



AWS Compute Blog

Using AWS Lambda for streaming analytics

by [James Beswick](#) | on 15 DEC 2020 | in [Amazon DynamoDB](#), [AWS Lambda](#), [Kinesis Data Streams](#), [Serverless](#) | [Permalink](#) | [Comments](#) | [Share](#)

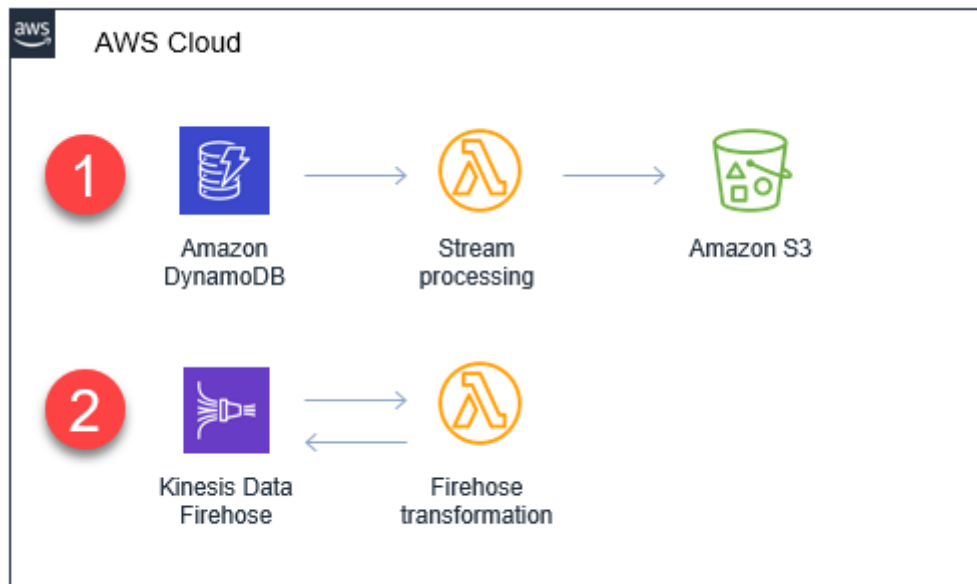
[AWS Lambda](#) now supports streaming analytics calculations for [Amazon Kinesis](#) and [Amazon DynamoDB](#). This allows developers to calculate aggregates in near-real time and pass state across multiple Lambda invocations. This feature provides an alternative way to build analytics in addition to services like [Amazon Kinesis Data Analytics](#).

In this blog post, I explain how this feature works with Kinesis Data Streams and DynamoDB Streams, together with example use-cases.

Overview

For workloads using [streaming data](#), data arrives continuously, often from different sources, and is processed incrementally. Discrete data processing tasks, such as operating on files, have a known beginning and end boundary for the data. For applications with streaming data, the processing function does not know when the data stream starts or ends. Consequently, this type of data is commonly processed in batches or windows.

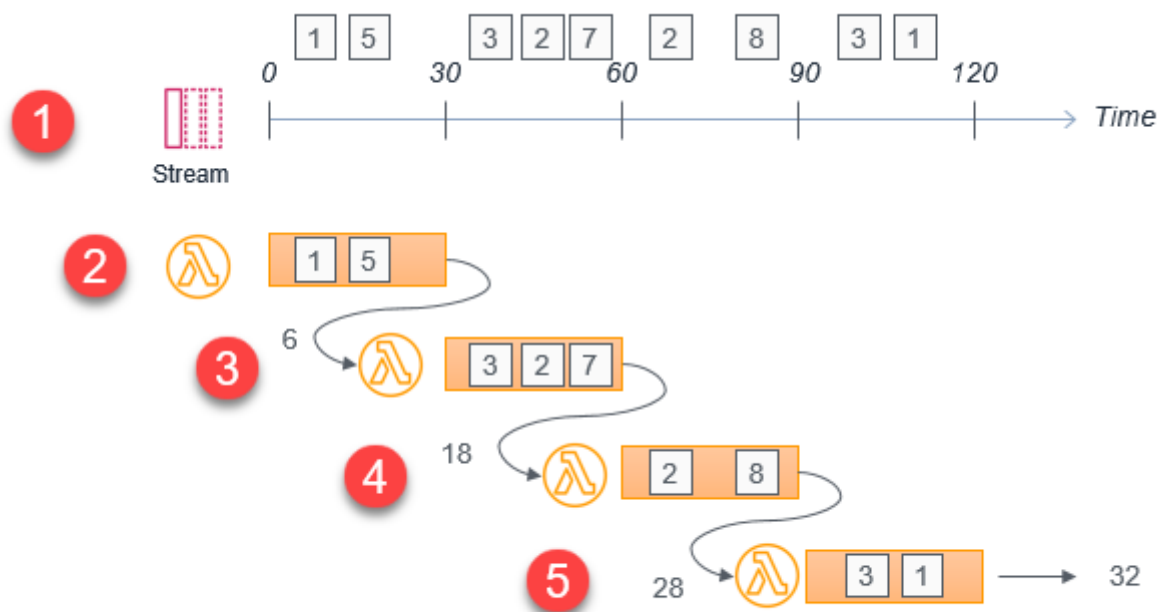
Before this feature, Lambda-based stream processing was limited to working on the incoming batch of data. For example, in [Amazon Kinesis Data Firehose](#), a Lambda function transforms the current batch of records with no information or state from previous batches. This is also the same for processing DynamoDB streams using Lambda functions. This existing approach works well for [MapReduce](#) or tasks focused exclusively on the data in the current batch.



