Amazon Kinesis Data Streams

Developer Guide

Amazon Kinesis Data Streams: Developer Guide

Table of Contents

What Is Amazon Kinesis Data Streams?

You can use Amazon Kinesis Data Streams to collect and process large streams of data records in real time. You can create data-processing applications, known as *Kinesis Data Streams applications*. A typical Kinesis Data Streams application reads data from a *data stream* as data records. These applications can use the Kinesis Client Library, and they can run on Amazon EC2 instances. You can send the processed records to dashboards, use them to generate alerts, dynamically change pricing and advertising strategies, or send data to a variety of other AWS services. For information about Kinesis Data Streams features and pricing, see Amazon Kinesis Data Streams.

Kinesis Data Streams is part of the Kinesis streaming data platform, along with Kinesis Data Firehose, Kinesis Video Streams, and Kinesis Data Analytics.

For more information about AWS big data solutions, see Big Data on AWS. For more information about AWS streaming data solutions, see What is Streaming Data?.

Topics

What Can I Do with Kinesis Data Streams?

You can use Kinesis Data Streams for rapid and continuous data intake and aggregation. The type of data used can include IT infrastructure log data, application logs, social media, market data feeds, and web clickstream data. Because the response time for the data intake and processing is in real time, the processing is typically lightweight.

The following are typical scenarios for using Kinesis Data Streams:

Accelerated log and data feed intake and processing

> You can have producers push data directly into a stream. For example, push system and application logs and they are available for processing in seconds. This prevents the log data from being lost if the front end or application server fails. Kinesis Data Streams provides accelerated data feed intake because you don't batch the data on the servers before you submit it for intake.

Real-time metrics and reporting

> You can use data collected into Kinesis Data Streams for simple data analysis and reporting in real time. For example, your data-processing application can work on metrics and reporting for system and application logs as the data is streaming in, rather than wait to receive batches of data.

Real-time data analytics

> This combines the power of parallel processing with the value of real-time data. For example, process website clickstreams in real time, and then analyze site usability engagement using multiple different Kinesis Data Streams applications running in parallel.

Complex stream processing

You can create Directed Acyclic Graphs (DAGs) of Kinesis Data Streams applications and data streams. This typically involves putting data from multiple Kinesis Data Streams applications into another stream for downstream processing by a different Kinesis Data Streams application.

Benefits of Using Kinesis Data Streams

Although you can use Kinesis Data Streams to solve a variety of streaming data problems, a common use is the real-time aggregation of data followed by loading the aggregate data into a data warehouse or map-reduce cluster.

Data is put into Kinesis data streams, which ensures durability and elasticity. The delay between the time a record is put into the stream and the time it can be retrieved (put-to-get delay) is typically less than 1 second. In other words, a Kinesis Data Streams application can start consuming the data from the stream almost immediately after the data is added. The managed service aspect of Kinesis Data Streams relieves you of the operational burden of creating and running a data intake pipeline. You can create streaming map-reduce–type applications. The elasticity of Kinesis Data Streams enables you to scale the stream up or down, so that you never lose data records before they expire.

Multiple Kinesis Data Streams applications can consume data from a stream, so that multiple actions, like archiving and processing, can take place concurrently and independently. For example, two applications can read data from the same stream. The first application calculates running aggregates and updates an Amazon DynamoDB table, and the second application compresses and archives data to a data store like Amazon Simple Storage Service (Amazon S3). The DynamoDB table with running aggregates is then read by a dashboard for up-to-the-minute reports.

The Kinesis Client Library enables fault-tolerant consumption of data from streams and provides scaling support for Kinesis Data Streams applications.

Related Services

For information about using Amazon EMR clusters to read and process Kinesis data streams directly, see Kinesis Connector.

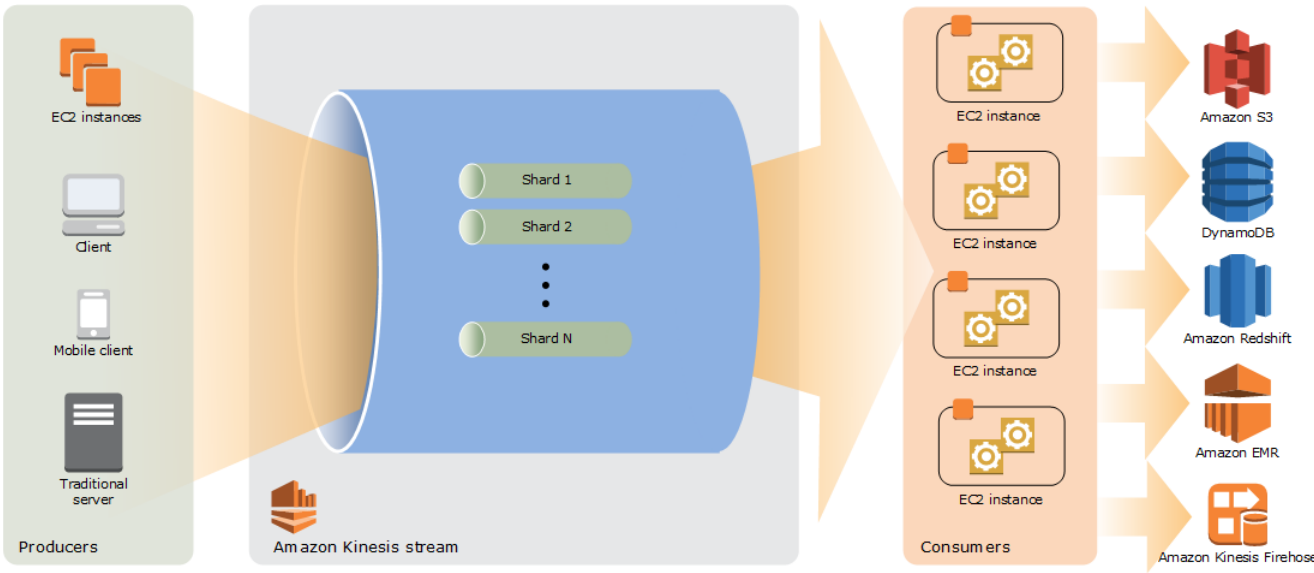
Amazon Kinesis Data Streams Terminology and Concepts

As you get started with Amazon Kinesis Data Streams, you can benefit from understanding its architecture and terminology.

Topics

Kinesis Data Streams High-Level Architecture

The following diagram illustrates the high-level architecture of Kinesis Data Streams. The *producers* continually push data to Kinesis Data Streams, and the *consumers* process the data in real time. Consumers (such as a custom application running on Amazon EC2 or an Amazon Kinesis Data Firehose delivery stream) can store their results using an AWS service such as Amazon DynamoDB, Amazon Redshift, or Amazon S3.



Kinesis Data Streams Terminology

Kinesis Data Stream

A *Kinesis data stream* is a set of shards (p. 5). Each shard has a sequence of data records. Each data record has a sequence number (p. 5) that is assigned by Kinesis Data Streams.

Data Record

A *data record* is the unit of data stored in a Kinesis data stream (p. 3). Data records are composed of a sequence number (p. 5), a partition key (p. 5), and a data blob, which is an immutable sequence of bytes. Kinesis Data Streams does not inspect, interpret, or change the data in the blob in any way. A data blob can be up to 1 MB.

Capacity Mode

A data stream *capacity mode* determines how capacity is managed and how you are charged for the usage of your data stream. Currenly, in Kinesis Data Streams, you can choose between an **on-demand** mode and a **provisioned** mode for your data streams. For more information, see Choosing the Data Stream Capacity Mode (p. 65).

With the **on-demand** mode, Kinesis Data Streams automatically manages the shards in order to provide the necessary throughput. You are charged only for the actual throughput that you use and Kinesis Data Streams automatically accommodates your workloads' throughput needs as they ramp up or down. For more information, see On-demand Mode (p. 66).

With the **provisioned** mode, you must specify the number of shards for the data stream. The total capacity of a data stream is the sum of the capacities of its shards. You can increase or decrease the number of shards in a data stream as needed and you are charged for the number of shards at an hourly rate. For more information, see Provisioned Mode (p. 67).

Retention Period

The *retention period* is the length of time that data records are accessible after they are added to the stream. A stream's retention period is set to a default of 24 hours after creation. You can increase the retention period up to 8760 hours (365 days) using the IncreaseStreamRetentionPeriod operation, and decrease the retention period down to a minimum of 24 hours using the DecreaseStreamRetentionPeriod operation. Additional charges apply for streams with a retention period set to more than 24 hours. For more information, see Amazon Kinesis Data Streams Pricing.

Producer

Producers put records into Amazon Kinesis Data Streams. For example, a web server sending log data to a stream is a producer.

Consumer

Consumers get records from Amazon Kinesis Data Streams and process them. These consumers are known as Amazon Kinesis Data Streams Application (p. 4).

Amazon Kinesis Data Streams Application

An *Amazon Kinesis Data Streams application* is a consumer of a stream that commonly runs on a fleet of EC2 instances.

There are two types of consumers that you can develop: shared fan-out consumers and enhanced fan-out consumers. To learn about the differences between them, and to see how you can create each type of consumer, see Reading Data from Amazon Kinesis Data Streams (p. 112).

The output of a Kinesis Data Streams application can be input for another stream, enabling you to create complex topologies that process data in real time. An application can also send data to a variety of other AWS services. There can be multiple applications for one stream, and each application can consume data from the stream independently and concurrently.

Shard

A *shard* is a uniquely identified sequence of data records in a stream. A stream is composed of one or more shards, each of which provides a fixed unit of capacity. Each shard can support up to 5 transactions per second for reads, up to a maximum total data read rate of 2 MB per second and up to 1,000 records per second for writes, up to a maximum total data write rate of 1 MB per second (including partition keys). The data capacity of your stream is a function of the number of shards that you specify for the stream. The total capacity of the stream is the sum of the capacities of its shards.

If your data rate increases, you can increase or decrease the number of shards allocated to your stream. For more information, see Resharding a Stream (p. 75).

Partition Key

A *partition key* is used to group data by shard within a stream. Kinesis Data Streams segregates the data records belonging to a stream into multiple shards. It uses the partition key that is associated with each data record to determine which shard a given data record belongs to. Partition keys are Unicode strings, with a maximum length limit of 256 characters for each key. An MD5 hash function is used to map partition keys to 128-bit integer values and to map associated data records to shards using the hash key ranges of the shards. When an application puts data into a stream, it must specify a partition key.

Sequence Number

Each data record has a *sequence number* that is unique per partition-key within its shard. Kinesis Data Streams assigns the sequence number after you write to the stream with `client.putRecords` or `client.putRecord`. Sequence numbers for the same partition key generally increase over time. The longer the time period between write requests, the larger the sequence numbers become.

> **Note**
> Sequence numbers cannot be used as indexes to sets of data within the same stream. To logically separate sets of data, use partition keys or create a separate stream for each dataset.

Kinesis Client Library

The Kinesis Client Library is compiled into your application to enable fault-tolerant consumption of data from the stream. The Kinesis Client Library ensures that for every shard there is a record processor running and processing that shard. The library also simplifies reading data from the stream. The Kinesis Client Library uses an Amazon DynamoDB table to store control data. It creates one table per application that is processing data.

There are two major versions of the Kinesis Client Library. Which one you use depends on the type of consumer you want to create. For more information, see Reading Data from Amazon Kinesis Data Streams (p. 112).

Application Name

The name of an Amazon Kinesis Data Streams application identifies the application. Each of your applications must have a unique name that is scoped to the AWS account and Region used by the application. This name is used as a name for the control table in Amazon DynamoDB and the namespace for Amazon CloudWatch metrics.

Server-Side Encryption

Amazon Kinesis Data Streams can automatically encrypt sensitive data as a producer enters it into a stream. Kinesis Data Streams uses AWS KMS master keys for encryption. For more information, see Data Protection in Amazon Kinesis Data Streams (p. 209).

Note
To read from or write to an encrypted stream, producer and consumer applications must have permission to access the master key. For information about granting permissions to producer and consumer applications, see the section called “Permissions to Use User-Generated KMS Master Keys” (p. 212).

Note
Using server-side encryption incurs AWS Key Management Service (AWS KMS) costs. For more information, see AWS Key Management Service Pricing.

Quotas and Limits

Amazon Kinesis Data Streams has the following stream and shard quotas and limits.

- There is no upper quota on the number of streams with the provisioned mode that you can have in an account.
- Within your AWS account, by default, you can create up to 50 data streams with the on-demand capacity mode. If you require an increase of this quota, contact AWS support.
- The default shard quota is 500 shards per AWS account for the following AWS regions: US East (N. Virginia), US West (Oregon), and Europe (Ireland). For all other regions, the default shard quota is 200 shards per AWS account. This limit is only applicable for data streams with the provisioned capacity mode.

  To request the shards per data stream quota increase, follow the procedure outlined in Requesting a Quota Increase.
- A single shard can ingest up to 1 MB of data per second (including partition keys) or 1,000 records per second for writes. Similarly, if you scale your stream to 5,000 shards, the stream can ingest up to 5 GB per second or 5 million records per second. If you need more ingest capacity, you can easily scale up the number of shards in the stream using the AWS Management Console or the UpdateShardCount API.
- The maximum size of the data payload of a record before base64-encoding is up to 1 MB.
- GetRecords can retrieve up to 10 MB of data per call from a single shard, and up to 10,000 records per call. Each call to `GetRecords` is counted as one read transaction.
- Each shard can support up to five read transactions per second. Each read transaction can provide up to 10,000 records with an upper quota of 10 MB per transaction.
- Each shard can support up to a maximum total data read rate of 2 MB per second via GetRecords. If a call to `GetRecords` returns 10 MB, subsequent calls made within the next 5 seconds throw an exception.
- By default, new data streams created with the on-demand capacity mode have 4 MB/s of 'write' and 8 MB/s of 'read' throughput. As the traffic increases, data streams with the on-demand capacity mode scale up to 200 MB/s of 'write' and 400 MB/s 'read' throughput. If you require additional throughput, contact AWS support.
- Within your AWS account, by default, you can create up to 50 data streams with the on-demand capacity mode. If you require an increase of this quota, contact AWS support.
- You can create 20 registered consumers (Enhanced Fan-out Limit) for each data stream.
- For each data stream in your AWS account, you can switch between the on-demand and provisioned capacity modes twice within 24 hours.

API Limits

Like most AWS APIs, Kinesis Data Streams API operations are rate-limited. The following limits apply per AWS account per region. For more information on Kinesis Data Streams APIs, see the Amazon Kinesis API Reference.

KDS Control Plane API Limits

The following section describes limits for the KDS control plane APIs. KDS control plane APIs enable you to create and manage your data streams. These limits apply per AWS account per region.

Control Plane API Limits

API	API call limit	Stream-level limit	Additional details
AddTagsToStream	5 transactions per second (TPS)	50 tags per data stream per account per region	
CreateStream	5 TPS	There is no upper quota on the number of streams you can have in an account.	You receive a LimitExceededException when making a CreateStream request when you try to do one of the following: • Have more than five streams in the `CREATING` state at any point in time. • Create more shards than are authorized for your account.
DecreaseStreamRetentionPeriod	5 TPS	The minimum value of a data stream's retention period is 24 hours.	
DeleteStream	5 TPS	N/A	
DeregisterStreamConsumer	5 TPS	N/A	
DescribeLimits	1 TPS		
DescribeStream	10 TPS	N/A	
DescribeStreamConsumer	20 TPS	N/A	
DescribeStreamSummary	20 TPS	N/A	
DisableEnhancedMonitoring	5 TPS	N/A	
EnableEnhancedMonitoring	5 TPS	N/A	
IncreaseStreamRetentionPeriod	5 TPS	The maximum value of a stream's retention period is 8760 hours (365 days).	
ListShards	100 TPS	N/A	
ListStreamConsumers	5 TPS	N/A	
ListStreams	5 TPS	N/A	
ListTagsForStream	5 TPS	N/A	
MergeShards	5 TPS	N/A	
RegisterStreamConsumer	5 TPS	You can register up to 20 consumers per data stream. A	

API	API call limit	Stream-level limit	Additional details
		given consumer can only be registered with one data stream at a time. Only 5 consumers can be created simultaneously. In other words, you cannot have more than 5 consumers in a CREATING status at the same time. Registering a 6th consumer while there are 5 in a CREATING status results in a LimitExceededException.	
RemoveTagsFromStream	5 TPS	N/A	
SplitShard	5 TPS	N/A	
StartStreamEncryption			You can successfully apply a new AWS KMS key for server-side encryption 25 times in a rolling 24-hour period.
StopStreamEncryption			You can successfully disable server-side encryption 25 times in a rolling 24-hour period.
UpdateShardCount		• Scale more than ten times per rolling 24-hour period per stream • Scale up to more than double your current shard count for a stream • Scale down below half your current shard count for a stream • Scale up to more than 10000 shards in a stream • Scale a stream with more than 10000 shards down unless the result is less than 10000 shards • Scale up to more than the shard limit for your account	

API	API call limit	Stream-level limit	Additional details
UpdateStreamMode		• For each data stream in your AWS account, you can switch between the on-demand and provisioned capacity modes twice within 24 hours.	

KDS Data Plane API Limits

The following section describes the limits for the KDS data plane APIs. KDS data plane APIs enable you to use your data streams for collecting and processing data records in real time. These limits apply per shard within your data streams.

Data Plane API limits

API	API call limit	Payload limit	Additional details
GetRecords	5 TPS	The maximum number of records that can be returned per call is 10,000. The maximum size of data that GetRecords can return is 10 MB.	If a call returns this amount of data, subsequent calls made within the next 5 seconds throw ProvisionedThroughputExceededExcep If there is insufficient provisioned throughput on the stream, subsequent calls made within the next 1 second throw ProvisionedThroughputExceededExcep
GetShardIterator	5 TPS		A shard iterator expires 5 minutes after it is returned to the requester. If a GetShardIterator request is made too often, you receive a ProvisionedThroughputExceededExcep
PutRecord	1000 TPS	Each shard can support writes up to 1,000 records per second, up to a maximum data write total of 1 MB per second.	
PutRecords		Each PutRecords request can support up to 500 records. Each record in the request can be as large as 1 MB,	

API	API call limit	Payload limit	Additional details
		up to a limit of 5 MB for the entire request, including partition keys. Each shard can support writes up to 1,000 records per second, up to a maximum data write total of 1 MB per second.	
SubscribeToShard	You can make one call to SubscribeToShard per second per registered consumer per shard.		If you call SubscribeToShard again with the same ConsumerARN and ShardId within 5 seconds of a successful call, you'll get a ResourceInUseException.

Increasing Quotas

You can use Service Quotas to request an increase for a quota, if the quota is adjustable. Some requests are automatically resolved, while others are submitted to AWS Support. You can track the status of a quota increase request that is submitted to AWS Support. Requests to increase service quotas do not receive priority support. If you have an urgent request, contact AWS Support. For more information, see What Is Service Quotas?

To request a service quota increase, follow the procedure outlined in Requesting a Quota Increase.

Setting Up for Amazon Kinesis Data Streams

Before you use Amazon Kinesis Data Streams for the first time, complete the following tasks.

Tasks

Sign Up for AWS

When you sign up for Amazon Web Services (AWS), your AWS account is automatically signed up for all services in AWS, including Kinesis Data Streams. You are charged only for the services that you use.

If you have an AWS account already, skip to the next task. If you don't have an AWS account, use the following procedure to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

Download Libraries and Tools

The following libraries and tools will help you work with Kinesis Data Streams:

- The Amazon Kinesis API Reference is the basic set of operations that Kinesis Data Streams supports. For more information about performing basic operations using Java code, see the following:
  - Developing Producers Using the Amazon Kinesis Data Streams API with the AWS SDK for Java (p. 95)
  - Developing Custom Consumers with Shared Throughput Using the AWS SDK for Java (p. 149)
  - Creating and Managing Streams (p. 65)
- The AWS SDKs for Go, Java, JavaScript, .NET, Node.js, PHP, Python, and Ruby include Kinesis Data Streams support and samples. If your version of the AWS SDK for Java does not include samples for Kinesis Data Streams, you can also download them from GitHub.
- The Kinesis Client Library (KCL) provides an easy-to-use programming model for processing data. The KCL can help you get started quickly with Kinesis Data Streams in Java, Node.js, .NET, Python, and Ruby. For more information see Reading Data from Streams (p. 112).
- The AWS Command Line Interface supports Kinesis Data Streams. The AWS CLI enables you to control multiple AWS services from the command line and automate them through scripts.

Configure Your Development Environment

To use the KCL, ensure that your Java development environment meets the following requirements:

- Java 1.7 (Java SE 7 JDK) or later. You can download the latest Java software from Java SE Downloads on the Oracle website.
- Apache Commons package (Code, HTTP Client, and Logging)
- Jackson JSON processor

Note that the AWS SDK for Java includes Apache Commons and Jackson in the third-party folder. However, the SDK for Java works with Java 1.6, while the Kinesis Client Library requires Java 1.7.

Getting Started with Amazon Kinesis Data Streams

The information in this section helps you get started using Amazon Kinesis Data Streams. If you are new to Kinesis Data Streams, start by becoming familiar with the concepts and terminology presented in Amazon Kinesis Data Streams Terminology and Concepts (p. 3).

This section shows you how to perform basic Amazon Kinesis Data Streams operations using the AWS Command Line Interface. You will learn fundamental Kinesis Data Streams data flow principles and the steps necessary to put and get data from an Kinesis data stream.

Topics

For CLI access, you need an access key ID and secret access key. Use IAM user access keys instead of AWS account root user access keys. IAM lets you securely control access to AWS services and resources in your AWS account. For more information about creating access keys, see Understanding and getting your security credentials in the *AWS General Reference*.

You can find detailed step-by-step IAM and security key set up instructions at Create an IAM User.

In this section, the specific commands discussed are given verbatim, except where specific values are necessarily different for each run. Also, the examples are using the US West (Oregon) region, but the steps in this section work in any of the regions where Kinesis Data Streams is supported.

Install and Configure the AWS CLI

Install AWS CLI

For detailed steps on how to install the AWS CLI for Windows and for Linux, OS X, and Unix operating systems, see Installing the AWS CLI.

Use the following command to list available options and services:

```
aws help
```

You will be using the Kinesis Data Streams service, so you can review the AWS CLI subcommands related to Kinesis Data Streams using the following command:

```
aws kinesis help
```

This command results in output that includes the available Kinesis Data Streams commands:

```
AVAILABLE COMMANDS

       o add-tags-to-stream

       o create-stream
```

```
    o delete-stream

    o describe-stream

    o get-records

    o get-shard-iterator

    o help

    o list-streams

    o list-tags-for-stream

    o merge-shards

    o put-record

    o put-records

    o remove-tags-from-stream

    o split-shard

    o wait
```

This command list corresponds to the Kinesis Data Streams API documented in the Amazon Kinesis Service API Reference. For example, the `create-stream` command corresponds to the `CreateStream` API action.

The AWS CLI is now successfully installed, but not configured. This is shown in the next section.

Configure AWS CLI

For general use, the `aws configure` command is the fastest way to set up your AWS CLI installation. For more information, see Configuring the AWS CLI.

Perform Basic Kinesis Data Stream Operations Using the AWS CLI

This section describes basic use of a Kinesis data stream from the command line using the AWS CLI. Be sure you are familiar with the concepts discussed in Amazon Kinesis Data Streams Terminology and Concepts (p. 3).

> **Note**
> After you create a stream, your account incurs nominal charges for Kinesis Data Streams usage because Kinesis Data Streams is not eligible for the AWS free tier. When you are finished with this tutorial, delete your AWS resources to stop incurring charges. For more information, see Step 4: Clean Up (p. 19).

Topics

Step 1: Create a Stream

Your first step is to create a stream and verify that it was successfully created. Use the following command to create a stream named "Foo":

```
aws kinesis create-stream --stream-name Foo
```

Next, issue the following command to check on the stream's creation progress:

```
aws kinesis describe-stream-summary --stream-name Foo
```

You should get output that is similar to the following example:

```
{
    "StreamDescriptionSummary": {
        "StreamName": "Foo",
        "StreamARN": "arn:aws:kinesis:us-west-2:123456789012:stream/Foo",
        "StreamStatus": "CREATING",
        "RetentionPeriodHours": 48,
        "StreamCreationTimestamp": 1572297168.0,
        "EnhancedMonitoring": [
            {
                "ShardLevelMetrics": []
            }
        ],
        "EncryptionType": "NONE",
        "OpenShardCount": 3,
        "ConsumerCount": 0
    }
}
```

In this example, the stream has a status CREATING, which means it is not quite ready to use. Check again in a few moments, and you should see output similar to the following example:

```
{
    "StreamDescriptionSummary": {
        "StreamName": "Foo",
        "StreamARN": "arn:aws:kinesis:us-west-2:123456789012:stream/Foo",
        "StreamStatus": "ACTIVE",
        "RetentionPeriodHours": 48,
        "StreamCreationTimestamp": 1572297168.0,
        "EnhancedMonitoring": [
            {
                "ShardLevelMetrics": []
            }
        ],
        "EncryptionType": "NONE",
        "OpenShardCount": 3,
        "ConsumerCount": 0
    }
}
```

There is information in this output that you don't need to be concerned about for this tutorial. The main thing for now is `"StreamStatus": "ACTIVE"`, which tells you that the stream is ready to be used, and the information on the single shard that you requested. You can also verify the existence of your new stream by using the `list-streams` command, as shown here:

```
aws kinesis list-streams
```

Output:

```
{
    "StreamNames": [
        "Foo"
    ]
}
```

Step 2: Put a Record

Now that you have an active stream, you are ready to put some data. For this tutorial, you will use the simplest possible command, `put-record`, which puts a single data record containing the text "testdata" into the stream:

```
aws kinesis put-record --stream-name Foo --partition-key 123 --data testdata
```

This command, if successful, will result in output similar to the following example:

```
{
    "ShardId": "shardId-000000000000",
    "SequenceNumber": "49546986683135544286507457936321625675700192471156785154"
}
```

Congratulations, you just added data to a stream! Next you will see how to get data out of the stream.

Step 3: Get the Record

GetShardIterator

Before you can get data from the stream you need to obtain the shard iterator for the shard you are interested in. A shard iterator represents the position of the stream and shard from which the consumer (`get-record` command in this case) will read. You'll use the `get-shard-iterator` command, as follows:

```
aws kinesis get-shard-iterator --shard-id shardId-000000000000 --shard-iterator-type
 TRIM_HORIZON --stream-name Foo
```

Recall that the `aws kinesis` commands have a Kinesis Data Streams API behind them, so if you are curious about any of the parameters shown, you can read about them in the `GetShardIterator` API reference topic. Successful execution will result in output similar to the following example (scroll horizontally to see the entire output):

```
{
    "ShardIterator": "AAAAAAAAAAHSywljv0zEgPX4NyKdZ5wryMzP9yALs8NeKbUjp1IxtZs1Sp
+KEd9I6AJ9ZG4lNR1EMi+9Md/nHvtLyxpfhEzYvkTZ4D9DQVz/mBYWRO6OTZRKnW9gd+efGN2aHFdkH1rJl4BL9Wyrk
+ghYG22D2T1Da2EyNSH1+LAbK33gQweTJADBdyMwlo5r6PqcP2dzhg="
}
```

The long string of seemingly random characters is the shard iterator (yours will be different). You will need to copy/paste the shard iterator into the get command, shown next. Shard iterators have a valid lifetime of 300 seconds, which should be enough time for you to copy/paste the shard iterator into the next command. Notice you will need to remove any newlines from your shard iterator before pasting to

the next command. If you get an error message that the shard iterator is no longer valid, simply execute the `get-shard-iterator` command again.

