Using AWS Lambda with Amazon DynamoDB

PDF (lambda-dg.pdf#with-ddb)
Kindle (https://www.amazon.com/dp/B07GFJLN6D)
RSS (lambda-updates.rss)

You can use an AWS Lambda function to process records in an Amazon DynamoDB stream (https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html) . With DynamoDB Streams, you can trigger a Lambda function to perform additional work each time a DynamoDB table is updated.

Lambda reads records from the stream and invokes your function synchronously (./invocation-sync.html) with an event that contains stream records. Lambda reads records in batches and invokes your function to process records from the batch.

Example DynamoDB Streams record event

```
{
  "Records": [
    {
      "eventID": "1",
      "eventVersion": "1.0",
      "dynamodb": {
        "Keys": {
          "Id": {
            "N": "101"
          }
        },
        "NewImage": {
          "Message": {
            "S": "New item!"
          },
          "Id": {
            "N": "101"
          }
        },
        "StreamViewType": "NEW_AND_OLD_IMAGES",
        "SequenceNumber": "111",
        "SizeBytes": 26
      },
      "awsRegion": "us-west-2",
      "eventName": "INSERT",
      "eventSourceARN": eventsourcearn,
      "eventSource": "aws:dynamodb"
    },
    {
      "eventID": "2",
      "eventVersion": "1.0",
      "dynamodb": {
        "OldImage": {
          "Message": {
            "S": "New item!"
          },
          "Id": {
            "N": "101"
          }
        },
        "SequenceNumber": "222",
        "Keys": {
          "Id": {
            "N": "101"
          }
        },
        "SizeBytes": 59,
        "NewImage": {
          "Message": {
            "S": "This item has changed"
          },
          "Id": {
            "N": "101"
          }
        },
        "StreamViewType": "NEW_AND_OLD_IMAGES"
      },
      "awsRegion": "us-west-2",
      "eventName": "MODIFY",
      "eventSourceARN": sourcearn,
      "eventSource": "aws:dynamodb"
    }
```

Lambda polls shards in your DynamoDB stream for records at a base rate of 4 times per second. When records are available, Lambda invokes your function and waits for the result. If processing succeeds, Lambda resumes polling until it receives more records.

By default, Lambda invokes your function as soon as records are available in the stream. If the batch that Lambda reads from the stream only has one record in it, Lambda sends only one record to the function. To avoid invoking the function with a small number of records, you can tell the event source to buffer records for up to five minutes by configuring a *batch window*. Before invoking the function, Lambda continues to read records from the stream until it has gathered a full batch, or until the batch window expires.

If your function returns an error, Lambda retries the batch until processing succeeds or the data expires. To avoid stalled shards, you can configure the event source mapping to retry with a smaller batch size, limit the number of retries, or discard records that are too old. To retain discarded events, you can configure the event source mapping to send details about failed batches to an SQS queue or SNS topic.

You can also increase concurrency by processing multiple batches from each shard in parallel. Lambda can process up to 10 batches in each shard simultaneously. If you increase the number of concurrent batches per shard, Lambda still ensures in-order processing at the partition-key level.

Configure the `ParallelizationFactor` setting to process one shard of a Kinesis or DynamoDB data stream with more than one Lambda invocation simultaneously. You can specify the number of concurrent batches that Lambda polls from a shard via a parallelization factor from 1 (default) to 10. For example, when `ParallelizationFactor` is set to 2, you can have 200 concurrent Lambda invocations at maximum to process 100 Kinesis data shards. This helps scale up the processing throughput when the data volume is volatile and the `IteratorAge` is high.

Sections

Execution role permissions

Lambda needs the following permissions to manage resources related to your DynamoDB stream. Add them to your function's execution role.

- dynamodb:DescribeStream (https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_streams_DescribeStream.html)
- dynamodb:GetRecords (https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_streams_GetRecords.html)
- dynamodb:GetShardIterator (https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_streams_GetShardIterator.html)
- dynamodb:ListStreams (https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_streams_ListStreams.html)

The `AWSLambdaDynamoDBExecutionRole` managed policy includes these permissions. For more information, see AWS Lambda execution role (./lambda-intro-execution-role.html) .

