



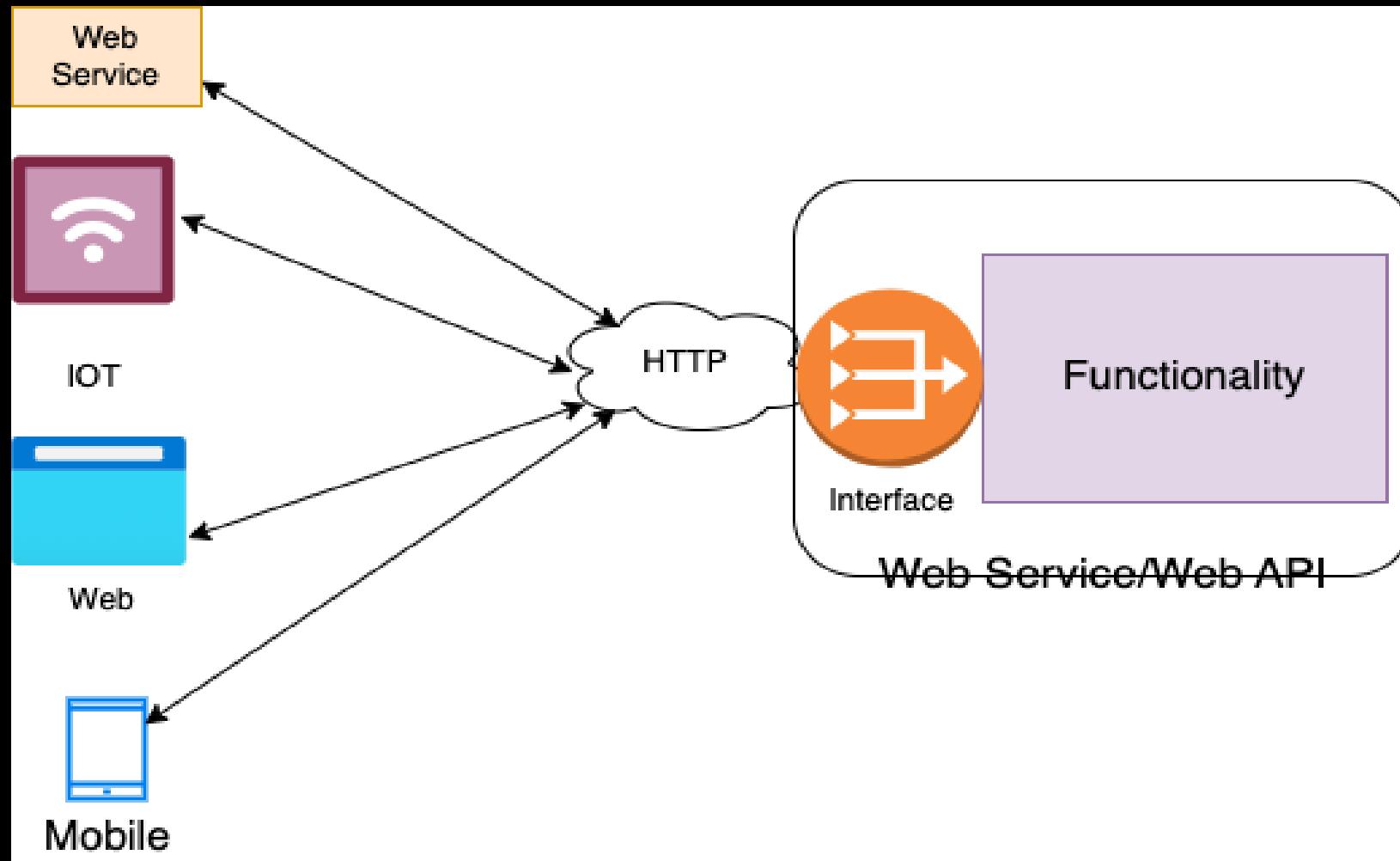
Serverless Web APIs

The AWS-related services

Serverless?

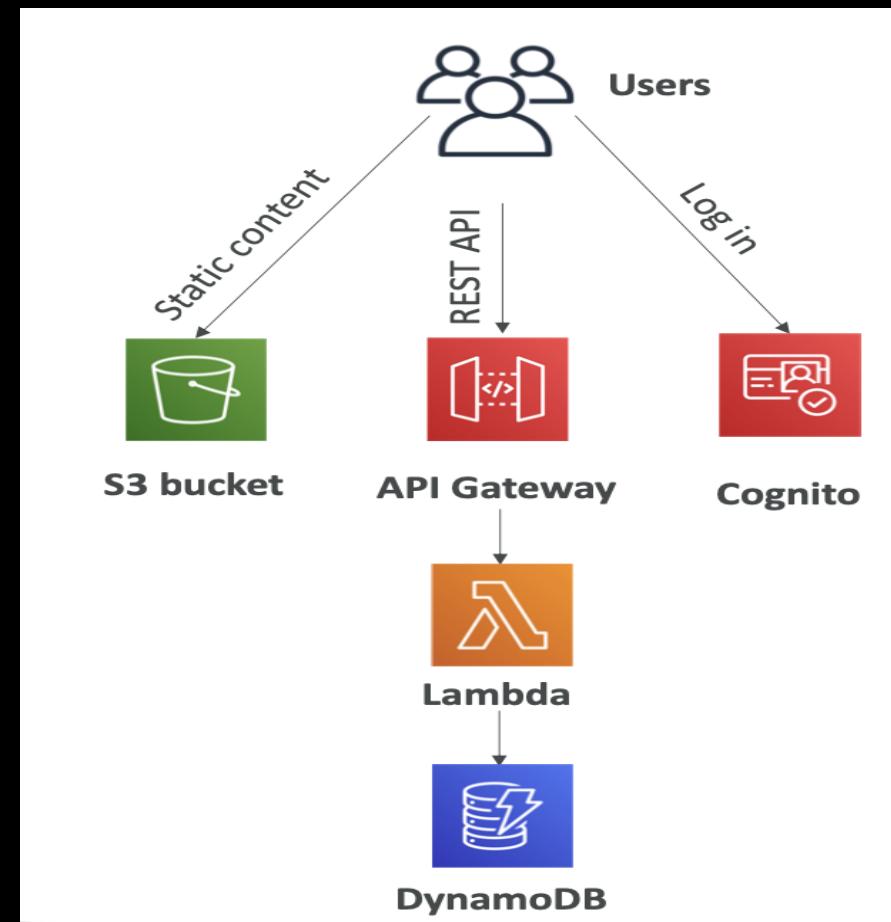
- Serverless relieves the developer of responsibilities:
 1. No servers to provision or manage.
 2. Auto-Scales with usage.
 3. No Up-front cost.
 4. Availability and fault tolerance built-in.