GetRecords

The `get-records` command gets data from the stream, and it resolves to a call to `GetRecords` in the Kinesis Data Streams API. The shard iterator specifies the position in the shard from which you want to start reading data records sequentially. If there are no records available in the portion of the shard that the iterator points to, `GetRecords` returns an empty list. Note that it might take multiple calls to get to a portion of the shard that contains records.

In the following example of the `get-records` command (scroll horizontally to see the entire command):

```
aws kinesis get-records --shard-iterator
 AAAAAAAAAAHSywljv0zEgPX4NyKdZ5wryMzP9yALs8NeKbUjp1IxtZs1Sp+KEd9I6AJ9ZG4lNR1EMi
+9Md/nHvtLyxpfhEzYvkTZ4D9DQVz/mBYWRO6OTZRKnW9gd+efGN2aHFdkH1rJl4BL9Wyrk
+ghYG22D2T1Da2EyNSH1+LAbK33gQweTJADBdyMwlo5r6PqcP2dzhg=
```

If you are running this tutorial from a Unix-type command processor such as bash, you can automate the acquisition of the shard iterator using a nested command, like this (scroll horizontally to see the entire command):

```
SHARD_ITERATOR=$(aws kinesis get-shard-iterator --shard-id shardId-000000000000 --shard-
iterator-type TRIM_HORIZON --stream-name Foo --query 'ShardIterator')

aws kinesis get-records --shard-iterator $SHARD_ITERATOR
```

If you are running this tutorial from a system that supports PowerShell, you can automate acquisition of the shard iterator using a command such as this (scroll horizontally to see the entire command):

```
aws kinesis get-records --shard-iterator ((aws kinesis get-shard-iterator --shard-id
 shardId-000000000000 --shard-iterator-type TRIM_HORIZON --stream-name Foo).split('"')[4])
```

The successful result of the `get-records` command will request records from your stream for the shard that you specified when you obtained the shard iterator, as in the following example (scroll horizontally to see the entire output):

```
{
  "Records":[ {
    "Data":"dGVzdGRhdGE=",
    "PartitionKey":"123",
    "ApproximateArrivalTimestamp": 1.441215410867E9,
    "SequenceNumber":"49544985256907370027570885864065577703022652638596431874"
  } ],
  "MillisBehindLatest":24000,

 "NextShardIterator":"AAAAAAAAAAEDOW3ugseWPE4503kqN1yN1UaodY8unE0sYslMUmC6lX9hlig5+t4RtZM0/
tALfiI4QGjunVgJvQsjxjh2aLyxaAaPr
+LaoENQ7eVs4EdYXgKyThTZGPcca2fVXYJWL3yafv9dsDwsYVedI66dbMZFC8rPMWc797zxQkv4pSKvPOZvrUIudb8UkH3VMzx58Is=
}
```

Note that `get-records` is described above as a *request*, which means you may receive zero or more records even if there are records in your stream, and any records returned may not represent all the records currently in your stream. This is perfectly normal, and production code will simply poll the stream for records at appropriate intervals (this polling speed will vary depending on your specific application design requirements).

The first thing you'll likely notice about your record in this part of the tutorial is that the data appears to be garbage –; it's not the clear text `testdata` we sent. This is due to the way `put-record` uses Base64

encoding to allow you to send binary data. However, the Kinesis Data Streams support in the AWS CLI does not provide Base64 *decoding* because Base64 decoding to raw binary content printed to stdout can lead to undesired behavior and potential security issues on certain platforms and terminals. If you use a Base64 decoder (for example, https://www.base64decode.org/) to manually decode `dGVzdGRhdGE=` you will see that it is, in fact, `testdata`. This is sufficient for the sake of this tutorial because, in practice, the AWS CLI is rarely used to consume data, but more often to monitor the state of the stream and obtain information, as shown previously (`describe-stream` and `list-streams`). Future tutorials will show you how to build production-quality consumer applications using the Kinesis Client Library (KCL), where Base64 is taken care of for you. For more information about the KCL, see Developing Custom Consumers with Shared Throughput Using KCL.

It's not always the case that `get-records` will return all records in the stream/shard specified. When that happens, use the `NextShardIterator` from the last result to get the next set of records. So if more data were being put into the stream (the normal situation in production applications), you could keep polling for data using `get-records` each time. However, if you do not call `get-records` using the next shard iterator within the 300 second shard iterator lifetime, you will get an error message, and you will need to use the `get-shard-iterator` command to get a fresh shard iterator.

Also provided in this output is `MillisBehindLatest`, which is the number of milliseconds the GetRecords operation's response is from the tip of the stream, indicating how far behind current time the consumer is. A value of zero indicates record processing is caught up, and there are no new records to process at this moment. In the case of this tutorial, you may see a number that's quite large if you've been taking time to read along as you go. That's not a problem, by default, data records stay in a stream for 24 hours waiting for you to retrieve them. This time frame is called the retention period and it is configurable up to 365 days.

Note that a successful `get-records` result will always have a `NextShardIterator` even if there are no more records currently in the stream. This is a polling model that assumes a producer is potentially putting more records into the stream at any given time. Although you can write your own polling routines, if you use the previously mentioned KCL for developing consumer applications, this polling is taken care of for you.

If you call `get-records` until there are no more records in the stream and shard you are pulling from, you will see output with empty records similar to the following example (scroll horizontally to see the entire output):

```
{
    "Records": [],
    "NextShardIterator": "AAAAAAAAAAGCJ5jzQNjmdhO6B/YDIDE56jmZmrmMA/r1WjoHXC/
kPJXc1rckt3TFL55dENfe5meNgdkyCRpUPGzJpMgYHaJ53C3nCAjQ6s7ZupjXeJGoUFs5oCuFwhP+Wul/
EhyNeSs5DYXLSSC5XCapmCAYGFjYER69QSdQjxMmBPE/hiybFDi5qtkT6/PsZNz6kFoqtDk="
}
```

Step 4: Clean Up

Finally, you'll want to delete your stream to free up resources and avoid unintended charges to your account, as previously noted. Do this in practice any time you have created a stream and will not be using it because charges accrue per stream whether you are putting and getting data with it or not. The clean-up command is simple:

```
aws kinesis delete-stream --stream-name Foo
```

Success results in no output, so you might want to use `describe-stream` to check on deletion progress:

```
aws kinesis describe-stream-summary --stream-name Foo
```

If you execute this command immediately after the delete command, you will likely see output part of which is similar to the following example:

```
{
    "StreamDescriptionSummary": {
        "StreamName": "samplestream",
        "StreamARN": "arn:aws:kinesis:us-west-2:123456789012:stream/samplestream",
        "StreamStatus": "ACTIVE",
```

After the stream is fully deleted, `describe-stream` will result in a "not found" error:

```
A client error (ResourceNotFoundException) occurred when calling the DescribeStreamSummary
 operation:
Stream Foo under account 123456789012 not found.
```

Example Tutorials for Amazon Kinesis Data Streams

The example tutorials in this section are designed to further assist you in understanding Amazon Kinesis Data Streams concepts and functionality.

Topics

Tutorial: Process Real-Time Stock Data Using KPL and KCL 2.x

The scenario for this tutorial involves ingesting stock trades into a data stream and writing a simple Amazon Kinesis Data Streams application that performs calculations on the stream. You will learn how to send a stream of records to Kinesis Data Streams and implement an application that consumes and processes the records in near-real time.

> **Important**
> After you create a stream, your account incurs nominal charges for Kinesis Data Streams usage because Kinesis Data Streams is not eligible for the AWS Free Tier. After the consumer application starts, it also incurs nominal charges for Amazon DynamoDB usage. The consumer application uses DynamoDB to track processing state. When you are finished with this application, delete your AWS resources to stop incurring charges. For more information, see Step 7: Finishing Up (p. 34).

The code does not access actual stock market data, but instead simulates the stream of stock trades. It does so by using a random stock trade generator that has a starting point of real market data for the top 25 stocks by market capitalization as of February 2015. If you have access to a real-time stream of stock trades, you might be interested in deriving useful, timely statistics from that stream. For example, you might want to perform a sliding window analysis where you determine the most popular stock purchased in the last 5 minutes. Or you might want a notification whenever there is a sell order that is too large (that is, it has too many shares). You can extend the code in this series to provide such functionality.

You can work through the steps in this tutorial on your desktop or laptop computer and run both the producer and consumer code on the same machine or any platform that supports the defined requirements.

The examples shown use the US West (Oregon) region, but they work on any of the AWS regions that support Kinesis Data Streams.

Tasks

Prerequisites

You must meet the following requirements in order to complete this tutorial:

Amazon Web Services Account

Before you begin, ensure that you are familiar with the concepts discussed in Amazon Kinesis Data Streams Terminology and Concepts (p. 3), particularly with streams, shards, producers, and consumers. It is also helpful to have completed the steps in the following guide: Install and Configure the AWS CLI (p. 14).

You must have an AWS account and a web browser to access the AWS Management Console.

For console access, use your IAM user name and password to sign in to the AWS Management Console from the IAM sign-in page. IAM lets you securely control access to AWS services and resources in your AWS account. For details about console and programmatic credentials, see Understanding and getting your security credentials in the *AWS General Reference*.

For more information about IAM and security key setup instructions, see Create an IAM User.

System Software Requirements

The system that you are using to run the application must have Java 7 or higher installed. To download and install the latest Java Development Kit (JDK), go to Oracle's Java SE installation site.

If you have a Java IDE, such as Eclipse, you can use it to open, edit, build, and run the source code.

You need the latest AWS SDK for Java version. If you are using Eclipse as your IDE, you can install the AWS Toolkit for Eclipse instead.

The consumer application requires the Kinesis Client Library (KCL) version 2.2.9 or higher, which you can obtain from GitHub at https://github.com/awslabs/amazon-kinesis-client/tree/master.

Next Steps

Step 1: Create a Data Stream (p. 22)

Step 1: Create a Data Stream

First, you must create the data stream that you will use in subsequent steps of this tutorial.

To create a stream

1. Sign in to the AWS Management Console and open the Kinesis console at https://console.aws.amazon.com/kinesis.
2. Choose **Data Streams** in the navigation pane.
3. In the navigation bar, expand the region selector and choose a region.
4. Choose **Create Kinesis stream**.

5. Enter a name for your data stream (for example, **`StockTradeStream`**).
6. Enter **`1`** for the number of shards, but leave **Estimate the number of shards you'll need** collapsed.
7. Choose **Create Kinesis stream**.

On the **Kinesis streams** list page, the status of your stream appears as `CREATING` while the stream is being created. When the stream is ready to use, the status changes to `ACTIVE`.

If you choose the name of your stream, in the page that appears, the **Details** tab displays a summary of your data stream configuration. The **Monitoring** section displays monitoring information for the stream.

Next Steps

Step 2: Create an IAM Policy and User (p. 23)

Step 2: Create an IAM Policy and User

Security best practices for AWS dictate the use of fine-grained permissions to control access to different resources. AWS Identity and Access Management (IAM) allows you to manage users and user permissions in AWS. An IAM policy explicitly lists actions that are allowed and the resources on which the actions are applicable.

The following are the minimum permissions generally required for Kinesis Data Streams producers and consumers.

Producer

Actions	Resource	Purpose
`DescribeStream`, `DescribeStreamSummary`, `DescribeStreamConsumer`	Kinesis data stream	Before attempting to read records, the consumer checks shards are contained in the data stream.
`SubscribeToShard`, `RegisterStreamConsumer`	Kinesis data stream	Subscribes and registers consumers to a shard.
`PutRecord`, `PutRecords`	Kinesis data stream	Writes records to Kinesis Data Streams.

Consumer

Actions	Resource	Purpose
`DescribeStream`	Kinesis data stream	Before attempting to read records, the consumer checks shards are contained in the data stream.
`GetRecords`, `GetShardIterator`	Kinesis data stream	Reads records from a shard.
`CreateTable`, `DescribeTable`, `GetItem`, `PutItem`, `Scan`, `UpdateItem`	Amazon DynamoDB table	If the consumer is developed using the Kinesis Client Lib permissions to a DynamoDB table to track the processin
`DeleteItem`	Amazon DynamoDB table	For when the consumer performs split/merge operation
`PutMetricData`	Amazon CloudWatch log	The KCL also uploads metrics to CloudWatch, which are

For this tutorial, you will create a single IAM policy that grants all of the above permissions. In production, you might want to create two policies, one for producers and one for consumers.

To create an IAM policy

1. Locate the Amazon Resource Name (ARN) for the new data stream that you created in the step above. You can find this ARN listed as **Stream ARN** at the top of the **Details** tab. The ARN format is as follows:

   ```
   arn:aws:kinesis:region:account:stream/name
   ```

   region

   The AWS region code; for example, `us-west-2`. For more information, see Region and Availability Zone Concepts.

   account

   The AWS account ID, as shown in Account Settings.

   name

   The name of the data stream that you created in the step above, which is `StockTradeStream`.
2. Determine the ARN for the DynamoDB table to be used by the consumer (and to be created by the first consumer instance). It must be in the following format:

   ```
   arn:aws:dynamodb:region:account:table/name
   ```

   The region and account ID are identical to the values in the ARN of the data stream that you're using for this tutorial, but the *name* is the name of the DynamoDB table created and used by the consumer application. KCL uses the application name as the table name. In this step, use `StockTradesProcessor` for the DynamoDB table name, because that is the application name used in later steps in this tutorial.
3. In the IAM console, in **Policies** (https://console.aws.amazon.com/iam/home#policies), choose **Create policy**. If this is the first time that you have worked with IAM policies, choose **Get Started**, **Create Policy**.
4. Choose **Select** next to **Policy Generator**.
5. Choose **Amazon Kinesis** as the AWS service.
6. Select `DescribeStream`, `GetShardIterator`, `GetRecords`, `PutRecord`, and `PutRecords` as the allowed actions.
7. Enter the ARN of the data stream that you're using in this tutorial.
8. Use **Add Statement** for each of the following:

AWS Service	Actions	ARN
Amazon DynamoDB	`CreateTable`, `DeleteItem`, `DescribeTable`, `GetItem`, `PutItem`, `Scan`, `UpdateItem`	The ARN of the DynamoDB table that you created in Step 2 of this procedure.
Amazon CloudWatch	`PutMetricData`	*

   The asterisk (*) that is used when specifying an ARN is not required. In this case, it's because there is no specific resource in CloudWatch on which the `PutMetricData` action is invoked.
9. Choose **Next Step**.

10. Change **Policy Name** to `StockTradeStreamPolicy`, review the code, and choose **Create Policy**.

The resulting policy document should look like this:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt123",
      "Effect": "Allow",
      "Action": [
        "kinesis:DescribeStream",
        "kinesis:PutRecord",
        "kinesis:PutRecords",
        "kinesis:GetShardIterator",
        "kinesis:GetRecords",
        "kinesis:ListShards",
        "kinesis:DescribeStreamSummary",
        "kinesis:RegisterStreamConsumer"
      ],
      "Resource": [
        "arn:aws:kinesis:us-west-2:123:stream/StockTradeStream"
      ]
    },
    {
      "Sid": "Stmt234",
      "Effect": "Allow",
      "Action": [
        "kinesis:SubscribeToShard",
        "kinesis:DescribeStreamConsumer"
      ],
      "Resource": [
        "arn:aws:kinesis:us-west-2:123:stream/StockTradeStream/*"
      ]
    },
    {
      "Sid": "Stmt456",
      "Effect": "Allow",
      "Action": [
        "dynamodb:*"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-west-2:123:table/StockTradesProcessor"
      ]
    },
    {
      "Sid": "Stmt789",
      "Effect": "Allow",
      "Action": [
        "cloudwatch:PutMetricData"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```

To create an IAM user

1. Open the IAM console at https://console.aws.amazon.com/iam/.
2. On the **Users** page, choose **Add user**.

3. For **User name**, type `StockTradeStreamUser`.
4. For **Access type**, choose **Programmatic access**, and then choose **Next: Permissions**.
5. Choose **Attach existing policies directly**.
6. Search by name for the policy that you created in the procedure above (`StockTradeStreamPolicy`. Select the box to the left of the policy name, and then choose **Next: Review**.
7. Review the details and summary, and then choose **Create user**.
8. Copy the **Access key ID**, and save it privately. Under **Secret access key**, choose **Show**, and save that key privately also.
9. Paste the access and secret keys to a local file in a safe place that only you can access. For this application, create a file named `~/.aws/credentials` (with strict permissions). The file should be in the following format:

```
[default]
aws_access_key_id=access key
aws_secret_access_key=secret access key
```

To attach an IAM policy to a user

1. In the IAM console, open Policies and choose **Policy Actions**.
2. Choose `StockTradeStreamPolicy` and **Attach**.
3. Choose `StockTradeStreamUser` and **Attach Policy**.

Next Steps

Step 3: Download and Build the Code (p. 26)

Step 3: Download and Build the Code

This topic provides sample implementation code for the sample stock trades ingestion into the data stream (*producer*) and the processing of this data (*consumer*).

To download and build the code

1. Download the source code from the https://github.com/aws-samples/amazon-kinesis-learning GitHub repo to your computer.
2. Create a project in your IDE with the source code, adhering to the provided directory structure.
3. Add the following libraries to the project:

   - Amazon Kinesis Client Library (KCL)
   - AWS SDK
   - Apache HttpCore
   - Apache HttpClient
   - Apache Commons Lang
   - Apache Commons Logging
   - Guava (Google Core Libraries For Java)
   - Jackson Annotations
   - Jackson Core
   - Jackson Databind
   - Jackson Dataformat: CBOR

- Joda Time

4. Depending on your IDE, the project might be built automatically. If not, build the project using the appropriate steps for your IDE.

If you complete these steps successfully, you are now ready to move to the next section, the section called “Step 4: Implement the Producer” (p. 27).

Next Steps

(p. 27)

Step 4: Implement the Producer

This tutorial uses the real-world scenario of stock market trade monitoring. The following principles briefly explain how this scenario maps to the producer and its supporting code structure.

Refer to the source code and review the following information.

StockTrade class

An individual stock trade is represented by an instance of the StockTrade class. This instance contains attributes such as the ticker symbol, price, number of shares, the type of the trade (buy or sell), and an ID uniquely identifying the trade. This class is implemented for you.

Stream record

A stream is a sequence of records. A record is a serialization of a `StockTrade` instance in JSON format. For example:

```
{
  "tickerSymbol": "AMZN",
  "tradeType": "BUY",
  "price": 395.87,
  "quantity": 16,
  "id": 3567129045
}
```

StockTradeGenerator class

StockTradeGenerator has a method called `getRandomTrade()` that returns a new randomly generated stock trade every time it is invoked. This class is implemented for you.

StockTradesWriter class

The `main` method of the producer, StockTradesWriter continuously retrieves a random trade and then sends it to Kinesis Data Streams by performing the following tasks:

1. Reads the data stream name and region name as input.
2. Uses the `KinesisAsyncClientBuilder` to set the region, credentials, and client configuration.
3. Checks that the stream exists and is active (if not, it exits with an error).
4. In a continuous loop, calls the `StockTradeGenerator.getRandomTrade()` method and then the `sendStockTrade` method to send the trade to the stream every 100 milliseconds.

The `sendStockTrade` method of the `StockTradesWriter` class has the following code:

```
private static void sendStockTrade(StockTrade trade, KinesisAsyncClient kinesisClient,
```

```
            String streamName) {
        byte[] bytes = trade.toJsonAsBytes();
        // The bytes could be null if there is an issue with the JSON serialization by
the Jackson JSON library.
        if (bytes == null) {
            LOG.warn("Could not get JSON bytes for stock trade");
            return;
        }

        LOG.info("Putting trade: " + trade.toString());
        PutRecordRequest request = PutRecordRequest.builder()
                .partitionKey(trade.getTickerSymbol()) // We use the ticker symbol as
the partition key, explained in the Supplemental Information section below.
                .streamName(streamName)
                .data(SdkBytes.fromByteArray(bytes))
                .build();
        try {
            kinesisClient.putRecord(request).get();
        } catch (InterruptedException e) {
            LOG.info("Interrupted, assuming shutdown.");
        } catch (ExecutionException e) {
            LOG.error("Exception while sending data to Kinesis. Will try again next
cycle.", e);
        }
    }
```

Refer to the following code breakdown:

- The `PutRecord` API expects a byte array, and you need to convert trade to JSON format. This single line of code performs that operation:

```
byte[] bytes = trade.toJsonAsBytes();
```

- Before you can send the trade, you create a new `PutRecordRequest` instance (called request in this case). Each `request` requires the stream name, partition key, and a data blob.

```
PutPutRecordRequest request = PutRecordRequest.builder()
    .partitionKey(trade.getTickerSymbol()) // We use the ticker symbol as the
 partition key, explained in the Supplemental Information section below.
    .streamName(streamName)
    .data(SdkBytes.fromByteArray(bytes))
    .build();
```

  The example uses a stock ticket as a partition key, which maps the record to a specific shard. In practice, you should have hundreds or thousands of partition keys per shard such that records are evenly dispersed across your stream. For more information about how to add data to a stream, see Writing Data to Amazon Kinesis Data Streams (p. 84).

  Now `request` is ready to send to the client (the put operation):

```
   kinesisClient.putRecord(request).get();
```

- Error checking and logging are always useful additions. This code logs error conditions:

```
if (bytes == null) {
    LOG.warn("Could not get JSON bytes for stock trade");
    return;
}
```

Add the try/catch block around the `put` operation:

```
try {
  kinesisClient.putRecord(request).get();
} catch (InterruptedException e) {
            LOG.info("Interrupted, assuming shutdown.");
} catch (ExecutionException e) {
            LOG.error("Exception while sending data to Kinesis. Will try again next
 cycle.", e);
}
```

This is because a Kinesis Data Streams put operation can fail because of a network error, or due to the data stream reaching its throughput limits and getting throttled. It is recommended that you carefully consider your retry policy for `put` operations to avoid data loss, such using as a simple retry.

- Status logging is helpful but optional:

```
LOG.info("Putting trade: " + trade.toString());
```

The producer shown here uses the Kinesis Data Streams API single record functionality, `PutRecord`. In practice, if an individual producer generates many records, it is often more efficient to use the multiple records functionality of `PutRecords` and send batches of records at a time. For more information, see Writing Data to Amazon Kinesis Data Streams (p. 84).

To run the producer

1. Verify that the access key and secret key pair retrieved in Step 2: Create an IAM Policy and User (p. 23) are saved in the file `~/.aws/credentials`.
2. Run the `StockTradeWriter` class with the following arguments:

```
StockTradeStream us-west-2
```

If you created your stream in a region other than `us-west-2`, you have to specify that region here instead.

You should see output similar to the following:

```
Feb 16, 2015 3:53:00 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter sendStockTrade
INFO: Putting trade: ID 8: SELL 996 shares of BUD for $124.18
Feb 16, 2015 3:53:00 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter sendStockTrade
INFO: Putting trade: ID 9: BUY 159 shares of GE for $20.85
Feb 16, 2015 3:53:01 PM
```

```
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter sendStockTrade
INFO: Putting trade: ID 10: BUY 322 shares of WMT for $90.08
```

Your stock trades are now being ingested by Kinesis Data Streams.

Next Steps

Step 5: Implement the Consumer (p. 30)

Step 5: Implement the Consumer

The consumer application in this tutorial continuously processes the stock trades in your data stream. It then outputs the most popular stocks being bought and sold every minute. The application is built on top of the Kinesis Client Library (KCL), which does much of the heavy lifting common to consumer apps. For more information, see Using the Kinesis Client Library (p. 113).

Refer to the source code and review the following information.

StockTradesProcessor class

Main class of the consumer, provided for you, which performs the following tasks:

- Reads the application, data stream, and region names, passed in as arguments.
- Creates a `KinesisAsyncClient` instance with the region name.
- Creates a `StockTradeRecordProcessorFactory` instance that serves instances of `ShardRecordProcessor`, implemented by a `StockTradeRecordProcessor` instance.
- Creates a `ConfigsBuilder` instance with the `KinesisAsyncClient`, `StreamName`, `ApplicationName` and `StockTradeRecordProcessorFactory` instance. This is useful for creating all configurations with default values.
- Creates a KCL scheduler (previously, in KCL versions 1.x it was known as the KCL worker) with the `ConfigsBuilder` instance.
- The scheduler creates a new thread for each shard (assigned to this consumer instance), which continuously loops to read records from the data stream. It then invokes the `StockTradeRecordProcessor` instance to process each batch of records received.

StockTradeRecordProcessor class

Implementation of the `StockTradeRecordProcessor` instance, which in turn implements five required methods: `initialize`, `processRecords`, `leaseLost`, `shardEnded`, and `shutdownRequested`.

The `initialize` and `shutdownRequested` methods are used by the KCL to let the record processor know when it should be ready to start receiving records and when it should expect to stop receiving records, respectively, so it can perform any application-specific setup and termination tasks. `leaseLost` and `shardEnded` are used to implement any logic for what to do when a lease is lost or a processing has reached the end of a shard. In this example, we simply log messages indicating these events.

The code for these methods is provided for you. The main processing happens in the `processRecords` method, which in turn uses `processRecord` for each record. This latter method is provided as the mostly empty skeleton code for you to implement in the next step, where it is explained in greater detail.

Also of note is the implementation of the support methods for `processRecord`: `reportStats`, and `resetStats`, which are empty in the original source code.

The `processRecords` method is implemented for you, and performs the following steps:

- For each record passed in, it calls `processRecord` on it.
- If at least 1 minute has elapsed since the last report, calls `reportStats()`, which prints out the latest stats, and then `resetStats()` which clears the stats so that the next interval includes only new records.
- Sets the next reporting time.
- If at least 1 minute has elapsed since the last checkpoint, calls `checkpoint()`.
- Sets the next checkpointing time.

This method uses 60-second intervals for the reporting and checkpointing rate. For more information about checkpointing, see Using the Kinesis Client Library.

StockStats class

This class provides data retention and statistics tracking for the most popular stocks over time. This code is provided for you and contains the following methods:

- `addStockTrade(StockTrade)`: injects the given `StockTrade` into the running statistics.
- `toString()`: returns the statistics in a formatted string.

This class keeps track of the most popular stock by keeping a running count of the total number of trades for each stock and the maximum count. It updates these counts whenever a stock trade arrives.

Add code to the methods of the `StockTradeRecordProcessor` class, as shown in the following steps.