To send records of failed batches to a queue or topic, your function needs additional permissions. Each destination service requires a different permission, as follows:

- **Amazon SQS** – sqs:SendMessage (https://docs.aws.amazon.com/AWSSimpleQueueService/latest/APIReference/API_SendMessage.html)
- **Amazon SNS** – sns:Publish (https://docs.aws.amazon.com/sns/latest/api/API_Publish.html)

Configuring a stream as an event source

Create an event source mapping to tell Lambda to send records from your stream to a Lambda function. You can create multiple event source mappings to process the same data with multiple Lambda functions, or to process items from multiple streams with a single function.

To configure your function to read from DynamoDB Streams in the Lambda console, create a **DynamoDB** trigger.

To create a trigger

1. Open the Functions page (https://console.aws.amazon.com/lambda/home#/functions) on the Lambda console.
2. Choose a function.
3. Under **Designer**, choose **Add trigger**.
4. Choose a trigger type.
5. Configure the required options and then choose **Add**.

Lambda supports the following options for DynamoDB event sources.

Event source options

- **DynamoDB table** – The DynamoDB table to read records from.
- **Batch size** – The number of records to send to the function in each batch, up to 10,000. Lambda passes all of the records in the batch to the function in a single call, as long as the total size of the events doesn't exceed the payload limit (./gettingstarted-limits.html) for synchronous invocation (6 MB).
- **Batch window** – Specify the maximum amount of time to gather records before invoking the function, in seconds.
- **Starting position** – Process only new records, or all existing records.
  - **Latest** – Process new records that are added to the stream.
  - **Trim horizon** – Process all records in the stream.

  After processing any existing records, the function is caught up and continues to process new records.
- **On-failure destination** – An SQS queue or SNS topic for records that can't be processed. When Lambda discards a batch of records because it's too old or has exhausted all retries, it sends details about the batch to the queue or topic.
- **Retry attempts** – The maximum number of times that Lambda retries when the function returns an error. This doesn't apply to service errors or throttles where the batch didn't reach the function.
- **Maximum age of record** – The maximum age of a record that Lambda sends to your function.
- **Split batch on error** – When the function returns an error, split the batch into two before retrying.
- **Concurrent batches per shard** – Process multiple batches from the same shard concurrently.
- **Enabled** – Set to true to enable the event source mapping. Set to false to stop processing records. Lambda keeps track of the last record processed and resumes processing from that point when the mapping is reenabled.

Note

You are not charged for GetRecords API calls invoked by Lambda as part of DynamoDB triggers.

To manage the event source configuration later, choose the trigger in the designer.

Event source mapping APIs

To manage an event source with the AWS CLI (https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-install.html) or AWS SDK (http://aws.amazon.com/getting-started/tools-sdks/) , you can use the following API operations:

- CreateEventSourceMapping (./API_CreateEventSourceMapping.html)
- ListEventSourceMappings (./API_ListEventSourceMappings.html)
- GetEventSourceMapping (./API_GetEventSourceMapping.html)
- UpdateEventSourceMapping (./API_UpdateEventSourceMapping.html)
- DeleteEventSourceMapping (./API_DeleteEventSourceMapping.html)

The following example uses the AWS CLI to map a function named `my-function` to a DynamoDB stream that is specified by its Amazon Resource Name (ARN), with a batch size of 500.

```
$ aws lambda create-event-source-mapping --function-name my-function --batch-size 500 --
starting-position LATEST \
--event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table/stream/2019-
06-10T19:26:16.525
{
    "UUID": "14e0db71-5d35-4eb5-b481-8945cf9d10c2",
    "BatchSize": 500,
    "MaximumBatchingWindowInSeconds": 0,
    "ParallelizationFactor": 1,
    "EventSourceArn": "arn:aws:dynamodb:us-east-2:123456789012:table/my-
table/stream/2019-06-10T19:26:16.525",
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
    "LastModified": 1560209851.963,
    "LastProcessingResult": "No records processed",
    "State": "Creating",
    "StateTransitionReason": "User action",
    "DestinationConfig": {},
    "MaximumRecordAgeInSeconds": 604800,
    "BisectBatchOnFunctionError": false,
    "MaximumRetryAttempts": 10000
}
```

