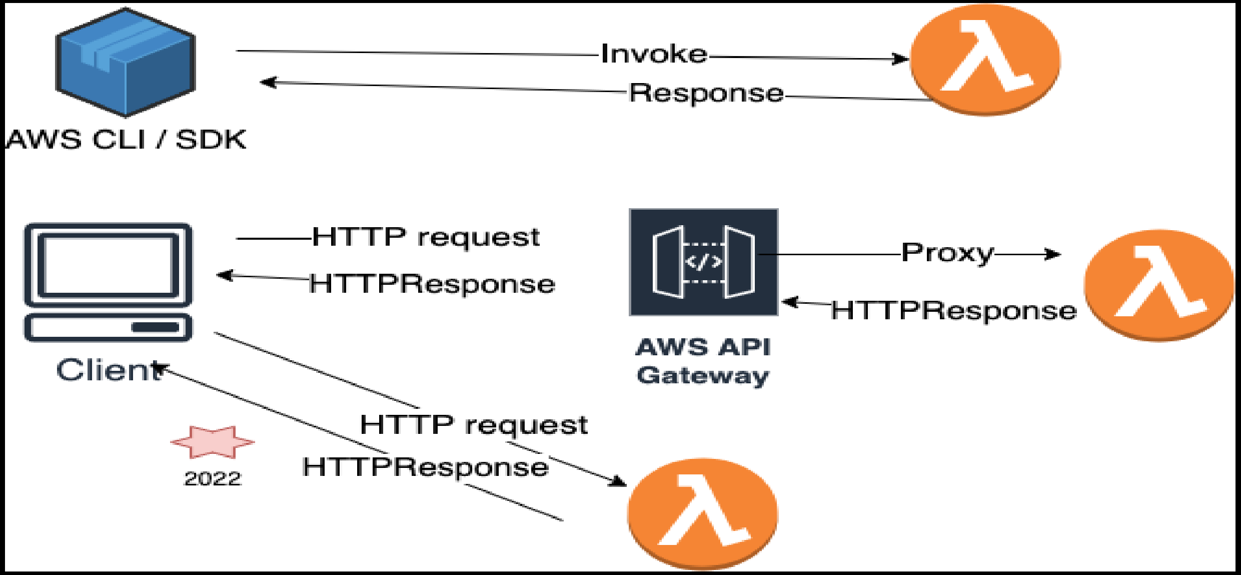


- Integration models:

  1. Synchronous.   2. Asynchronous.   3. Poll-based

Synchronous Integration

- Trigger (Event Source) options: CLI, SDK, API Gateway, Load Balancers, Function URLs.
- Client (Event source) waits for the function's response, i.e. synchronous.
- Client should handle error responses (retries, exponential backoff, etc.)

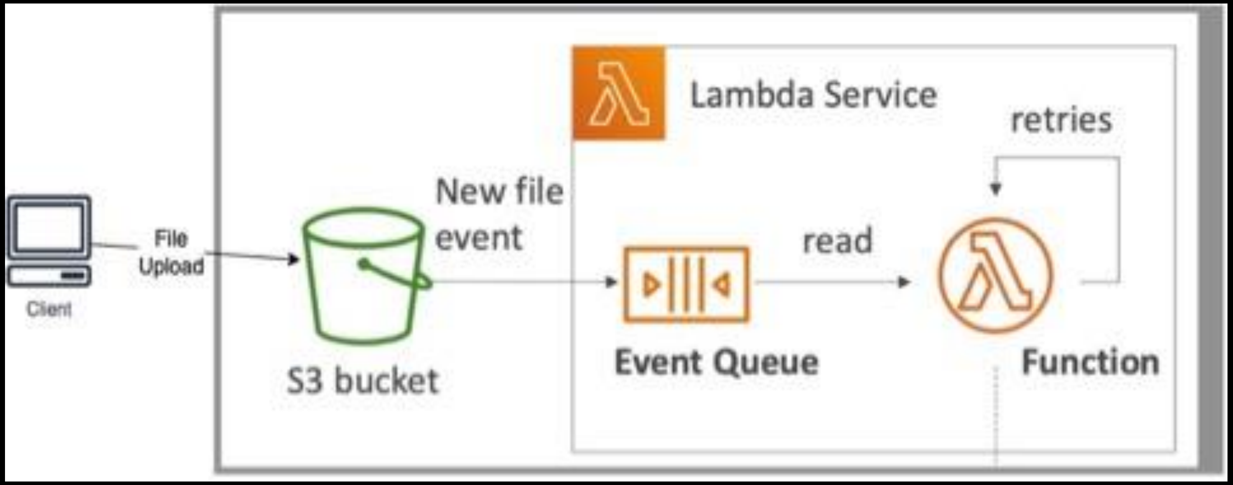


Demo

- Objective:
  - Use the CDK to:
    1. Provision a lambda function.
    2. Get the Lambda service to generate a URL endpoint.
  - Test the URL with Postman client.
  - Configure the endpoint as private and invoke it.

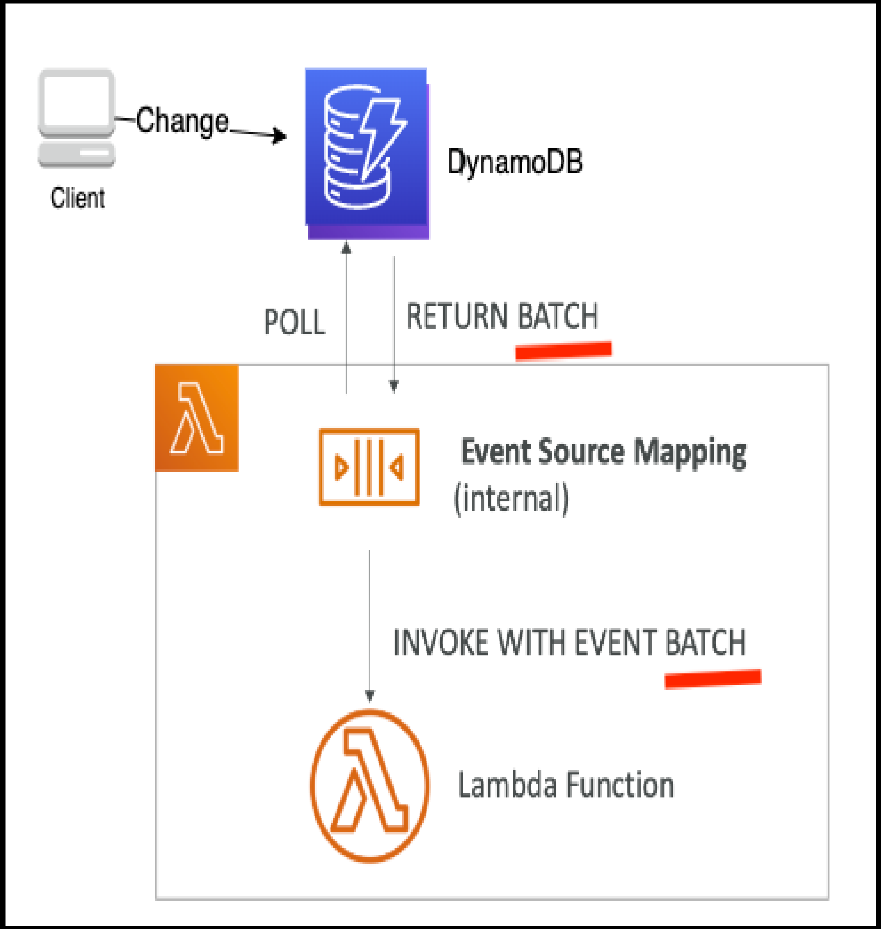
Asynchronous Integration.

- Trigger (Event Source): S3, SNS, CloudWatch.
- Lambda service places the events in a queue (see below image).
- Lambda service retries on errors – 3 retries, using exponential backoff algorithm.
- Function’s processing should be idempotent (due to retries)
- Suitable when the client does not require the function result immediately.