To implement the consumer

1. Implement the `processRecord` method by instantiating a correctly sized `StockTrade` object and adding the record data to it, logging a warning if there's a problem.

```
byte[] arr = new byte[record.data().remaining()];
record.data().get(arr);
StockTrade trade = StockTrade.fromJsonAsBytes(arr);
    if (trade == null) {
        log.warn("Skipping record. Unable to parse record into StockTrade. Partition
 Key: " + record.partitionKey());
        return;
        }
stockStats.addStockTrade(trade);
```

2. Implement a simple `reportStats` method. Feel free to modify the output format to suit your preferences.

```
System.out.println("****** Shard " + kinesisShardId + " stats for last 1 minute ******
\n" +
stockStats + "\n" +
"*****************************************************************\n");
```

3. Implement the `resetStats` method, which creates a new `stockStats` instance.

```
stockStats = new StockStats();
```

4. Implement the following methods required by `ShardRecordProcessor` interface

```
@Override
public void leaseLost(LeaseLostInput leaseLostInput) {
    log.info("Lost lease, so terminating.");
}

@Override
public void shardEnded(ShardEndedInput shardEndedInput) {
    try {
        log.info("Reached shard end checkpointing.");
        shardEndedInput.checkpointer().checkpoint();
    } catch (ShutdownException | InvalidStateException e) {
        log.error("Exception while checkpointing at shard end. Giving up.", e);
    }
}

@Override
public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
    log.info("Scheduler is shutting down, checkpointing.");
    checkpoint(shutdownRequestedInput.checkpointer());
}

private void checkpoint(RecordProcessorCheckpointer checkpointer) {
    log.info("Checkpointing shard " + kinesisShardId);
    try {
        checkpointer.checkpoint();
    } catch (ShutdownException se) {
        // Ignore checkpoint if the processor instance has been shutdown (fail over).
        log.info("Caught shutdown exception, skipping checkpoint.", se);
    } catch (ThrottlingException e) {
        // Skip checkpoint when throttled. In practice, consider a backoff and retry
 policy.
        log.error("Caught throttling exception, skipping checkpoint.", e);
    } catch (InvalidStateException e) {
        // This indicates an issue with the DynamoDB table (check for table,
 provisioned IOPS).
        log.error("Cannot save checkpoint to the DynamoDB table used by the Amazon
 Kinesis Client Library.", e);
    }
}
```

To run the consumer

1. Run the producer that you wrote in (p. 27) to inject simulated stock trade records into your stream.
2. Verify that the access key and secret key pair retrieved earlier (when creating the IAM user) are saved in the file `~/.aws/credentials`.
3. Run the `StockTradesProcessor` class with the following arguments:

```
StockTradesProcessor StockTradeStream us-west-2
```

   Note that if you created your stream in a region other than `us-west-2`, you have to specify that region here instead.

After a minute, you should see output like the following, refreshed every minute thereafter:

```
****** Shard shardId-000000000001 stats for last 1 minute ******
Most popular stock being bought: WMT, 27 buys.
Most popular stock being sold: PTR, 14 sells.
****************************************************************
```

Next Steps

Step 6: (Optional) Extending the Consumer (p. 33)

Step 6: (Optional) Extending the Consumer

This optional section shows how you can extend the consumer code for a slightly more elaborate scenario.

If you want to know about the biggest sell orders each minute, you can modify the `StockStats` class in three places to accommodate this new priority.

To extend the consumer

1. Add new instance variables:

```
// Ticker symbol of the stock that had the largest quantity of shares sold
private String largestSellOrderStock;
// Quantity of shares for the largest sell order trade
private long largestSellOrderQuantity;
```

2. Add the following code to `addStockTrade`:

```
if (type == TradeType.SELL) {
     if (largestSellOrderStock == null || trade.getQuantity() >
 largestSellOrderQuantity) {
         largestSellOrderStock = trade.getTickerSymbol();
         largestSellOrderQuantity = trade.getQuantity();
     }
 }
```

3. Modify the `toString` method to print the additional information:

```
public String toString() {
    return String.format(
        "Most popular stock being bought: %s, %d buys.%n" +
        "Most popular stock being sold: %s, %d sells.%n" +
        "Largest sell order: %d shares of %s.",
        getMostPopularStock(TradeType.BUY), getMostPopularStockCount(TradeType.BUY),
        getMostPopularStock(TradeType.SELL), getMostPopularStockCount(TradeType.SELL),
        largestSellOrderQuantity, largestSellOrderStock);
}
```

If you run the consumer now (remember to run the producer also), you should see output similar to this:

```
****** Shard shardId-000000000001 stats for last 1 minute ******
```

```
Most popular stock being bought: WMT, 27 buys.
Most popular stock being sold: PTR, 14 sells.
Largest sell order: 996 shares of BUD.
************************************************************
```

Next Steps

Step 7: Finishing Up (p. 34)

Step 7: Finishing Up

Because you are paying to use the Kinesis data stream, make sure that you delete it and the corresponding Amazon DynamoDB table when you are done with it. Nominal charges occur on an active stream even when you aren't sending and getting records. This is because an active stream is using resources by continuously "listening" for incoming records and requests to get records.

To delete the stream and table

1. Shut down any producers and consumers that you may still have running.
2. Open the Kinesis console at https://console.aws.amazon.com/kinesis.
3. Choose the stream that you created for this application (`StockTradeStream`).
4. Choose **Delete Stream**.
5. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
6. Delete the `StockTradesProcessor` table.

Summary

Processing a large amount of data in near-real time doesn't require writing any magical code or developing a huge infrastructure. It is as simple as writing logic to process a small amount of data (like writing `processRecord(Record)`) but using Kinesis Data Streams to scale so that it works for a large amount of streaming data. You don't have to worry about how your processing would scale because Kinesis Data Streams handles it for you. All you have to do is send your streaming records to Kinesis Data Streams and write the logic to process each new record received.

Here are some potential enhancements for this application.

Aggregate across all shards

Currently, you get stats resulting from aggregation of the data records received by a single worker from a single shard. (A shard cannot be processed by more than one worker in a single application at the same time.) Of course, when you scale and have more than one shard, you might want to aggregate across all shards. You can do this by having a pipeline architecture where the output of each worker is fed into another stream with a single shard, which is processed by a worker that aggregates the outputs of the first stage. Because the data from the first stage is limited (one sample per minute per shard), it would easily be handled by one shard.

Scale processing

When the stream scales up to have many shards (because many producers are sending data), the way to scale the processing is to add more workers. You can run the workers in Amazon EC2 instances and use Auto Scaling groups.

Use connectors to Amazon S3/DynamoDB/Amazon Redshift/Storm

As a stream is continuously processed, its output can be sent to other destinations. AWS provides connectors to integrate Kinesis Data Streams with other AWS services and third-party tools.

Tutorial: Process Real-Time Stock Data Using KPL and KCL 1.x

The scenario for this tutorial involves ingesting stock trades into a data stream and writing a simple Amazon Kinesis Data Streams application that performs calculations on the stream. You will learn how to send a stream of records to Kinesis Data Streams and implement an application that consumes and processes the records in near-real time.

> **Important**
> After you create a stream, your account incurs nominal charges for Kinesis Data Streams usage because Kinesis Data Streams is not eligible for the AWS Free Tier. After the consumer application starts, it also incurs nominal charges for Amazon DynamoDB usage. The consumer application uses DynamoDB to track processing state. When you are finished with this application, delete your AWS resources to stop incurring charges. For more information, see Step 7: Finishing Up (p. 47).

The code does not access actual stock market data, but instead simulates the stream of stock trades. It does so by using a random stock trade generator that has a starting point of real market data for the top 25 stocks by market capitalization as of February 2015. If you have access to a real-time stream of stock trades, you might be interested in deriving useful, timely statistics from that stream. For example, you might want to perform a sliding window analysis where you determine the most popular stock purchased in the last 5 minutes. Or you might want a notification whenever there is a sell order that is too large (that is, it has too many shares). You can extend the code in this series to provide such functionality.

You can work through the steps in this tutorial on your desktop or laptop computer and run both the producer and consumer code on the same machine or any platform that supports the defined requirements, such as Amazon Elastic Compute Cloud (Amazon EC2).

The examples shown use the US West (Oregon) Region, but they work on any of the AWS Regions that support Kinesis Data Streams.

Tasks

Prerequisites

The following are requirements for completing the Tutorial: Process Real-Time Stock Data Using KPL and KCL 1.x (p. 35).

Amazon Web Services Account

Before you begin, ensure that you are familiar with the concepts discussed in Amazon Kinesis Data Streams Terminology and Concepts (p. 3), particularly streams, shards, producers, and consumers. It is also helpful to have completed Install and Configure the AWS CLI (p. 14).

You need an AWS account and a web browser to access the AWS Management Console.

For console access, use your IAM user name and password to sign in to the AWS Management Console from the IAM sign-in page. IAM lets you securely control access to AWS services and resources in your AWS account. For details about console and programmatic credentials, see Understanding and getting your security credentials in the *AWS General Reference*.

For more information about IAM and security key setup instructions, see Create an IAM User.

System Software Requirements

The system used to run the application must have Java 7 or higher installed. To download and install the latest Java Development Kit (JDK), go to Oracle's Java SE installation site.

If you have a Java IDE, such as Eclipse, you can open the source code, edit, build, and run it.

You need the latest AWS SDK for Java version. If you are using Eclipse as your IDE, you can install the AWS Toolkit for Eclipse instead.

The consumer application requires the Kinesis Client Library (KCL) version 1.2.1 or higher, which you can obtain from GitHub at Kinesis Client Library (Java).

Next Steps

Step 1: Create a Data Stream (p. 36)

Step 1: Create a Data Stream

In the first step of the Tutorial: Process Real-Time Stock Data Using KPL and KCL 1.x (p. 35), you create the stream that you will use in subsequent steps.

To create a stream

1. Sign in to the AWS Management Console and open the Kinesis console at https://console.aws.amazon.com/kinesis.
2. Choose **Data Streams** in the navigation pane.
3. In the navigation bar, expand the Region selector and choose a Region.
4. Choose **Create Kinesis stream**.
5. Enter a name for your stream (for example, **`StockTradeStream`**).
6. Enter **`1`** for the number of shards, but leave **Estimate the number of shards you'll need** collapsed.
7. Choose **Create Kinesis stream**.

On the **Kinesis streams** list page, the status of your stream is `CREATING` while the stream is being created. When the stream is ready to use, the status changes to `ACTIVE`. Choose the name of your stream. In the page that appears, the **Details** tab displays a summary of your stream configuration. The **Monitoring** section displays monitoring information for the stream.

Additional Information About Shards

When you begin to use Kinesis Data Streams outside of this tutorial, you might need to plan the stream creation process more carefully. You should plan for expected maximum demand when you provision shards. Using this scenario as an example, US stock market trading traffic peaks during the day (Eastern time) and demand estimates should be sampled from that time of day. You then have a choice to provision for the maximum expected demand, or scale your stream up and down in response to demand fluctuations.

A *shard* is a unit of throughput capacity. On the **Create Kinesis stream** page, expand **Estimate the number of shards you'll need**. Enter the average record size, the maximum records written per second, and the number of consuming applications, using the following guidelines:

Average record size

An estimate of the calculated average size of your records. If you don't know this value, use the estimated maximum record size for this value.

Max records written

Take into account the number of entities providing data and the approximate number of records per second produced by each. For example, if you are getting stock trade data from 20 trading servers and each generates 250 trades per second, the total number of trades (records) per second is 5000/second.

Number of consuming applications

The number of applications that independently read from the stream to process the stream in a different way and produce different output. Each application can have multiple instances running on different machines (that is, run in a cluster) so that it can keep up with a high volume stream.

If the estimated number of shards shown exceeds your current shard limit, you might need to submit a request to increase that limit before you can create a stream with that number of shards. To request an increase to your shard limit, use the Kinesis Data Streams Limits form. For more information about streams and shards, see Creating and Managing Streams (p. 65).

Next Steps

Step 2: Create an IAM Policy and User (p. 37)

Step 2: Create an IAM Policy and User

Security best practices for AWS dictate the use of fine-grained permissions to control access to different resources. AWS Identity and Access Management (IAM) allows you to manage users and user permissions in AWS. An IAM policy explicitly lists actions that are allowed and the resources on which the actions are applicable.

The following are the minimum permissions generally required for a Kinesis Data Streams producer and consumer.

Producer

Actions	Resource	Purpose
`DescribeStream`, `DescribeStreamSummary`, `DescribeStreamConsumer`	Kinesis data stream	Before attempting to write records, the producer checks are contained in the stream, and if the stream has a cons
`SubscribeToShard`, `RegisterStreamConsumer`	Kinesis data stream	Subscribes and register a consumers to a Kinesis Data St
`PutRecord`, `PutRecords`	Kinesis data stream	Write records to Kinesis Data Streams.

Consumer

Actions	Resource	Purpose
`DescribeStream`	Kinesis data stream	Before attempting to read records, the consumer checks are contained in the stream.

Actions	Resource	Purpose
`GetRecords`, `GetShardIterator`	Kinesis data stream	Read records from a Kinesis Data Streams shard.
`CreateTable`, `DescribeTable`, `GetItem`, `PutItem`, `Scan`, `UpdateItem`	Amazon DynamoDB table	If the consumer is developed using the Kinesis Client Lib table to track the processing state of the application. Th
`DeleteItem`	Amazon DynamoDB table	For when the consumer performs split/merge operation
`PutMetricData`	Amazon CloudWatch log	The KCL also uploads metrics to CloudWatch, which are

For this application, you create a single IAM policy that grants all of the preceding permissions. In practice, you might want to consider creating two policies, one for producers and one for consumers.

To create an IAM policy

1. Locate the Amazon Resource Name (ARN) for the new stream. You can find this ARN listed as **Stream ARN** at the top of the **Details** tab. The ARN format is as follows:

   ```
   arn:aws:kinesis:region:account:stream/name
   ```

   region

   The Region code; for example, `us-west-2`. For more information, see Region and Availability Zone Concepts.

   account

   The AWS account ID, as shown in Account Settings.

   name

   The name of the stream from Step 1: Create a Data Stream (p. 36), which is `StockTradeStream`.
2. Determine the ARN for the DynamoDB table to be used by the consumer (and created by the first consumer instance). It must be in the following format:

   ```
   arn:aws:dynamodb:region:account:table/name
   ```

   The Region and account are from the same place as the previous step, but this time *name* is the name of the table created and used by the consumer application. The KCL used by the consumer uses the application name as the table name. Use `StockTradesProcessor`, which is the application name used later.
3. In the IAM console, in **Policies** (https://console.aws.amazon.com/iam/home#policies), choose **Create policy**. If this is the first time that you have worked with IAM policies, choose **Get Started**, **Create Policy**.
4. Choose **Select** next to **Policy Generator**.
5. Choose **Amazon Kinesis** as the AWS service.
6. Select `DescribeStream`, `GetShardIterator`, `GetRecords`, `PutRecord`, and `PutRecords` as the allowed actions.
7. Enter the ARN that you created in Step 1.
8. Use **Add Statement** for each of the following:

AWS Service	Actions	ARN
Amazon DynamoDB	`CreateTable`, `DeleteItem`, `DescribeTable`, `GetItem`, `PutItem`, `Scan`, `UpdateItem`	The ARN you created in Step 2
Amazon CloudWatch	`PutMetricData`	*

The asterisk (*) that is used when specifying an ARN is not required. In this case, it's because there is no specific resource in CloudWatch on which the `PutMetricData` action is invoked.

9. Choose **Next Step**.
10. Change **Policy Name** to `StockTradeStreamPolicy`, review the code, and choose **Create Policy**.

The resulting policy document should look something like the following:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt123",
      "Effect": "Allow",
      "Action": [
        "kinesis:DescribeStream",
        "kinesis:PutRecord",
        "kinesis:PutRecords",
        "kinesis:GetShardIterator",
        "kinesis:GetRecords",
        "kinesis:ListShards",
        "kinesis:DescribeStreamSummary",
        "kinesis:RegisterStreamConsumer"
      ],
      "Resource": [
        "arn:aws:kinesis:us-west-2:123:stream/StockTradeStream"
      ]
    },
    {
      "Sid": "Stmt234",
      "Effect": "Allow",
      "Action": [
        "kinesis:SubscribeToShard",
        "kinesis:DescribeStreamConsumer"
      ],
      "Resource": [
        "arn:aws:kinesis:us-west-2:123:stream/StockTradeStream/*"
      ]
    },
    {
      "Sid": "Stmt456",
      "Effect": "Allow",
      "Action": [
        "dynamodb:*"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-west-2:123:table/StockTradesProcessor"
      ]
    },
    {
      "Sid": "Stmt789",
      "Effect": "Allow",
```

```
      "Action": [
        "cloudwatch:PutMetricData"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```

To create an IAM user

1. Open the IAM console at https://console.aws.amazon.com/iam/.
2. On the **Users** page, choose **Add user**.
3. For **User name**, type `StockTradeStreamUser`.
4. For **Access type**, choose **Programmatic access**, and then choose **Next: Permissions**.
5. Choose **Attach existing policies directly**.
6. Search by name for the policy that you created. Select the box to the left of the policy name, and then choose **Next: Review**.
7. Review the details and summary, and then choose **Create user**.
8. Copy the **Access key ID**, and save it privately. Under **Secret access key**, choose **Show**, and save that key privately also.
9. Paste the access and secret keys to a local file in a safe place that only you can access. For this application, create a file named `~/.aws/credentials` (with strict permissions). The file should be in the following format:

```
[default]
aws_access_key_id=access key
aws_secret_access_key=secret access key
```

To attach an IAM policy to a user

1. In the IAM console, open Policies and choose **Policy Actions**.
2. Choose `StockTradeStreamPolicy` and **Attach**.
3. Choose `StockTradeStreamUser` and **Attach Policy**.

Next Steps

Step 3: Download and Build the Implementation Code (p. 40)

Step 3: Download and Build the Implementation Code

Skeleton code is provided for the the section called "Tutorial: Process Real-Time Stock Data Using KPL and KCL 1.x" (p. 35). It contains a stub implementation for both the stock trade stream ingestion (*producer*) and the processing of the data (*consumer*). The following procedure shows how to complete the implementations.

To download and build the implementation code

1. Download the source code to your computer.

2. Create a project in your favorite IDE with the source code, adhering to the provided directory structure.
3. Add the following libraries to the project:

   - Amazon Kinesis Client Library (KCL)
   - AWS SDK
   - Apache HttpCore
   - Apache HttpClient
   - Apache Commons Lang
   - Apache Commons Logging
   - Guava (Google Core Libraries For Java)
   - Jackson Annotations
   - Jackson Core
   - Jackson Databind
   - Jackson Dataformat: CBOR
   - Joda Time
4. Depending on your IDE, the project might be built automatically. If not, build the project using the appropriate steps for your IDE.

If you complete these steps successfully, you are now ready to move to the next section, the section called "Step 4: Implement the Producer" (p. 41). If your build generates errors at any stage, investigate and fix them before proceeding.

Next Steps

(p. 41)

Step 4: Implement the Producer

The application in the Tutorial: Process Real-Time Stock Data Using KPL and KCL 1.x (p. 35) uses the real-world scenario of stock market trade monitoring. The following principles briefly explain how this scenario maps to the producer and supporting code structure.

Refer to the source code and review the following information.

StockTrade class

An individual stock trade is represented by an instance of the `StockTrade` class. This instance contains attributes such as the ticker symbol, price, number of shares, the type of the trade (buy or sell), and an ID uniquely identifying the trade. This class is implemented for you.

Stream record

A stream is a sequence of records. A record is a serialization of a `StockTrade` instance in JSON format. For example:

```
{
  "tickerSymbol": "AMZN",
  "tradeType": "BUY",
  "price": 395.87,
  "quantity": 16,
  "id": 3567129045
}
```

StockTradeGenerator class

`StockTradeGenerator` has a method called `getRandomTrade()` that returns a new randomly generated stock trade every time it is invoked. This class is implemented for you.

StockTradesWriter class

The `main` method of the producer, `StockTradesWriter` continuously retrieves a random trade and then sends it to Kinesis Data Streams by performing the following tasks:

1. Reads the stream name and Region name as input.
2. Creates an `AmazonKinesisClientBuilder`.
3. Uses the client builder to set the Region, credentials, and client configuration.
4. Builds an `AmazonKinesis` client using the client builder.
5. Checks that the stream exists and is active (if not, it exits with an error).
6. In a continuous loop, calls the `StockTradeGenerator.getRandomTrade()` method and then the `sendStockTrade` method to send the trade to the stream every 100 milliseconds.

The `sendStockTrade` method of the `StockTradesWriter` class has the following code:

```
private static void sendStockTrade(StockTrade trade, AmazonKinesis kinesisClient,
 String streamName) {
    byte[] bytes = trade.toJsonAsBytes();
    // The bytes could be null if there is an issue with the JSON serialization by the
 Jackson JSON library.
    if (bytes == null) {
        LOG.warn("Could not get JSON bytes for stock trade");
        return;
    }

    LOG.info("Putting trade: " + trade.toString());
    PutRecordRequest putRecord = new PutRecordRequest();
    putRecord.setStreamName(streamName);
    // We use the ticker symbol as the partition key, explained in the Supplemental
 Information section below.
    putRecord.setPartitionKey(trade.getTickerSymbol());
    putRecord.setData(ByteBuffer.wrap(bytes));

    try {
        kinesisClient.putRecord(putRecord);
    } catch (AmazonClientException ex) {
        LOG.warn("Error sending record to Amazon Kinesis.", ex);
    }
}
```

Refer to the following code breakdown:

- The `PutRecord` API expects a byte array, and you need to convert `trade` to JSON format. This single line of code performs that operation:

  ```
  byte[] bytes = trade.toJsonAsBytes();
  ```

- Before you can send the trade, you create a new `PutRecordRequest` instance (called `putRecord` in this case):

  ```
  PutRecordRequest putRecord = new PutRecordRequest();
  ```

  Each `PutRecord` call requires the stream name, partition key, and data blob. The following code populates these fields in the `putRecord` object using its `setXxxx()` methods:

  ```
  putRecord.setStreamName(streamName);
  ```

```
putRecord.setPartitionKey(trade.getTickerSymbol());
putRecord.setData(ByteBuffer.wrap(bytes));
```

The example uses a stock ticket as a partition key, which maps the record to a specific shard. In practice, you should have hundreds or thousands of partition keys per shard such that records are evenly dispersed across your stream. For more information about how to add data to a stream, see Adding Data to a Stream (p. 96).

Now `putRecord` is ready to send to the client (the `put` operation):

```
kinesisClient.putRecord(putRecord);
```

- Error checking and logging are always useful additions. This code logs error conditions:

```
if (bytes == null) {
    LOG.warn("Could not get JSON bytes for stock trade");
    return;
}
```

Add the try/catch block around the `put` operation:

```
try {
       kinesisClient.putRecord(putRecord);
} catch (AmazonClientException ex) {
       LOG.warn("Error sending record to Amazon Kinesis.", ex);
}
```

This is because a Kinesis Data Streams `put` operation can fail because of a network error, or due to the stream reaching its throughput limits and getting throttled. We recommend carefully considering your retry policy for `put` operations to avoid data loss, such using as a simple retry.

- Status logging is helpful but optional:

```
LOG.info("Putting trade: " + trade.toString());
```

The producer shown here uses the Kinesis Data Streams API single record functionality, `PutRecord`. In practice, if an individual producer generates many records, it is often more efficient to use the multiple records functionality of `PutRecords` and send batches of records at a time. For more information, see Adding Data to a Stream (p. 96).

To run the producer

1. Verify that the access key and secret key pair retrieved earlier (when creating the IAM user) are saved in the file `~/.aws/credentials`.
2. Run the `StockTradeWriter` class with the following arguments:

```
StockTradeStream us-west-2
```

If you created your stream in a Region other than `us-west-2`, you have to specify that Region here instead.

You should see output similar to the following:

```
Feb 16, 2015 3:53:00 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter sendStockTrade
```

```
INFO: Putting trade: ID 8: SELL 996 shares of BUD for $124.18
Feb 16, 2015 3:53:00 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter sendStockTrade
INFO: Putting trade: ID 9: BUY 159 shares of GE for $20.85
Feb 16, 2015 3:53:01 PM
com.amazonaws.services.kinesis.samples.stocktrades.writer.StockTradesWriter sendStockTrade
INFO: Putting trade: ID 10: BUY 322 shares of WMT for $90.08
```

Your stock trade stream is now being ingested by Kinesis Data Streams.

Next Steps

Step 5: Implement the Consumer (p. 44)

Step 5: Implement the Consumer

The consumer application in the Tutorial: Process Real-Time Stock Data Using KPL and KCL 1.x (p. 35) continuously processes the stock trades stream that you created in (p. 41). It then outputs the most popular stocks being bought and sold every minute. The application is built on top of the Kinesis Client Library (KCL), which does much of the heavy lifting common to consumer apps. For more information, see Developing KCL 1.x Consumers (p. 125).

Refer to the source code and review the following information.

StockTradesProcessor class

Main class of the consumer, provided for you, which performs the following tasks:

- Reads the application, stream, and Region names, passed in as arguments.
- Reads credentials from `~/.aws/credentials`.
- Creates a `RecordProcessorFactory` instance that serves instances of `RecordProcessor`, implemented by a `StockTradeRecordProcessor` instance.
- Creates a KCL worker with the `RecordProcessorFactory` instance and a standard configuration including the stream name, credentials, and application name.
- The worker creates a new thread for each shard (assigned to this consumer instance), which continuously loops to read records from Kinesis Data Streams. It then invokes the `RecordProcessor` instance to process each batch of records received.

StockTradeRecordProcessor class

Implementation of the `RecordProcessor` instance, which in turn implements three required methods: `initialize`, `processRecords`, and `shutdown`.

As the names suggest, `initialize` and `shutdown` are used by the Kinesis Client Library to let the record processor know when it should be ready to start receiving records and when it should expect to stop receiving records, respectively, so it can do any application-specific setup and termination tasks. The code for these is provided for you. The main processing happens in the `processRecords` method, which in turn uses `processRecord` for each record. This latter method is provided as mostly empty skeleton code for you to implement in the next step, where it is explained further.

Also of note is the implementation of support methods for `processRecord`: `reportStats`, and `resetStats`, which are empty in the original source code.

The `processRecords` method is implemented for you, and performs the following steps:

- For each record passed in, calls `processRecord` on it.
- If at least 1 minute has elapsed since the last report, calls `reportStats()`, which prints out the latest stats, and then `resetStats()` which clears the stats so that the next interval includes only new records.

- Sets the next reporting time.
- If at least 1 minute has elapsed since the last checkpoint, calls `checkpoint()`.
- Sets the next checkpointing time.

This method uses 60-second intervals for the reporting and checkpointing rate. For more information about checkpointing, see Additional Information About the Consumer (p. 46).

StockStats class

This class provides data retention and statistics tracking for the most popular stocks over time. This code is provided for you and contains the following methods:

- `addStockTrade(StockTrade)`: Injects the given `StockTrade` into the running statistics.
- `toString()`: Returns the statistics in a formatted string.

This class keeps track of the most popular stock by keeping a running count of the total number of trades for each stock and the maximum count. It updates these counts whenever a stock trade arrives.

Add code to the methods of the `StockTradeRecordProcessor` class, as shown in the following steps.

To implement the consumer

1. Implement the `processRecord` method by instantiating a correctly sized `StockTrade` object and adding the record data to it, logging a warning if there's a problem.

```
StockTrade trade = StockTrade.fromJsonAsBytes(record.getData().array());
if (trade == null) {
    LOG.warn("Skipping record. Unable to parse record into StockTrade. Partition Key: "
 + record.getPartitionKey());
    return;
}
stockStats.addStockTrade(trade);
```

2. Implement a simple `reportStats` method. Feel free to modify the output format to your preferences.

```
System.out.println("****** Shard " + kinesisShardId + " stats for last 1 minute ******
\n" +
                   stockStats + "\n" +
                   "****************************************************************
\n");
```

3. Finally, implement the `resetStats` method, which creates a new `stockStats` instance.

```
stockStats = new StockStats();
```

To run the consumer

1. Run the producer that you wrote in (p. 41) to inject simulated stock trade records into your stream.
2. Verify that the access key and secret key pair retrieved earlier (when creating the IAM user) are saved in the file `~/.aws/credentials` .
3. Run the `StockTradesProcessor` class with the following arguments:

```
StockTradesProcessor StockTradeStream us-west-2
```

Note that if you created your stream in a Region other than `us-west-2`, you have to specify that Region here instead.

After a minute, you should see output like the following, refreshed every minute thereafter:

```
****** Shard shardId-000000000001 stats for last 1 minute ******
Most popular stock being bought: WMT, 27 buys.
Most popular stock being sold: PTR, 14 sells.
****************************************************************
```

Additional Information About the Consumer

If you are familiar with the advantages of the Kinesis Client Library, discussed in Developing KCL 1.x Consumers (p. 125) and elsewhere, you might wonder why you should use it here. Although you use only a single shard stream and a single consumer instance to process it, it is still easier to implement the consumer using the KCL. Compare the code implementation steps in the producer section to the consumer, and you can see the comparative ease of implementing a consumer. This is largely due to the services that the KCL provides.