1. DynamoDB streams invoke a processing Lambda function asynchronously. After processing, the function may then store the results in a downstream service, such as [Amazon S3](#).
2. Kinesis Data Firehose invokes a transformation Lambda function synchronously, which returns the transformed data back to the service.

This new feature introduces the concept of a *tumbling window*, which is a fixed-size, non-overlapping time interval of up to 15 minutes. To use this, you specify a tumbling window duration in the event-source mapping between the stream and the Lambda function. When you apply a tumbling window to a stream, items in the stream are grouped by window and sent to the processing Lambda function. The function returns a state value that is passed to the next tumbling window.

You can use this to calculate aggregates over multiple windows. For example, you can calculate the total value of a data item in a stream using 30-second tumbling windows:

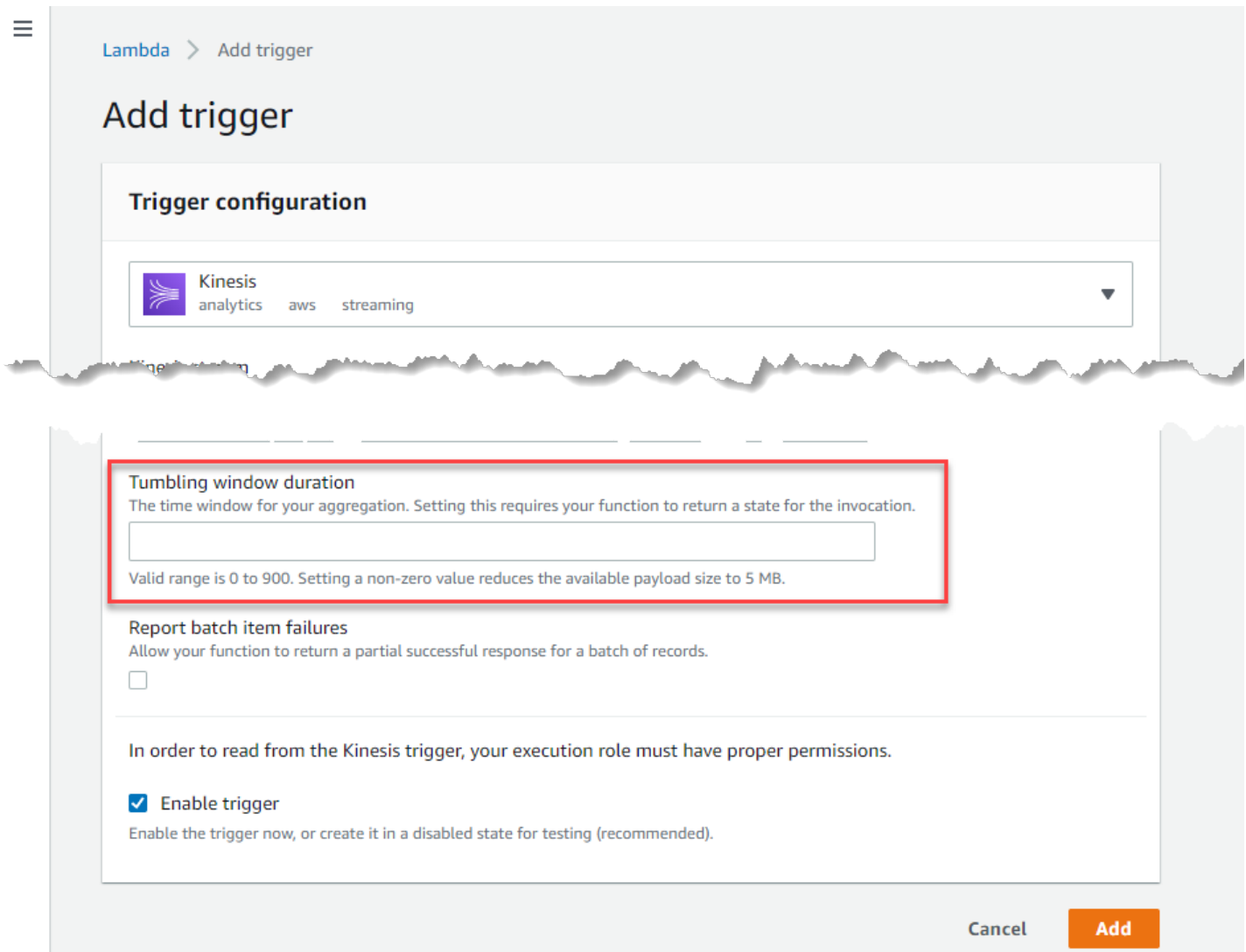


1. Integer data arrives in the stream at irregular time intervals.
2. The first tumbling window consists of data in the 0–30 second range, passed to the Lambda function. It adds the items and returns the total of 6 as a state value.
3. The second tumbling window invokes the Lambda function with the state value of 6 and the 30–60 second batch of stream data. This adds the items to the existing total, returning 18.
4. The third tumbling window invokes the Lambda function with a state value of 18 and the next window of values. The running total is now 28 and returned as the state value.
5. The fourth tumbling window invokes the Lambda function with a state value of 28 and the 90–120 second batch of data. The final total is 32.

This feature is useful in workloads where you need to calculate aggregates continuously. For example, for a retailer streaming order information from point-of-sale systems, it can generate near-live sales data for downstream reporting. Using Lambda to generate aggregates only requires minimal code, and the function can access other AWS services as needed.

Using tumbling windows with Lambda functions

When you configure an event source mapping between Kinesis or DynamoDB and a Lambda function, use the new setting, *Tumbling window duration*. This appears in the trigger configuration in the Lambda console:



Add trigger

Trigger configuration

Kinesis analytics aws streaming ▼

Tumbling window duration
The time window for your aggregation. Setting this requires your function to return a state for the invocation.

Valid range is 0 to 900. Setting a non-zero value reduces the available payload size to 5 MB.

Report batch item failures
Allow your function to return a partial successful response for a batch of records.

☐

In order to read from the Kinesis trigger, your execution role must have proper permissions.

☒ **Enable trigger**
Enable the trigger now, or create it in a disabled state for testing (recommended).