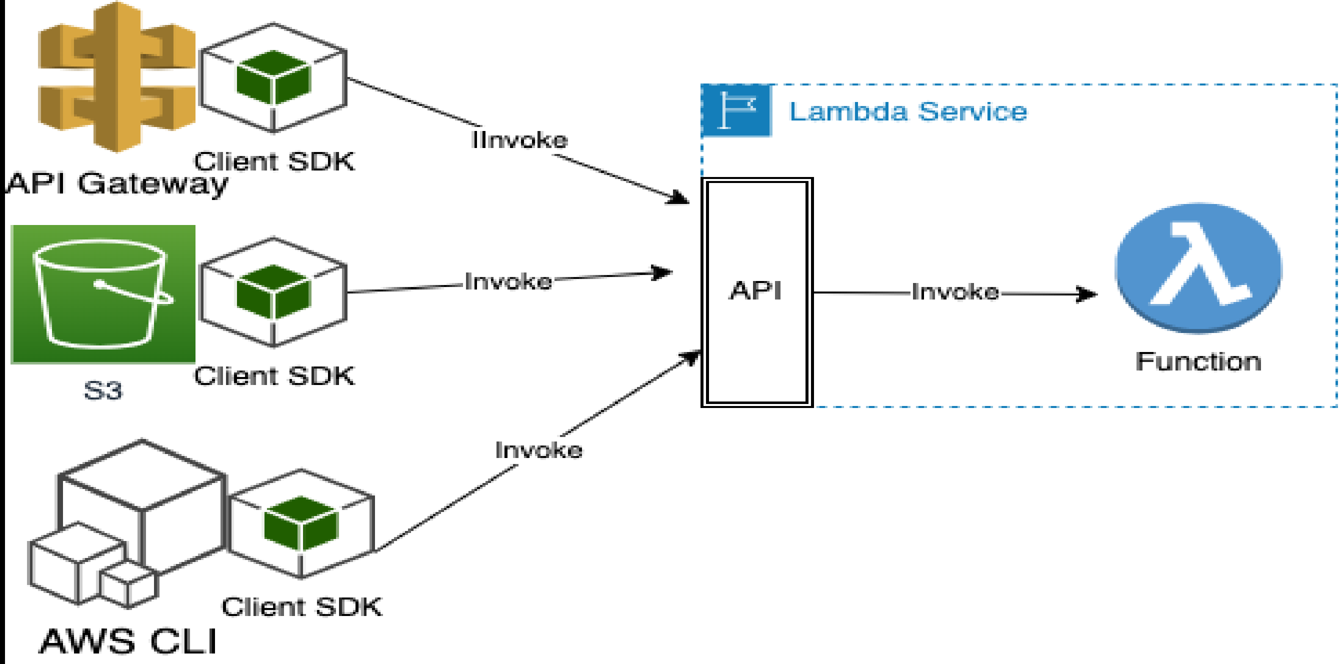
Event source mapping. (Poll-based Integration)

- Event Sources/Trigger: DynamoDB streams, SQS, Kinesis streams.
- Lambda service polls the source for event records.
- Lambda service invokes the function synchronously.



Lambda service API & SDK

- Lambda Service provides a web API and an accompanying SDK.
- All other services use the SDK to trigger Lambda functions in all models (sync, async, poll-based).
- Can pass any event payload you want.



Execution Role (IAM Role)

- Grants the lambda function permissions to access specified AWS services / resources.
- Many predefined / Managed policies, e.g.
  - AWSLambdaBasicExecutionRole – Upload logs to CloudWatch.
  - AWSLambdaDynamoDBExecutionRole – Read from DynamoDB Streams.
  - AWSLambdaSQSQueueExecutionRole – Read from SQS queue
  - AWSLambdaVPCAccessExecutionRole – Deploy function in VPC.

- It’s best practice to create one Execution Role per function.

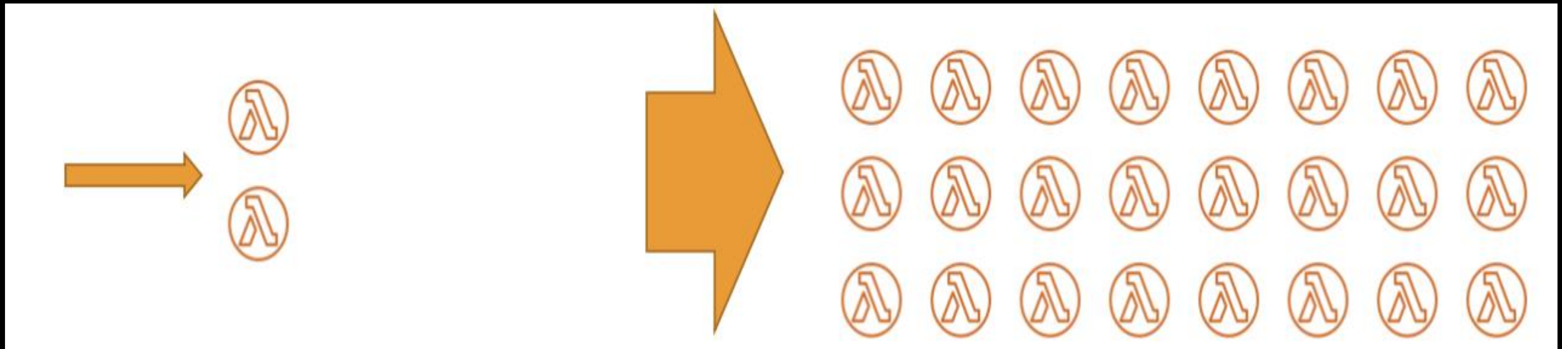
Resource based Policies.

- Use resource-based policies to give other AWS services (and accounts) permission to use your Lambda resource/function.

- An IAM principal (e.g. user, service) can access a Lambda resource when:
    1. An IAM policy attached to the principal authorizes it,
    2. OR the function’s resource-based policy authorizes it.

- Ex.: The AWS S3 service can trigger a Lambda function if the function’s resource-based policy permits it.