Web APIs?



Serverless Services on AWS

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





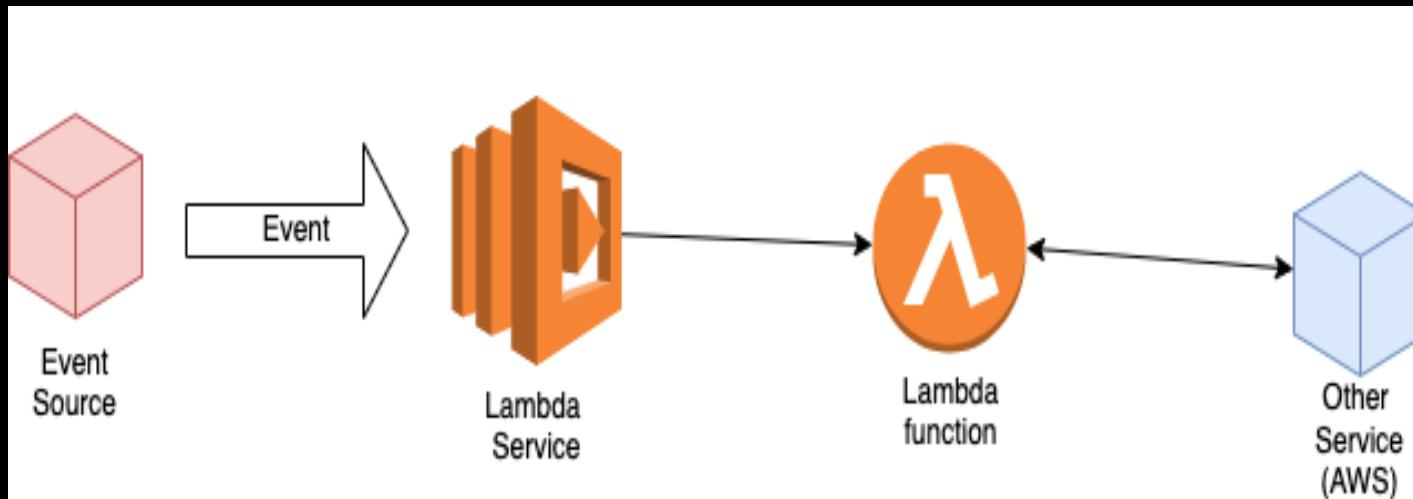
AWS Lambda Service

Serverless Compute

AWS Lambda

- The Lambda Service is an event-driven, serverless computing platform.
- It runs code (a function) in response to events.
- It manages the computing resources (CPU, memory, networking) required by that code.
- “Custom code that runs in an ephemeral container” Mike Robins
- FaaS (Functions as a Service)
 - Preceded by IaaS, PaaS, SaaS.

Lambda runtime model.



- Event Source (Trigger), e.g. HTTP request; Data state change, e.g. database, S3; Resource state change, 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 be executed upon 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