Cancel Add

You can also set this value in [AWS CloudFormation](#) and [AWS SAM](#) templates. After the event source mapping is created, events delivered to the Lambda function have several new attributes:

```

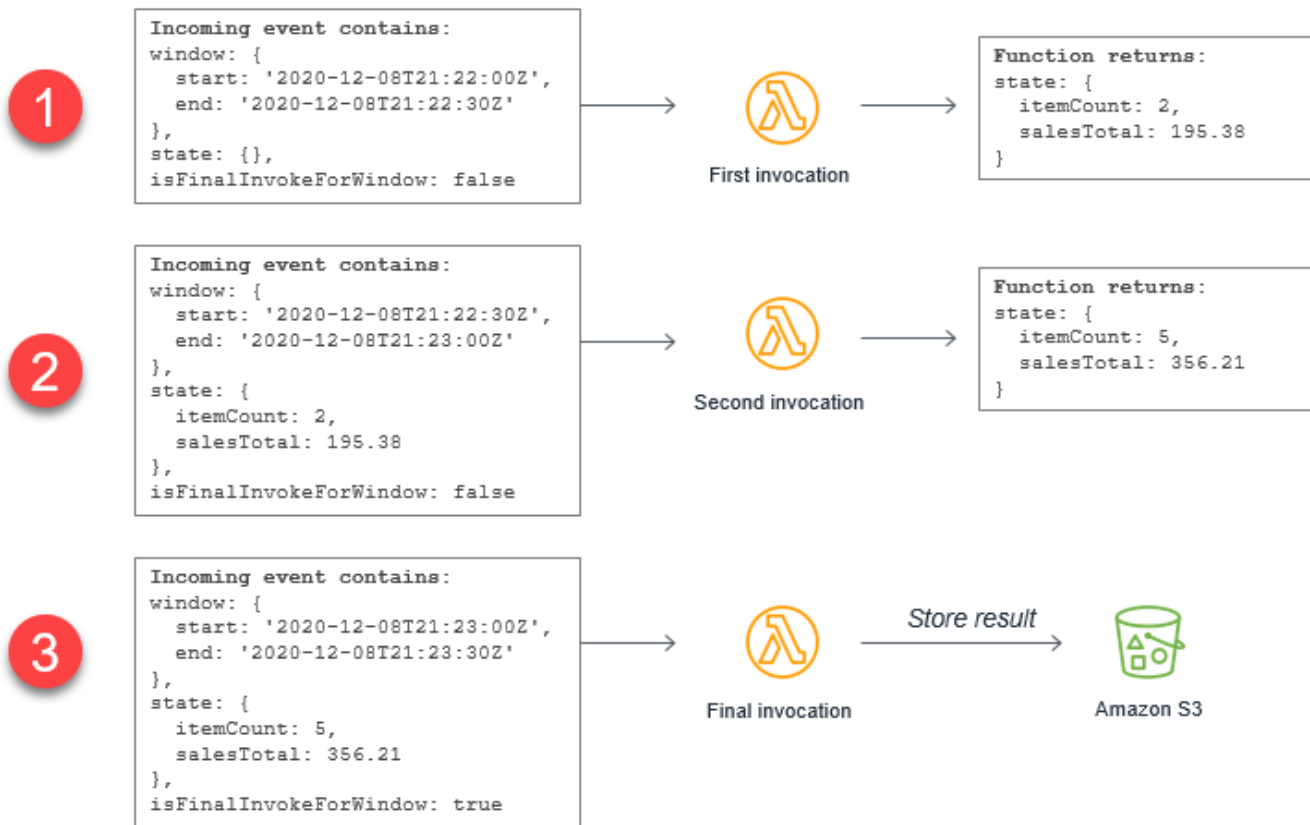
2020-12-08T21:22:06.802Z    1f21edc4-c2c6-480f-9bef-eedaa0611be0    INFO    {
  Records: [
    {
      eventID: '0b13737885b69c401f4b88dfc2174d17',
      eventName: 'INSERT',
      eventVersion: '1.1',
      eventSource: 'aws:dynamodb',
      awsRegion: 'us-east-1',
      dynamodb: [Object],
      eventSourceARN: 'arn:aws:dynamodb:us-east-1:██████████:table/tumblingWindows/stream/2020-12-07T21:57:44.846'
    },
    {
      eventID: 'ab11750975516705189d8ef1da6bef4b',
      eventName: 'INSERT',
      eventVersion: '1.1',
      eventSource: 'aws:dynamodb',
      awsRegion: 'us-east-1',
      dynamodb: [Object],
      eventSourceARN: 'arn:aws:dynamodb:us-east-1:██████████:table/tumblingWindows/stream/2020-12-07T21:57:44.846'
    }
  ],
  shardId: 'shardId-00000001607452260937-107fa67e',
  eventSourceARN: 'arn:aws:dynamodb:us-east-1:██████████:table/tumblingWindows/stream/2020-12-07T21:57:44.846',
  window: { start: '2020-12-08T21:22:00Z', end: '2020-12-08T21:22:30Z' },
  state: {},
  isFinalInvokeForWindow: false,
  isWindowTerminatedEarly: false
}

```

These include:

- **Window start and end:** the beginning and ending timestamps for the current tumbling window.
- **State:** an object containing the state returned from the previous window, which is initially empty. The state object can contain up to 1 MB of data.
- **isFinalInvokeForWindow:** indicates if this is the last invocation for the tumbling window. This only occurs once per window period.
- **isWindowTerminatedEarly:** a window ends early only if the state exceeds the maximum allowed size of 1 MB.