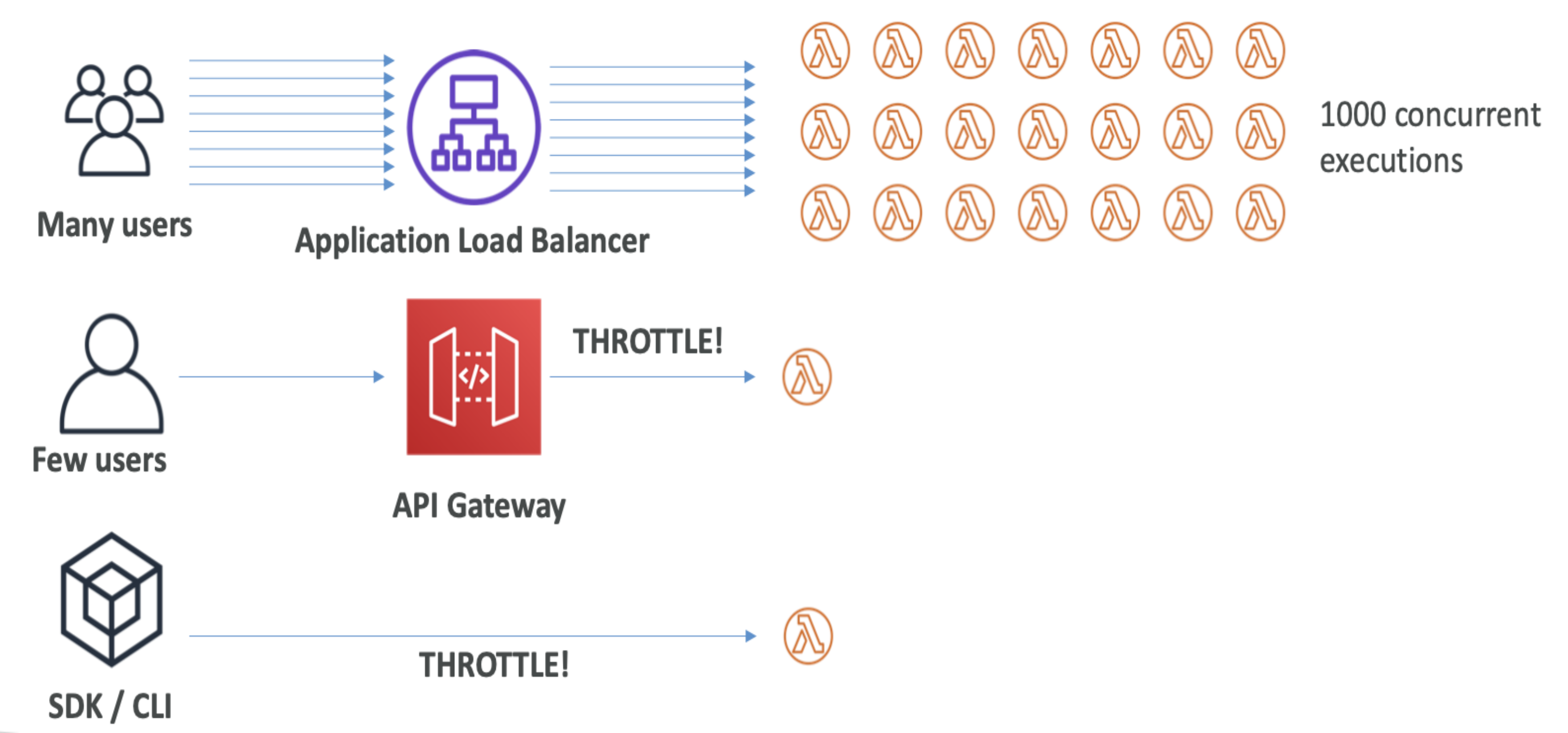
Concurrency and Throttling



- All AWS accounts have a concurrency limit of 1000 concurrent executions.
- Can set a reserved concurrency at the function level.
  - “Throttle” error response when limit exceeded.
- Throttle behavior:
  - Synchronous invocation => return ThrottleError – HTTP status 429
  - Asynchronous invocation => retry automatically and then go to a DLQ (Dead Letter Queue).

Concurrency issues

- The following can happen when no reserve concurrency is set:



Cold Starts & Provisioned Concurrency.

- Cold Start:
  - Source Event → New micro VM instance created → function’s initialization code executes → handler code executes.
- Warm start – Micro VM is reused; Init code not executed.
- Lengthy init code (LoC, connection setup, e.g. d/b) effects overall event processing time.
  - Greater latency with cold start executions.
- Solution - Provisioned Concurrency:
  - Micro VMs are pre-allocated in advance; Init code executed during pre-allocation.
  - lower latency on average.

AWS DynamoDB Service

Serverless NoSQL Database

Features

- NoSQL database - not a relational database.
- Fully Managed (Serverless), Highly available with replication across 3 AZ.
- Schema-less.
- No support for table joins.
  - Consider denormalization instead.
- Integrated with IAM for security, authorization and administration.
- Supports event-driven application architecture via DynamoDB Streams.

DynamoDB Basics

- A DynamoDB database is made up of tables.
- A table’s entries are termed items (= rows/records).
- Each item has attributes.
    - Schema-less – attributes can be added over time; can be null.
    - Primary key attribute must be declared at creation-time.
    - Maximum size of an item is 400 KB.

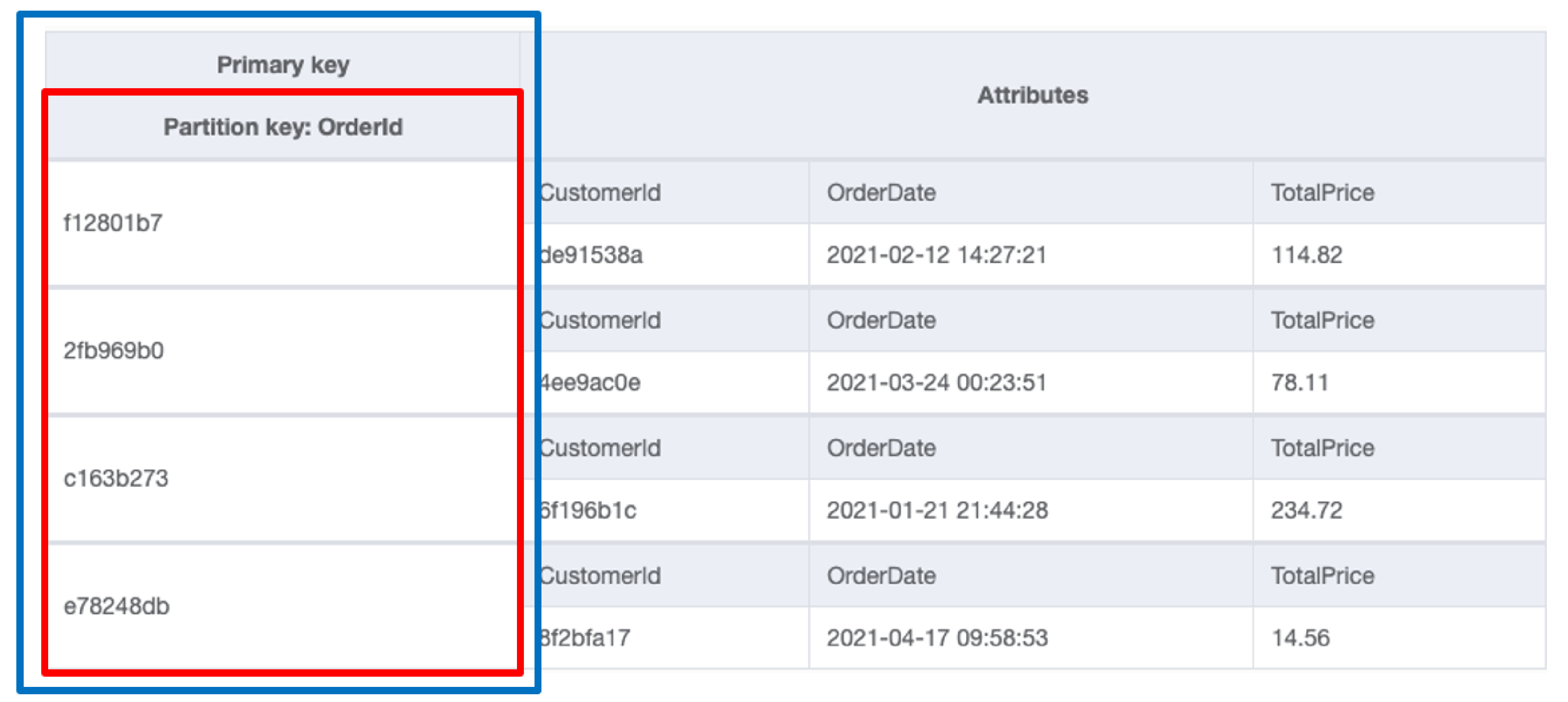
- Native Data types:
    - Scalar Types: String, Number, Binary, Boolean, Null.
    - Document Types: List, Map.
    - Set Types: String Set, Number Set, Binary Set.

The Primary Key attribute

- Option 1: Simple – Partition key (Hash key) only.
    - Partition key must be unique for each item.
    - Partition key value range must be “diverse” so that the data is distributed evenly.
    - Example: user_id for a users table.