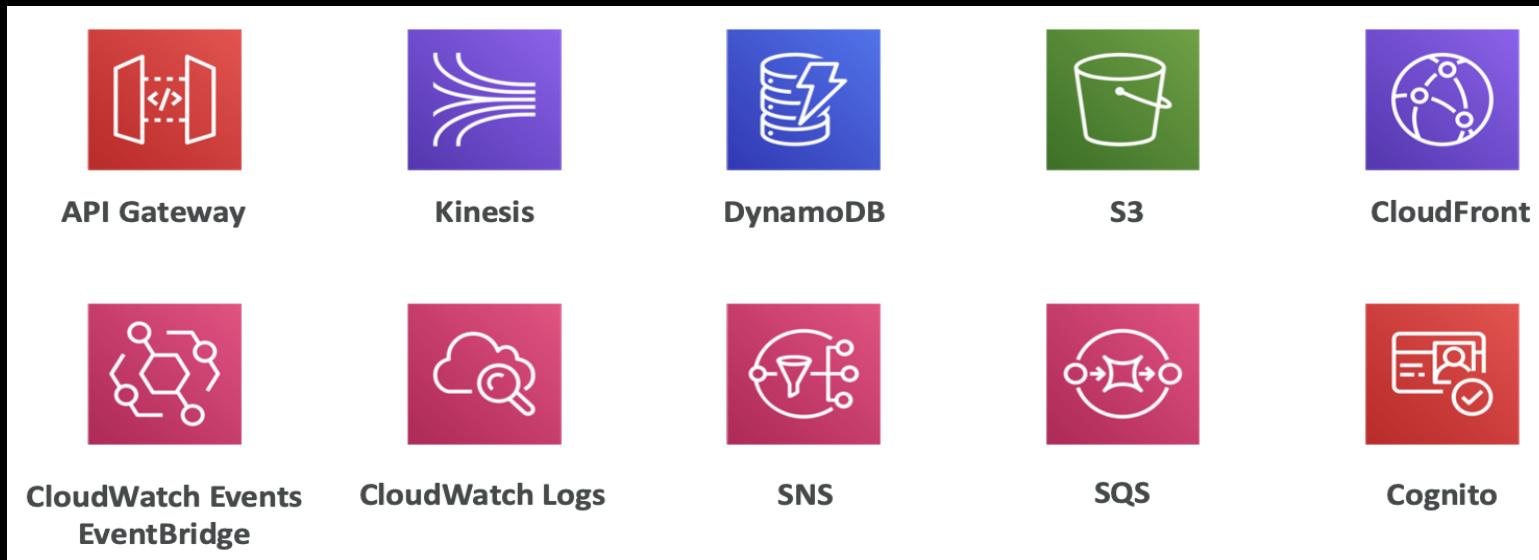
- Lambda service provides a memory control to configure a function's compute power requirements.
 - The % of CPU core and network capacity are computed in proportion to its RAM.
- RAM:
 - From 128MB to 3,008 MB in 64 MB increments
 - The more RAM you add, the more vCPU credits you get.
 - 1,792 MB RAM 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.
 - Invoke it (trigger it) using the AWS CLI.
 - Check `console.log()` statement output in the CloudWatch Logs.

Lambda integration

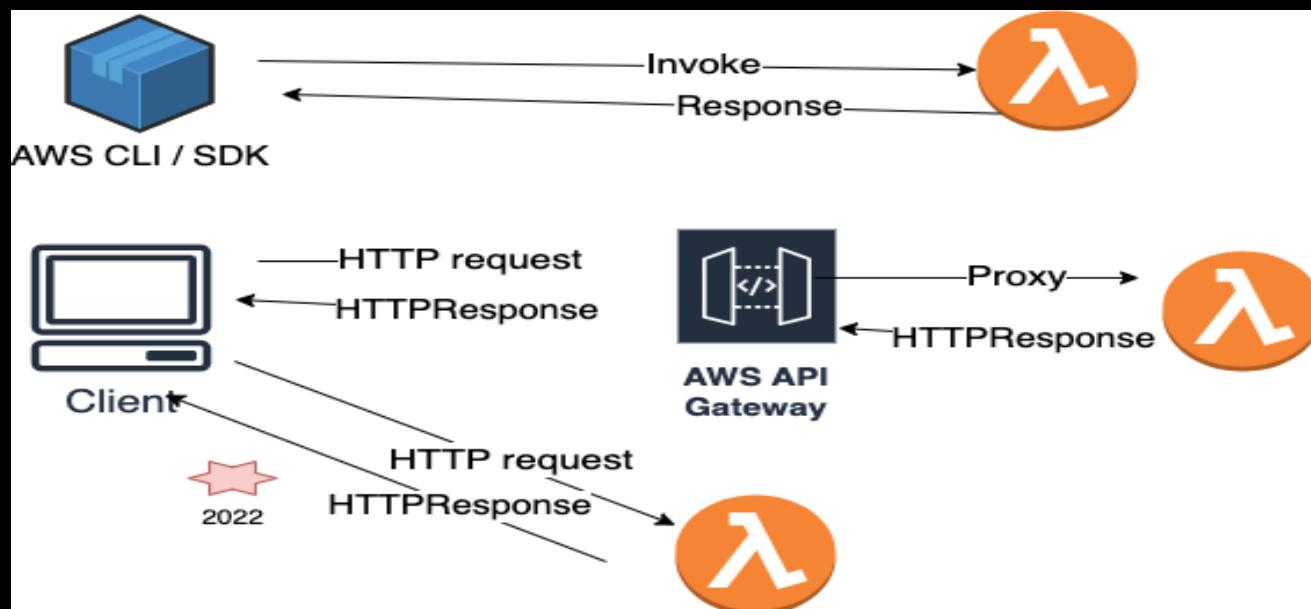
- What services can trigger a lambda function?