In any tumbling window, there is a series of Lambda invocations following this pattern:



1. The first invocation contains an empty state object in the event. The function returns a state object containing custom attributes that are specific to the custom logic in the aggregation.
2. The second invocation contains the state object provided by the first Lambda invocation. This function returns an updated state object with new aggregated values. Subsequent invocations follow this same sequence.
3. The final invocation in the tumbling window has the `isFinalInvokeForWindow` flag set to the true. This contains the state returned by the most recent Lambda invocation. This invocation is responsible for storing the result in S3 or in another data store, such as a DynamoDB table. There is no state returned in this final invocation.

Using tumbling windows with DynamoDB

DynamoDB streams can invoke Lambda function using tumbling windows, enabling you to generate aggregates per shard. In this example, an ecommerce workload saves orders in a DynamoDB table and uses a tumbling window to calculate the near-real time sales total.

First, I create a DynamoDB table to capture the order data and a second DynamoDB table to store the aggregate calculation. I create a Lambda function with a trigger from the first orders table. The event source mapping is created with a *Tumbling window duration* of 30 seconds:

DynamoDB (1)

Find triggers

☐ Trigger

DynamoDB: tumblingWindows (Enabled)

arn:aws:dynamodb:us-east-1:780018668030:table/tumblingWindows/stream/2020-12-07T21:57:44.846

▼ Details

Batch size: **10**
Batch window: **None**
Concurrent batches per shard: **1**
Last processing result: **OK**
Maximum age of record: **-1**
On-failure destination:

{
"onFailure": {}
}

☐

Retry attempts: **-1**
Split batch on error: **No**
Starting position: **LATEST**
Tumbling window duration: **30**

I use the following code in the Lambda function:

JavaScript

```
const AWS = require('aws-sdk')
AWS.config.update({ region: process.env.AWS_REGION })
const docClient = new AWS.DynamoDB.DocumentClient()
const TableName = 'tumblingWindowsAggregation'

function isEmpty(obj) { return Object.keys(obj).length === 0 }

exports.handler = async (event) => {
  // Save aggregation result in the final invocation
  if (event.isFinalInvokeForWindow) {
    console.log('Final: ', event)

    const params = {
      TableName,
      Item: {
        windowEnd: event.window.end,
        windowStart: event.window.start,
        sales: event.state.sales,
        shardId: event.shardId
      }
    }
  }
}
```

This function code processes the incoming event to aggregate a sales attribute, and return this aggregated result in a state object. In the final invocation, it stores the aggregated value in another DynamoDB table.

I then use this Node.js script to generate random sample order data:

JavaScript

```
const AWS = require('aws-sdk')
AWS.config.update({ region: 'us-east-1' })
const docClient = new AWS.DynamoDB.DocumentClient()

const TableName = 'tumblingWindows'
const ITERATIONS = 100
const SLEEP_MS = 100

let totalSales = 0

function sleep(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

const createSales = async () => {
  for (let i = 0; i < ITERATIONS; i++) {

    let sales = Math.round (parseFloat(100 * Math.random()))
    totalSales += sales
  }
}
```

Once the script is complete, the console shows the individual order transactions and the total sales:

```
{ i: 92, sales: 5, totalSales: 4279 }
{ i: 93, sales: 94, totalSales: 4373 }
{ i: 94, sales: 99, totalSales: 4472 }
{ i: 95, sales: 50, totalSales: 4522 }
{ i: 96, sales: 65, totalSales: 4587 }
{ i: 97, sales: 16, totalSales: 4603 }
{ i: 98, sales: 36, totalSales: 4639 }
{ i: 99, sales: 10, totalSales: 4649 }
Total Sales: 4649
```

After the tumbling window duration is finished, the second DynamoDB table shows the aggregate values calculated and stored by the Lambda function:

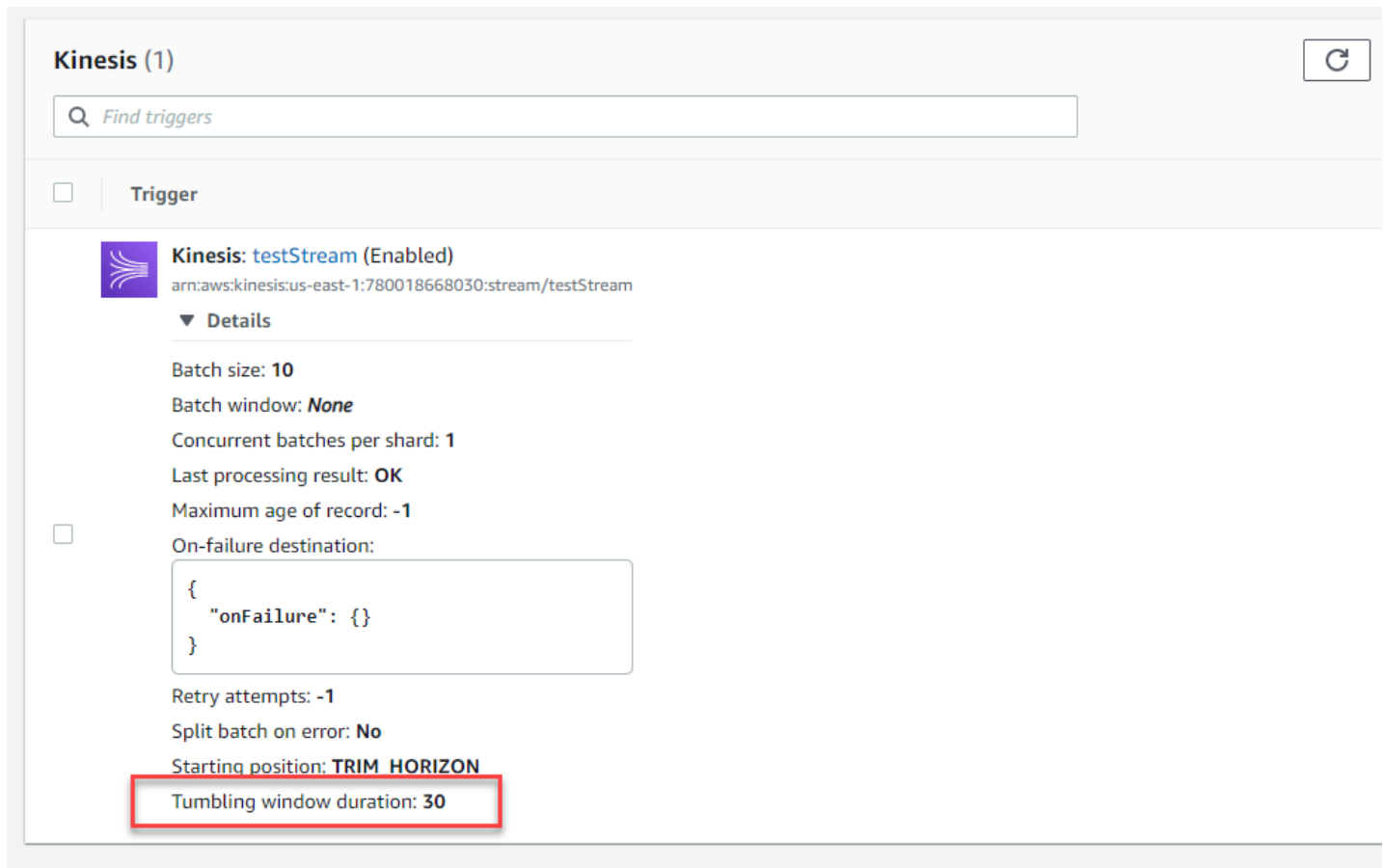
<input type="checkbox"/>	windowEnd ⓘ	shardId	sales	windowStart
<input type="checkbox"/>	2020-12-09T14:26:00Z	shardId-00000001607510792308-dfcadc33	1268	2020-12-09T14:25:30Z
<input type="checkbox"/>	2020-12-09T14:26:00Z	shardId-00000001607510816856-e5987fbe	1300	2020-12-09T14:25:30Z
<input type="checkbox"/>	2020-12-09T14:26:00Z	shardId-00000001607511368862-284c6792	1080	2020-12-09T14:25:30Z
<input type="checkbox"/>	2020-12-09T14:26:00Z	shardId-00000001607522243755-4bdc3eb3	1001	2020-12-09T14:25:30Z

Since aggregation for each shard is independent, the totals are stored by *shardId*. If I continue to run the test data script, the aggregation function continues to calculate and store more totals per tumbling window period.

Using tumbling windows with Kinesis

Kinesis data streams can also invoke a Lambda function using a tumbling window in a similar way. The biggest difference is that you control how many shards are used in the data stream. Since aggregation occurs per shard, this controls the total number aggregate results per tumbling window.