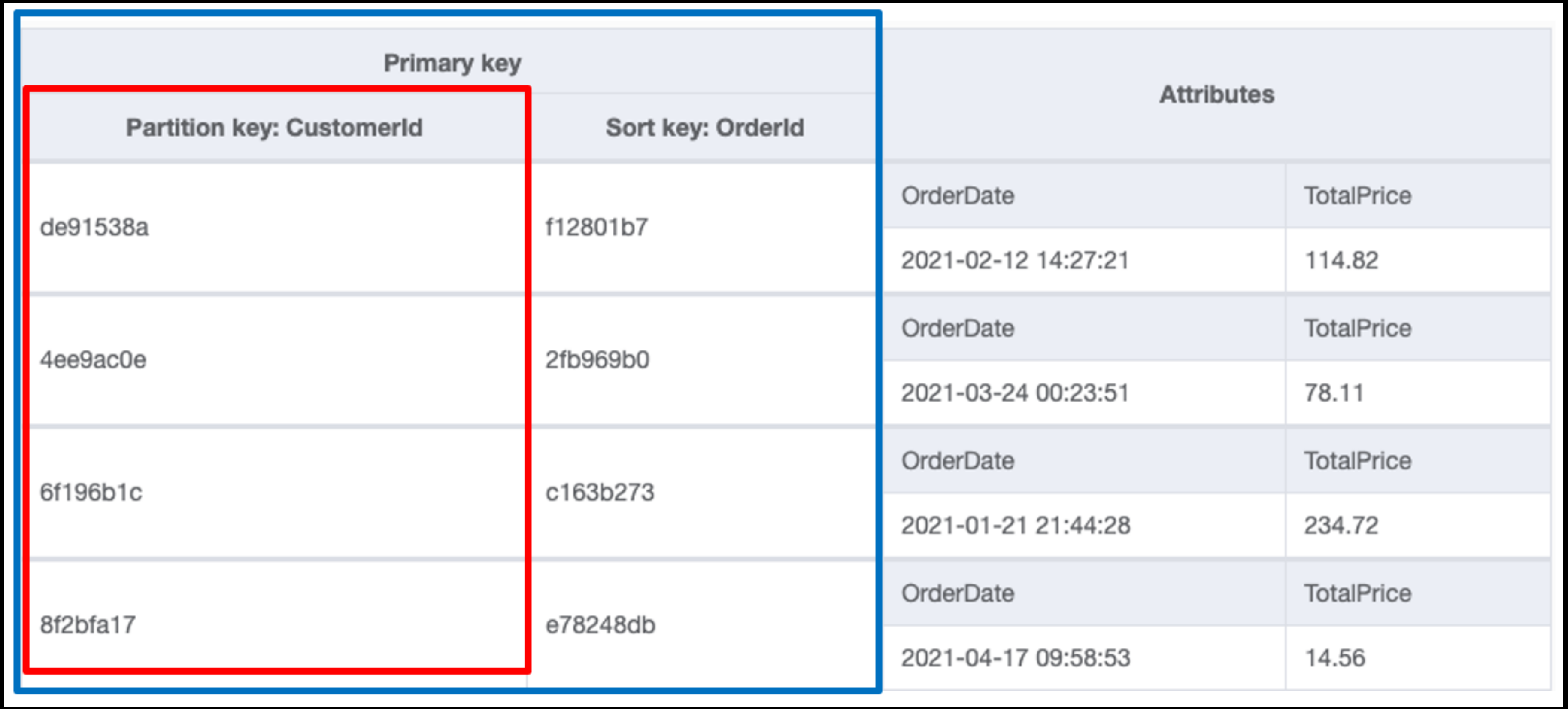
- Option 2: Composite - Partition key + Sort Key (Range key).
    - The combination must be unique.
    - Results in item collections within a table:
        - grouped by partition key.
        - ordered by the sort key.
    - Example: users-games table (Games are played by many users)
        - user_id (Partition key) + game_id (Sort key).

Simple Primary key



Primary key	Attributes		
Partition key: OrderId			
f12801b7	CustomerId	OrderDate	TotalPrice
	de91538a	2021-02-12 14:27:21	114.82
2fb969b0	CustomerId	OrderDate	TotalPrice
	4ee9ac0e	2021-03-24 00:23:51	78.11
c163b273	CustomerId	OrderDate	TotalPrice
	6f196b1c	2021-01-21 21:44:28	234.72
e78248db	CustomerId	OrderDate	TotalPrice
	8f2bfa17	2021-04-17 09:58:53	14.56