In this application, you focus on implementing a record processor class that can process individual records. You don't have to worry about how the records are fetched from Kinesis Data Streams; The KCL fetches the records and invoke the record processor whenever there are new records available. Also, you don't have to worry about how many shards and consumer instances there are. If the stream is scaled up, you don't have to rewrite your application to handle more than one shard or one consumer instance.

The term *checkpointing* means recording the point in the stream up to the data records that have been consumed and processed thus far, so that if the application crashes, the stream is read from that point and not from the beginning of the stream. The subject of checkpointing and the various design patterns and best practices for it are outside the scope of this chapter. However, it is something you may encounter in production environments.

As you learned in (p. 41), the `put` operations in the Kinesis Data Streams API take a *partition key* as input. Kinesis Data Streams uses a partition key as a mechanism to split records across multiple shards (when there is more than one shard in the stream). The same partition key always routes to the same shard. This allows the consumer that processes a particular shard to be designed with the assumption that records with the same partition key are only sent to that consumer, and no records with the same partition key end up at any other consumer. Therefore, a consumer's worker can aggregate all records with the same partition key without worrying that it might be missing needed data.

In this application, the consumer's processing of records is not intensive, so you can use one shard and do the processing in the same thread as the KCL thread. However, in practice, consider first scaling up the number of shards. In some cases you may want to switch processing to a different thread, or use a thread pool if your record processing is expected to be intensive. In this way, the KCL can fetch new records more quickly while the other threads can process the records in parallel. Multithreaded design is not trivial and should be approached with advanced techniques, so increasing your shard count is usually the most effective and easiest way to scale up.

Next Steps

Step 6: (Optional) Extending the Consumer (p. 46)

Step 6: (Optional) Extending the Consumer

The application in the Tutorial: Process Real-Time Stock Data Using KPL and KCL 1.x (p. 35) might already be sufficient for your purposes. This optional section shows how you can extend the consumer code for a slightly more elaborate scenario.

If you want to know about the biggest sell orders each minute, you can modify the `StockStats` class in three places to accommodate this new priority.

To extend the consumer

1. Add new instance variables:

```
// Ticker symbol of the stock that had the largest quantity of shares sold
private String largestSellOrderStock;
// Quantity of shares for the largest sell order trade
private long largestSellOrderQuantity;
```

2. Add the following code to `addStockTrade`:

```
if (type == TradeType.SELL) {
    if (largestSellOrderStock == null || trade.getQuantity() >
largestSellOrderQuantity) {
        largestSellOrderStock = trade.getTickerSymbol();
        largestSellOrderQuantity = trade.getQuantity();
    }
}
```

3. Modify the `toString` method to print the additional information:

```
public String toString() {
    return String.format(
            "Most popular stock being bought: %s, %d buys.%n" +
            "Most popular stock being sold: %s, %d sells.%n" +
            "Largest sell order: %d shares of %s.",
            getMostPopularStock(TradeType.BUY),
getMostPopularStockCount(TradeType.BUY),
            getMostPopularStock(TradeType.SELL),
getMostPopularStockCount(TradeType.SELL),
            largestSellOrderQuantity, largestSellOrderStock);
}
```

If you run the consumer now (remember to run the producer also), you should see output similar to this:

```
****** Shard shardId-000000000001 stats for last 1 minute ******
 Most popular stock being bought: WMT, 27 buys.
 Most popular stock being sold: PTR, 14 sells.
 Largest sell order: 996 shares of BUD.
 ************************************************************
```

Next Steps

Step 7: Finishing Up (p. 47)

Step 7: Finishing Up

Because you are paying to use the Kinesis data stream, make sure that you delete it and the corresponding Amazon DynamoDB table when you are done with it. Nominal charges occur on an active stream even when you aren't sending and getting records. This is because an active stream is using resources by continuously "listening" for incoming records and requests to get records.

To delete the stream and table

1. Shut down any producers and consumers that you may still have running.

2. Open the Kinesis console at https://console.aws.amazon.com/kinesis.
3. Choose the stream that you created for this application (`StockTradeStream`).
4. Choose **Delete Stream**.
5. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/.
6. Delete the `StockTradesProcessor` table.

Summary

Processing a large amount of data in near-real time doesn't require writing any magical code or developing a huge infrastructure. It is as simple as writing logic to process a small amount of data (like writing `processRecord(Record)`) but using Kinesis Data Streams to scale so that it works for a large amount of streaming data. You don't have to worry about how your processing would scale because Kinesis Data Streams handles it for you. All you have to do is send your streaming records to Kinesis Data Streams and write the logic to process each new record received.

Here are some potential enhancements for this application.

Aggregate across all shards

Currently, you get stats resulting from aggregation of the data records received by a single worker from a single shard. (A shard cannot be processed by more than one worker in a single application at the same time.) Of course, when you scale and have more than one shard, you might want to aggregate across all shards. You can do this by having a pipeline architecture where the output of each worker is fed into another stream with a single shard, which is processed by a worker that aggregates the outputs of the first stage. Because the data from the first stage is limited (one sample per minute per shard), it would easily be handled by one shard.

Scale processing

When the stream scales up to have many shards (because many producers are sending data), the way to scale the processing is to add more workers. You can run the workers in Amazon EC2 instances and use Auto Scaling groups.

Use connectors to Amazon S3/DynamoDB/Amazon Redshift/Storm

As a stream is continuously processed, its output can be sent to other destinations. AWS provides connectors to integrate Kinesis Data Streams with other AWS services and third-party tools.

Next Steps

- For more information about using Kinesis Data Streams API operations, see Developing Producers Using the Amazon Kinesis Data Streams API with the AWS SDK for Java (p. 95), Developing Custom Consumers with Shared Throughput Using the AWS SDK for Java (p. 149), and Creating and Managing Streams (p. 65).
- For more information about the Kinesis Client Library, see Developing KCL 1.x Consumers (p. 125).
- For more information about how to optimize your application, see Advanced Topics (p. 174).

Tutorial: Analyze Real-Time Stock Data Using Kinesis Data Analytics for Flink Applications

The scenario for this tutorial involves ingesting stock trades into a data stream and writing a simple Amazon Kinesis Data Analytics application that performs calculations on the stream. You will learn how

to send a stream of records to Kinesis Data Streams and implement an application that consumes and processes the records in near-real time.

With Amazon Kinesis Data Analytics for Flink Applications, you can use Java or Scala to process and analyze streaming data. The service enables you to author and run Java or Scala code against streaming sources to perform time-series analytics, feed real-time dashboards, and create real-time metrics.

You can build Flink applications in Kinesis Data Analytics using open-source libraries based on Apache Flink. Apache Flink is a popular framework and engine for processing data streams.

> **Important**
> After you create two data streams and an application, your account incurs nominal charges for Kinesis Data Streams and Kinesis Data Analytics usage because they are not eligible for the AWS Free Tier. When you are finished with this application, delete your AWS resources to stop incurring charges.

The code does not access actual stock market data, but instead simulates the stream of stock trades. It does so by using a random stock trade generator. If you have access to a real-time stream of stock trades, you might be interested in deriving useful, timely statistics from that stream. For example, you might want to perform a sliding window analysis where you determine the most popular stock purchased in the last 5 minutes. Or you might want a notification whenever there is a sell order that is too large (that is, it has too many shares). You can extend the code in this series to provide such functionality.

The examples shown use the US West (Oregon) Region, but they work on any of the AWS Regions that support Kinesis Data Analytics.

Tasks

Prerequisites for Completing the Exercises

To complete the steps in this guide, you must have the following:

- Java Development Kit (JDK) version 8. Set the `JAVA_HOME` environment variable to point to your JDK install location.
- We recommend that you use a development environment (such as Eclipse Java Neon or IntelliJ Idea) to develop and compile your application.
- Git Client. Install the Git client if you haven't already.
- Apache Maven Compiler Plugin. Maven must be in your working path. To test your Apache Maven installation, enter the following:

```
$ mvn -version
```

To get started, go to Step 1: Set Up an AWS Account and Create an Administrator User (p. 49).

Step 1: Set Up an AWS Account and Create an Administrator User

Before you use Amazon Kinesis Data Analytics for Flink Applications for the first time, complete the following tasks:

1. Sign Up for AWS (p. 50)
2. Create an IAM User (p. 50)

Sign Up for AWS

When you sign up for Amazon Web Services (AWS), your AWS account is automatically signed up for all services in AWS, including Amazon Kinesis Data Analytics. You are charged only for the services that you use.

With Kinesis Data Analytics, you pay only for the resources that you use. If you are a new AWS customer, you can get started with Kinesis Data Analytics for free. For more information, see AWS Free Tier.

If you already have an AWS account, skip to the next task. If you don't have an AWS account, follow these steps to create one.

To create an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

Note your AWS account ID because you'll need it for the next task.

Create an IAM User

Services in AWS, such as Amazon Kinesis Data Analytics, require that you provide credentials when you access them. This is so that the service can determine whether you have permissions to access the resources that are owned by that service. The AWS Management Console requires that you enter your password.

You can create access keys for your AWS account to access the AWS Command Line Interface (AWS CLI) or API. However, we don't recommend that you access AWS using the credentials for your AWS account. Instead, we recommend that you use AWS Identity and Access Management (IAM). Create an IAM user, add the user to an IAM group with administrative permissions, and then grant administrative permissions to the IAM user that you created. You can then access AWS using a special URL and that IAM user's credentials.

If you signed up for AWS, but you haven't created an IAM user for yourself, you can create one using the IAM console.

The getting started exercises in this guide assume that you have a user (`adminuser`) with administrator permissions. Follow the procedure to create `adminuser` in your account.

To create a group for administrators

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane, choose **Groups**, and then choose **Create New Group**.
3. For **Group Name**, enter a name for your group, such as **`Administrators`**, and then choose **Next Step**.
4. In the list of policies, select the check box next to the **AdministratorAccess** policy. You can use the **Filter** menu and the **Search** box to filter the list of policies.

5. Choose **Next Step**, and then choose **Create Group**.

Your new group is listed under **Group Name**.

To create an IAM user for yourself, add it to the Administrators group, and create a password

1. In the navigation pane, choose **Users**, and then choose **Add user**.
2. In the **User name** box, enter a user name.
3. Choose both **Programmatic access** and **AWS Management Console access**.
4. Choose **Next: Permissions**.
5. Select the check box next to the **Administrators** group. Then choose **Next: Review**.
6. Choose **Create user**.

To sign in as the new IAM user

1. Sign out of the AWS Management Console.
2. Use the following URL format to sign in to the console:

   `https://aws_account_number.signin.aws.amazon.com/console/`

   The *aws_account_number* is your AWS account ID without any hyphens. For example, if your AWS account ID is 1234-5678-9012, replace *aws_account_number* with **123456789012**. For information about how to find your account number, see Your AWS Account ID and Its Alias in the *IAM User Guide*.
3. Enter the IAM user name and password that you just created. When you're signed in, the navigation bar displays *your_user_name* @ *your_aws_account_id*.

   Note
   If you don't want the URL for your sign-in page to contain your AWS account ID, you can create an account alias.

To create or remove an account alias

1. Open the IAM console at https://console.aws.amazon.com/iam/.
2. On the navigation pane, choose **Dashboard**.
3. Find the IAM users sign-in link.
4. To create the alias, choose **Customize**. Enter the name you want to use for your alias, and then choose **Yes, Create**.
5. To remove the alias, choose **Customize**, and then choose **Yes, Delete**. The sign-in URL reverts to using your AWS account ID.

To sign in after you create an account alias, use the following URL:

`https://your_account_alias.signin.aws.amazon.com/console/`

To verify the sign-in link for IAM users for your account, open the IAM console and check under **IAM users sign-in link** on the dashboard.

For more information about IAM, see the following:

- AWS Identity and Access Management (IAM)
- Getting started

- IAM User Guide

Next Step

Step 2: Set Up the AWS Command Line Interface (AWS CLI) (p. 52)

Step 2: Set Up the AWS Command Line Interface (AWS CLI)

In this step, you download and configure the AWS CLI to use with Amazon Kinesis Data Analytics for Flink Applications.

Note
The getting started exercises in this guide assume that you are using administrator credentials (`adminuser`) in your account to perform the operations.

Note
If you already have the AWS CLI installed, you might need to upgrade to get the latest functionality. For more information, see Installing the AWS Command Line Interface in the *AWS Command Line Interface User Guide*. To check the version of the AWS CLI, run the following command:

```
aws --version
```

The exercises in this tutorial require the following AWS CLI version or later:

```
aws-cli/1.16.63
```

To set up the AWS CLI

1. Download and configure the AWS CLI. For instructions, see the following topics in the *AWS Command Line Interface User Guide*:
   - Installing the AWS Command Line Interface
   - Configuring the AWS CLI
2. Add a named profile for the administrator user in the AWS CLI config file. You use this profile when executing the AWS CLI commands. For more information about named profiles, see Named Profiles in the *AWS Command Line Interface User Guide*.

   ```
   [profile adminuser]
   aws_access_key_id = adminuser access key ID
   aws_secret_access_key = adminuser secret access key
   region = aws-region
   ```

   For a list of available AWS Regions, see AWS Regions and Endpoints in the *Amazon Web Services General Reference*.
3. Verify the setup by entering the following help command at the command prompt:

   ```
   aws help
   ```

After you set up an AWS account and the AWS CLI, you can try the next exercise, in which you configure a sample application and test the end-to-end setup.

Next Step

Step 3: Create and Run a Kinesis Data Analytics for Flink Application (p. 53)

Step 3: Create and Run a Kinesis Data Analytics for Flink Application

In this exercise, you create a Kinesis Data Analytics for Flink application with data streams as a source and a sink.

This section contains the following steps:

Create Two Amazon Kinesis Data Streams

Before you create a Kinesis Data Analytics for Flink application for this exercise, create two Kinesis data streams (`ExampleInputStream` and `ExampleOutputStream`). Your application uses these streams for the application source and destination streams.

You can create these streams using either the Amazon Kinesis console or the following AWS CLI command. For console instructions, see Creating and Updating Data Streams.

To create the data streams (AWS CLI)

1. To create the first stream (`ExampleInputStream`), use the following Amazon Kinesis `create-stream` AWS CLI command.

```
$ aws kinesis create-stream \
--stream-name ExampleInputStream \
--shard-count 1 \
--region us-west-2 \
--profile adminuser
```

2. To create the second stream that the application uses to write output, run the same command, changing the stream name to `ExampleOutputStream`.

```
$ aws kinesis create-stream \
--stream-name ExampleOutputStream \
--shard-count 1 \
--region us-west-2 \
--profile adminuser
```

Write Sample Records to the Input Stream

In this section, you use a Python script to write sample records to the stream for the application to process.

Note
This section requires the AWS SDK for Python (Boto).

1. Create a file named `stock.py` with the following contents:

```
import datetime
import json
import random
import boto3

STREAM_NAME = "ExampleInputStream"


def get_data():
    return {
        'EVENT_TIME': datetime.datetime.now().isoformat(),
        'TICKER': random.choice(['AAPL', 'AMZN', 'MSFT', 'INTC', 'TBV']),
        'PRICE': round(random.random() * 100, 2)}


def generate(stream_name, kinesis_client):
    while True:
        data = get_data()
        print(data)
        kinesis_client.put_record(
            StreamName=stream_name,
            Data=json.dumps(data),
            PartitionKey="partitionkey")


if __name__ == '__main__':
    generate(STREAM_NAME, boto3.client('kinesis'))
```

2. Later in the tutorial, you run the `stock.py` script to send data to the application.

```
$ python stock.py
```

Download and Examine the Apache Flink Streaming Java Code

The Java application code for this examples is available from GitHub. To download the application code, do the following:

1. Clone the remote repository with the following command:

```
git clone https://github.com/aws-samples/amazon-kinesis-data-analytics-java-
examples.git
```

2. Navigate to the `GettingStarted` directory.

The application code is located in the `CustomSinkStreamingJob.java` and `CloudWatchLogSink.java` files. Note the following about the application code:

- The application uses a Kinesis source to read from the source stream. The following snippet creates the Kinesis sink:

```
return env.addSource(new FlinkKinesisConsumer<>(inputStreamName,
                new SimpleStringSchema(), inputProperties));
```

Compile the Application Code

In this section, you use the Apache Maven compiler to create the Java code for the application. For information about installing Apache Maven and the Java Development Kit (JDK), see Prerequisites for Completing the Exercises (p. 49).

Your Java application requires the following components:

- A Project Object Model (pom.xml) file. This file contains information about the application's configuration and dependencies, including the Kinesis Data Analytics for Flink Applications libraries.
- A `main` method that contains the application's logic.

Note
In order to use the Kinesis connector for the following application, you need to download the source code for the connector and build it as described in the Apache Flink documentation.

To create and compile the application code

1. Create a Java/Maven application in your development environment. For information about creating an application, see the documentation for your development environment:
   - Creating your first Java project (Eclipse Java Neon)
   - Creating, Running and Packaging Your First Java Application (IntelliJ Idea)
2. Use the following code for a file named `StreamingJob.java`.

```
package com.amazonaws.services.kinesisanalytics;

import com.amazonaws.services.kinesisanalytics.runtime.KinesisAnalyticsRuntime;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kinesis.FlinkKinesisConsumer;
import org.apache.flink.streaming.connectors.kinesis.FlinkKinesisProducer;
import org.apache.flink.streaming.connectors.kinesis.config.ConsumerConfigConstants;

import java.io.IOException;
import java.util.Map;
import java.util.Properties;



public class StreamingJob {

    private static final String region = "us-east-1";
    private static final String inputStreamName = "ExampleInputStream";
    private static final String outputStreamName = "ExampleOutputStream";

    private static DataStream<String>
 createSourceFromStaticConfig(StreamExecutionEnvironment env) {
        Properties inputProperties = new Properties();
        inputProperties.setProperty(ConsumerConfigConstants.AWS_REGION, region);
        inputProperties.setProperty(ConsumerConfigConstants.STREAM_INITIAL_POSITION,
 "LATEST");

        return env.addSource(new FlinkKinesisConsumer<>(inputStreamName, new
 SimpleStringSchema(), inputProperties));
    }
```

```
   private static DataStream<String>
createSourceFromApplicationProperties(StreamExecutionEnvironment env) throws
IOException {
       Map<String, Properties> applicationProperties =
KinesisAnalyticsRuntime.getApplicationProperties();
       return env.addSource(new FlinkKinesisConsumer<>(inputStreamName, new
SimpleStringSchema(),
               applicationProperties.get("ConsumerConfigProperties")));
   }

   private static FlinkKinesisProducer<String> createSinkFromStaticConfig() {
       Properties outputProperties = new Properties();
       outputProperties.setProperty(ConsumerConfigConstants.AWS_REGION, region);
       outputProperties.setProperty("AggregationEnabled", "false");

       FlinkKinesisProducer<String> sink = new FlinkKinesisProducer<>(new
SimpleStringSchema(), outputProperties);
       sink.setDefaultStream(outputStreamName);
       sink.setDefaultPartition("0");
       return sink;
   }

   private static FlinkKinesisProducer<String> createSinkFromApplicationProperties()
throws IOException {
       Map<String, Properties> applicationProperties =
KinesisAnalyticsRuntime.getApplicationProperties();
       FlinkKinesisProducer<String> sink = new FlinkKinesisProducer<>(new
SimpleStringSchema(),
               applicationProperties.get("ProducerConfigProperties"));

       sink.setDefaultStream(outputStreamName);
       sink.setDefaultPartition("0");
       return sink;
   }

   public static void main(String[] args) throws Exception {
       // set up the streaming execution environment
       final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();


        /* if you would like to use runtime configuration properties, uncomment the
lines below
        * DataStream<String> input = createSourceFromApplicationProperties(env);
        */

       DataStream<String> input = createSourceFromStaticConfig(env);


        /* if you would like to use runtime configuration properties, uncomment the
lines below
        * input.addSink(createSinkFromApplicationProperties())
        */

       input.addSink(createSinkFromStaticConfig());

       env.execute("Flink Streaming Java API Skeleton");
   }
}
```

Note the following about the preceding code example:

- This file contains the `main` method that defines the application's functionality.

- Your application creates source and sink connectors to access external resources using a `StreamExecutionEnvironment` object.
- The application creates source and sink connectors using static properties. To use dynamic application properties, use the `createSourceFromApplicationProperties` and `createSinkFromApplicationProperties` methods to create the connectors. These methods read the application's properties to configure the connectors.

3. To use your application code, you compile and package it into a JAR file. You can compile and package your code in one of two ways:

   - Use the command line Maven tool. Create your JAR file by running the following command in the directory that contains the `pom.xml` file:

     ```
     mvn package
     ```

   - Use your development environment. See your development environment documentation for details.

   You can either upload your package as a JAR file, or you can compress your package and upload it as a ZIP file. If you create your application using the AWS CLI, you specify your code content type (JAR or ZIP).
4. If there are errors while compiling, verify that your `JAVA_HOME` environment variable is correctly set.

If the application compiles successfully, the following file is created:

`target/java-getting-started-1.0.jar`

Upload the Apache Flink Streaming Java Code

In this section, you create an Amazon Simple Storage Service (Amazon S3) bucket and upload your application code.

To upload the application code

1. Open the Amazon S3 console at https://console.aws.amazon.com/s3/.
2. Choose **Create bucket**.
3. Enter **ka-app-code-*<username>*** in the **Bucket name** field. Add a suffix to the bucket name, such as your user name, to make it globally unique. Choose **Next**.
4. In the **Configure options** step, keep the settings as they are, and choose **Next**.
5. In the **Set permissions** step, keep the settings as they are, and choose **Next**.
6. Choose **Create bucket**.
7. In the Amazon S3 console, choose the **ka-app-code-*<username>*** bucket, and choose **Upload**.
8. In the **Select files** step, choose **Add files**. Navigate to the `java-getting-started-1.0.jar` file that you created in the previous step. Choose **Next**.
9. In the **Set permissions** step, keep the settings as they are. Choose **Next**.
10. In the **Set properties** step, keep the settings as they are. Choose **Upload**.

Your application code is now stored in an Amazon S3 bucket where your application can access it.

Create and Run the Kinesis Data Analytics Application

You can create and run a Kinesis Data Analytics for Flink application using either the console or the AWS CLI.

Note
When you create the application using the console, your AWS Identity and Access Management (IAM) and Amazon CloudWatch Logs resources are created for you. When you create the application using the AWS CLI, you create these resources separately.

Topics

Create and Run the Application (Console)

Follow these steps to create, configure, update, and run the application using the console.

Create the Application

1. Open the Kinesis console at https://console.aws.amazon.com/kinesis.
2. On the Amazon Kinesis dashboard, choose **Create analytics application**.
3. On the **Kinesis Analytics - Create application** page, provide the application details as follows:
   - For **Application name**, enter **`MyApplication`**.
   - For **Description**, enter **`My java test app`**.
   - For **Runtime**, choose **Apache Flink 1.6**.
4. For **Access permissions**, choose **Create / update IAM role `kinesis-analytics-MyApplication-us-west-2`**.
5. Choose **Create application**.

Note
When you create a Kinesis Data Analytics for Flink application using the console, you have the option of having an IAM role and policy created for your application. Your application uses this role and policy to access its dependent resources. These IAM resources are named using your application name and Region as follows:

- Policy: `kinesis-analytics-service-`*`MyApplication`*`-`*`us-west-2`*
- Role: `kinesis-analytics-`*`MyApplication`*`-`*`us-west-2`*

Edit the IAM Policy

Edit the IAM policy to add permissions to access the Kinesis data streams.

1. Open the IAM console at https://console.aws.amazon.com/iam/.
2. Choose **Policies**. Choose the **`kinesis-analytics-service-MyApplication-us-west-2`** policy that the console created for you in the previous section.
3. On the **Summary** page, choose **Edit policy**. Choose the **JSON** tab.
4. Add the highlighted section of the following policy example to the policy. Replace the sample account IDs (*`012345678901`*) with your account ID.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ReadCode",
            "Effect": "Allow",
            "Action": [
```

```
                "s3:GetObject",
                "s3:GetObjectVersion"
            ],
            "Resource": [
                "arn:aws:s3:::ka-app-code-username/java-getting-started-1.0.jar"
            ]
        },
        {
            "Sid": "ListCloudwatchLogGroups",
            "Effect": "Allow",
            "Action": [
                "logs:DescribeLogGroups"
            ],
            "Resource": [
                "arn:aws:logs:us-west-2:012345678901:log-group:*"
            ]
        },
        {
            "Sid": "ListCloudwatchLogStreams",
            "Effect": "Allow",
            "Action": [
                "logs:DescribeLogStreams"
            ],
            "Resource": [
                "arn:aws:logs:us-west-2:012345678901:log-group:/aws/kinesis-analytics/
MyApplication:log-stream:*"
            ]
        },
        {
            "Sid": "PutCloudwatchLogs",
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents"
            ],
            "Resource": [
                "arn:aws:logs:us-west-2:012345678901:log-group:/aws/kinesis-analytics/
MyApplication:log-stream:kinesis-analytics-log-stream"
            ]
        },
        {
            "Sid": "ReadInputStream",
            "Effect": "Allow",
            "Action": "kinesis:*",
            "Resource": "arn:aws:kinesis:us-west-2:012345678901:stream/
ExampleInputStream"
        },
        {
            "Sid": "WriteOutputStream",
            "Effect": "Allow",
            "Action": "kinesis:*",
            "Resource": "arn:aws:kinesis:us-west-2:012345678901:stream/
ExampleOutputStream"
        }
    ]
}
```

Configure the Application

1. On the **MyApplication** page, choose **Configure**.
2. On the **Configure application** page, provide the **Code location**:
   - For **Amazon S3 bucket**, enter **`ka-app-code-<username>`**.
   - For **Path to Amazon S3 object**, enter **`java-getting-started-1.0.jar`**.

3. Under **Access to application resources**, for **Access permissions**, choose **Create / update IAM role `kinesis-analytics-MyApplication-us-west-2`**.
4. Under **Properties**, for **Group ID**, enter **`ProducerConfigProperties`**.
5. Enter the following application properties and values:

Key	Value
`flink.inputstream.initpos`	**`LATEST`**
`aws:region`	**`us-west-2`**
`AggregationEnabled`	**`false`**

6. Under **Monitoring**, ensure that the **Monitoring metrics level** is set to **Application**.
7. For **CloudWatch logging**, select the **Enable** check box.
8. Choose **Update**.

Note
When you choose to enable CloudWatch logging, Kinesis Data Analytics creates a log group and log stream for you. The names of these resources are as follows:

- Log group: `/aws/kinesis-analytics/MyApplication`
- Log stream: `kinesis-analytics-log-stream`

Run the Application

1. On the **MyApplication** page, choose **Run**. Confirm the action.
2. When the application is running, refresh the page. The console shows the **Application graph**.

Stop the Application

On the **MyApplication** page, choose **Stop**. Confirm the action.

Update the Application

Using the console, you can update application settings such as application properties, monitoring settings, and the location or file name of the application JAR. You can also reload the application JAR from the Amazon S3 bucket if you need to update the application code.

On the **MyApplication** page, choose **Configure**. Update the application settings and choose **Update**.

Create and Run the Application (AWS CLI)

In this section, you use the AWS CLI to create and run the Kinesis Data Analytics application. Kinesis Data Analytics for Flink Applications uses the `kinesisanalyticsv2` AWS CLI command to create and interact with Kinesis Data Analytics applications.

Create a Permissions Policy

First, you create a permissions policy with two statements: one that grants permissions for the `read` action on the source stream, and another that grants permissions for `write` actions on the sink stream. You then attach the policy to an IAM role (which you create in the next section). Thus, when Kinesis Data Analytics assumes the role, the service has the necessary permissions to read from the source stream and write to the sink stream.

Use the following code to create the `KAReadSourceStreamWriteSinkStream` permissions policy. Replace *username* with the user name that you used to create the Amazon S3 bucket to store the application code. Replace the account ID in the Amazon Resource Names (ARNs) (*012345678901*) with your account ID.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "S3",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:GetObjectVersion"
            ],
            "Resource": ["arn:aws:s3:::ka-app-code-username",
                "arn:aws:s3:::ka-app-code-username/*"
            ]
        },
        {
            "Sid": "ReadInputStream",
            "Effect": "Allow",
            "Action": "kinesis:*",
            "Resource": "arn:aws:kinesis:us-west-2:012345678901:stream/ExampleInputStream"
        },
        {
            "Sid": "WriteOutputStream",
            "Effect": "Allow",
            "Action": "kinesis:*",
            "Resource": "arn:aws:kinesis:us-west-2:012345678901:stream/ExampleOutputStream"
        }
    ]
}
```

For step-by-step instructions to create a permissions policy, see Tutorial: Create and Attach Your First Customer Managed Policy in the *IAM User Guide*.