Configure additional options to customize how batches are processed and to specify when to discard records that can't be processed. The following example updates an event source mapping to send a failure record to an SQS queue after two retry attempts, or if the records are more than an hour old.

```
$ aws lambda update-event-source-mapping --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--maximum-retry-attempts 2  --maximum-record-age-in-seconds 3600
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-east-
2:123456789012:dlq"}}'
{
    "UUID": "f89f8514-cdd9-4602-9e1f-01a5b77d449b",
    "BatchSize": 100,
    "MaximumBatchingWindowInSeconds": 0,
    "ParallelizationFactor": 1,
    "EventSourceArn": "arn:aws:dynamodb:us-east-2:123456789012:table/my-
table/stream/2019-06-10T19:26:16.525",
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
    "LastModified": 1573243620.0,
    "LastProcessingResult": "PROBLEM: Function call failed",
    "State": "Updating",
    "StateTransitionReason": "User action",
    "DestinationConfig": {},
    "MaximumRecordAgeInSeconds": 604800,
    "BisectBatchOnFunctionError": false,
    "MaximumRetryAttempts": 10000
}
```

Updated settings are applied asynchronously and aren't reflected in the output until the process completes. Use the `get-event-source-mapping` command to view the current status.

```
$ aws lambda get-event-source-mapping --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b
{
    "UUID": "f89f8514-cdd9-4602-9e1f-01a5b77d449b",
    "BatchSize": 100,
    "MaximumBatchingWindowInSeconds": 0,
    "ParallelizationFactor": 1,
    "EventSourceArn": "arn:aws:dynamodb:us-east-2:123456789012:table/my-
table/stream/2019-06-10T19:26:16.525",
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
    "LastModified": 1573244760.0,
    "LastProcessingResult": "PROBLEM: Function call failed",
    "State": "Enabled",
    "StateTransitionReason": "User action",
    "DestinationConfig": {
        "OnFailure": {
            "Destination": "arn:aws:sqs:us-east-2:123456789012:dlq"
        }
    },
    "MaximumRecordAgeInSeconds": 3600,
    "BisectBatchOnFunctionError": false,
    "MaximumRetryAttempts": 2
}
```

To process multiple batches concurrently, use the `--parallelization-factor` option.

```
$ aws lambda update-event-source-mapping --uuid 2b733gdc-8ac3-cdf5-af3a-1827b3b11284 \
--parallelization-factor 5
```

Error handling

The event source mapping that reads records from your DynamoDB stream invokes your function synchronously and retries on errors. If the function is throttled or the Lambda service returns an error without invoking the function, Lambda retries until the records expire or exceed the maximum age that you configure on the event source mapping.

If the function receives the records but returns an error, Lambda retries until the records in the batch expire, exceed the maximum age, or reach the configured retry quota. For function errors, you can also configure the event source mapping to split a failed batch into two batches. Retrying with smaller batches isolates bad records and works around timeout issues. Splitting a batch does not count towards the retry quota.

If the error handling measures fail, Lambda discards the records and continues processing batches from the stream. With the default settings, this means that a bad record can block processing on the affected shard for up to one day. To avoid this, configure your function's event source mapping with a reasonable number of retries and a maximum record age that fits your use case.

To retain a record of discarded batches, configure a failed-event destination. Lambda sends a document to the destination queue or topic with details about the batch.

To configure a destination for failed-event records

1. Open the Functions page (https://console.aws.amazon.com/lambda/home#/functions) on the Lambda console.
2. Choose a function.
3. Under **Designer**, choose **Add destination**.
4. For **Source**, choose **Stream invocation**.
5. For **Stream**, choose a stream that is mapped to the function.
6. For **Destination type**, choose the type of resource that receives the invocation record.
7. For **Destination**, choose a resource.
8. Choose **Save**.

The following example shows an invocation record for a DynamoDB stream.