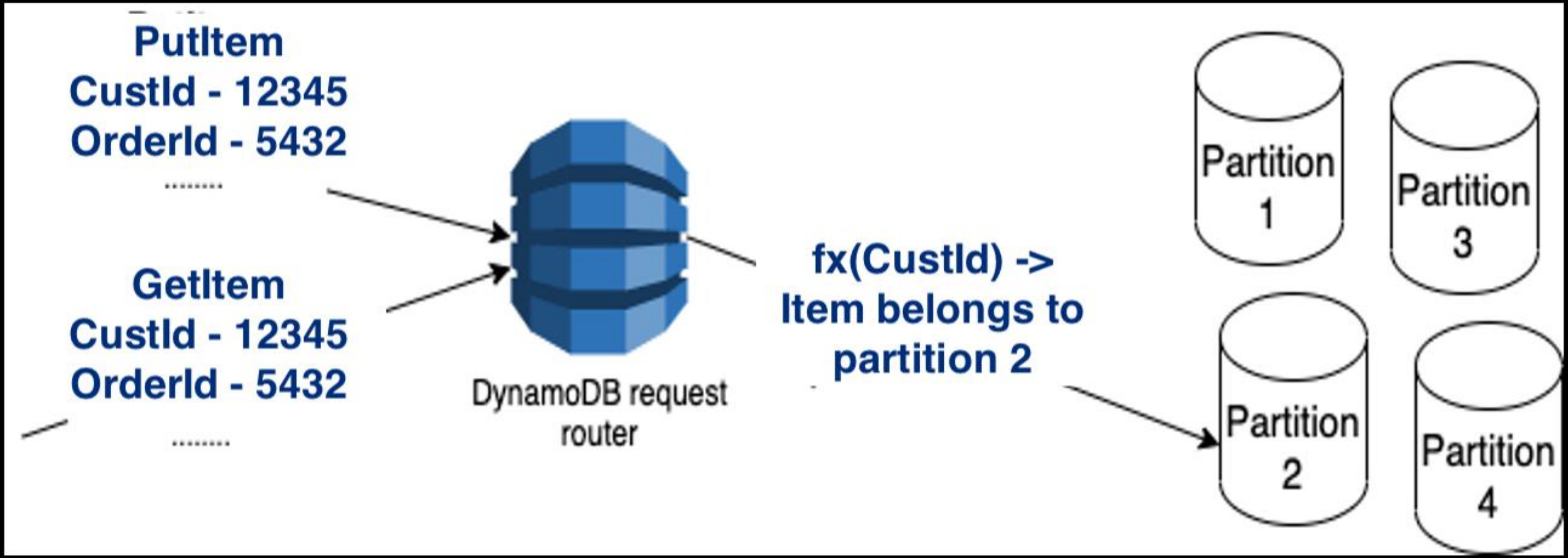
Composite Primary key



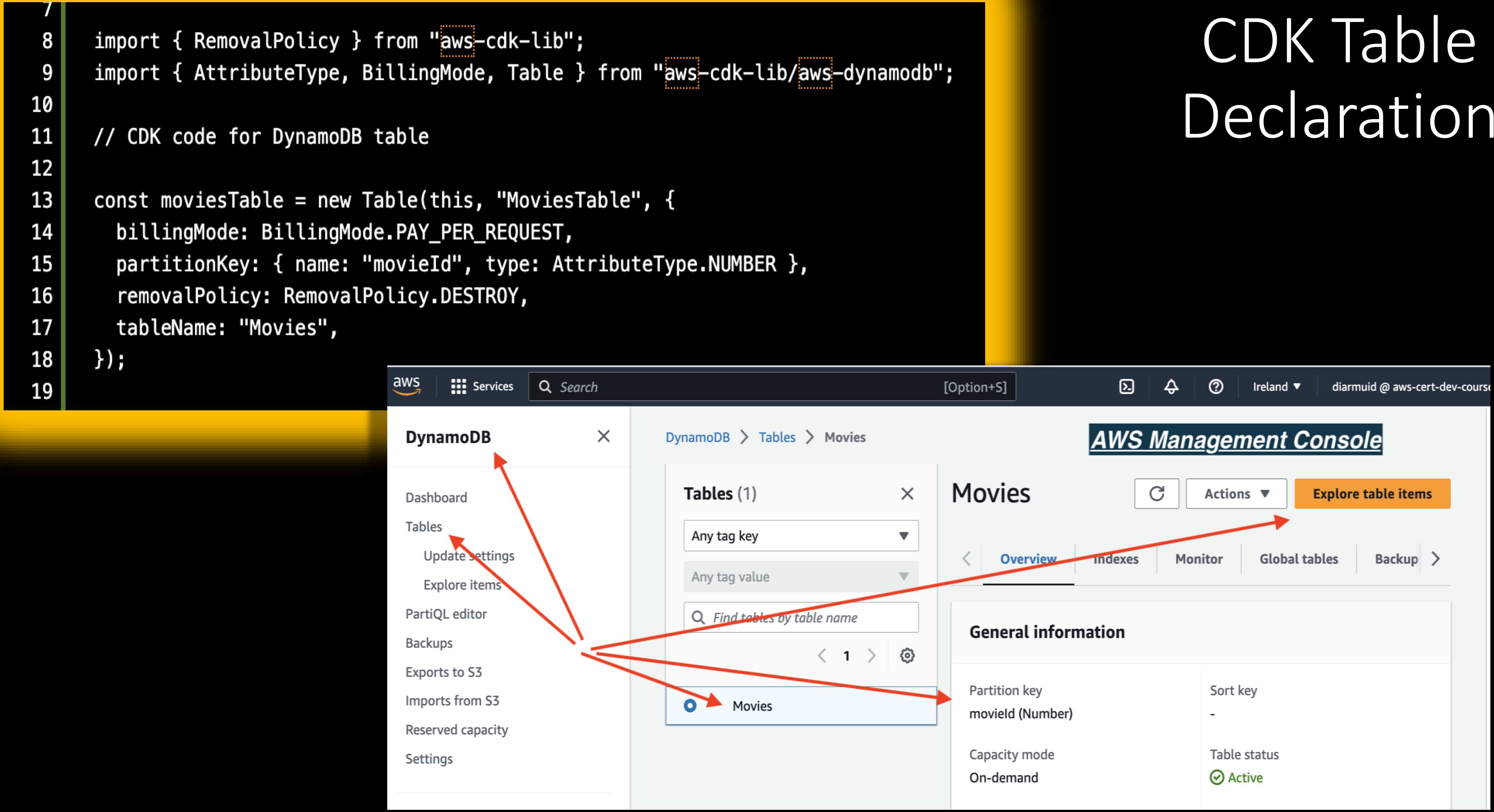
Primary key		Attributes	
Partition key: CustomerId	Sort key: OrderId		
de91538a	f12801b7	OrderDate	TotalPrice
		2021-02-12 14:27:21	114.82
4ee9ac0e	2fb969b0	OrderDate	TotalPrice
		2021-03-24 00:23:51	78.11
6f196b1c	c163b273	OrderDate	TotalPrice
		2021-01-21 21:44:28	234.72
8f2bfa17	e78248db	OrderDate	TotalPrice
		2021-04-17 09:58:53	14.56

Horizontal Scaling

- Data is distributed across a fleet of computers, where a single node holds a subset of a table's data, called 'partitions’ (max. 10GB).
- Adv – Scalability; Consistent performance.



CDK Table Declaration



```
import { RemovalPolicy } from "aws-cdk-lib";
import { AttributeType, BillingMode, Table } from "aws-cdk-lib/aws-dynamodb";

// CDK code for DynamoDB table

const moviesTable = new Table(this, "MoviesTable", {
  billingMode: BillingMode.PAY_PER_REQUEST,
  partitionKey: { name: "movieId", type: AttributeType.NUMBER },
  removalPolicy: RemovalPolicy.DESTROY,
  tableName: "Movies",
});
```

Movies

General information

Partition key
movieId (Number)

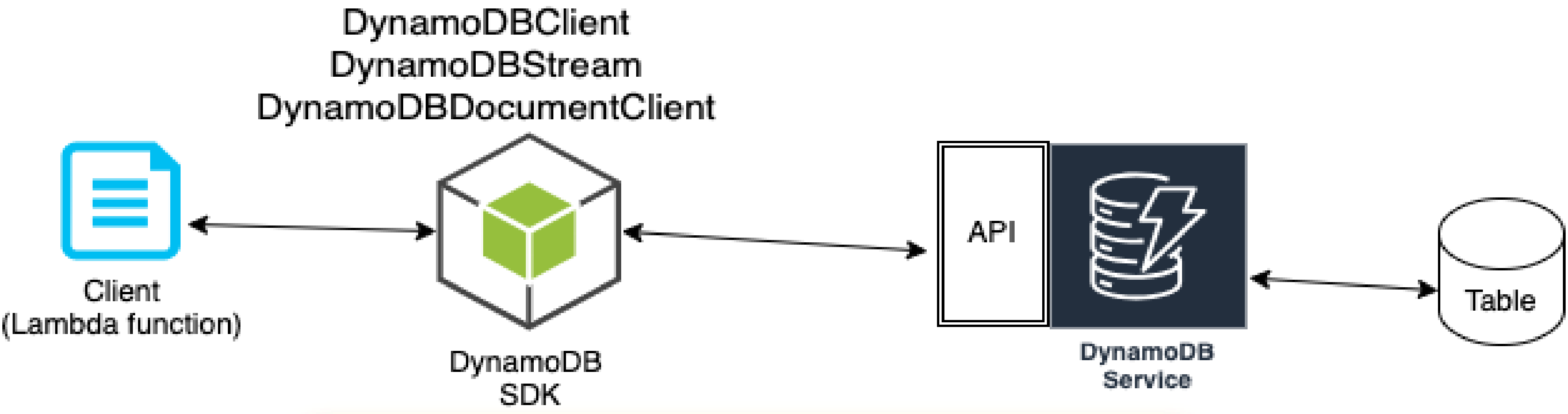
Sort key
-

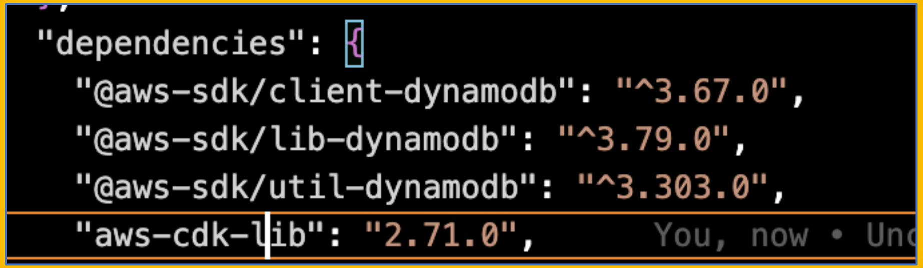
Capacity mode
On-demand

Table status
Active

DynamoDB API & SDK

- The SDK provides three classes for client interaction:
  - DynamoDBClient, DynamoDBStreams, and DynamoDBDocumentClient.