- Integration models:
 1. Synchronous.
 2. Asynchronous.
 3. Poll-based

Synchronous Integration

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

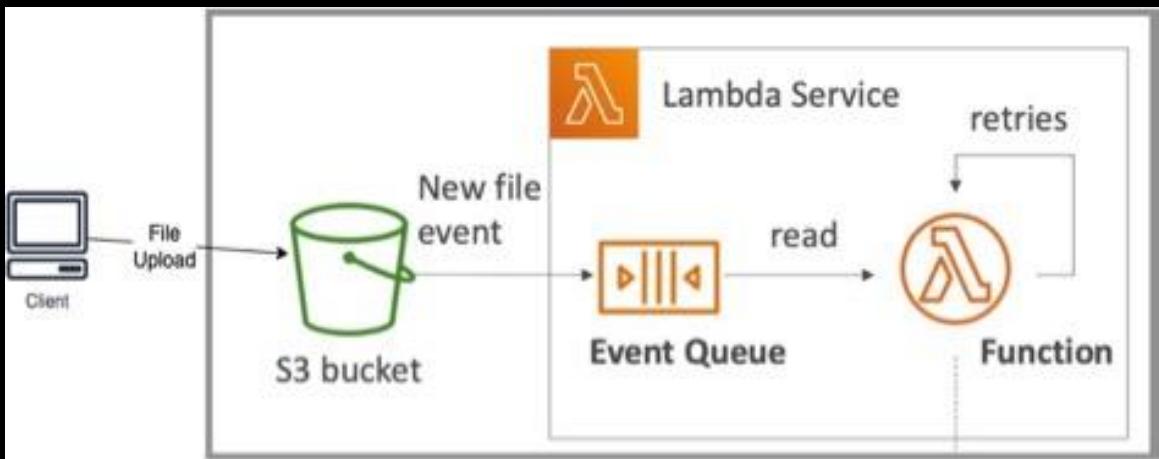


Demo

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

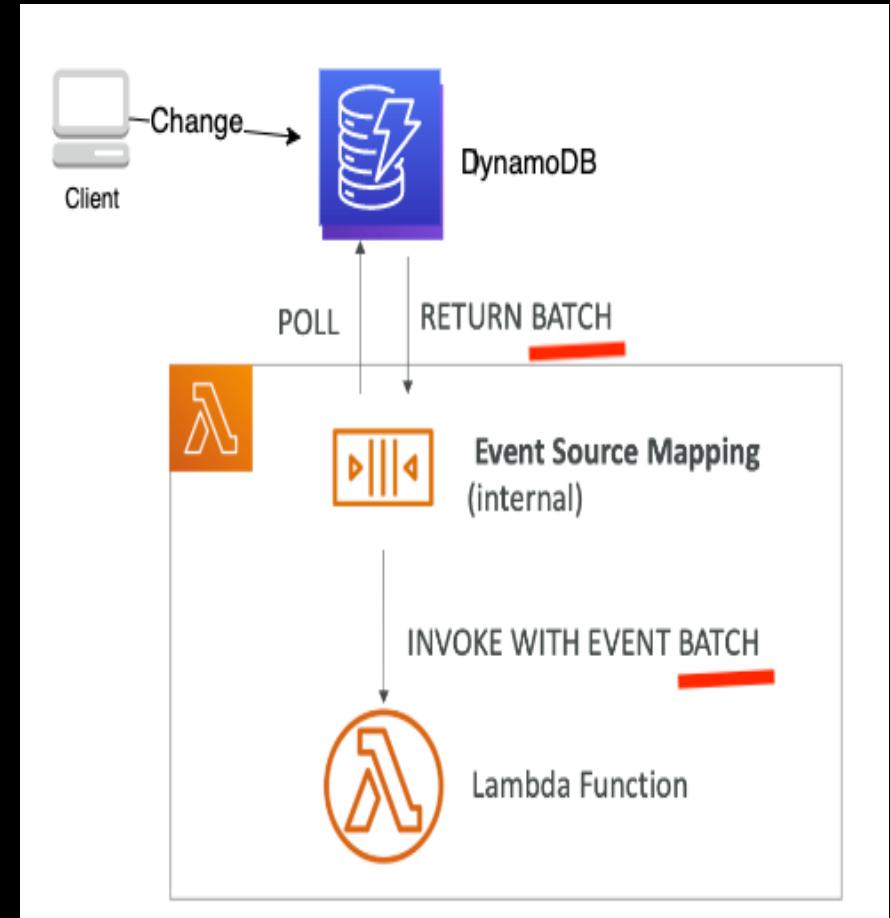
Asynchronous Integration.

- Trigger (Event Source): S3, SNS, CloudWatch.
- Lambda service places the events in an internal queue.
- 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.