Note
To access other AWS services, you can use the AWS SDK for Java. Kinesis Data Analytics automatically sets the credentials required by the SDK to those of the service execution IAM role that is associated with your application. No additional steps are needed.

Create an IAM Role

In this section, you create an IAM role that the Kinesis Data Analytics for Flink application can assume to read a source stream and write to the sink stream.

Kinesis Data Analytics cannot access your stream without permissions. You grant these permissions via an IAM role. Each IAM role has two policies attached. The trust policy grants Kinesis Data Analytics permission to assume the role, and the permissions policy determines what Kinesis Data Analytics can do after assuming the role.

You attach the permissions policy that you created in the preceding section to this role.

To create an IAM role

1. Open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane, choose **Roles**, **Create Role**.
3. Under **Select type of trusted identity**, choose **AWS Service**. Under **Choose the service that will use this role**, choose **Kinesis**. Under **Select your use case**, choose **Kinesis Analytics**.

   Choose **Next: Permissions**.

4. On the **Attach permissions policies** page, choose **Next: Review**. You attach permissions policies after you create the role.
5. On the **Create role** page, enter **`KA-stream-rw-role`** for the **Role name**. Choose **Create role**.

   Now you have created a new IAM role called `KA-stream-rw-role`. Next, you update the trust and permissions policies for the role.
6. Attach the permissions policy to the role.

   Note
   For this exercise, Kinesis Data Analytics assumes this role for both reading data from a Kinesis data stream (source) and writing output to another Kinesis data stream. So you attach the policy that you created in the previous step, the section called "Create a Permissions Policy" (p. 60).

   a. On the **Summary** page, choose the **Permissions** tab.
   b. Choose **Attach Policies**.
   c. In the search box, enter **`KAReadSourceStreamWriteSinkStream`** (the policy that you created in the previous section).
   d. Choose the **KAReadInputStreamWriteOutputStream** policy, and choose **Attach policy**.

You now have created the service execution role that your application uses to access resources. Make a note of the ARN of the new role.

For step-by-step instructions for creating a role, see Creating an IAM Role (Console) in the *IAM User Guide*.

Create the Kinesis Data Analytics Application

1. Save the following JSON code to a file named `create_request.json`. Replace the sample role ARN with the ARN for the role that you created previously. Replace the bucket ARN suffix (*`username`*) with the suffix that you chose in the previous section. Replace the sample account ID (*`012345678901`*) in the service execution role with your account ID.

```
{
    "ApplicationName": "test",
    "ApplicationDescription": "my java test app",
    "RuntimeEnvironment": "FLINK-1_6",
    "ServiceExecutionRole": "arn:aws:iam::012345678901:role/KA-stream-rw-role",
    "ApplicationConfiguration": {
        "ApplicationCodeConfiguration": {
            "CodeContent": {
                "S3ContentLocation": {
                    "BucketARN": "arn:aws:s3:::ka-app-code-username",
                    "FileKey": "java-getting-started-1.0.jar"
                }
            },
            "CodeContentType": "ZIPFILE"
        },
        "EnvironmentProperties":  {
         "PropertyGroups": [
            {
               "PropertyGroupId": "ProducerConfigProperties",
               "PropertyMap" : {
                    "flink.stream.initpos" : "LATEST",
                    "aws.region" : "us-west-2",
                    "AggregationEnabled" : "false"
               }
            },
            {
               "PropertyGroupId": "ConsumerConfigProperties",
```

```
                "PropertyMap" : {
                     "aws.region" : "us-west-2"
                }
            }
        ]
    }
}
```

2. Execute the `CreateApplication` action with the preceding request to create the application:

```
aws kinesisanalyticsv2 create-application --cli-input-json file://create_request.json
```

The application is now created. You start the application in the next step.

Start the Application

In this section, you use the `StartApplication` action to start the application.

To start the application

1. Save the following JSON code to a file named `start_request.json`.

```
{
    "ApplicationName": "test",
    "RunConfiguration": {
        "ApplicationRestoreConfiguration": {
         "ApplicationRestoreType": "RESTORE_FROM_LATEST_SNAPSHOT"
         }
    }
}
```

2. Execute the `StartApplication` action with the preceding request to start the application:

```
aws kinesisanalyticsv2 start-application --cli-input-json file://start_request.json
```

The application is now running. You can check the Kinesis Data Analytics metrics on the Amazon CloudWatch console to verify that the application is working.

Stop the Application

In this section, you use the `StopApplication` action to stop the application.

To stop the application

1. Save the following JSON code to a file named `stop_request.json`.

```
{"ApplicationName": "test"
}
```

2. Execute the `StopApplication` action with the following request to stop the application:

```
aws kinesisanalyticsv2 stop-application --cli-input-json file://stop_request.json
```

The application is now stopped.

Tutorial: Using AWS Lambda with Amazon Kinesis Data Streams

In this tutorial, you create a Lambda function to consume events from a Kinesis data stream. In this example scenario, a custom application writes records to a Kinesis data stream. AWS Lambda then polls this data stream and, when it detects new data records, invokes your Lambda function. AWS Lambda then executes the Lambda function by assuming the execution role that you specified when you created the Lambda function.

For the detailed step by step instructions, see Tutorial: Using AWS Lambda with Amazon Kinesis.

> **Note**
> This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in Getting Started with AWS Lambda to create your first Lambda function.

AWS Streaming Data Solution for Amazon Kinesis

The AWS Streaming Data Solution for Amazon Kinesis automatically configures the AWS services necessary to easily capture, store, process, and deliver streaming data. The solution provides multiple options for solving streaming data use cases that use multiple AWS services including Kinesis Data Streams, AWS Lambda, Amazon API Gateway, and Amazon Kinesis Data Analytics.

Each solution includes the following components:

- A AWS CloudFormation package to deploy the complete example.
- A CloudWatch dashboard for displaying application metrics.
- CloudWatch alarms on the most relevant application metrics.
- All necessary IAM roles and policies.

The solution can be found here: Streaming Data Solution for Amazon Kinesis

Creating and Managing Streams

Amazon Kinesis Data Streams ingests a large amount of data in real time, durably stores the data, and makes the data available for consumption. The unit of data stored by Kinesis Data Streams is a *data record*. A *data stream* represents a group of data records. The data records in a data stream are distributed into shards.

A *shard* has a sequence of data records in a stream. It serves as a base throughput unit of a Kinesis data stream. A shard supports 1 MB/s and 1000 records per second for *writes* and 2 MB/s for *reads* in both on-demand and provisioned capacity modes. The shard limits ensure predictable performance, making it easier to design and operate a highly reliable data streaming workflow.

Topics

Choosing the Data Stream Capacity Mode

Topics

What is a Data Stream Capacity Mode?

A capacity mode determines how the capacity of a data stream is managed and how you are charged for the usage of your data stream. In Amazon Kinesis Data Streams, you can choose between an **on-demand** mode and a **provisioned** mode for your data streams.

- **On-demand** - data streams with an on-demand mode require no capacity planning and automatically scale to handle gigabytes of write and read throughput per minute. With the on-demand mode, Kinesis Data Streams automatically manages the shards in order to provide the necessary throughput.
- **Provisioned** - for the data streams with a provisioned mode, you must specify the number of shards for the data stream. The total capacity of a data stream is the sum of the capacities of its shards. You can increase or decrease the number of shards in a data stream as needed.

You can use Kinesis Data Streams `PutRecord` and `PutRecords` APIs to write data into your data streams in both on-demand and provisioned capacity modes. To retrieve data, both capacity modes

support default consumers that use the `GetRecords` API and Enhanced Fan-Out (EFO) consumers that use the `SubscribeToShard` API.

All Kinesis Data Streams capabilities, including retention mode, encryption, monitoring metrics, and others, are supported for both the on-demand and provisioned modes. Kinesis Data Streams provides the high durability and availability in both the on-demand and provisioned capacity modes.

On-demand Mode

Data streams in the on-demand mode require no capacity planning and automatically scale to handle gigabytes of write and read throughput per minute. On-demand mode simplifies ingesting and storing large data volumes at a low-latency because it eliminates provisioning and managing servers, storage, or throughput. You can ingest billions of records per day without any operational overhead.

On-demand mode is ideal for addressing the needs of highly variable and unpredictable application traffic. You no longer have to provision these workloads for peak capacity, which can result in higher costs due to low utilization. On-demand mode is suited for workloads with unpredictable and highly-variable traffic patterns.

With the on-demand capacity mode, you pay per GB of data written and read from your data streams. You do not need to specify how much read and write throughput you expect your application to perform. Kinesis Data Streams instantly accommodates your workloads as they ramp up or down. For more information, see Amazon Kinesis Data Streams pricing.

You can create a new data stream with the on-demand mode by using the Kinesis Data Streams console, APIs, or CLI commands.

A data stream in the on-demand mode accommodates up to double the peak write throughput observed in the previous 30 days. As your data stream's write throughput reaches a new peak, Kinesis Data Streams scales the data stream's capacity automatically. For example, if your data stream has a write throughput that varies between 10 MB/s and 40 MB/s, then Kinesis Data Streams ensures that you can easily burst to double your previous peak throughput, or 80 MB/s. If the same data stream sustains a new peak throughput of 50 MB/s, Kinesis Data Streams ensures that there is enough capacity to ingest 100 MB/s of write throughput. However, write throttling can occur if your traffic increases to more than double the previous peak within a 15-minute duration. You need to retry these throttled requests.

The aggregate read capacity of a data stream with the on-demand mode increases proportionally to write throughput. This helps to ensure that consumer applications always have adequate read throughput to process incoming data in real time. You get at least twice the write throughput compared to read data using the `GetRecords` API. We recommend that you use one consumer application with the `GetRecord` API, so that it has enough room to catch up when the application needs to recover from downtime. It is recommended that you use the Enhanced Fan-Out capability of Kinesis Data Streams for scenarios that require adding more than one consumer application. Enhanced Fan-Out supports adding up to 20 consumer applications to a data stream using the `SubscribeToShard` API, with each consumer application having dedicated throughput.

Handling Read and Write Throughput Exceptions

With the on-demand capacity mode (same as with the provisioned capacity mode), you must specify a partition key with each record to write data into your data stream. Kinesis Data Streams uses your partition keys to distribute data across shards. Kinesis Data Streams monitors traffic for each shard. When the incoming traffic exceeds 500 KB/s per shard, it splits the shard within 15 minutes. The parent shard's hash key values are redistributed evenly across child shards.

If your incoming traffic exceeds twice your prior peak, you can experience read or write exceptions for about 15 minutes, even when your data is distributed evenly across the shards. We recommend that you retry all such requests so that all the records are properly stored in Kinesis Data Streams.

You may experience read and write exceptions if you are using a partition key that leads to uneven data distribution, and the records assigned to a particular shard exceed its limits. With on-demand mode, the data stream automatically adapts to handle uneven data distribution patterns unless a single partition key exceeds a shard's 1 MB/s throughput and 1000 records per second limits.

In the on-demand mode, Kinesis Data Streams splits the shards evenly when it detects an increase in traffic. However, it does not detect and isolate hash keys that are driving a higher portion of incoming traffic to a particular shard. If you are using highly uneven partition keys you may continue to receive write exceptions. For such use cases, we recommend that you use the provisioned capacity mode that supports granular shard splits.

Provisioned Mode

With provisioned mode, after you create the data stream, you can dynamically scale your shard capacity up or down using the AWS Management Console or the UpdateShardCount API. You can make updates while there is a Kinesis Data Streams producer or consumer application writing to or reading data from the stream.

The provisioned mode is suited for predictable traffic with capacity requirements that are easy to forecast. You can use the provisioned mode if you want fine-grained control over how data is distributed across shards.

With the provisioned mode, you must specify the number of shards for the data stream. To determine the size of a data stream with the provisioned mode, you need the following input values:

- The average size of the data record written to the stream in kilobytes (KB), rounded up to the nearest 1 KB (`average_data_size_in_KB`).
- The number of data records written to and read from the stream per second (`records_per_second`).
- The number of consumers, which are Kinesis Data Streams applications that consume data concurrently and independently from the stream (`number_of_consumers`).
- The incoming write bandwidth in KB (`incoming_write_bandwidth_in_KB`), which is equal to the `average_data_size_in_KB` multiplied by the `records_per_second`.
- The outgoing read bandwidth in KB (`outgoing_read_bandwidth_in_KB`), which is equal to the `incoming_write_bandwidth_in_KB` multiplied by the `number_of_consumers`.

You can calculate the number of shards (`number_of_shards`) that your stream needs by using the input values in the following formula.

```
number_of_shards = max(incoming_write_bandwidth_in_KiB/1024,
 outgoing_read_bandwidth_in_KiB/2048)
```

You may still experience read and write throughput exceptions in the provisioned mode if you don't configure your data stream to handle your peak throughput. In this case, you must manually scale your data stream to accommodate your data traffic.

You may also experience read and write exceptions if you're using a partition key that leads to uneven data distribution and the records assigned to a shard exceed its limits. To resolve this issue in the provisioned mode, identify such shards and manually split them to better accommodate your traffic. For more information, see Resharding a Stream.

Switching Between Capacity Modes

You can switch the capacity mode of your data stream from on-demand to provisioned, or from provisioned to on-demand. For each data stream in your AWS account, you can switch between the on-demand and provisioned capacity modes twice within 24 hours.

Switching between capacity modes of a data stream does not cause any disruptions to your applications that use this data stream. You can continue writing to and reading from this data stream. As you are switching between capacity modes, either from on-demand to provisioned or from provisioned to on-demand, the status of the stream is set to *Updating*. You must wait for the data stream status to get to *Active* before you can modify its properties again.

When you switch from provisioned to on-demand capacity mode, your data stream initially retains whatever shard count it had before the transition, and from this point on, Kinesis Data Streams monitors your data traffic and scales the shard count of this on-demand data stream depending on your write throughput.

When you switch from on-demand to provisioned mode, your data stream also initially retains whatever shard count it had before the transition, but from this point on, you are responsible for monitoring and adjusting the shard count of this data stream to properly accomodate your write throughput.

Creating a Stream via the AWS Management Console

You can create a stream using the Kinesis Data Streams console, the Kinesis Data Streams API, or the AWS Command Line Interface (AWS CLI).

To create a data stream using the console

1. Sign in to the AWS Management Console and open the Kinesis console at https://console.aws.amazon.com/kinesis.
2. In the navigation bar, expand the Region selector and choose a Region.
3. Choose **Create data stream**.
4. On the **Create Kinesis stream** page, enter a name for your data stream and then choose either the **On-demand** or **Provisioned** capacity mode. The **On-demand** mode is selected by default. For more information, see Choosing the Data Stream Capacity Mode (p. 65).

   With the **On-demand** mode, you can then choose **Create Kinesis stream** to create your data stream. With the **Provisioned** mode, you must then specify the number of shards you need, and then choose **Create Kinesis stream**.

   On the **Kinesis streams** page, your stream's **Status** is **Creating** while the stream is being created. When the stream is ready to use, the **Status** changes to **Active**.
5. Choose the name of your stream. The **Stream Details** page displays a summary of your stream configuration, along with monitoring information.

To create a stream using the Kinesis Data Streams API

- For information about creating a stream using the Kinesis Data Streams API, see Creating a Stream via the APIs (p. 68).

To create a stream using the AWS CLI

- For information about creating a stream using the AWS CLI, see the create-stream command.

Creating a Stream via the APIs

Use the following steps to create your Kinesis data stream.

Build the Kinesis Data Streams Client

Before you can work with Kinesis data streams, you must build a client object. The following Java code instantiates a client builder and uses it to set the Region, credentials, and the client configuration. It then builds a client object.

```
AmazonKinesisClientBuilder clientBuilder = AmazonKinesisClientBuilder.standard();

clientBuilder.setRegion(regionName);
clientBuilder.setCredentials(credentialsProvider);
clientBuilder.setClientConfiguration(config);

AmazonKinesis client = clientBuilder.build();
```

For more information, see Kinesis Data Streams Regions and Endpoints in the *AWS General Reference*.

Create the Stream

Now that you have created your Kinesis Data Streams client, you can create a stream to work with, which you can accomplish with the Kinesis Data Streams console, or programmatically. To create a stream programmatically, instantiate a `CreateStreamRequest` object and specify a name for the stream and (if you want to use provisioned mode) the number of shards for the stream to use.

- **On-demand**:

```
CreateStreamRequest createStreamRequest = new CreateStreamRequest();
createStreamRequest.setStreamName( myStreamName );
```

- **Provisioned**:

```
CreateStreamRequest createStreamRequest = new CreateStreamRequest();
createStreamRequest.setStreamName( myStreamName );
createStreamRequest.setShardCount( myStreamSize );
```

The stream name identifies the stream. The name is scoped to the AWS account used by the application. It is also scoped by Region. That is, two streams in two different AWS accounts can have the same name, and two streams in the same AWS account but in two different Regions can have the same name, but not two streams on the same account and in the same Region.

The throughput of the stream is a function of the number of shards; more shards are required for greater provisioned throughput. More shards also increase the cost that AWS charges for the stream. For more information about calculating an appropriate number of shards for your application, see Choosing the Data Stream Capacity Mode (p. 65).

After the `createStreamRequest` object is configured, create a stream by calling the `createStream` method on the client. After calling `createStream`, wait for the stream to reach the `ACTIVE` state before performing any operations on the stream. To check the state of the stream, call the `describeStream` method. However, `describeStream` throws an exception if the stream does not exist. Therefore, enclose the `describeStream` call in a `try/catch` block.

```
client.createStream( createStreamRequest );
DescribeStreamRequest describeStreamRequest = new DescribeStreamRequest();
describeStreamRequest.setStreamName( myStreamName );

long startTime = System.currentTimeMillis();
long endTime = startTime + ( 10 * 60 * 1000 );
```

```
while ( System.currentTimeMillis() < endTime ) {
  try {
    Thread.sleep(20 * 1000);
  }
  catch ( Exception e ) {}

  try {
    DescribeStreamResult describeStreamResponse =
 client.describeStream( describeStreamRequest );
    String streamStatus = describeStreamResponse.getStreamDescription().getStreamStatus();
    if ( streamStatus.equals( "ACTIVE" ) ) {
      break;
    }
    //
    // sleep for one second
    //
    try {
      Thread.sleep( 1000 );
    }
    catch ( Exception e ) {}
  }
  catch ( ResourceNotFoundException e ) {}
}
if ( System.currentTimeMillis() >= endTime ) {
  throw new RuntimeException( "Stream " + myStreamName + " never went active" );
}
```

Updating a Stream

You can update the details of a stream using the Kinesis Data Streams console, the Kinesis Data Streams API, or the AWS CLI.

> **Note**
> You can enable server-side encryption for existing streams, or for streams that you have recently created.

To update a data stream using the console

1. Open the Amazon Kinesis console at https://console.aws.amazon.com/kinesis/.
2. In the navigation bar, expand the Region selector and choose a Region.
3. Choose the name of your stream in the list. The **Stream Details** page displays a summary of your stream configuration and monitoring information.
4. To switch between on-demand and provisioned capacity modes for a data stream, choose **Edit capacity mode** in the **Configuration** tab. For more information, see Choosing the Data Stream Capacity Mode (p. 65).

   > **Important**
   > For each data stream in your AWS account, you can switch between the on-demand and provisioned modes twice within 24 hours.
5. For a data stream with the provisioned mode, to edit the number of shards, choose **Edit provisioned shards** in the **Configuration** tab, and then enter a new shard count.
6. To enable server-side encryption of data records, choose **Edit** in the **Server-side encryption** section. Choose a KMS key to use as the master key for encryption, or use the default master key, **aws/kinesis**, managed by Kinesis. If you enable encryption for a stream and use your own AWS KMS master key, ensure that your producer and consumer applications have access to the AWS KMS master key that you used. To assign permissions to an application to access a user-generated AWS KMS key, see the section called “Permissions to Use User-Generated KMS Master Keys” (p. 212).

7. To edit the data retention period, choose **Edit** in the **Data retention period** section, and then enter a new data retention period.
8. If you have enabled custom metrics on your account, choose **Edit** in the **Shard level metrics** section, and then specify metrics for your stream. For more information, see the section called "Monitoring the Service with CloudWatch" (p. 180).

Updating a Stream Using the API

To update stream details using the API, see the following methods:

- AddTagsToStream
- DecreaseStreamRetentionPeriod
- DisableEnhancedMonitoring
- EnableEnhancedMonitoring
- IncreaseStreamRetentionPeriod
- RemoveTagsFromStream
- StartStreamEncryption
- StopStreamEncryption
- UpdateShardCount

Updating a Stream Using the AWS CLI

For information about updating a stream using the AWS CLI, see the Kinesis CLI reference.

Listing Streams

As described in the previous section, streams are scoped to the AWS account associated with the AWS credentials used to instantiate the Kinesis Data Streams client and also to the Region specified for the client. An AWS account could have many streams active at one time. You can list your streams in the Kinesis Data Streams console, or programmatically. The code in this section shows how to list all the streams for your AWS account.

```
ListStreamsRequest listStreamsRequest = new ListStreamsRequest();
listStreamsRequest.setLimit(20);
ListStreamsResult listStreamsResult = client.listStreams(listStreamsRequest);
List<String> streamNames = listStreamsResult.getStreamNames();
```

This code example first creates a new instance of `ListStreamsRequest` and calls its `setLimit` method to specify that a maximum of 20 streams should be returned for each call to `listStreams`. If you do not specify a value for `setLimit`, Kinesis Data Streams returns a number of streams less than or equal to the number in the account. The code then passes `listStreamsRequest` to the `listStreams` method of the client. The return value `listStreams` is stored in a `ListStreamsResult` object. The code calls the `getStreamNames` method on this object and stores the returned stream names in the `streamNames` list. Note that Kinesis Data Streams might return fewer streams than specified by the specified limit even if there are more streams than that in the account and Region. To ensure that you retrieve all the streams, use the `getHasMoreStreams` method as described in the next code example.

```
while (listStreamsResult.getHasMoreStreams())
{
    if (streamNames.size() > 0) {
```

```
      listStreamsRequest.setExclusiveStartStreamName(streamNames.get(streamNames.size() -
 1));
    }
    listStreamsResult = client.listStreams(listStreamsRequest);
    streamNames.addAll(listStreamsResult.getStreamNames());
}
```

This code calls the `getHasMoreStreams` method on `listStreamsRequest` to check if there are additional streams available beyond the ones returned in the initial call to `listStreams`. If so, the code calls the `setExclusiveStartStreamName` method with the name of the last stream that was returned in the previous call to `listStreams`. The `setExclusiveStartStreamName` method causes the next call to `listStreams` to start after that stream. The group of stream names returned by that call is then added to the `streamNames` list. This process continues until all the stream names have been collected in the list.

The streams returned by `listStreams` can be in one of the following states:

- `CREATING`
- `ACTIVE`
- `UPDATING`
- `DELETING`

You can check the state of a stream using the `describeStream` method, as shown in the previous section, Creating a Stream via the APIs (p. 68).

Listing Shards

A data stream can have one or more shards. There are two methods for listing (or retrieving) shards from a data stream.

Topics

ListShards API - Recommended

The recommended method for listing or retrieving the shards from a data stream is to use the ListShards API. The following example shows how you can get a list of the shards in a data stream. For a full description of the main operation used in this example and all of the parameters you can set for the operation, see ListShards.

```
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
import software.amazon.awssdk.services.kinesis.model.ListShardsRequest;
import software.amazon.awssdk.services.kinesis.model.ListShardsResponse;

import java.util.concurrent.TimeUnit;

public class ShardSample {

    public static void main(String[] args) {

        KinesisAsyncClient client = KinesisAsyncClient.builder().build();

        ListShardsRequest request = ListShardsRequest
```

```
                .builder().streamName("myFirstStream")
                .build();

        try {
            ListShardsResponse response = client.listShards(request).get(5000,
 TimeUnit.MILLISECONDS);
            System.out.println(response.toString());
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

To run the previous code example you can use a POM file like the following one.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>kinesis.data.streams.samples</groupId>
    <artifactId>shards</artifactId>
    <version>1.0-SNAPSHOT</version>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>8</source>
                    <target>8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <dependencies>
        <dependency>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>kinesis</artifactId>
            <version>2.0.0</version>
        </dependency>
    </dependencies>
</project>
```

With the `ListShards` API, you can use the ShardFilter parameter to filter out the response of the API. You can only specify one filter at a time.

If you use the `ShardFilter` parameter when invoking the ListShards API, the `Type` is the required property and must be specified. If you specify the `AT_TRIM_HORIZON`, `FROM_TRIM_HORIZON`, or `AT_LATEST` types, you do not need to specify either the `ShardId` or the `Timestamp` optional properties.

If you specify the `AFTER_SHARD_ID` type, you must also provide the value for the optional `ShardId` property. The `ShardId` property is identical in functionality to the `ExclusiveStartShardId` parameter of the ListShards API. When `ShardId` property is specified, the response includes the shards starting with the shard whose ID immediately follows the `ShardId` that you provided.

If you specify the `AT_TIMESTAMP` or `FROM_TIMESTAMP_ID` type, you must also provide the value for the optional `Timestamp` property. If you specify the `AT_TIMESTAMP` type, then all shards that were open at the provided timestamp are returned. If you specify the `FROM_TIMESTAMP` type, then all shards starting from the provided timestamp to TIP are returned.

Important
`DescribeStreamSummary` and `ListShard` APIs provide a more scalable way to retrieve information about your data streams. More specifically, the quotas for the DescribeStream API can cause throttling. For more information, see Quotas and Limits (p. 7). Note also that `DescribeStream` quotas are shared across all applications that interact with all data streams in your AWS account. The quotas for the ListShards API, on the other hand, are specific to a single data stream. So not only do you get higher TPS with the ListShards API, but the action scales better as you create more data streams.
We recommend that you migrate all of your producers and consumers that call the DescribeStream API to instead invoke the DescribeStreamSummary and the ListShard APIs. To identify these producers and consumers, we recommend using Athena to parse CloudTrail logs as user agents for KPL and KCL are captured in the API calls.

```
SELECT useridentity.sessioncontext.sessionissuer.username,
useridentity.arn,eventname,useragent, count(*) FROM
cloudtrail_logs WHERE Eventname IN ('DescribeStream')  AND
eventtime
    BETWEEN ''
        AND ''
GROUP BY
 useridentity.sessioncontext.sessionissuer.username,useridentity.arn,eventname,useragent
ORDER BY  count(*) DESC LIMIT 100
```

We also recommend that the AWS Lambda and Kinesis Data Firehose integrations with Kinesis Data Streams that invoke the `DescribeStream` API are reconfigured so that the integrations instead invoke `DescribeStreamSummary` and `ListShards`. Specifically, for AWS Lambda, you must update your event source mapping. For Kinesis Data Firehose, the corresponding IAM permissions must be updated so that they include the `ListShards` IAM permission.