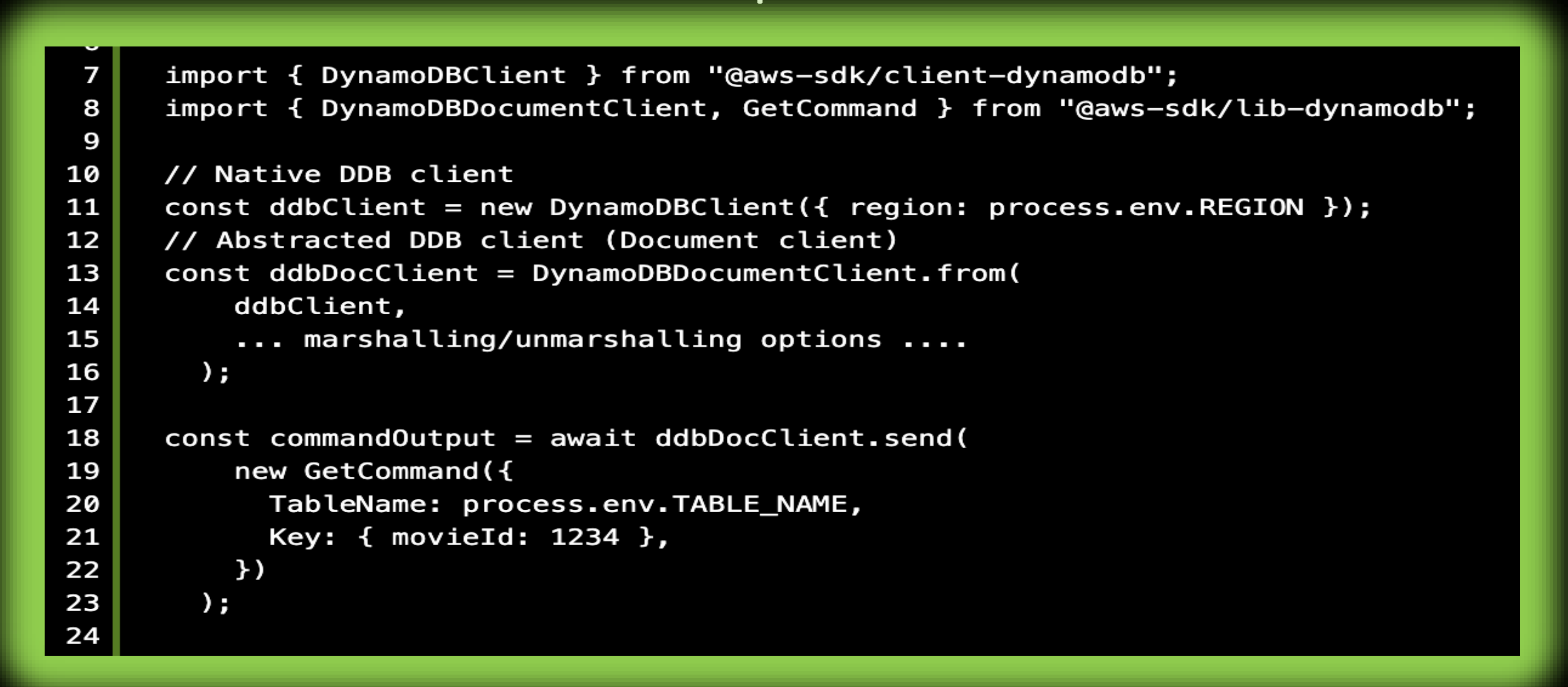
```
"dependencies": {
  "@aws-sdk/client-dynamodb": "^3.67.0",
  "@aws-sdk/lib-dynamodb": "^3.79.0",
  "@aws-sdk/util-dynamodb": "^3.303.0",
  "aws-cdk-lib": "2.71.0",        You, now • Unc
```

DynamoDB Table requests

- Three main types of actions:

1. Single-item requests - acts on a single, specific table item and requires the full primary key.
   - (CRUD) PutItem, GetItem, UpdateItem, DeleteItem.
2. Query - read a range of items;.
   - Only suitable for tables with composite primary key. ***
   - Must specify partition key; sort key is optional.
   - Query result are always from the same partition, i.e. an item collection.
3. Scan - reads all items.
   - Performed across all table partitions - very inefficient.
   - Client-side filtering possible

Sample Code



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

// Native DDB client
const ddbClient = new DynamoDBClient({ region: process.env.REGION });
// Abstracted DDB client (Document client)
const ddbDocClient = DynamoDBDocumentClient.from(
    ddbClient,
    ... marshalling/unmarshalling options ....
  );