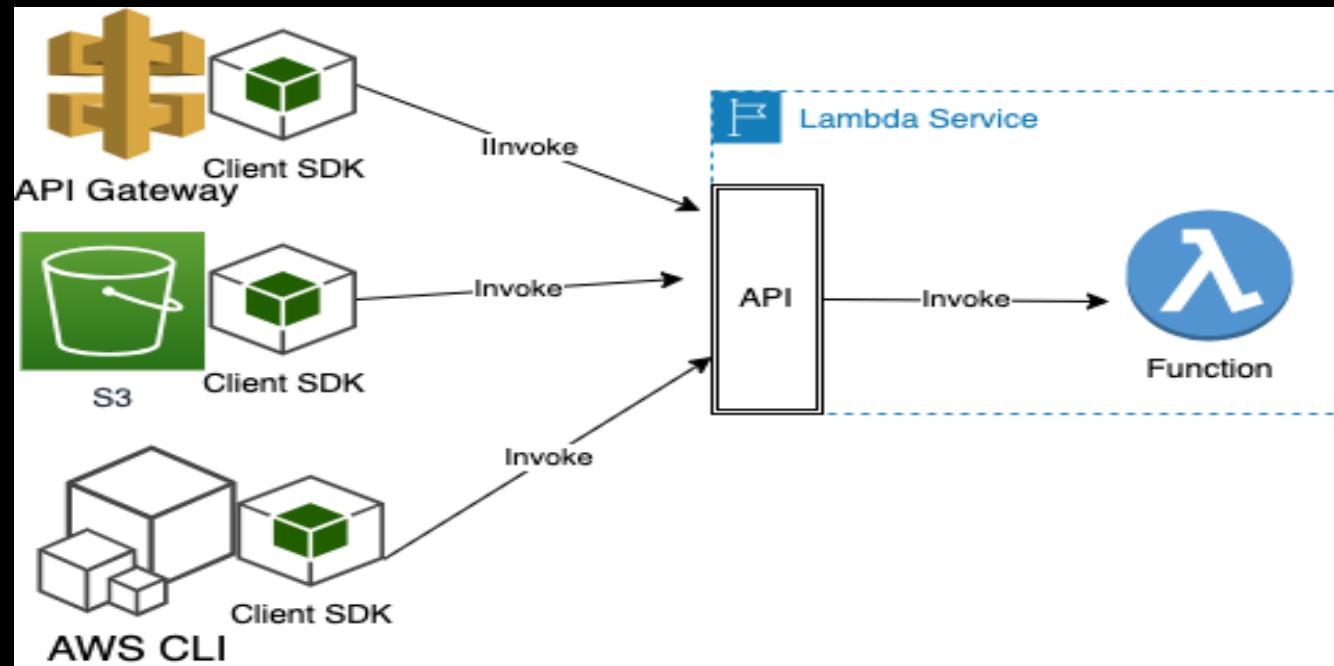
Event source mapping. (Poll-based Integration)

- 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 an API.
- Used by all other services that trigger Lambda functions across all integration models.
- Can pass any event payload structure you want.



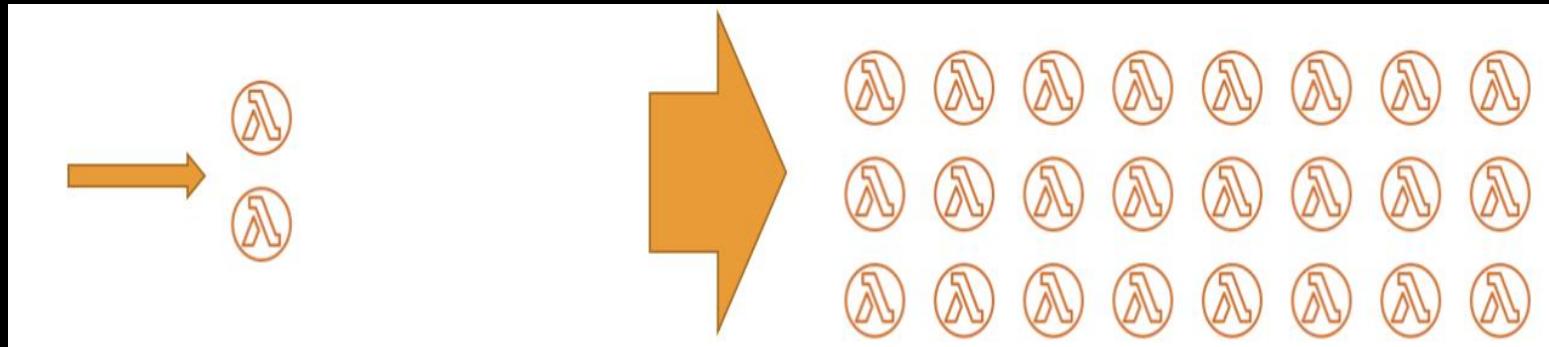
Execution Role (IAM Role)

- 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.
- Best practice: 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 if:
 - The IAM policy attached to the principal authorizes it, or
 - The function's resource-based policy authorizes it.
- Ex. An AWS S3 service can trigger a Lambda function if the function's resource-based policy permits it.