DescribeStream API - Deprecated

Important
The information below describes a currently deprecated way of retrieving shards from a data stream via the DescribeStream API. It is currently highly recommended that you use the `ListShards` API to retrieve the shards that comprise the data stream.

The response object returned by the `describeStream` method enables you to retrieve information about the shards that comprise the stream. To retrieve the shards, call the `getShards` method on this object. This method might not return all the shards from the stream in a single call. In the following code, we check the `getHasMoreShards` method on `getStreamDescription` to see if there are additional shards that were not returned. If so, that is, if this method returns `true`, we continue to call `getShards` in a loop, adding each new batch of returned shards to our list of shards. The loop exits when `getHasMoreShards` returns `false`; that is, all shards have been returned. Note that `getShards` does not return shards that are in the `EXPIRED` state. For more information about shard states, including the `EXPIRED` state, see Data Routing, Data Persistence, and Shard State after a Reshard (p. 79).

```
DescribeStreamRequest describeStreamRequest = new DescribeStreamRequest();
describeStreamRequest.setStreamName( myStreamName );
List<Shard> shards = new ArrayList<>();
String exclusiveStartShardId = null;
do {
    describeStreamRequest.setExclusiveStartShardId( exclusiveStartShardId );
    DescribeStreamResult describeStreamResult =
 client.describeStream( describeStreamRequest );
    shards.addAll( describeStreamResult.getStreamDescription().getShards() );
    if (describeStreamResult.getStreamDescription().getHasMoreShards() && shards.size() >
 0) {
```

```
        exclusiveStartShardId = shards.get(shards.size() - 1).getShardId();
    } else {
        exclusiveStartShardId = null;
    }
} while ( exclusiveStartShardId != null );
```

Deleting a Stream

You can delete a stream with the Kinesis Data Streams console, or programmatically. To delete a stream programmatically, use `DeleteStreamRequest`, as shown in the following code.

```
DeleteStreamRequest deleteStreamRequest = new DeleteStreamRequest();
deleteStreamRequest.setStreamName(myStreamName);
client.deleteStream(deleteStreamRequest);
```

Shut down any applications that are operating on the stream before you delete it. If an application attempts to operate on a deleted stream, it receives `ResourceNotFound` exceptions. Also, if you subsequently create a new stream that has the same name as your previous stream, and applications that were operating on the previous stream are still running, these applications might try to interact with the new stream as though it were the previous stream—with unpredictable results.

Resharding a Stream

Important
You can reshard your stream using the UpdateShardCount API. Otherwise, you can continue to perform splits and merges as explained here.

Amazon Kinesis Data Streams supports *resharding*, which lets you adjust the number of shards in your stream to adapt to changes in the rate of data flow through the stream. Resharding is considered an advanced operation. If you are new to Kinesis Data Streams, return to this subject after you are familiar with all the other aspects of Kinesis Data Streams.

There are two types of resharding operations: shard split and shard merge. In a shard split, you divide a single shard into two shards. In a shard merge, you combine two shards into a single shard. Resharding is always *pairwise* in the sense that you cannot split into more than two shards in a single operation, and you cannot merge more than two shards in a single operation. The shard or pair of shards that the resharding operation acts on are referred to as *parent* shards. The shard or pair of shards that result from the resharding operation are referred to as *child* shards.

Splitting increases the number of shards in your stream and therefore increases the data capacity of the stream. Because you are charged on a per-shard basis, splitting increases the cost of your stream. Similarly, merging reduces the number of shards in your stream and therefore decreases the data capacity—and cost—of the stream.

Resharding is typically performed by an administrative application that is distinct from the producer (put) applications and the consumer (get) applications. Such an administrative application monitors the overall performance of the stream based on metrics provided by Amazon CloudWatch or based on metrics collected from the producers and consumers. The administrative application also needs a broader set of IAM permissions than the consumers or producers because the consumers and producers usually should not need access to the APIs used for resharding. For more information about IAM permissions for Kinesis Data Streams, see Controlling Access to Amazon Kinesis Data Streams Resources Using IAM (p. 216).

For more information about resharding, see How do I change the number of open shards in Kinesis Data Streams?

Topics

Strategies for Resharding

The purpose of resharding in Amazon Kinesis Data Streams is to enable your stream to adapt to changes in the rate of data flow. You split shards to increase the capacity (and cost) of your stream. You merge shards to reduce the cost (and capacity) of your stream.

One approach to resharding could be to split every shard in the stream—which would double the stream's capacity. However, this might provide more additional capacity than you actually need and therefore create unnecessary cost.

You can also use metrics to determine which are your *hot* or *cold* shards, that is, shards that are receiving much more data, or much less data, than expected. You could then selectively split the hot shards to increase capacity for the hash keys that target those shards. Similarly, you could merge cold shards to make better use of their unused capacity.

You can obtain some performance data for your stream from the Amazon CloudWatch metrics that Kinesis Data Streams publishes. However, you can also collect some of your own metrics for your streams. One approach would be to log the hash key values generated by the partition keys for your data records. Recall that you specify the partition key at the time that you add the record to the stream.

```
putRecordRequest.setPartitionKey( String.format( "myPartitionKey" ) );
```

Kinesis Data Streams uses MD5 to compute the hash key from the partition key. Because you specify the partition key for the record, you could use MD5 to compute the hash key value for that record and log it.

You could also log the IDs of the shards that your data records are assigned to. The shard ID is available by using the `getShardId` method of the `putRecordResults` object returned by the `putRecords` method, and the `putRecordResult` object returned by the `putRecord` method.

```
String shardId = putRecordResult.getShardId();
```

With the shard IDs and the hash key values, you can determine which shards and hash keys are receiving the most or least traffic. You can then use resharding to provide more or less capacity, as appropriate for these keys.

Splitting a Shard

To split a shard in Amazon Kinesis Data Streams, you need to specify how hash key values from the parent shard should be redistributed to the child shards. When you add a data record to a stream, it is assigned to a shard based on a hash key value. The hash key value is the MD5 hash of the partition key that you specify for the data record at the time that you add the data record to the stream. Data records that have the same partition key also have the same hash key value.

The possible hash key values for a given shard constitute a set of ordered contiguous non-negative integers. This range of possible hash key values is given by the following:

```
shard.getHashKeyRange().getStartingHashKey();
shard.getHashKeyRange().getEndingHashKey();
```

When you split the shard, you specify a value in this range. That hash key value and all higher hash key values are distributed to one of the child shards. All the lower hash key values are distributed to the other child shard.

The following code demonstrates a shard split operation that redistributes the hash keys evenly between each of the child shards, essentially splitting the parent shard in half. This is just one possible way of dividing the parent shard. You could, for example, split the shard so that the lower one-third of the keys from the parent go to one child shard and the upper two-thirds of the keys go to the other child shard. However, for many applications, splitting shards in half is an effective approach.

The code assumes that `myStreamName` holds the name of your stream and the object variable `shard` holds the shard to split. Begin by instantiating a new `splitShardRequest` object and setting the stream name and shard ID.

```
SplitShardRequest splitShardRequest = new SplitShardRequest();
splitShardRequest.setStreamName(myStreamName);
splitShardRequest.setShardToSplit(shard.getShardId());
```

Determine the hash key value that is half-way between the lowest and highest values in the shard. This is the starting hash key value for the child shard that will contain the upper half of the hash keys from the parent shard. Specify this value in the `setNewStartingHashKey` method. You need specify only this value. Kinesis Data Streams automatically distributes the hash keys below this value to the other child shard that is created by the split. The last step is to call the `splitShard` method on the Kinesis Data Streams client.

```
BigInteger startingHashKey = new BigInteger(shard.getHashKeyRange().getStartingHashKey());
BigInteger endingHashKey   = new BigInteger(shard.getHashKeyRange().getEndingHashKey());
String newStartingHashKey  = startingHashKey.add(endingHashKey).divide(new
 BigInteger("2")).toString();

splitShardRequest.setNewStartingHashKey(newStartingHashKey);
client.splitShard(splitShardRequest);
```

The first step after this procedure is shown in Waiting for a Stream to Become Active Again (p. 78).

Merging Two Shards

A shard merge operation takes two specified shards and combines them into a single shard. After the merge, the single child shard receives data for all hash key values covered by the two parent shards.

Shard Adjacency

To merge two shards, the shards must be *adjacent*. Two shards are considered adjacent if the union of the hash key ranges for the two shards forms a contiguous set with no gaps. For example, suppose that you have two shards, one with a hash key range of 276...381 and the other with a hash key range of 382...454. You could merge these two shards into a single shard that would have a hash key range of 276...454.

To take another example, suppose that you have two shards, one with a hash key range of 276..381 and the other with a hash key range of 455...560. You could not merge these two shards because there would be one or more shards between these two that cover the range 382..454.

The set of all `OPEN` shards in a stream—as a group—always spans the entire range of MD5 hash key values. For more information about shard states—such as `CLOSED`—see Data Routing, Data Persistence, and Shard State after a Reshard (p. 79).

To identify shards that are candidates for merging, you should filter out all shards that are in a `CLOSED` state. Shards that are `OPEN`—that is, not `CLOSED`—have an ending sequence number of `null`. You can test the ending sequence number for a shard using:

```
if( null == shard.getSequenceNumberRange().getEndingSequenceNumber() )
{
  // Shard is OPEN, so it is a possible candidate to be merged.
}
```

After filtering out the closed shards, sort the remaining shards by the highest hash key value supported by each shard. You can retrieve this value using:

```
shard.getHashKeyRange().getEndingHashKey();
```

If two shards are adjacent in this filtered, sorted list, they can be merged.

Code for the Merge Operation

The following code merges two shards. The code assumes that `myStreamName` holds the name of your stream and the object variables `shard1` and `shard2` hold the two adjacent shards to merge.

For the merge operation, begin by instantiating a new `mergeShardsRequest` object. Specify the stream name with the `setStreamName` method. Then specify the two shards to merge using the `setShardToMerge` and `setAdjacentShardToMerge` methods. Finally, call the `mergeShards` method on the Kinesis Data Streams client to carry out the operation.

```
MergeShardsRequest mergeShardsRequest = new MergeShardsRequest();
mergeShardsRequest.setStreamName(myStreamName);
mergeShardsRequest.setShardToMerge(shard1.getShardId());
mergeShardsRequest.setAdjacentShardToMerge(shard2.getShardId());
client.mergeShards(mergeShardsRequest);
```

The first step after this procedure is shown in Waiting for a Stream to Become Active Again (p. 78).

After Resharding

After any kind of resharding procedure in Amazon Kinesis Data Streams, and before normal record processing resumes, other procedures and considerations are required. The following sections describe these.

Topics

Waiting for a Stream to Become Active Again

After you call a resharding operation, either `splitShard` or `mergeShards`, you need to wait for the stream to become active again. The code to use is the same as when you wait for a stream to become active after creating a stream (p. 68). That code is as follows:

```
DescribeStreamRequest describeStreamRequest = new DescribeStreamRequest();
describeStreamRequest.setStreamName( myStreamName );

long startTime = System.currentTimeMillis();
long endTime = startTime + ( 10 * 60 * 1000 );
while ( System.currentTimeMillis() < endTime )
{
  try {
    Thread.sleep(20 * 1000);
  }
  catch ( Exception e ) {}
```

```
  try {
    DescribeStreamResult describeStreamResponse =
 client.describeStream( describeStreamRequest );
    String streamStatus = describeStreamResponse.getStreamDescription().getStreamStatus();
    if ( streamStatus.equals( "ACTIVE" ) ) {
      break;
    }
   //
    // sleep for one second
    //
    try {
      Thread.sleep( 1000 );
    }
    catch ( Exception e ) {}
  }
  catch ( ResourceNotFoundException e ) {}
}
if ( System.currentTimeMillis() >= endTime )
{
  throw new RuntimeException( "Stream " + myStreamName + " never went active" );
}
```

Data Routing, Data Persistence, and Shard State after a Reshard

Kinesis Data Streams is a real-time data streaming service, which is to say that your applications should assume that data is flowing continuously through the shards in your stream. When you reshard, data records that were flowing to the parent shards are re-routed to flow to the child shards based on the hash key values that the data-record partition keys map to. However, any data records that were in the parent shards before the reshard remain in those shards. In other words, the parent shards do not disappear when the reshard occurs. They persist along with the data they contained before the reshard. The data records in the parent shards are accessible using the `getShardIterator` and `getRecords` (p. 150) operations in the Kinesis Data Streams API, or through the Kinesis Client Library.

> **Note**
> Data records are accessible from the time they are added to the stream to the current retention period. This holds true regardless of any changes to the shards in the stream during that time period. For more information about a stream's retention period, see Changing the Data Retention Period (p. 80).

In the process of resharding, a parent shard transitions from an `OPEN` state to a `CLOSED` state to an `EXPIRED` state.

- **OPEN**: Before a reshard operation, a parent shard is in the `OPEN` state, which means that data records can be both added to the shard and retrieved from the shard.
- **CLOSED**: After a reshard operation, the parent shard transitions to a `CLOSED` state. This means that data records are no longer added to the shard. Data records that would have been added to this shard are now added to a child shard instead. However, data records can still be retrieved from the shard for a limited time.
- **EXPIRED**: After the stream's retention period has expired, all the data records in the parent shard have expired and are no longer accessible. At this point, the shard itself transitions to an `EXPIRED` state. Calls to `getStreamDescription().getShards` to enumerate the shards in the stream do not include `EXPIRED` shards in the list shards returned. For more information about a stream's retention period, see Changing the Data Retention Period (p. 80).

After the reshard has occurred and the stream is again in an `ACTIVE` state, you could immediately begin to read data from the child shards. However, the parent shards that remain after the reshard could still contain data that you haven't read yet that was added to the stream before the reshard. If you read data from the child shards before having read all data from the parent shards, you could read data for a

particular hash key out of the order given by the data records' sequence numbers. Therefore, assuming that the order of the data is important, you should, after a reshard, always continue to read data from the parent shards until it is exhausted. Only then should you begin reading data from the child shards. When `getRecordsResult.getNextShardIterator` returns `null`, it indicates that you have read all the data in the parent shard. If you are reading data using the Kinesis Client Library, the library ensures that you receive the data in order even if a reshard occurs.

Changing the Data Retention Period

Amazon Kinesis Data Streams supports changes to the data record retention period of your data stream. A Kinesis data stream is an ordered sequence of data records meant to be written to and read from in real time. Data records are therefore stored in shards in your stream temporarily. The time period from when a record is added to when it is no longer accessible is called the *retention period*. A Kinesis data stream stores records from 24 hours by default, up to 8760 hours (365 days).

You can update the retention period via the Kinesis Data Streams console or by using the IncreaseStreamRetentionPeriod and the DecreaseStreamRetentionPeriod operations. With the Kinesis Data Streams console, you can bulk edit the retention period of more than one data stream at the same time. You can increase the retention period up to a maximum of 8760 hours (365 days) using the IncreaseStreamRetentionPeriod operation or the Kinesis Data Streams console. You can decrease the retention period down to a minimum of 24 hours using the DecreaseStreamRetentionPeriod operation or the Kinesis Data Streams console. The request syntax for both operations includes the stream name and the retention period in hours. Finally, you can check the current retention period of a stream by calling the DescribeStream operation.

The following is an example of changing the retention period using the AWS CLI:

```
aws kinesis increase-stream-retention-period --stream-name retentionPeriodDemo --retention-period-hours 72
```

Kinesis Data Streams stops making records inaccessible at the old retention period within several minutes of increasing the retention period. For example, changing the retention period from 24 hours to 48 hours means that records added to the stream 23 hours 55 minutes prior are still available after 24 hours.

Kinesis Data Streams almost immediately makes records older than the new retention period inaccessible upon decreasing the retention period. Therefore, take great care when calling the DecreaseStreamRetentionPeriod operation.

Set your data retention period to ensure that your consumers are able to read data before it expires, if problems occur. You should carefully consider all possibilities, such as an issue with your record processing logic or a downstream dependency being down for a long period of time. Think of the retention period as a safety net to allow more time for your data consumers to recover. The retention period API operations allow you to set this up proactively or to respond to operational events reactively.

Additional charges apply for streams with a retention period set above 24 hours. For more information, see Amazon Kinesis Data Streams Pricing.

Tagging Your Streams in Amazon Kinesis Data Streams

You can assign your own metadata to streams you create in Amazon Kinesis Data Streams in the form of *tags*. A tag is a key-value pair that you define for a stream. Using tags is a simple yet powerful way to manage AWS resources and organize data, including billing data.

Contents

Tag Basics

You use the Kinesis Data Streams console, AWS CLI, or Kinesis Data Streams API to complete the following tasks:

- Add tags to a stream
- List the tags for your streams
- Remove tags from a stream

You can use tags to categorize your streams. For example, you can categorize streams by purpose, owner, or environment. Because you define the key and value for each tag, you can create a custom set of categories to meet your specific needs. For example, you might define a set of tags that helps you track streams by owner and associated application. Here are several examples of tags:

- Project: Project name
- Owner: Name
- Purpose: Load testing
- Application: Application name
- Environment: Production

Tracking Costs Using Tagging

You can use tags to categorize and track your AWS costs. When you apply tags to your AWS resources, including streams, your AWS cost allocation report includes usage and costs aggregated by tags. You can apply tags that represent business categories (such as cost centers, application names, or owners) to organize your costs across multiple services. For more information, see Use Cost Allocation Tags for Custom Billing Reports in the *AWS Billing User Guide*.

Tag Restrictions

The following restrictions apply to tags.

Basic restrictions

- The maximum number of tags per resource (stream) is 50.
- Tag keys and values are case-sensitive.
- You can't change or edit tags for a deleted stream.

Tag key restrictions

- Each tag key must be unique. If you add a tag with a key that's already in use, your new tag overwrites the existing key-value pair.

- You can't start a tag key with `aws:` because this prefix is reserved for use by AWS. AWS creates tags that begin with this prefix on your behalf, but you can't edit or delete them.
- Tag keys must be between 1 and 128 Unicode characters in length.
- Tag keys must consist of the following characters: Unicode letters, digits, white space, and the following special characters: `_ . / = + - @`.

Tag value restrictions

- Tag values must be between 0 and 255 Unicode characters in length.
- Tag values can be blank. Otherwise, they must consist of the following characters: Unicode letters, digits, white space, and any of the following special characters: `_ . / = + - @`.

Tagging Streams Using the Kinesis Data Streams Console

You can add, list, and remove tags using the Kinesis Data Streams console.

To view the tags for a stream

1. Open the Kinesis Data Streams console. In the navigation bar, expand the region selector and select a region.
2. On the **Stream List** page, select a stream.
3. On the **Stream Details** page, click the **Tags** tab.

To add a tag to a stream

1. Open the Kinesis Data Streams console. In the navigation bar, expand the region selector and select a region.
2. On the **Stream List** page, select a stream.
3. On the **Stream Details** page, click the **Tags** tab.
4. Specify the tag key in the **Key** field, optionally specify a tag value in the **Value** field, and then click **Add Tag**.

   If the **Add Tag** button is not enabled, either the tag key or tag value that you specified don't meet the tag restrictions. For more information, see Tag Restrictions (p. 81).
5. To view your new tag in the list on the **Tags** tab, click the refresh icon.

To remove a tag from a stream

1. Open the Kinesis Data Streams console. In the navigation bar, expand the region selector and select a region.
2. On the Stream List page, select a stream.
3. On the Stream Details page, click the **Tags** tab, and then click the **Remove** icon for the tag.
4. In the **Delete Tag** dialog box, click **Yes, Delete**.

Tagging Streams Using the AWS CLI

You can add, list, and remove tags using the AWS CLI. For examples, see the following documentation.

add-tags-to-stream

Adds or updates tags for the specified stream.

list-tags-for-stream

Lists the tags for the specified stream.

remove-tags-from-stream

Removes tags from the specified stream.

Tagging Streams Using the Kinesis Data Streams API

You can add, list, and remove tags using the Kinesis Data Streams API. For examples, see the following documentation:

AddTagsToStream

Adds or updates tags for the specified stream.

ListTagsForStream

Lists the tags for the specified stream.

RemoveTagsFromStream

Removes tags from the specified stream.

Writing Data to Amazon Kinesis Data Streams

A *producer* is an application that writes data to Amazon Kinesis Data Streams. You can build producers for Kinesis Data Streams using the AWS SDK for Java and the Kinesis Producer Library.

If you are new to Kinesis Data Streams, start by becoming familiar with the concepts and terminology presented in What Is Amazon Kinesis Data Streams? (p. 1) and Getting Started with Amazon Kinesis Data Streams (p. 14).

> **Important**
> Kinesis Data Streams supports changes to the data record retention period of your data stream. For more information, see Changing the Data Retention Period (p. 80).

To put data into the stream, you must specify the name of the stream, a partition key, and the data blob to be added to the stream. The partition key is used to determine which shard in the stream the data record is added to.

All the data in the shard is sent to the same worker that is processing the shard. Which partition key you use depends on your application logic. The number of partition keys should typically be much greater than the number of shards. This is because the partition key is used to determine how to map a data record to a particular shard. If you have enough partition keys, the data can be evenly distributed across the shards in a stream.

Contents

Developing Producers Using the Amazon Kinesis Producer Library

An Amazon Kinesis Data Streams producer is an application that puts user data records into a Kinesis data stream (also called *data ingestion*). The Kinesis Producer Library (KPL) simplifies producer application development, allowing developers to achieve high write throughput to a Kinesis data stream.

You can monitor the KPL with Amazon CloudWatch. For more information, see Monitoring the Kinesis Producer Library with Amazon CloudWatch (p. 204).

Contents

Note
It is recommended that you upgrade to the latest KPL version. KPL is regularly updated with newer releases that include the latest dependency and security patches, bug fixes, and backward-compatible new features. For more information, see https://github.com/awslabs/amazon-kinesis-producer/releases/.

Role of the KPL

The KPL is an easy-to-use, highly configurable library that helps you write to a Kinesis data stream. It acts as an intermediary between your producer application code and the Kinesis Data Streams API actions. The KPL performs the following primary tasks:

- Writes to one or more Kinesis data streams with an automatic and configurable retry mechanism
- Collects records and uses `PutRecords` to write multiple records to multiple shards per request
- Aggregates user records to increase payload size and improve throughput
- Integrates seamlessly with the Kinesis Client Library (KCL) to de-aggregate batched records on the consumer
- Submits Amazon CloudWatch metrics on your behalf to provide visibility into producer performance

Note that the KPL is different from the Kinesis Data Streams API that is available in the AWS SDKs. The Kinesis Data Streams API helps you manage many aspects of Kinesis Data Streams (including creating streams, resharding, and putting and getting records), while the KPL provides a layer of abstraction specifically for ingesting data. For information about the Kinesis Data Streams API, see the Amazon Kinesis API Reference.

Advantages of Using the KPL

The following list represents some of the major advantages to using the KPL for developing Kinesis Data Streams producers.

The KPL can be used in either synchronous or asynchronous use cases. We suggest using the higher performance of the asynchronous interface unless there is a specific reason to use synchronous behavior. For more information about these two use cases and example code, see Writing to your Kinesis Data Stream Using the KPL (p. 90).

Performance Benefits

The KPL can help build high-performance producers. Consider a situation where your Amazon EC2 instances serve as a proxy for collecting 100-byte events from hundreds or thousands of low

power devices and writing records into a Kinesis data stream. These EC2 instances must each write thousands of events per second to your data stream. To achieve the throughput needed, producers must implement complicated logic, such as batching or multithreading, in addition to retry logic and record de-aggregation at the consumer side. The KPL performs all of these tasks for you.

Consumer-Side Ease of Use

For consumer-side developers using the KCL in Java, the KPL integrates without additional effort. When the KCL retrieves an aggregated Kinesis Data Streams record consisting of multiple KPL user records, it automatically invokes the KPL to extract the individual user records before returning them to the user.

For consumer-side developers who do not use the KCL but instead use the API operation `GetRecords` directly, a KPL Java library is available to extract the individual user records before returning them to the user.

Producer Monitoring

You can collect, monitor, and analyze your Kinesis Data Streams producers using Amazon CloudWatch and the KPL. The KPL emits throughput, error, and other metrics to CloudWatch on your behalf, and is configurable to monitor at the stream, shard, or producer level.

Asynchronous Architecture

Because the KPL may buffer records before sending them to Kinesis Data Streams, it does not force the caller application to block and wait for a confirmation that the record has arrived at the server before continuing execution. A call to put a record into the KPL always returns immediately and does not wait for the record to be sent or a response to be received from the server. Instead, a `Future` object is created that receives the result of sending the record to Kinesis Data Streams at a later time. This is the same behavior as asynchronous clients in the AWS SDK.

When Not to Use the KPL

The KPL can incur an additional processing delay of up to `RecordMaxBufferedTime` within the library (user-configurable). Larger values of `RecordMaxBufferedTime` results in higher packing efficiencies and better performance. Applications that cannot tolerate this additional delay may need to use the AWS SDK directly. For more information about using the AWS SDK with Kinesis Data Streams, see Developing Producers Using the Amazon Kinesis Data Streams API with the AWS SDK for Java (p. 95). For more information about `RecordMaxBufferedTime` and other user-configurable properties of the KPL, see Configuring the Kinesis Producer Library (p. 91).

Installing the KPL

Amazon provides pre-built binaries of the C++ Kinesis Producer Library (KPL) for macOS, Windows, and recent Linux distributions (for supported platform details, see the next section). These binaries are packaged as part of Java .jar files and are automatically invoked and used if you are using Maven to install the package. To locate the latest versions of the KPL and KCL, use the following Maven search links:

- KPL
- KCL

The Linux binaries have been compiled with the GNU Compiler Collection (GCC) and statically linked against libstdc++ on Linux. They are expected to work on any 64-bit Linux distribution that includes a glibc version 2.5 or higher.

Users of older Linux distributions can build the KPL using the build instructions provided along with the source on GitHub. To download the KPL from GitHub, see Kinesis Producer Library.

Transitioning to Amazon Trust Services (ATS) Certificates for the Kinesis Producer Library

On February 9, 2018, at 9:00 AM PST, Amazon Kinesis Data Streams installed ATS certificates. To continue to be able to write records to Kinesis Data Streams using the Kinesis Producer Library (KPL), you must upgrade your installation of the KPL to version 0.12.6 or later. This change affects all AWS Regions.

For information about the move to ATS, please see How to Prepare for AWS's Move to Its Own Certificate Authority.

If you encounter problems and need technical support, create a case with the AWS Support Center.

KPL Supported Platforms

The Kinesis Producer Library (KPL) is written in C++ and runs as a child process to the main user process. Precompiled 64-bit native binaries are bundled with the Java release and are managed by the Java wrapper.

The Java package runs without the need to install any additional libraries on the following operating systems:

- Linux distributions with kernel 2.6.18 (September 2006) and later
- Apple OS X 10.9 and later
- Windows Server 2008 and later

  Important
  Windows Server 2008 and later is supported for all KPL versions up to version 0.14.0. The Windows platform is NOT supported starting with KPL version 0.14.0 or higher.