Using the same sales example, first I [create a Kinesis data stream](#) with one shard. I use the same DynamoDB tables from the previous example, then create a Lambda function with a trigger from the first orders table. The event source mapping is created with a *Tumbling window duration* of 30 seconds:



I use the following code in the Lambda function, modified to process the incoming Kinesis data event:

JavaScript

```
const AWS = require('aws-sdk')
AWS.config.update({ region: process.env.AWS_REGION })
const docClient = new AWS.DynamoDB.DocumentClient()
const TableName = 'tumblingWindowsAggregation'

function isEmpty(obj) {
  return Object.keys(obj).length === 0
}
```

```
exports.handler = async (event) => {

    // Save aggregation result in the final invocation
    if (event.isFinalInvokeForWindow) {
        console.log('Final: ', event)

        const params = {
            TableName,
            ...
        }
    }
}
```

This function code processes the incoming event in the same way as the previous example. I then use this Node.js script to generate random sample order data, modified to put the data on the Kinesis stream:

JavaScript

```
const AWS = require('aws-sdk')
AWS.config.update({ region: 'us-east-1' })
const kinesis = new AWS.Kinesis()

const StreamName = 'testStream'
const ITERATIONS = 100
const SLEEP_MS = 10

let totalSales = 0

function sleep(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}

const createSales = async () => {

    for (let i = 0; i < ITERATIONS; i++) {

        let sales = Math.round (parseFloat(100 * Math.random()))
    }
}
```

Once the script is complete, the console shows the individual order transactions and the total sales:

```
{ i: 94, sales: 49, totalSales: 5198 }
{ i: 95, sales: 19, totalSales: 5217 }
{ i: 96, sales: 5, totalSales: 5222 }
{ i: 97, sales: 92, totalSales: 5314 }
{ i: 98, sales: 96, totalSales: 5410 }
{ i: 99, sales: 64, totalSales: 5474 }
```

After the tumbling window duration is finished, the second DynamoDB table shows the aggregate values calculated and stored by the Lambda function:

<input type="checkbox"/>	windowEnd ⓘ	shardId	sales	windowStart
<input type="checkbox"/>	2020-12-09T15:14:30Z	shardId-000000000000	5474	2020-12-09T15:14:00Z

As there is only one shard in this Kinesis stream, there is only one aggregation value for all the data items in the test.

Conclusion

With tumbling windows, you can calculate aggregate values in near-real time for Kinesis data streams and DynamoDB streams. Unlike existing stream-based invocations, state can be passed forward by Lambda invocations. This makes it easier to calculate sums, averages, and counts on values across multiple batches of data.

In this post, I walk through an example that aggregates sales data stored in Kinesis and DynamoDB. In each case, I create an aggregation function with an event source mapping that uses the new tumbling window duration attribute. I show how state is passed between invocations and how to persist the aggregated value at the end of the tumbling window.

To learn more about how to use this feature, read the developer documentation for [DynamoDB](#) and [Kinesis Streams](#). To learn more about building with serverless technology, visit [Serverless Land](#).

TAGS: [serverless](#)



AWS Podcast

Subscribe for weekly AWS news and interviews

[Learn more »](#)



AWS Partner Network

Find an APN member to support your cloud business needs

[Learn more »](#)



AWS Training & Certifications

Free digital courses to help you develop your skills

[Learn more »](#)

Resources

[Serverless Computing and Applications](#)

[Amazon Container Services](#)

[AWS Messaging](#)

[Cloud Compute with AWS](#)

[Desktop and Application Streaming](#)

Follow

 [Twitter](#)

 [Facebook](#)

-  [LinkedIn](#)
-  [Twitch](#)
-  [Email Updates](#)



New Launches From re:Invent

Discover the latest services and features from AWS

[Visit the News Blog »](#)

Related Posts

[Detecting sensitive data in DynamoDB with Macie](#)

[BandLab welcomes users by the millions with AWS](#)

[ICYMI: Serverless pre:Invent 2020](#)

[Detect change points in your event data stream using Amazon Kinesis Data Streams, Amazon DynamoDB and AWS Lambda](#)

[Integrating Amazon ElastiCache with other AWS services: The serverless way](#)

[New – Export Amazon DynamoDB Table Data to Your Data Lake in Amazon S3, No Code Writing Required](#)

[How to deliver headless commerce in retail](#)

[Optimizing batch processing with custom checkpoints in AWS Lambda](#)