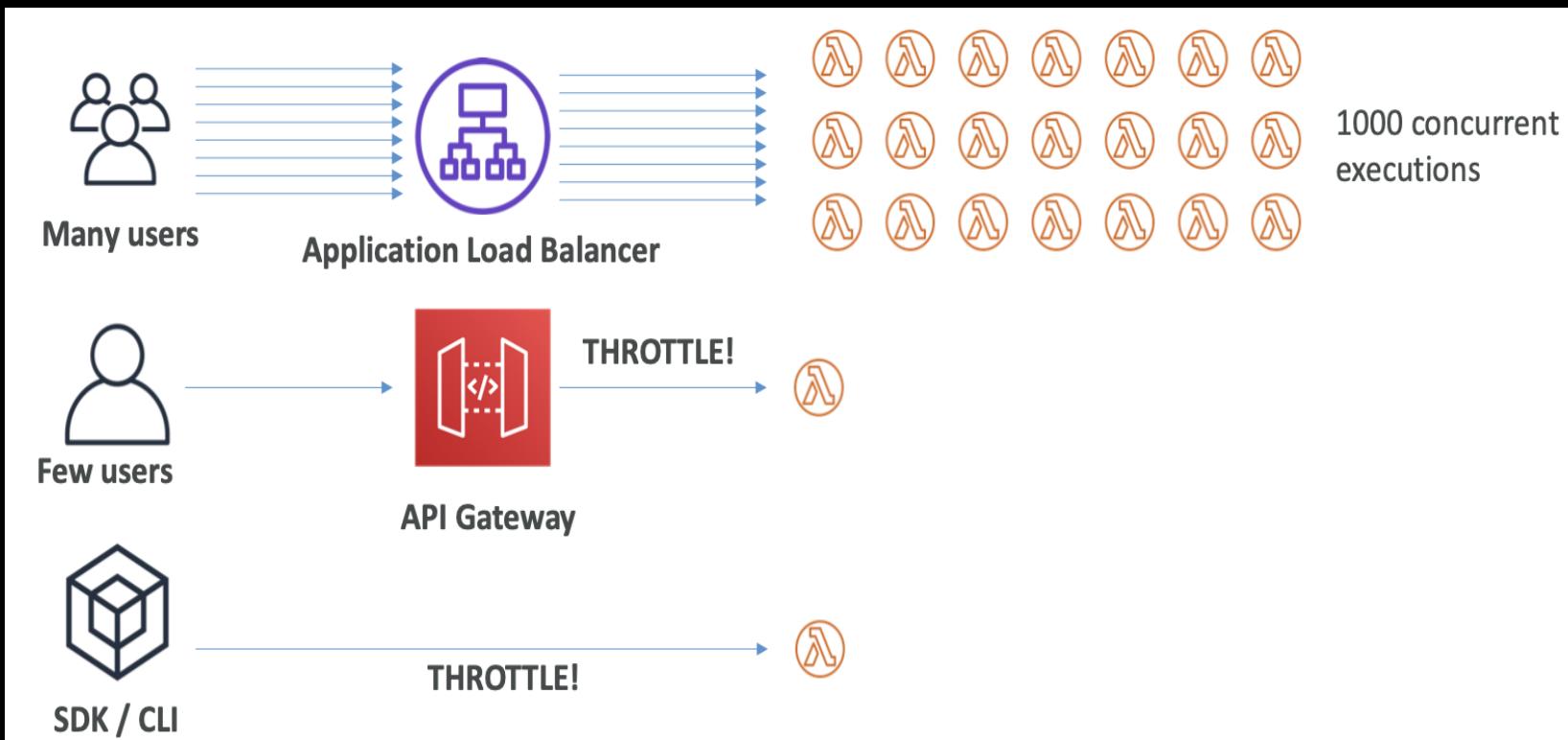
Concurrency and Throttling



- AWS account lambda concurrency limit is set to 1000 executions.
- Can set a reserved concurrency at the function level (= limit).
 - “Throttle” error response when limit is 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

- Use reserved concurrency to avoid the following:



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 reused; Init code not executed.
- Problem: Lengthy init code (LoC, dependencies, SDK) effects overall event processing time.
 - Greater latency with cold start executions.
- Solution – Use Provisioned Concurrency:
 - Micro VMs are pre-allocated in advance; Init code executed during pre-allocation.
 - Cold starts avoided (or minimized)
 - 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.
 - Records in the same table can have different attributes.
- No support for 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.
- Each table has a primary key (must be decided at creation time).
- A table's entries are termed items (= rows/records).
- Each item has attributes.
 - Schema-less - can be added over time; can be null.
 - Primary key attribute(s) declared at creation-time.
 - Maximum size of an item is 400 KB.
- Data types supported are:
 - Scalar Types: String, Number, Binary, Boolean, Null.
 - Document Types: List, Map.
 - Set Types: String Set, Number Set, Binary Set.

The Primary Key

- Option 1: Simple - Partition key (Hash key) only.
 - Partition key must be unique for each item.
 - Partition key value range must be “diverse” to ensure 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. and
 - ordered by the sort key.
 - Example: users-games table (Games played by users)
 - user_id (Partition key) + game_id (Sort key).

Simple Primary key example

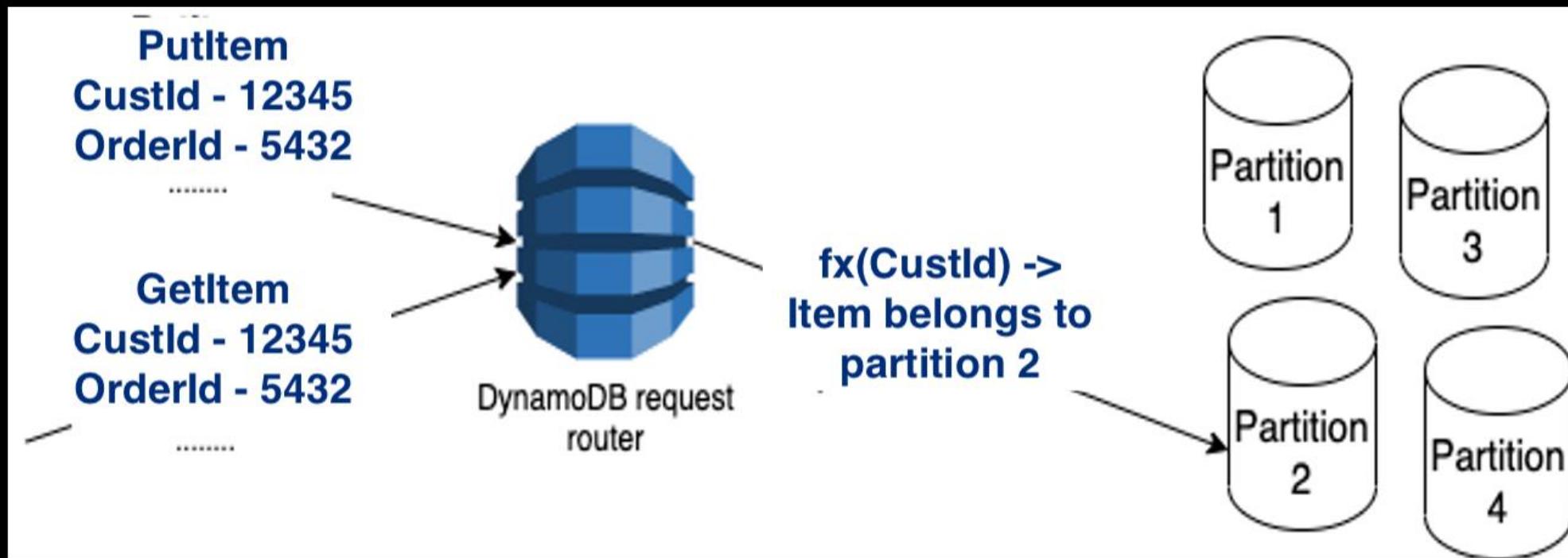
Primary key		Attributes		
Partition key: OrderId		CustomerId	OrderDate	TotalPrice
f12801b7		de91538a	2021-02-12 14:27:21	114.82
2fb969b0		4ee9ac0e	2021-03-24 00:23:51	78.11
c163b273		6f196b1c	2021-01-21 21:44:28	234.72
e78248db		8f2bfa17	2021-04-17 09:58:53	14.56