Note that the KPL is 64-bit only.

Source Code

If the binaries provided in the KPL installation are not sufficient for your environment, the core of the KPL is written as a C++ module. The source code for the C++ module and the Java interface are released under the Amazon Public License and are available on GitHub at Kinesis Producer Library. Although the KPL can be used on any platform for which a recent standards-compliant C++ compiler and JRE are available, Amazon doesn't officially support any platform that is not on the supported platforms list.

KPL Key Concepts

The following sections contain concepts and terminology necessary to understand and benefit from the Kinesis Producer Library (KPL).

Topics

Records

In this guide, we distinguish between *KPL user records* and *Kinesis Data Streams records*. When we use the term *record* without a qualifier, we refer to a *KPL user record*. When we refer to a Kinesis Data Streams record, we explicitly say *Kinesis Data Streams record*.

A KPL user record is a blob of data that has particular meaning to the user. Examples include a JSON blob representing a UI event on a website, or a log entry from a web server.

A Kinesis Data Streams record is an instance of the `Record` data structure defined by the Kinesis Data Streams service API. It contains a partition key, sequence number, and a blob of data.

Batching

Batching refers to performing a single action on multiple items instead of repeatedly performing the action on each individual item.

In this context, the "item" is a record, and the action is sending it to Kinesis Data Streams. In a non-batching situation, you would place each record in a separate Kinesis Data Streams record and make one HTTP request to send it to Kinesis Data Streams. With batching, each HTTP request can carry multiple records instead of just one.

The KPL supports two types of batching:

- *Aggregation* – Storing multiple records within a single Kinesis Data Streams record.
- *Collection* – Using the API operation `PutRecords` to send multiple Kinesis Data Streams records to one or more shards in your Kinesis data stream.

The two types of KPL batching are designed to coexist and can be turned on or off independently of one another. By default, both are turned on.

Aggregation

Aggregation refers to the storage of multiple records in a Kinesis Data Streams record. Aggregation allows customers to increase the number of records sent per API call, which effectively increases producer throughput.

Kinesis Data Streams shards support up to 1,000 Kinesis Data Streams records per second, or 1 MB throughput. The Kinesis Data Streams records per second limit binds customers with records smaller than 1 KB. Record aggregation allows customers to combine multiple records into a single Kinesis Data Streams record. This allows customers to improve their per shard throughput.

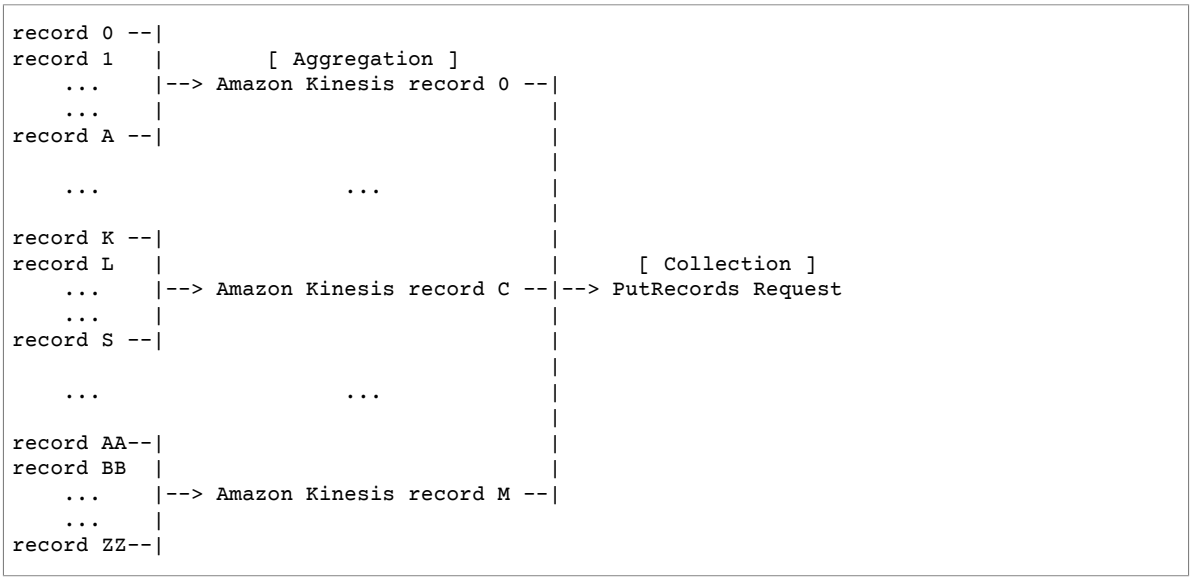
Consider the case of one shard in region us-east-1 that is currently running at a constant rate of 1,000 records per second, with records that are 512 bytes each. With KPL aggregation, you can pack 1,000 records into only 10 Kinesis Data Streams records, reducing the RPS to 10 (at 50 KB each).

Collection

Collection refers to batching multiple Kinesis Data Streams records and sending them in a single HTTP request with a call to the API operation `PutRecords`, instead of sending each Kinesis Data Streams record in its own HTTP request.

This increases throughput compared to using no collection because it reduces the overhead of making many separate HTTP requests. In fact, `PutRecords` itself was specifically designed for this purpose.

Collection differs from aggregation in that it is working with groups of Kinesis Data Streams records. The Kinesis Data Streams records being collected can still contain multiple records from the user. The relationship can be visualized as such:



Integrating the KPL with Producer Code

The Kinesis Producer Library (KPL) runs in a separate process, and communicates with your parent user process using IPC. This architecture is sometimes called a microservice, and is chosen for two main reasons:

1) Your user process will not crash even if the KPL crashes

Your process could have tasks unrelated to Kinesis Data Streams, and may be able to continue operation even if the KPL crashes. It is also possible for your parent user process to restart the KPL and recover to a fully working state (this functionality is in the official wrappers).

An example is a web server that sends metrics to Kinesis Data Streams; the server can continue serving pages even if the Kinesis Data Streams part has stopped working. Crashing the whole server because of a bug in the KPL would therefore cause an unnecessary outage.

2) Arbitrary clients can be supported

There are always customers who use languages other than the ones officially supported. These customers should also be able to use the KPL easily.

Recommended Usage Matrix

The following usage matrix enumerates the recommended settings for different users and advises you about whether and how you should use the KPL. Keep in mind that if aggregation is enabled, de-aggregation must also be used to extract your records on the consumer side.

Producer side language	Consumer side language	KCL Version	Checkpoint logic	Can you use the KPL?	Caveats
Anything but Java	*	*	*	No	N/A
Java	Java	Uses Java SDK directly	N/A	Yes	If aggregation is used, you have to use the

Producer side language	Consumer side language	KCL Version	Checkpoint logic	Can you use the KPL?	Caveats
					provided de-aggregation library after `GetRecords` calls.
Java	Anything but Java	Uses SDK directly	N/A	Yes	Must disable aggregation.
Java	Java	1.3.x	N/A	Yes	Must disable aggregation.
Java	Java	1.4.x	Calls checkpoint without any arguments	Yes	None
Java	Java	1.4.x	Calls checkpoint with an explicit sequence number	Yes	Either disable aggregation, or change the code to use extended sequence numbers for checkpointing.
Java	Anything but Java	1.3.x + Multilanguage daemon + language-specific wrapper	N/A	Yes	Must disable aggregation.

Writing to your Kinesis Data Stream Using the KPL

The following sections show sample code in a progression from the simplest possible "bare-bones" producer on through to fully asynchronous code.

Barebones Producer Code

The following code is all that is needed to write a minimal working producer. The Kinesis Producer Library (KPL) user records are processed in the background.

```
// KinesisProducer gets credentials automatically like
// DefaultAWSCredentialsProviderChain.
// It also gets region automatically from the EC2 metadata service.
KinesisProducer kinesis = new KinesisProducer();
// Put some records
for (int i = 0; i < 100; ++i) {
    ByteBuffer data = ByteBuffer.wrap("myData".getBytes("UTF-8"));
    // doesn't block
    kinesis.addUserRecord("myStream", "myPartitionKey", data);
}
// Do other stuff ...
```

Responding to Results Synchronously

In the previous example, the code didn't check whether the KPL user records succeeded. The KPL performs any retries needed to account for failures. But if you want to check on the results, you can examine them using the `Future` objects that are returned from `addUserRecord`, as in the following example (previous example shown for context):

```
KinesisProducer kinesis = new KinesisProducer();

// Put some records and save the Futures
List<Future<UserRecordResult>> putFutures = new LinkedList<Future<UserRecordResult>>();
for (int i = 0; i < 100; i++) {
    ByteBuffer data = ByteBuffer.wrap("myData".getBytes("UTF-8"));
    // doesn't block
    putFutures.add(
        kinesis.addUserRecord("myStream", "myPartitionKey", data));
}

// Wait for puts to finish and check the results
for (Future<UserRecordResult> f : putFutures) {
    UserRecordResult result = f.get(); // this does block
    if (result.isSuccessful()) {
        System.out.println("Put record into shard " +
                           result.getShardId());
    } else {
        for (Attempt attempt : result.getAttempts()) {
            // Analyze and respond to the failure
        }
    }
}
```

Responding to Results Asynchronously

The previous example is calling `get()` on a `Future` object, which blocks execution. If you don't want to block execution, you can use an asynchronous callback, as shown in the following example:

```
KinesisProducer kinesis = new KinesisProducer();

FutureCallback<UserRecordResult> myCallback = new FutureCallback<UserRecordResult>() {
    @Override public void onFailure(Throwable t) {
        /* Analyze and respond to the failure  */
    };
    @Override public void onSuccess(UserRecordResult result) {
        /* Respond to the success */
    };
};

for (int i = 0; i < 100; ++i) {
    ByteBuffer data = ByteBuffer.wrap("myData".getBytes("UTF-8"));
    ListenableFuture<UserRecordResult> f = kinesis.addUserRecord("myStream",
 "myPartitionKey", data);
    // If the Future is complete by the time we call addCallback, the callback will be
 invoked immediately.
    Futures.addCallback(f, myCallback);
}
```

Configuring the Kinesis Producer Library

Although the default settings should work well for most use cases, you may want to change some of the default settings to tailor the behavior of the `KinesisProducer` to your needs. An instance of the

`KinesisProducerConfiguration` class can be passed to the `KinesisProducer` constructor to do so, for example:

```
KinesisProducerConfiguration config = new KinesisProducerConfiguration()
        .setRecordMaxBufferedTime(3000)
        .setMaxConnections(1)
        .setRequestTimeout(60000)
        .setRegion("us-west-1");

final KinesisProducer kinesisProducer = new KinesisProducer(config);
```

You can also load a configuration from a properties file:

```
KinesisProducerConfiguration config =
 KinesisProducerConfiguration.fromPropertiesFile("default_config.properties");
```

You can substitute any path and file name that the user process has access to. You can additionally call set methods on the `KinesisProducerConfiguration` instance created this way to customize the config.

The properties file should specify parameters using their names in PascalCase. The names match those used in the set methods in the `KinesisProducerConfiguration` class. For example:

```
RecordMaxBufferedTime = 100
MaxConnections = 4
RequestTimeout = 6000
Region = us-west-1
```

For more information about configuration parameter usage rules and value limits, see the sample configuration properties file on GitHub.

Note that after `KinesisProducer` is initialized, changing the `KinesisProducerConfiguration` instance that was used has no further effect. `KinesisProducer` does not currently support dynamic reconfiguration.

Consumer De-aggregation

Beginning with release 1.4.0, the KCL supports automatic de-aggregation of KPL user records. Consumer application code written with previous versions of the KCL will compile without any modification after you update the KCL. However, if KPL aggregation is being used on the producer side, there is a subtlety involving checkpointing: all subrecords within an aggregated record have the same sequence number, so additional data has to be stored with the checkpoint if you need to distinguish between subrecords. This additional data is referred to as the *subsequence number*.

Migrating from Previous Versions of the KCL

You are not required to change your existing calls to do checkpointing in conjunction with aggregation. It is still guaranteed that you can retrieve all records successfully stored in Kinesis Data Streams. The KCL now provides two new checkpoint operations to support particular use cases, described below.

In the event that your existing code was written for the KCL prior to KPL support, and your checkpoint operation is called without arguments, it is equivalent to checkpointing the sequence number of the last KPL user record in the batch. If your checkpoint operation is called with a sequence number string, it is equivalent to checkpointing the given sequence number of the batch along with the implicit subsequence number 0 (zero).

Calling the new KCL checkpoint operation `checkpoint()` without any arguments is semantically equivalent to checkpointing the sequence number of the last `Record` call in the batch, along with the implicit subsequence number 0 (zero).

Calling the new KCL checkpoint operation `checkpoint(Record record)` is semantically equivalent to checkpointing the given `Record`'s sequence number along with the implicit subsequence number 0 (zero). If the `Record` call is actually a `UserRecord`, the `UserRecord` sequence number and subsequence number are checkpointed.

Calling the new KCL checkpoint operation `checkpoint(String sequenceNumber, long subSequenceNumber)` explicitly checkpoints the given sequence number along with the given subsequence number.

In any of these cases, after the checkpoint is stored in the Amazon DynamoDB checkpoint table, the KCL can correctly resume retrieving records even when the application crashes and restarts. If more records are contained within the sequence, retrieval occurs starting with the next subsequence number record within the record with the most recently checkpointed sequence number. If the most recent checkpoint included the very last subsequence number of the previous sequence number record, retrieval occurs starting with the record with the next sequence number.

The next section discusses details of sequence and subsequence checkpointing for consumers that need to avoid skipping and duplication of records. If skipping (or duplication) of records when stopping and restarting your consumer's record processing is not important, you can run your existing code with no modification.

KCL Extensions for KPL De-aggregation

As previously discussed, KPL de-aggregation can involve subsequence checkpointing. To facilitate using subsequence checkpointing, a `UserRecord` class has been added to the KCL:

```
public class UserRecord extends Record {
    public long getSubSequenceNumber() {
    /* ... */
    }
    @Override
    public int hashCode() {
    /* contract-satisfying implementation */
    }
    @Override
    public boolean equals(Object obj) {
    /* contract-satisfying implementation */
    }
}
```

This class is now used instead of `Record`. This does not break existing code because it is a subclass of `Record`. The `UserRecord` class represents both actual subrecords and standard, non-aggregated records. Non-aggregated records can be thought of as aggregated records with exactly one subrecord.

In addition, two new operations are added to`IRecordProcessorCheckpointer`:

```
public void checkpoint(Record record);
public void checkpoint(String sequenceNumber, long subSequenceNumber);
```

To begin using subsequence number checkpointing, you can perform the following conversion. Change the following form code:

```
checkpointer.checkpoint(record.getSequenceNumber());
```

New form code:

```
checkpointer.checkpoint(record);
```

We recommend that you use the `checkpoint(Record record)` form for subsequence checkpointing. However, if you are already storing `sequenceNumbers` in strings to use for checkpointing, you should now also store `subSequenceNumber`, as shown in the following example:

```
String sequenceNumber = record.getSequenceNumber();
long subSequenceNumber = ((UserRecord) record).getSubSequenceNumber();  // ... do other
 processing
checkpointer.checkpoint(sequenceNumber, subSequenceNumber);
```

The cast from `Record`to`UserRecord` always succeeds because the implementation always uses `UserRecord` under the hood. Unless there is a need to perform arithmetic on the sequence numbers, this approach is not recommended.

While processing KPL user records, the KCL writes the subsequence number into Amazon DynamoDB as an extra field for each row. Previous versions of the KCL used `AFTER_SEQUENCE_NUMBER` to fetch records when resuming checkpoints. The current KCL with KPL support uses `AT_SEQUENCE_NUMBER` instead. When the record at the checkpointed sequence number is retrieved, the checkpointed subsequence number is checked, and subrecords are dropped as appropriate (which may be all of them, if the last subrecord is the one checkpointed). Again, non-aggregated records can be thought of as aggregated records with a single subrecord, so the same algorithm works for both aggregated and non-aggregated records.

Using GetRecords Directly

You can also choose not to use the KCL but instead invoke the API operation `GetRecords` directly to retrieve Kinesis Data Streams records. To unpack these retrieved records into your original KPL user records, call one of the following static operations in `UserRecord.java`:

```
public static List<Record> deaggregate(List<Record> records)

public static List<UserRecord> deaggregate(List<UserRecord> records, BigInteger
 startingHashKey, BigInteger endingHashKey)
```

The first operation uses the default value `0` (zero) for `startingHashKey` and the default value `2^128 -1` for `endingHashKey`.

Each of these operations de-aggregates the given list of Kinesis Data Streams records into a list of KPL user records. Any KPL user records whose explicit hash key or partition key falls outside the range of the `startingHashKey` (inclusive) and the `endingHashKey` (inclusive) are discarded from the returned list of records.

Using the KPL with Kinesis Data Firehose

If you use the Kinesis Producer Library (KPL) to write data to a Kinesis data stream, you can use aggregation to combine the records that you write to that Kinesis data stream. If you then use that data stream as a source for your Kinesis Data Firehose delivery stream, Kinesis Data Firehose de-aggregates the records before it delivers them to the destination. If you configure your delivery stream to transform the data, Kinesis Data Firehose de-aggregates the records before it delivers them to AWS Lambda. For more information, see Writing to Kinesis Data Firehose Using Kinesis Data Streams.

Using the KPL with the AWS Glue Schema Registry

You can integrate your Kinesis data streams with the AWS Glue schema registry. The AWS Glue schema registry allows you to centrally discover, control, and evolve schemas, while ensuring data produced is

continuously validated by a registered schema. A schema defines the structure and format of a data record. A schema is a versioned specification for reliable data publication, consumption, or storage. The AWS Glue Schema Registry enables you to improve end-to-end data quality and data governance within your streaming applications. For more information, see AWS Glue Schema Registry. One of the ways to set up this integration is through the KPL and Kinesis Client Library (KCL) libraries in Java.

Important
Currently, Kinesis Data Streams and AWS Glue schema registry integration is only supported for the Kinesis data streams that use KPL producers implemented in Java. Multi-language support is not provided.

For detailed instructions on how to set up integration of Kinesis Data Streams with Schema Registry using the KPL, see the "Interacting with Data Using the KPL/KCL Libraries" section in Use Case: Integrating Amazon Kinesis Data Streams with the AWS Glue Schema Registry.

KPL Proxy Configuration

For applications that cannot directly connect to the internet, all AWS SDK clients support the use of HTTP or HTTPS proxies. In a typical enterprise environment, all outbound network traffic has to go through proxy servers. If your application uses Kinesis Producer Library (KPL) to collect and send data to AWS in an environment that uses proxy servers, your application will require KPL proxy configuration. KPL is a high level library built on top of the AWS Kinesis SDK. It is split into a native process and a wrapper. The native process performs all of the jobs of processing and sending records, while the wrapper manages the native process and communicates with it. For more information, see Implementing Efficient and Reliable Producers with the Amazon Kinesis Producer Library.

The wrapper is written in Java and the native process is written in C++ with the use of Kinesis SDK. KPL version 0.14.7 and higher now supports proxy configuration in the Java wrapper which can pass all proxy configurations to the native process. For more information, see https://github.com/awslabs/amazon-kinesis-producer/releases/tag/v0.14.7.

You can use the following code to add proxy configurations to your KPL applications.

```
KinesisProducerConfiguration configuration = new KinesisProducerConfiguration();
// Next 4 lines used to configure proxy
configuration.setProxyHost("10.0.0.0"); // required
configuration.setProxyPort(3128); // default port is set to 443
configuration.setProxyUserName("username"); // no default
configuration.setProxyPassword("password"); // no default

KinesisProducer kinesisProducer = new KinesisProducer(configuration);
```

Developing Producers Using the Amazon Kinesis Data Streams API with the AWS SDK for Java

You can develop producers using the Amazon Kinesis Data Streams API with the AWS SDK for Java. If you are new to Kinesis Data Streams, start by becoming familiar with the concepts and terminology presented in What Is Amazon Kinesis Data Streams? (p. 1) and Getting Started with Amazon Kinesis Data Streams (p. 14).

These examples discuss the Kinesis Data Streams API and use the AWS SDK for Java to add (put) data to a stream. However, for most use cases, you should prefer the Kinesis Data Streams KPL library. For more information, see Developing Producers Using the Amazon Kinesis Producer Library (p. 84).

The Java example code in this chapter demonstrates how to perform basic Kinesis Data Streams API operations, and is divided up logically by operation type. These examples do not represent production-ready code, in that they do not check for all possible exceptions, or account for all possible security or performance considerations. Also, you can call the Kinesis Data Streams API using other programming languages. For more information about all available AWS SDKs, see Start Developing with Amazon Web Services.

Each task has prerequisites; for example, you cannot add data to a stream until you have created a stream, which requires you to create a client . For more information, see Creating and Managing Streams (p. 65).

Topics

Adding Data to a Stream

Once a stream is created, you can add data to it in the form of records. A record is a data structure that contains the data to be processed in the form of a data blob. After you store the data in the record, Kinesis Data Streams does not inspect, interpret, or change the data in any way. Each record also has an associated sequence number and partition key.

There are two different operations in the Kinesis Data Streams API that add data to a stream, `PutRecords` and `PutRecord`. The `PutRecords` operation sends multiple records to your stream per HTTP request, and the singular `PutRecord` operation sends records to your stream one at a time (a separate HTTP request is required for each record). You should prefer using `PutRecords` for most applications because it will achieve higher throughput per data producer. For more information about each of these operations, see the separate subsections below.

Topics

Always keep in mind that, as your source application is adding data to the stream using the Kinesis Data Streams API, there are most likely one or more consumer applications that are simultaneously processing data off the stream. For information about how consumers get data using the Kinesis Data Streams API, see Getting Data from a Stream (p. 150).

> **Important**
> Changing the Data Retention Period (p. 80)

Adding Multiple Records with PutRecords

The `PutRecords` operation sends multiple records to Kinesis Data Streams in a single request. By using `PutRecords`, producers can achieve higher throughput when sending data to their Kinesis data stream. Each `PutRecords` request can support up to 500 records. Each record in the request can be as large as 1 MB, up to a limit of 5 MB for the entire request, including partition keys. As with the single `PutRecord` operation described below, `PutRecords` uses sequence numbers and partition keys. However, the `PutRecord` parameter `SequenceNumberForOrdering` is not included in a `PutRecords` call. The `PutRecords` operation attempts to process all records in the natural order of the request.

Each data record has a unique sequence number. The sequence number is assigned by Kinesis Data Streams after you call `client.putRecords` to add the data records to the stream. Sequence numbers for the same partition key generally increase over time; the longer the time period between `PutRecords` requests, the larger the sequence numbers become.

Note
Sequence numbers cannot be used as indexes to sets of data within the same stream. To logically separate sets of data, use partition keys or create a separate stream for each data set.

A `PutRecords` request can include records with different partition keys. The scope of the request is a stream; each request may include any combination of partition keys and records up to the request limits. Requests made with many different partition keys to streams with many different shards are generally faster than requests with a small number of partition keys to a small number of shards. The number of partition keys should be much larger than the number of shards to reduce latency and maximize throughput.

PutRecords Example

The following code creates 100 data records with sequential partition keys and puts them in a stream called `DataStream`.

```
        AmazonKinesisClientBuilder clientBuilder = AmazonKinesisClientBuilder.standard();

        clientBuilder.setRegion(regionName);
        clientBuilder.setCredentials(credentialsProvider);
        clientBuilder.setClientConfiguration(config);

        AmazonKinesis kinesisClient = clientBuilder.build();

        PutRecordsRequest putRecordsRequest  = new PutRecordsRequest();
        putRecordsRequest.setStreamName(streamName);
        List <PutRecordsRequestEntry> putRecordsRequestEntryList  = new ArrayList<>();
        for (int i = 0; i < 100; i++) {
            PutRecordsRequestEntry putRecordsRequestEntry  = new PutRecordsRequestEntry();
            putRecordsRequestEntry.setData(ByteBuffer.wrap(String.valueOf(i).getBytes()));
            putRecordsRequestEntry.setPartitionKey(String.format("partitionKey-%d", i));
            putRecordsRequestEntryList.add(putRecordsRequestEntry);
        }

        putRecordsRequest.setRecords(putRecordsRequestEntryList);
        PutRecordsResult putRecordsResult  = kinesisClient.putRecords(putRecordsRequest);
        System.out.println("Put Result" + putRecordsResult);
```

The `PutRecords` response includes an array of response `Records`. Each record in the response array directly correlates with a record in the request array using natural ordering, from the top to the bottom of the request and response. The response `Records` array always includes the same number of records as the request array.

Handling Failures When Using PutRecords

By default, failure of individual records within a request does not stop the processing of subsequent records in a `PutRecords` request. This means that a response `Records` array includes both successfully and unsuccessfully processed records. You must detect unsuccessfully processed records and include them in a subsequent call.

Successful records include `SequenceNumber` and `ShardID` values, and unsuccessful records include `ErrorCode` and `ErrorMessage` values. The `ErrorCode` parameter reflects the type of error and can be one of the following values: `ProvisionedThroughputExceededException` or `InternalFailure`. `ErrorMessage` provides more detailed information about the `ProvisionedThroughputExceededException` exception including the account ID, stream name, and shard ID of the record that was throttled. The example below has three records in a `PutRecords` request. The second record fails and is reflected in the response.

Example PutRecords Request Syntax

```
{
```

```
    "Records": [
        {
     "Data": "XzxkYXRhPl8w",
     "PartitionKey": "partitionKey1"
        },
        {
     "Data": "AbceddeRFfg12asd",
     "PartitionKey": "partitionKey1"
        },
        {
     "Data": "KFpcd98*7nd1",
     "PartitionKey": "partitionKey3"
        }
    ],
    "StreamName": "myStream"
}
```

Example PutRecords Response Syntax

```
{
    "FailedRecordCount”: 1,
    "Records": [
        {
     "SequenceNumber": "21269319989900637946712965403778482371",
     "ShardId": "shardId-000000000001"

        },
        {
     “ErrorCode”:”ProvisionedThroughputExceededException”,
     “ErrorMessage": "Rate exceeded for shard shardId-000000000001 in stream
 exampleStreamName under account 111111111111."

        },
        {
     "SequenceNumber": "21269319989999637946712965403778482985",
     "ShardId": "shardId-000000000002"
        }
    ]
}
```

Records that were unsuccessfully processed can be included in subsequent `PutRecords` requests. First, check the `FailedRecordCount` parameter in the `putRecordsResult` to confirm if there are failed records in the request. If so, each `putRecordsEntry` that has an `ErrorCode` that is not `null` should be added to a subsequent request. For an example of this type of handler, refer to the following code.

Example PutRecords failure handler

```
PutRecordsRequest putRecordsRequest = new PutRecordsRequest();
putRecordsRequest.setStreamName(myStreamName);
List<PutRecordsRequestEntry> putRecordsRequestEntryList = new ArrayList<>();
for (int j = 0; j < 100; j++) {
    PutRecordsRequestEntry putRecordsRequestEntry = new PutRecordsRequestEntry();
    putRecordsRequestEntry.setData(ByteBuffer.wrap(String.valueOf(j).getBytes()));
    putRecordsRequestEntry.setPartitionKey(String.format("partitionKey-%d", j));
    putRecordsRequestEntryList.add(putRecordsRequestEntry);
}

putRecordsRequest.setRecords(putRecordsRequestEntryList);
PutRecordsResult putRecordsResult = amazonKinesisClient.putRecords(putRecordsRequest);

while (putRecordsResult.getFailedRecordCount() > 0) {
    final List<PutRecordsRequestEntry> failedRecordsList = new ArrayList<>();
```

```
    final List<PutRecordsResultEntry> putRecordsResultEntryList =
 putRecordsResult.getRecords();
    for (int i = 0; i < putRecordsResultEntryList.size(); i++) {
        final PutRecordsRequestEntry putRecordRequestEntry =
 putRecordsRequestEntryList.get(i);
        final PutRecordsResultEntry putRecordsResultEntry =
 putRecordsResultEntryList.get(i);
        if (putRecordsResultEntry.getErrorCode() != null) {
            failedRecordsList.add(putRecordRequestEntry);
        }
    }
    putRecordsRequestEntryList = failedRecordsList;
    putRecordsRequest.setRecords(putRecordsRequestEntryList);
    putRecordsResult = amazonKinesisClient.putRecords(putRecordsRequest);
}
```

Adding a Single Record with PutRecord

Each call to `PutRecord` operates on a single record. Prefer the `PutRecords` operation described in Adding Multiple Records with PutRecords (p. 96) unless your application specifically needs to always send single records per request, or some other reason `PutRecords` can't be used.