const commandOutput = await ddbDocClient.send(
    new GetCommand({
      TableName: process.env.TABLE_NAME,
      Key: { movieId: 1234 },
    })
  );

```

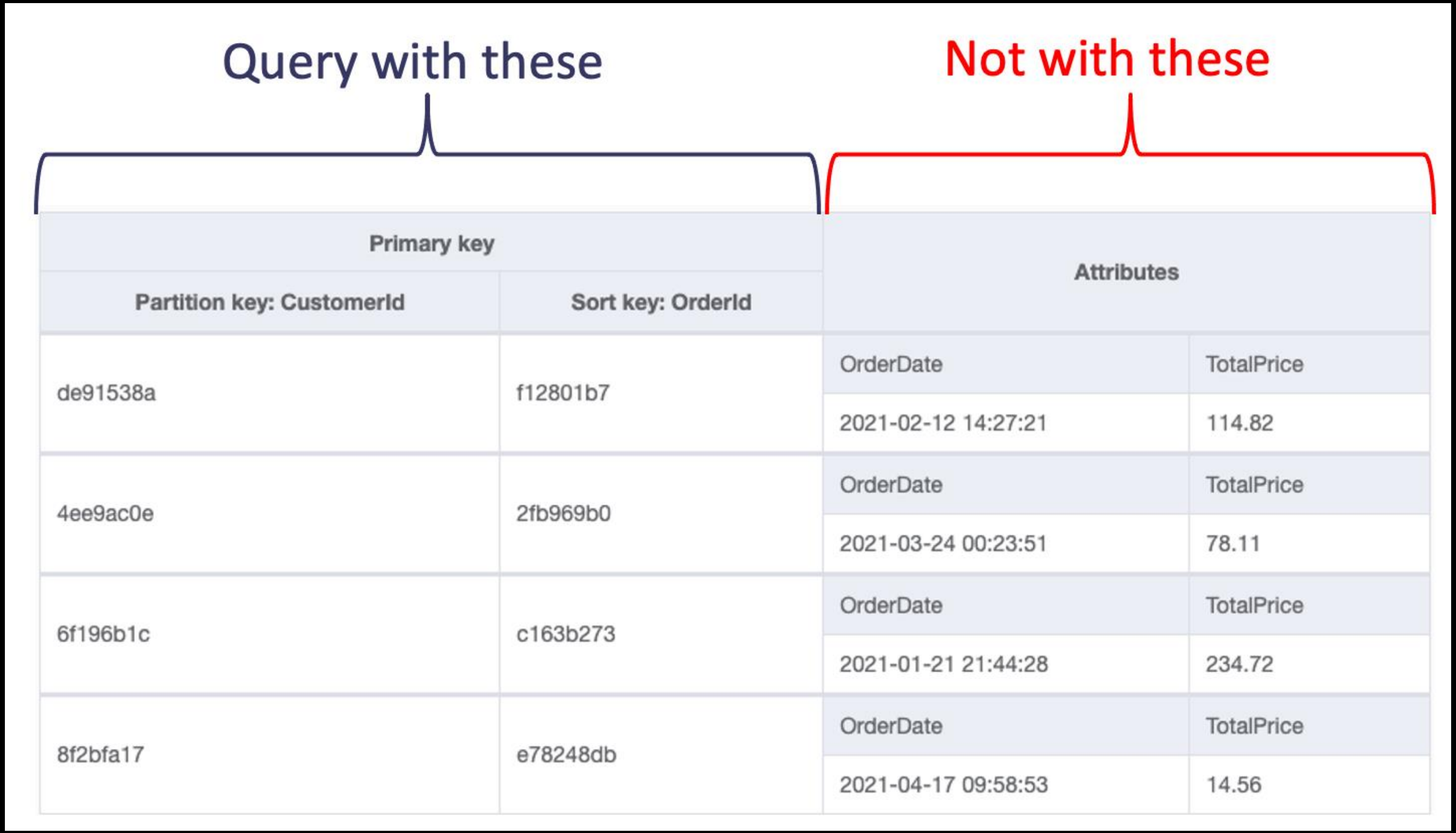
- Marshalling and Unmarshalling Data - The document client allows JS data types instead of DynamoDB's native AttributeValues. JS objects passed as parameters are marshalled into the AttributeValue shapes required by DDB. Responses from DDB are unmarshalled into plain JS objects.

Query actions

- Query requests return items based on:
  - Partition Key (equals (=) operator only)

    + Sort Key (=, <, <=, >, >=, Between, BeginsWith) – optional.
- Can include a Filter Expression for further filtering (performed on the client side).
- Query response:
  - Up to 1MB of data, or
  - Use a Limit parameter to reduce response size.
- Supports Pagination response.

Query actions



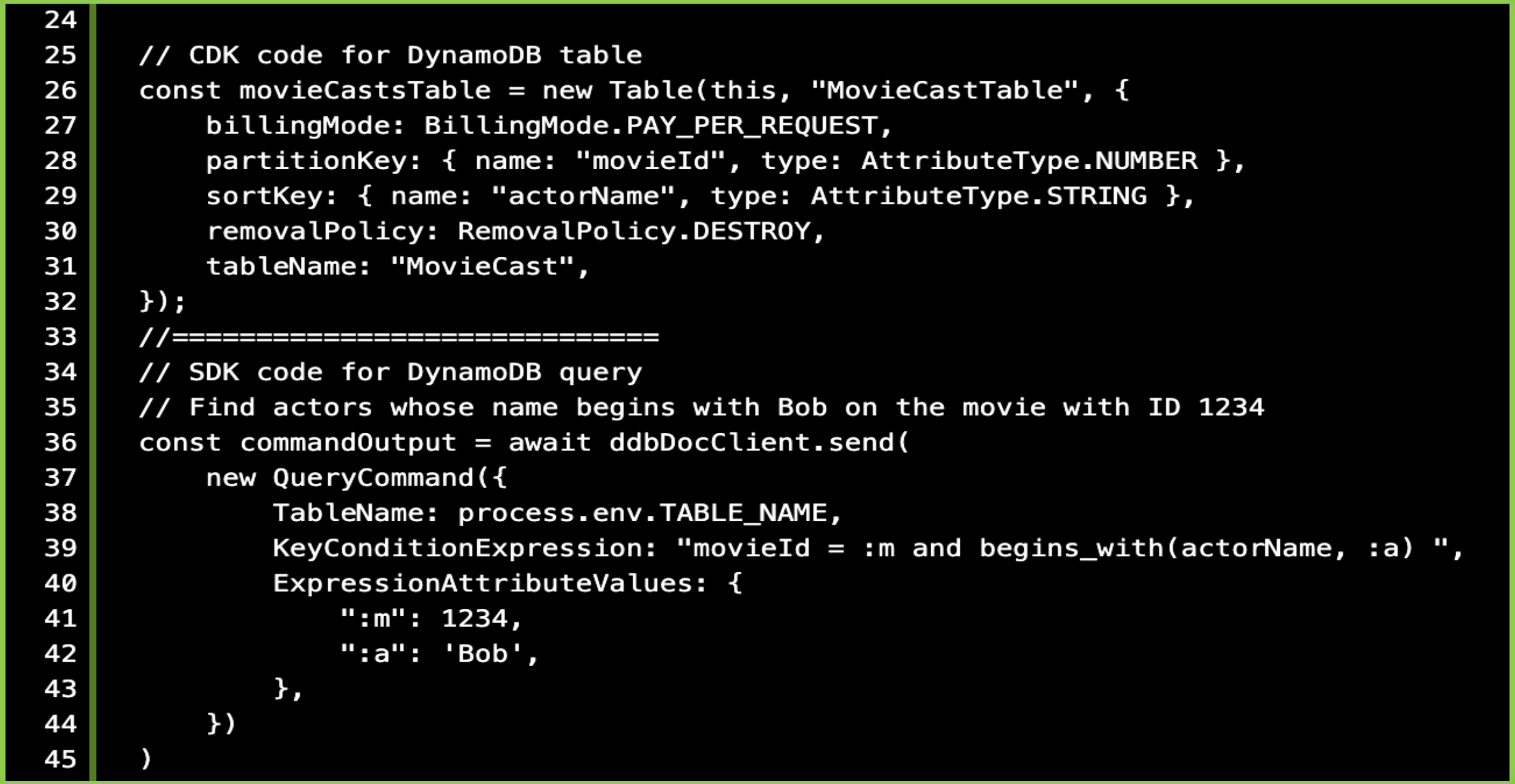
Query with these

Not with these

Primary key		Attributes	
Partition key: CustomerId	Sort key: OrderId		
de91538a	f12801b7	OrderDate	TotalPrice
		2021-02-12 14:27:21	114.82
4ee9ac0e	2fb969b0	OrderDate	TotalPrice
		2021-03-24 00:23:51	78.11
6f196b1c	c163b273	OrderDate	TotalPrice
		2021-01-21 21:44:28	234.72
8f2bfa17	e78248db	OrderDate	TotalPrice
		2021-04-17 09:58:53	14.56

Query Example

- A table stores the movie cast data, with one item per cast member.



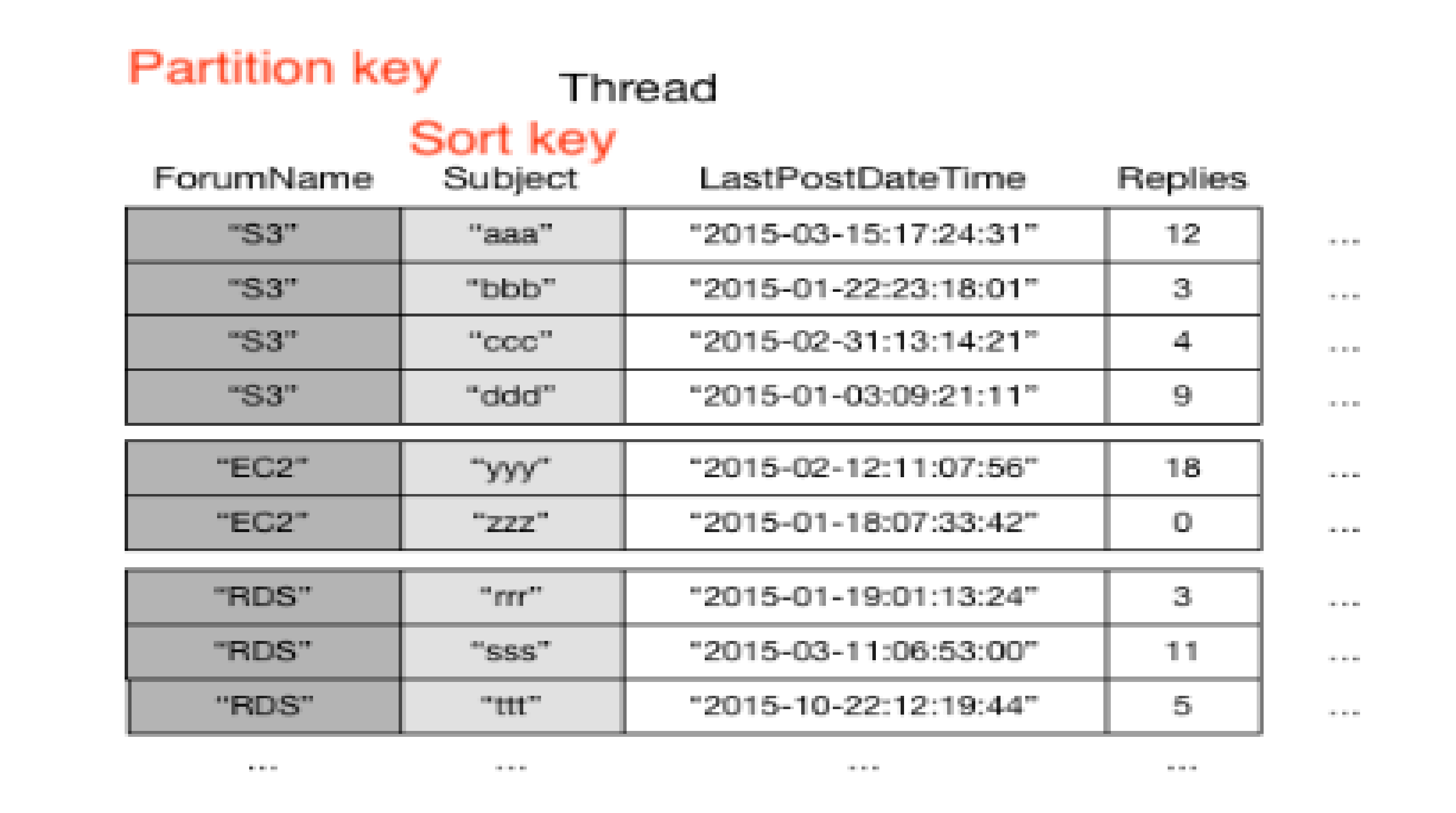
```