Example Invocation Record

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:13:49.717Z",
    "DDBStreamBatchInfo": {
        "shardId": "shardId-00000001573689847184-864758bb",
        "startSequenceNumber": "800000000003126276362",
        "endSequenceNumber": "800000000003126276362",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:13:19Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:13:19Z",
        "batchSize": 1,
        "streamArn": "arn:aws:dynamodb:us-east-2:123456789012:table/mytable/stream/2019-
11-14T00:04:06.388"
    }
}
```

You can use this information to retrieve the affected records from the stream for troubleshooting. The actual records aren't included, so you must process this record and retrieve them from the stream before they expire and are lost.

Amazon CloudWatch metrics

Lambda emits the `IteratorAge` metric when your function finishes processing a batch of records. The metric indicates how old the last record in the batch was when processing finished. If your function is processing new events, you can use the iterator age to estimate the latency between when a record is added and when the function processes it.

An increasing trend in iterator age can indicate issues with your function. For more information, see Working with AWS Lambda function metrics (./monitoring-metrics.html) .

Time windows

Lambda functions can run continuous stream processing applications. A stream represents unbounded data that flows continuously through your application. To analyze information from this continuously updating input, you can bound the included records using a window defined in terms of time.

Lambda invocations are stateless—you cannot use them for processing data across multiple continuous invocations without an external database. However, with windowing enabled, you can maintain your state across invocations. This state contains the aggregate result of the messages previously processed for the current window. Your state can be a maximum of 1 MB per shard. If it exceeds that size, Lambda terminates the window early.

Tumbling windows

Lambda functions can aggregate data using tumbling windows: distinct time windows that open and close at regular intervals. Tumbling windows enable you to process streaming data sources through contiguous, non-overlapping time windows.

Each record of a stream belongs to a specific window. A record is processed only once, when Lambda processes the window that the record belongs to. In each window, you can perform calculations, such as a sum or average, at the partition key (https://docs.aws.amazon.com/streams/latest/dev/key-concepts.html#partition-key) level within a shard.

Aggregation and processing

Your user managed function is invoked both for aggregation and for processing the final results of that aggregation. Lambda aggregates all records received in the window. You can receive these records in multiple batches, each as a separate invocation. Each invocation receives a state. You can also process records and return a new state, which is passed in the next invocation. Lambda returns a `TimeWindowEventResponse` in JSON in the following format:

Example `TimeWindowEventReponse` values

```
{
    "state": {
        "1": 282,
        "2": 715
    },
    "batchItemFailures": []
}
```

Note

For Java functions, we recommend using a Map<String, String> to represent the state.

At the end of the window, the flag `isFinalInvokeForWindow` is set to `true` to indicate that this is the final state and that it's ready for processing. After processing, the window completes and your final invocation completes, and then the state is dropped.

At the end of your window, Lambda uses final processing for actions on the aggregation results. Your final processing is synchronously invoked. After successful invocation, your function checkpoints the sequence number and stream processing continues. If invocation is unsuccessful, your Lambda function suspends further processing until a successful invocation.

Example DynamodbTimeWindowEvent

```
{
   "Records":[
      {
         "eventID":"1",
         "eventName":"INSERT",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "NewImage":{
               "Message":{
                  "S":"New item!"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"111",
            "SizeBytes":26,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      },
      {
         "eventID":"2",
         "eventName":"MODIFY",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "NewImage":{
               "Message":{
                  "S":"This item has changed"
               },
               "Id":{
                  "N":"101"
               }
            },
            "OldImage":{
               "Message":{
                  "S":"New item!"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"222",
            "SizeBytes":59,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      },
      {
         "eventID":"3",
         "eventName":"REMOVE",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "OldImage":{
               "Message":{
                  "S":"This item has changed"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"333",
            "SizeBytes":38,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      }
   ],
    "window": {
        "start": "2020-07-30T17:00:00Z",
        "end": "2020-07-30T17:05:00Z"
    },
    "state": {
        "1": "state1"
    },
    "shardId": "shard123456789",
    "eventSourceARN": "stream-ARN",
    "isFinalInvokeForWindow": false,
    "isWindowTerminatedEarly": false
}
```

Configuration

You can configure tumbling windows when you create or update an event source mapping (./invocation-eventsourcemapping.html) . To configure a tumbling window, specify the window in seconds. The following example AWS Command Line Interface (AWS CLI) command creates a streaming event source mapping that has a tumbling window of 120 seconds. The Lambda function defined for aggregation and processing is named `tumbling-window-example-function`.