Composite Primary key example

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

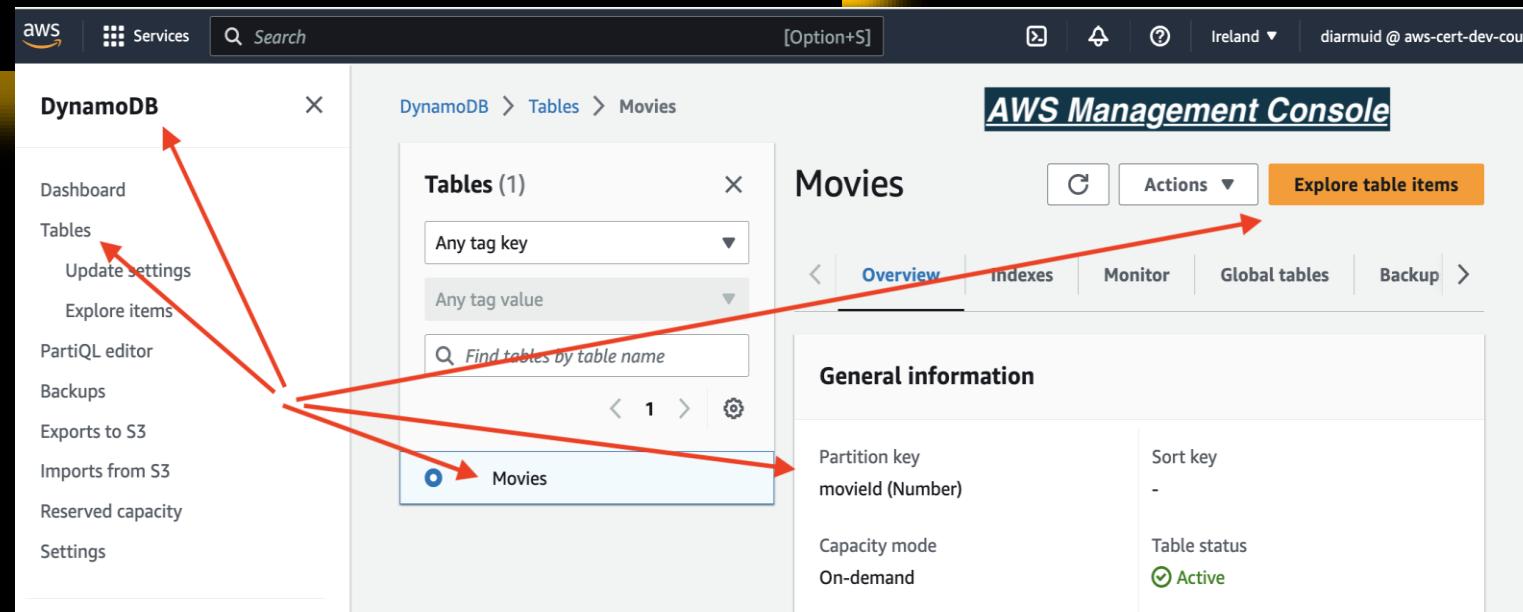
Horizontal Scaling

- 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.