// CDK code for DynamoDB table
const movieCastsTable = new Table(this, "MovieCastTable", {
    billingMode: BillingMode.PAY_PER_REQUEST,
    partitionKey: { name: "movieId", type: AttributeType.NUMBER },
    sortKey: { name: "actorName", type: AttributeType.STRING },
    removalPolicy: RemovalPolicy.DESTROY,
    tableName: "MovieCast",
});
//=============================
// SDK code for DynamoDB query
// Find actors whose name begins with Bob on the movie with ID 1234
const commandOutput = await ddbDocClient.send(
    new QueryCommand({
        TableName: process.env.TABLE_NAME,
        KeyConditionExpression: "movieId = :m and begins_with(actorName, :a) ",
        ExpressionAttributeValues: {
            ":m": 1234,
            ":a": 'Bob',
        },
    })
)
```

Local Secondary Index (LSI)

- Only apply to tables with composite primary key.
- LSIs are based on an alternate sort key attribute.
- LSIs is stored local to the partition key.
- Max of five LSIs per table.
- Used with Query actions.
- The attribute chosen for LSI sort key must be a scalar String, Number, or Binary.
- LSI must be defined at table creation time.
- Each LSI entry is a projection of the table item.
- A table has: (1) A primary/main index (2) (optional) multiple LSIs + (optional) multiple GSIs (Global secondary indices)

Table Indices – Example (1/2)

- Example: Table of Discussion threads for a social media app.
  - The main / primary index



Partition key

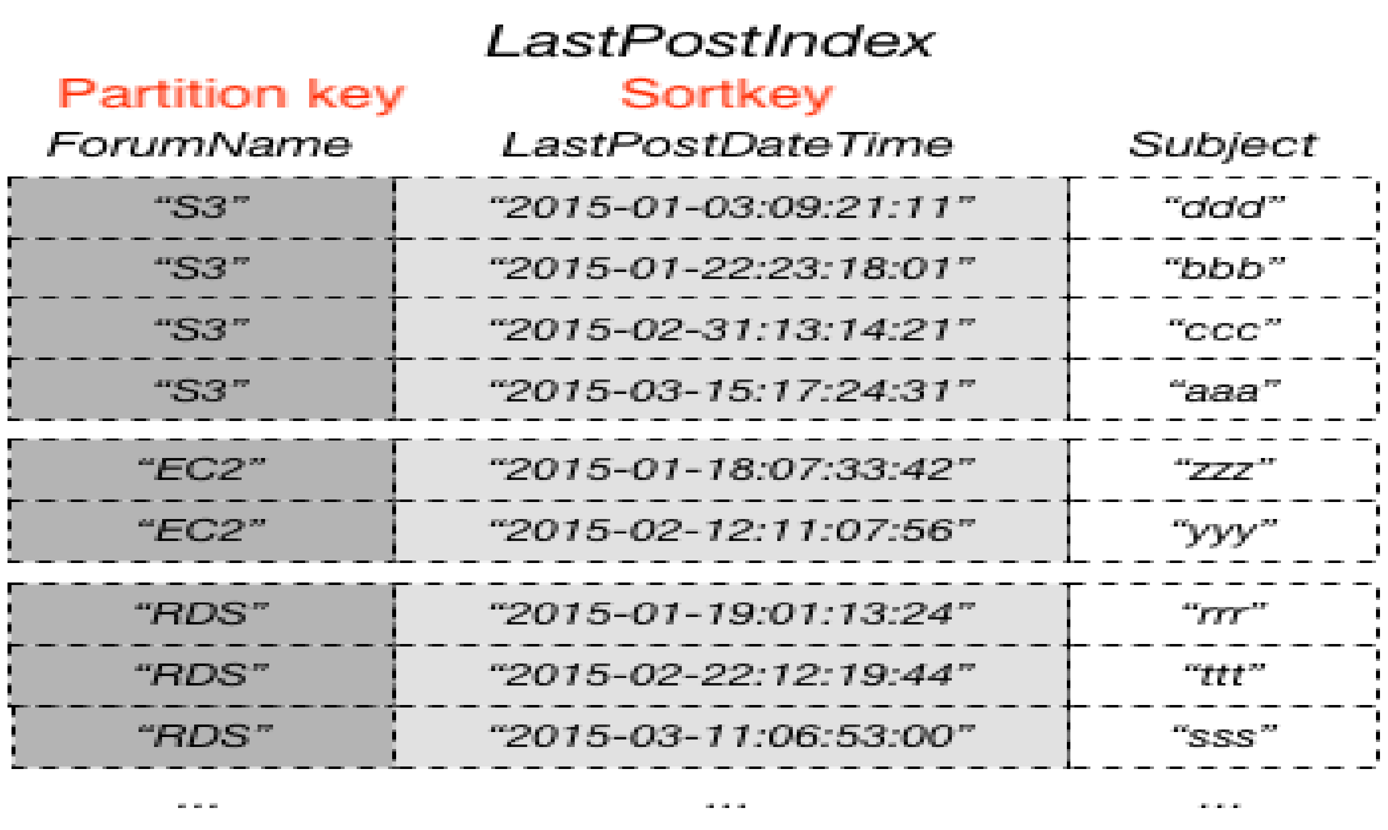
Thread

Sort key

ForumName	Subject	LastPostDateTime	Replies	
"S3"	"aaa"	"2015-03-15:17:24:31"	12	...
"S3"	"bbb"	"2015-01-22:23:18:01"	3	...
"S3"	"ccc"	"2015-02-31:13:14:21"	4	...
"S3"	"ddd"	"2015-01-03:09:21:11"	9	...
"EC2"	"yyy"	"2015-02-12:11:07:56"	18	...
"EC2"	"zzz"	"2015-01-18:07:33:42"	0	...
"RDS"	"rrr"	"2015-01-19:01:13:24"	3	...
"RDS"	"sss"	"2015-03-11:06:53:00"	11	...
"RDS"	"ttt"	"2015-10-22:12:19:44"	5	...
...	...	...	...	

Table Indices – Example (2/2)

- Example: Table of Discussion threads for a social media app.
  - A local secondary index (LSI)



LastPostIndex

Partition key	Sortkey	
ForumName	LastPostDateTime	Subject
“S3”	“2015-01-03:09:21:11”	“ddd”
“S3”	“2015-01-22:23:18:01”	“bbb”
“S3”	“2015-02-31:13:14:21”	“ccc”
“S3”	“2015-03-15:17:24:31”	“aaa”
“EC2”	“2015-01-18:07:33:42”	“zzz”
“EC2”	“2015-02-12:11:07:56”	“yyy”
“RDS”	“2015-01-19:01:13:24”	“rrr”
“RDS”	“2015-02-22:12:19:44”	“ttt”
“RDS”	“2015-03-11:06:53:00”	“sss”
...	...	...

Information

- Watch https://www.youtube.com/watch?v=ErPrf74RUDY