```
$ aws lambda create-event-source-mapping --event-source-arn arn:aws:dynamodb:us-east-
1:123456789012:stream/lambda-stream --function-name "arn:aws:lambda:us-east-
1:123456789018:function:tumbling-window-example-function" --region us-east-1 --starting-
position TRIM_HORIZON --tumbling-window-in-seconds 120
```

Lambda determines tumbling window boundaries based on the time when records were inserted into the stream. All records have an approximate timestamp available that Lambda uses in boundary determinations.

Tumbling window aggregations do not support resharding. When the shard ends, Lambda considers the window closed, and the child shards start their own window in a fresh state.

Tumbling windows fully support the existing retry policies `maxRetryAttempts` and `maxRecordAge`.

Example Handler.py – Aggregation and processing

The following Python function demonstrates how to aggregate and then process your final state:

```
def lambda_handler(event, context):
    print('Incoming event: ', event)
    print('Incoming state: ', event['state'])

#Check if this is the end of the window to either aggregate or process.
    if event['isFinalInvokeForWindow']:
        # logic to handle final state of the window
        print('Destination invoke')
    else:
        print('Aggregate invoke')

#Check for early terminations
    if event['isWindowTerminatedEarly']:
        print('Window terminated early')

    #Aggregation logic
    state = event['state']
    for record in event['Records']:
        state[record['dynamodb']['NewImage']['Id']] = state.get(record['dynamodb']
['NewImage']['Id'], 0) + 1

    print('Returning state: ', state)
    return {'state': state}
```

Reporting batch item failures

When consuming and processing streaming data from an event source, by default Lambda checkpoints to the highest sequence number of a batch only when the batch is a complete success. Lambda treats all other results as a complete failure and retries processing the batch up to the retry limit. To allow for partial successes while processing batches from a stream, turn on `ReportBatchItemFailures`. Allowing partial successes can help to reduce the number of retries on a record, though it doesn't entirely prevent the possibility of retries in a successful record.

To turn on `ReportBatchItemFailures`, include the enum value `ReportBatchItemFailures` in the `FunctionResponseTypes` list. This list indicates which response types are enabled for your function. You can configure this list when you create or update an event source mapping (./invocation-eventsourcemapping.html) .

Report syntax

When configuring reporting on batch item failures, the `StreamsEventResponse` class is returned with a list of batch item failures. You can use a `StreamsEventResponse` object to return the sequence number of the first failed record in the batch. You can also create your own custom class using the correct response syntax. The following JSON structure shows the required response syntax:

```
{
  "BatchItemFailures": [
        {
            "ItemIdentifier": "<id>"
        }
    ]
}
```

Success and failure conditions

Lambda treats a batch as a complete success if you return any of the following:

- An empty `batchItemFailure` list
- A null `batchItemFailure` list
- An empty `EventResponse`
- A null `EventResponse`

Lambda treats a batch as a complete failure if you return any of the following:

- An empty string `itemIdentifier`
- A null `itemIdentifier`
- An `itemIdentifier` with a bad key name

Lambda retries failures based on your retry strategy.

Bisecting a batch

If your invocation fails and `BisectBatchOnFunctionError` is turned on, the batch is bisected regardless of your `ReportBatchItemFailures` setting.

When a partial batch success response is received and both `BisectBatchOnFunctionError` and `ReportBatchItemFailures` are turned on, the batch is bisected at the returned sequence number and Lambda retries only the remaining records.

Java (#java) Python (#python)

Example Handler.java – return new StreamsEventResponse()

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessDynamodbRecords implements RequestHandler<DynamodbEvent,
Serializable> {

    @Override
    public Serializable handleRequest(DynamodbEvent input, Context context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
ArrayList<*gt;();
        String curRecordSequenceNumber = "";
        for (DynamodbEvent.DynamodbEventRecord dynamodbEventRecord :
input.getRecords()) {
            try {
                //Process your record
                DynamodbEvent.Record dynamodbRecord =
dynamodbEventRecord.getDynamodb();
                curRecordSequenceNumber = dynamodbRecord.getSequenceNumber();

            } catch (Exception e) {
                //Return failed record's sequence number
                batchItemFailures.add(new
StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                    return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse(batchItemFailures);
    }
}
```