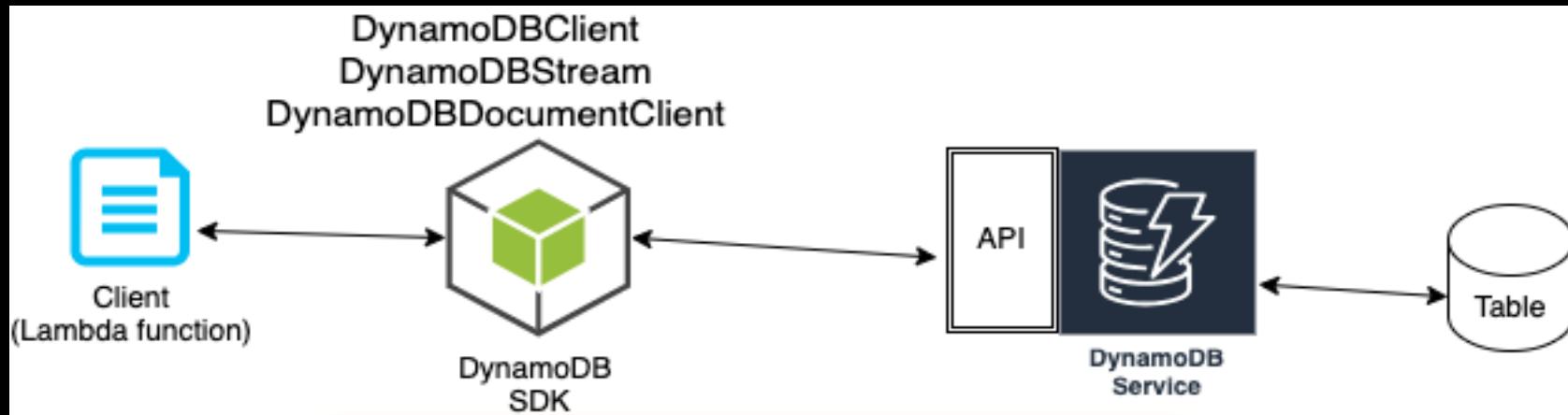
CDK Table Declaration

```
/  
8 import { RemovalPolicy } from "aws-cdk-lib";  
9 import { AttributeType, BillingMode, Table } from "aws-cdk-lib/aws-dynamodb";  
10 // CDK code for DynamoDB table  
11  
13 const moviesTable = new Table(this, "MoviesTable", {  
14   billingMode: BillingMode.PAY_PER_REQUEST,  
15   partitionKey: { name: "movieId", type: AttributeType.NUMBER },  
16   removalPolicy: RemovalPolicy.DESTROY,  
17   tableName: "Movies",  
18 });  
19
```



DynamoDB API & SDK

- The SDK provides three classes for client interaction:
 - DynamoDBClient, DynamoDBStream, 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 • Undo
```

DynamoDB API

- Three types of database actions:
 1. Single-item requests - acts on a single, specific table item and requires the full primary key.
 - PutItem, GetItem, UpdateItem, DeleteItem.
 2. Query - reads a range/collection of item, all with the same partition key.
 - Only suitable for tables with composite primary key. ***
 - Query result are always from the same partition (item collection).
 3. Scan - reads a set of items, but searches across the entire table; an inefficient operation.

Sample Lambda Code

```
7 import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
8 import { DynamoDBDocumentClient, GetCommand } from "@aws-sdk/lib-dynamodb";
9
10 // Native DDB client
11 const ddbClient = new DynamoDBClient({ region: process.env.REGION });
12 // Abstracted DDB client (Document client)
13 const ddbDocClient = DynamoDBDocumentClient.from(
14     ddbClient,
15     ... marshalling/unmarshalling options ...
16 );
17
18 const commandOutput = await ddbDocClient.send(
19     new GetCommand({
20         TableName: process.env.TABLE_NAME,
21         Key: { movieId: 1234 },
22     })
23 );
24
```

- Marshalling and Unmarshalling Data – Our code uses JS/TS types instead of DynamoDB's native AttributeValue types.
 - JS objects are marshalled into AttributeValue shapes for DB write operations.
 - Responses from DDB are unmarshalled into plain JS objects.

Query actions

- Query requests return items based on:
 - Partition Key (=) 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	OrderDate	TotalPrice
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.

```
24
25 // CDK code for DynamoDB table
26 const movieCastsTable = new Table(this, "MovieCastTable", {
27   billingMode: BillingMode.PAY_PER_REQUEST,
28   partitionKey: { name: "movieId", type: AttributeType.NUMBER },
29   sortKey: { name: "actorName", type: AttributeType.STRING },
30   removalPolicy: RemovalPolicy.DESTROY,
31   tableName: "MovieCast",
32 });
33 =====
34 // SDK code for DynamoDB query
35 // Find actors whose name begins with Bob on the movie with ID 1234
36 const commandOutput = await ddbDocClient.send(
37   new QueryCommand({
38     TableName: process.env.TABLE_NAME,
39     KeyConditionExpression: "movieId = :m and begins_with(actorName, :a) ",
40     ExpressionAttributeValues: {
41       ":m": 1234,
42       ":a": 'Bob',
43     },
44   })
45 )
```

Local Secondary Index (LSI)

- Only apply to table's with composite primary key.
- LSIs are based on an alternate sort key attribute.
- LSIs are 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)

LSI example

Primary key		Attributes	
Partition key: CustomerId	Sort key: OrderId	OrderDate	TotalPrice
de91538a	f12801b7	2021-02-12 14:27:21	114.82
4ee9ac0e	2fb969b0	2021-03-24 00:23:51	78.11
6f196b1c	c163b273	LSI 1: Customer ID (PK) + OrderDate (SK - string)	
8f2bfaf17	e78248db	LSI 2: Customer ID (PK) + TotalPrice (SK)	

LSI example

- Sample Query:
- LSI 1
 - Get orders for customer 1234 with a date that begins with 2021.
 - Get orders for customer 1234 with a date that begins with 2021-06.
 - Get order for customer 1234 on 2022-93-22.
- LSI 2
 - Get orders with a total price < 200 for customer 1234.
 - Get o
 - rders with a total price >= 300 for customer 1234.

Information

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