Each data record has a unique sequence number. The sequence number is assigned by Kinesis Data Streams after you call `client.putRecord` to add the data record to the stream. Sequence numbers for the same partition key generally increase over time; the longer the time period between `PutRecord` requests, the larger the sequence numbers become.

When puts occur in quick succession, the returned sequence numbers are not guaranteed to increase because the put operations appear essentially as simultaneous to Kinesis Data Streams. To guarantee strictly increasing sequence numbers for the same partition key, use the `SequenceNumberForOrdering` parameter, as shown in the PutRecord Example (p. 99) code sample.

Whether or not you use `SequenceNumberForOrdering`, records that Kinesis Data Streams receives through a `GetRecords` call are strictly ordered by sequence number.

> **Note**
> Sequence numbers cannot be used as indexes to sets of data within the same stream. To logically separate sets of data, use partition keys or create a separate stream for each data set.

A partition key is used to group data within the stream. A data record is assigned to a shard within the stream based on its partition key. Specifically, Kinesis Data Streams uses the partition key as input to a hash function that maps the partition key (and associated data) to a specific shard.

As a result of this hashing mechanism, all data records with the same partition key map to the same shard within the stream. However, if the number of partition keys exceeds the number of shards, some shards necessarily contain records with different partition keys. From a design standpoint, to ensure that all your shards are well utilized, the number of shards (specified by the `setShardCount` method of `CreateStreamRequest`) should be substantially less than the number of unique partition keys, and the amount of data flowing to a single partition key should be substantially less than the capacity of the shard.

PutRecord Example

The following code creates ten data records, distributed across two partition keys, and puts them in a stream called `myStreamName`.

```
for (int j = 0; j < 10; j++)
{
  PutRecordRequest putRecordRequest = new PutRecordRequest();
  putRecordRequest.setStreamName( myStreamName );
```

```
  putRecordRequest.setData(ByteBuffer.wrap( String.format( "testData-%d",
 j ).getBytes() ));
  putRecordRequest.setPartitionKey( String.format( "partitionKey-%d", j/5 ));
  putRecordRequest.setSequenceNumberForOrdering( sequenceNumberOfPreviousRecord );
  PutRecordResult putRecordResult = client.putRecord( putRecordRequest );
  sequenceNumberOfPreviousRecord = putRecordResult.getSequenceNumber();
}
```

The preceding code sample uses `setSequenceNumberForOrdering` to guarantee strictly increasing ordering within each partition key. To use this parameter effectively, set the `SequenceNumberForOrdering` of the current record (record *n*) to the sequence number of the preceding record (record *n-1*). To get the sequence number of a record that has been added to the stream, call `getSequenceNumber` on the result of `putRecord`.

The `SequenceNumberForOrdering` parameter ensures strictly increasing sequence numbers for the same partition key. `SequenceNumberForOrdering` does not provide ordering of records across multiple partition keys.

Interacting with Data Using the AWS Glue Schema Registry

You can integrate your Kinesis data streams with the AWS Glue schema registry. The AWS Glue schema registry allows you to centrally discover, control, and evolve schemas, while ensuring data produced is continuously validated by a registered schema. A schema defines the structure and format of a data record. A schema is a versioned specification for reliable data publication, consumption, or storage. The AWS Glue Schema Registry enables you to improve end-to-end data quality and data governance within your streaming applications. For more information, see AWS Glue Schema Registry. One of the ways to set up this integration is through the `PutRecords` and `PutRecord` Kinesis Data Streams APIs available in the AWS Java SDK.

For detailed instructions on how to set up integration of Kinesis Data Streams with Schema Registry using the PutRecords and PutRecord Kinesis Data Streams APIs, see the "Interacting with Data Using the Kinesis Data Streams APIs" section in Use Case: Integrating Amazon Kinesis Data Streams with the AWS Glue Schema Registry.

Writing to Amazon Kinesis Data Streams Using Kinesis Agent

Kinesis Agent is a stand-alone Java software application that offers an easy way to collect and send data to Kinesis Data Streams. The agent continuously monitors a set of files and sends new data to your stream. The agent handles file rotation, checkpointing, and retry upon failures. It delivers all of your data in a reliable, timely, and simple manner. It also emits Amazon CloudWatch metrics to help you better monitor and troubleshoot the streaming process.

By default, records are parsed from each file based on the newline (`'\n'`) character. However, the agent can also be configured to parse multi-line records (see Agent Configuration Settings (p. 102)).

You can install the agent on Linux-based server environments such as web servers, log servers, and database servers. After installing the agent, configure it by specifying the files to monitor and the stream for the data. After the agent is configured, it durably collects data from the files and reliably sends it to the stream.

Topics

Prerequisites

- Your operating system must be either Amazon Linux AMI with version 2015.09 or later, or Red Hat Enterprise Linux version 7 or later.
- If you are using Amazon EC2 to run your agent, launch your EC2 instance.
- Manage your AWS credentials using one of the following methods:
  - Specify an IAM role when you launch your EC2 instance.
  - Specify AWS credentials when you configure the agent (see awsAccessKeyId (p.) and awsSecretAccessKey (p.)).
  - Edit `/etc/sysconfig/aws-kinesis-agent` to specify your region and AWS access keys.
  - If your EC2 instance is in a different AWS account, create an IAM role to provide access to the Kinesis Data Streams service, and specify that role when you configure the agent (see assumeRoleARN (p.) and assumeRoleExternalId (p.)). Use one of the previous methods to specify the AWS credentials of a user in the other account who has permission to assume this role.
- The IAM role or AWS credentials that you specify must have permission to perform the Kinesis Data Streams PutRecords operation for the agent to send data to your stream. If you enable CloudWatch monitoring for the agent, permission to perform the CloudWatch PutMetricData operation is also needed. For more information, see Controlling Access to Amazon Kinesis Data Streams Resources Using IAM (p. 216), Monitoring Kinesis Data Streams Agent Health with Amazon CloudWatch (p. 190), and CloudWatch Access Control.

Download and Install the Agent

First, connect to your instance. For more information, see Connect to Your Instance in the *Amazon EC2 User Guide for Linux Instances*. If you have trouble connecting, see Troubleshooting Connecting to Your Instance in the *Amazon EC2 User Guide for Linux Instances*.

To set up the agent using the Amazon Linux AMI

Use the following command to download and install the agent:

```
sudo yum install -y aws-kinesis-agent
```

To set up the agent using Red Hat Enterprise Linux

Use the following command to download and install the agent:

```
sudo yum install -y https://s3.amazonaws.com/streaming-data-agent/aws-kinesis-agent-
latest.amzn1.noarch.rpm
```

To set up the agent using GitHub

1. Download the agent from awlabs/amazon-kinesis-agent.
2. Install the agent by navigating to the download directory and running the following command:

```
sudo ./setup --install
```

Configure and Start the Agent

To configure and start the agent

1. Open and edit the configuration file (as superuser if using default file access permissions): `/etc/aws-kinesis/agent.json`

   In this configuration file, specify the files (`"filePattern"`) from which the agent collects data, and the name of the stream (`"kinesisStream"`) to which the agent sends data. Note that the file name is a pattern, and the agent recognizes file rotations. You can rotate files or create new files no more than once per second. The agent uses the file creation timestamp to determine which files to track and tail into your stream; creating new files or rotating files more frequently than once per second does not allow the agent to differentiate properly between them.

   ```
   {
      "flows": [
           {
               "filePattern": "/tmp/app.log*",
               "kinesisStream": "yourkinesisstream"
           }
      ]
   }
   ```

2. Start the agent manually:

   ```
   sudo service aws-kinesis-agent start
   ```

3. (Optional) Configure the agent to start on system startup:

   ```
   sudo chkconfig aws-kinesis-agent on
   ```

The agent is now running as a system service in the background. It continuously monitors the specified files and sends data to the specified stream. Agent activity is logged in `/var/log/aws-kinesis-agent/aws-kinesis-agent.log`.

Agent Configuration Settings

The agent supports the two mandatory configuration settings, `filePattern` and `kinesisStream`, plus optional configuration settings for additional features. You can specify both mandatory and optional configuration in `/etc/aws-kinesis/agent.json`.

Whenever you change the configuration file, you must stop and start the agent, using the following commands:

```
sudo service aws-kinesis-agent stop
sudo service aws-kinesis-agent start
```

Alternatively, you could use the following command:

```
sudo service aws-kinesis-agent restart
```

The following are the general configuration settings.

Configuration Setting	Description
`assumeRoleARN`	The ARN of the role to be assumed by the user. For more information, see Delegate Access Across AWS Accounts Using IAM Roles in the *IAM User Guide*.
`assumeRoleExternalId`	An optional identifier that determines who can assume the role. For more information, see How to Use an External ID in the *IAM User Guide*.
`awsAccessKeyId`	AWS access key ID that overrides the default credentials. This setting takes precedence over all other credential providers.
`awsSecretAccessKey`	AWS secret key that overrides the default credentials. This setting takes precedence over all other credential providers.
`cloudwatch.emitMetrics`	Enables the agent to emit metrics to CloudWatch if set (true). Default: true
`cloudwatch.endpoint`	The regional endpoint for CloudWatch. Default: `monitoring.us-east-1.amazonaws.com`
`kinesis.endpoint`	The regional endpoint for Kinesis Data Streams. Default: `kinesis.us-east-1.amazonaws.com`

The following are the flow configuration settings.

Configuration Setting	Description
`dataProcessingOptions`	The list of processing options applied to each parsed record before it is sent to the stream. The processing options are performed in the specified order. For more information, see Use the Agent to Pre-process Data (p. 105).
`kinesisStream`	[Required] The name of the stream.
`filePattern`	[Required] A glob for the files that must be monitored by the agent. Any file that matches this pattern is picked up by the agent automatically and monitored. For all files matching this pattern, read permission must be granted to `aws-kinesis-agent-user`. For the directory containing the files, read and execute permissions must be granted to `aws-kinesis-agent-user`.
`initialPosition`	The initial position from which the file started to be parsed. Valid values are `START_OF_FILE` and `END_OF_FILE`. Default: `END_OF_FILE`
`maxBufferAgeMillis`	The maximum time, in milliseconds, for which the agent buffers data before sending it to the stream. Value range: 1,000 to 900,000 (1 second to 15 minutes) Default: 60,000 (1 minute)
`maxBufferSizeBytes`	The maximum size, in bytes, for which the agent buffers data before sending it to the stream.

Configuration Setting	Description
	Value range: 1 to 4,194,304 (4 MB) Default: 4,194,304 (4 MB)
`maxBufferSizeRecords`	The maximum number of records for which the agent buffers data before sending it to the stream. Value range: 1 to 500 Default: 500
`minTimeBetweenFilePollsMillis`	The minimum interval, in milliseconds, at which the agent polls and parses the monitored files for new data. Value range: 1 or more Default: 100
`multiLineStartPattern`	The pattern for identifying the start of a record. A record is made of a line that matches the pattern and any following lines that don't match the pattern. The valid values are regular expressions. By default, each new line in the log files is parsed as one record.
`partitionKeyOption`	The method for generating the partition key. Valid values are `RANDOM` (randomonly generated integer) and `DETERMINISTIC` (a hash value computed from the data). Default: `RANDOM`
`skipHeaderLines`	The number of lines for the agent to skip parsing at the beginning of monitored files. Value range: 0 or more Default: 0 (zero)
`truncatedRecordTerminator`	The string that the agent uses to truncate a parsed record when the record size exceeds the Kinesis Data Streams record size limit. (1,000 KB) Default: `'\n'` (newline)

Monitor Multiple File Directories and Write to Multiple Streams

By specifying multiple flow configuration settings, you can configure the agent to monitor multiple file directories and send data to multiple streams. In the following configuration example, the agent monitors two file directories and sends data to an Kinesis stream and a Kinesis Data Firehose delivery stream respectively. Note that you can specify different endpoints for Kinesis Data Streams and Kinesis Data Firehose so that your Kinesis stream and Kinesis Data Firehose delivery stream don't need to be in the same region.

```
{
    "cloudwatch.emitMetrics": true,
    "kinesis.endpoint": "https://your/kinesis/endpoint",
    "firehose.endpoint": "https://your/firehose/endpoint",
    "flows": [
```

```
        {
            "filePattern": "/tmp/app1.log*",
            "kinesisStream": "yourkinesisstream"
        },
        {
            "filePattern": "/tmp/app2.log*",
            "deliveryStream": "yourfirehosedeliverystream"
        }
    ]
}
```

For more detailed information about using the agent with Kinesis Data Firehose, see Writing to Amazon Kinesis Data Firehose with Kinesis Agent.

Use the Agent to Pre-process Data

The agent can pre-process the records parsed from monitored files before sending them to your stream. You can enable this feature by adding the `dataProcessingOptions` configuration setting to your file flow. One or more processing options can be added and they will be performed in the specified order.

The agent supports the following processing options listed. Because the agent is open-source, you can further develop and extend its processing options. You can download the agent from Kinesis Agent.

Processing Options

`SINGLELINE`

Converts a multi-line record to a single line record by removing newline characters, leading spaces, and trailing spaces.

```
{
    "optionName": "SINGLELINE"
}
```

`CSVTOJSON`

Converts a record from delimiter separated format to JSON format.

```
{
    "optionName": "CSVTOJSON",
    "customFieldNames": [ "field1", "field2", ... ],
    "delimiter": "yourdelimiter"
}
```

`customFieldNames`

[Required] The field names used as keys in each JSON key value pair. For example, if you specify `["f1", "f2"]`, the record "v1, v2" will be converted to `{"f1":"v1","f2":"v2"}`.

`delimiter`

The string used as the delimiter in the record. The default is a comma (,).

`LOGTOJSON`

Converts a record from a log format to JSON format. The supported log formats are **Apache Common Log**, **Apache Combined Log**, **Apache Error Log**, and **RFC3164 Syslog**.

```
{
    "optionName": "LOGTOJSON",
    "logFormat": "logformat",
    "matchPattern": "yourregexpattern",
```

```
    "customFieldNames": [ "field1", "field2", … ]
}
```

`logFormat`

[Required] The log entry format. The following are possible values:

- `COMMONAPACHELOG` — The Apache Common Log format. Each log entry has the following pattern by default: `"%{host} %{ident} %{authuser} [%{datetime}] \"%{request}\" %{response} %{bytes}"`.
- `COMBINEDAPACHELOG` — The Apache Combined Log format. Each log entry has the following pattern by default: `"%{host} %{ident} %{authuser} [%{datetime}] \"%{request}\" %{response} %{bytes} %{referrer} %{agent}"`.
- `APACHEERRORLOG` — The Apache Error Log format. Each log entry has the following pattern by default: `"[%{timestamp}] [%{module}:%{severity}] [pid %{processid}:tid %{threadid}] [client: %{client}] %{message}"`.
- `SYSLOG` — The RFC3164 Syslog format. Each log entry has the following pattern by default: `"%{timestamp} %{hostname} %{program}[%{processid}]: %{message}"`.

`matchPattern`

The regular expression pattern used to extract values from log entries. This setting is used if your log entry is not in one of the predefined log formats. If this setting is used, you must also specify `customFieldNames`.

`customFieldNames`

The custom field names used as keys in each JSON key value pair. You can use this setting to define field names for values extracted from `matchPattern`, or override the default field names of predefined log formats.

Example : LOGTOJSON Configuration

Here is one example of a `LOGTOJSON` configuration for an Apache Common Log entry converted to JSON format:

```
{
    "optionName": "LOGTOJSON",
    "logFormat": "COMMONAPACHELOG"
}
```

Before conversion:

```
64.242.88.10 - - [07/Mar/2004:16:10:02 -0800] "GET /mailman/listinfo/hsdivision HTTP/1.1"
 200 6291
```

After conversion:

```
{"host":"64.242.88.10","ident":null,"authuser":null,"datetime":"07/
Mar/2004:16:10:02 -0800","request":"GET /mailman/listinfo/hsdivision
 HTTP/1.1","response":"200","bytes":"6291"}
```

Example : LOGTOJSON Configuration With Custom Fields

Here is another example `LOGTOJSON` configuration:

```
{
    "optionName": "LOGTOJSON",
    "logFormat": "COMMONAPACHELOG",
```

```
    "customFieldNames": ["f1", "f2", "f3", "f4", "f5", "f6", "f7"]
}
```

With this configuration setting, the same Apache Common Log entry from the previous example is converted to JSON format as follows:

```
{"f1":"64.242.88.10","f2":null,"f3":null,"f4":"07/Mar/2004:16:10:02 -0800","f5":"GET /
mailman/listinfo/hsdivision HTTP/1.1","f6":"200","f7":"6291"}
```

Example : Convert Apache Common Log Entry

The following flow configuration converts an Apache Common Log entry to a single line record in JSON format:

```
{
    "flows": [
        {
            "filePattern": "/tmp/app.log*",
            "kinesisStream": "my-stream",
            "dataProcessingOptions": [
                {
                    "optionName": "LOGTOJSON",
                    "logFormat": "COMMONAPACHELOG"
                }
            ]
        }
    ]
}
```

Example : Convert Multi-Line Records

The following flow configuration parses multi-line records whose first line starts with "`[SEQUENCE=`". Each record is first converted to a single line record. Then, values are extracted from the record based on a tab delimiter. Extracted values are mapped to specified `customFieldNames` values to form a single-line record in JSON format.

```
{
    "flows": [
        {
            "filePattern": "/tmp/app.log*",
            "kinesisStream": "my-stream",
            "multiLineStartPattern": "\\[SEQUENCE=",
            "dataProcessingOptions": [
                {
                    "optionName": "SINGLELINE"
                },
                {
                    "optionName": "CSVTOJSON",
                    "customFieldNames": [ "field1", "field2", "field3" ],
                    "delimiter": "\\t"
                }
            ]
        }
    ]
}
```

Example : LOGTOJSON Configuration with Match Pattern

Here is one example of a `LOGTOJSON` configuration for an Apache Common Log entry converted to JSON format, with the last field (bytes) omitted:

```
{
    "optionName": "LOGTOJSON",
    "logFormat": "COMMONAPACHELOG",
    "matchPattern": "^([\\d.]+) (\\S+) (\\S+) \\[([\\w:/]+\\s[+\\-]\\d{4})\\] \"(.+?)\" (\
\d{3})",
    "customFieldNames": ["host", "ident", "authuser", "datetime", "request", "response"]
}
```

Before conversion:

```
123.45.67.89 - - [27/Oct/2000:09:27:09 -0400] "GET /java/javaResources.html HTTP/1.0" 200
```

After conversion:

```
{"host":"123.45.67.89","ident":null,"authuser":null,"datetime":"27/Oct/2000:09:27:09
 -0400","request":"GET /java/javaResources.html HTTP/1.0","response":"200"}
```

Agent CLI Commands

Automatically start the agent on system startup:

```
sudo chkconfig aws-kinesis-agent on
```

Check the status of the agent:

```
sudo service aws-kinesis-agent status
```

Stop the agent:

```
sudo service aws-kinesis-agent stop
```

Read the agent's log file from this location:

```
/var/log/aws-kinesis-agent/aws-kinesis-agent.log
```

Uninstall the agent:

```
sudo yum remove aws-kinesis-agent
```

Troubleshooting Amazon Kinesis Data Streams Producers

The following sections offer solutions to some common problems you may find while working with Amazon Kinesis Data Streams producers.

- Producer Application is Writing at a Slower Rate Than Expected (p. 109)
- Unauthorized KMS master key permission error (p. 110)
- Common issues, questions, and troubleshooting ideas for producers (p. 110)

Producer Application is Writing at a Slower Rate Than Expected

The most common reasons for write throughput being slower than expected are as follows.

- Service Limits Exceeded (p. 109)
- Producer Optimization (p. 109)

Service Limits Exceeded

To find out if service limits are being exceeded, check to see if your producer is throwing throughput exceptions from the service, and validate what API operations are being throttled. Keep in mind that there are different limits based on the call, see Quotas and Limits (p. 7). For example, in addition to the shard-level limits for writes and reads that are most commonly known, there are the following stream-level limits:

- CreateStream
- DeleteStream
- ListStreams
- GetShardIterator
- MergeShards
- DescribeStream
- DescribeStreamSummary

The operations `CreateStream`, `DeleteStream`, `ListStreams`, `GetShardIterator`, and `MergeShards` are limited to 5 calls per second. The `DescribeStream` operation is limited to 10 calls per second. The `DescribeStreamSummary` operation is limited to 20 calls per second.

If these calls aren't the issue, make sure you've selected a partition key that allows you to distribute *put* operations evenly across all shards, and that you don't have a particular partition key that's bumping into the service limits when the rest are not. This requires that you measure peak throughput and take into account the number of shards in your stream. For more information about managing streams, see Creating and Managing Streams (p. 65).

> **Tip**
> Remember to round up to the nearest kilobyte for throughput throttling calculations when using the single-record operation PutRecord, while the multi-record operation PutRecords rounds on the cumulative sum of the records in each call. For example, a `PutRecords` request with 600 records that are 1.1 KB in size will not get throttled.

Producer Optimization

Before you begin optimizing your producer, there are some key tasks to be completed. First, identify your desired peak throughput in terms of record size and records per second. Next, rule out stream capacity as the limiting factor (Service Limits Exceeded (p. 109)). If you've ruled out stream capacity, use the following troubleshooting tips and optimization guidelines for the two common types of producers.

Large Producer

A large producer is usually running from an on-premises server or Amazon EC2 instance. Customers who need higher throughput from a large producer typically care about per-record latency. Strategies for dealing with latency include the following: If the customer can micro-batch/buffer records, use the Kinesis Producer Library (which has advanced aggregation logic), the multi-record operation PutRecords,

or aggregate records into a larger file before using the single-record operation PutRecord. If you are unable to batch/buffer, use multiple threads to write to the Kinesis Data Streams service at the same time. The AWS SDK for Java and other SDKs include async clients that can do this with very little code.

Small Producer

A small producer is usually a mobile app, IoT device, or web client. If it's a mobile app, we recommend using the `PutRecords` operation or the Kinesis Recorder in the AWS Mobile SDKs. For more information, see AWS Mobile SDK for Android Getting Started Guide and AWS Mobile SDK for iOS Getting Started Guide. Mobile apps must handle intermittent connections inherently and need some sort of batch put, such as `PutRecords`. If you are unable to batch for some reason, see the Large Producer information above. If your producer is a browser, the amount of data being generated is typically very small. However, you are putting the *put* operations on the critical path of the application, which we don't recommend.

Unauthorized KMS master key permission error

This error occurs when a producer application writes to an encrypted stream without permissions on the KMS master key. To assign permissions to an application to access a KMS key, see Using Key Policies in AWS KMS and Using IAM Policies with AWS KMS.

Common issues, questions, and troubleshooting ideas for producers

- Why is my Kinesis data stream returning a 500 Internal Server Error?
- How do I troubleshoot timeout errors when writing from Flink to Kinesis Data Streams?
- How do I troubleshoot throttling errors in Kinesis Data Streams?
- Why is my Kinesis data stream throttling?
- How can I put data records into a Kinesis data stream using the KPL?

Advanced Topics for Kinesis Data Streams Producers

This section discusses how to optimize your Amazon Kinesis Data Streams producers.

Topics

KPL Retries and Rate Limiting

When you add Kinesis Producer Library (KPL) user records using the KPL `addUserRecord()` operation, a record is given a time stamp and added to a buffer with a deadline set by the `RecordMaxBufferedTime` configuration parameter. This time stamp/deadline combination sets the buffer priority. Records are flushed from the buffer based on the following criteria:

- Buffer priority
- Aggregation configuration
- Collection configuration

The aggregation and collection configuration parameters affecting buffer behavior are as follows:

- `AggregationMaxCount`
- `AggregationMaxSize`
- `CollectionMaxCount`
- `CollectionMaxSize`

Records flushed are then sent to your Kinesis data stream as Amazon Kinesis Data Streams records using a call to the Kinesis Data Streams API operation `PutRecords`. The `PutRecords` operation sends requests to your stream that occasionally exhibit full or partial failures. Records that fail are automatically added back to the KPL buffer. The new deadline is set based on the minimum of these two values:

- Half the current `RecordMaxBufferedTime` configuration
- The record's time-to-live value

This strategy allows retried KPL user records to be included in subsequent Kinesis Data Streams API calls, to improve throughput and reduce complexity while enforcing the Kinesis Data Streams record's time-to-live value. There is no backoff algorithm, making this a relatively aggressive retry strategy. Spamming due to excessive retries is prevented by rate limiting, discussed in the next section.

Rate Limiting

The KPL includes a rate limiting feature, which limits per-shard throughput sent from a single producer. Rate limiting is implemented using a token bucket algorithm with separate buckets for both Kinesis Data Streams records and bytes. Each successful write to a Kinesis data stream adds a token (or multiple tokens) to each bucket, up to a certain threshold. This threshold is configurable but by default is set 50 percent higher than the actual shard limit, to allow shard saturation from a single producer.

You can lower this limit to reduce spamming due to excessive retries. However, the best practice is for each producer to retry for maximum throughput aggressively and to handle any resulting throttling determined as excessive by expanding the capacity of the stream and implementing an appropriate partition key strategy.

Considerations When Using KPL Aggregation

While the sequence number scheme of the resulting Amazon Kinesis Data Streams records remains the same, aggregation causes the indexing of Kinesis Producer Library (KPL) user records contained within an aggregated Kinesis Data Streams record to start at 0 (zero); however, as long as you do not rely on sequence numbers to uniquely identify your KPL user records, your code can ignore this, as the aggregation (of your KPL user records into a Kinesis Data Streams record) and subsequent de-aggregation (of a Kinesis Data Streams record into your KPL user records) automatically takes care of this for you. This applies whether your consumer is using the KCL or the AWS SDK. To use this aggregation functionality, you'll need to pull the Java part of the KPL into your build if your consumer is written using the API provided in the AWS SDK.

If you intend to use sequence numbers as unique identifiers for your KPL user records, we recommend that you use the contract-abiding `public int hashCode()` and `public boolean equals(Object obj)` operations provided in `Record` and `UserRecord` to enable the comparison of your KPL user records. Additionally, if you want to examine the subsequence number of your KPL user record, you can cast it to a `UserRecord` instance and retrieve its subsequence number.

For more information, see Consumer De-aggregation (p. 92).

Reading Data from Amazon Kinesis Data Streams

A *consumer* is an application that processes all data from a Kinesis data stream. When a consumer uses *enhanced fan-out*, it gets its own 2 MB/sec allotment of read throughput, allowing multiple consumers to read data from the same stream in parallel, without contending for read throughput with other consumers. To use the enhanced fan-out capability of shards, see Developing Custom Consumers with Dedicated Throughput (Enhanced Fan-Out) (p. 154).

By default, shards in a stream provide 2 MB/sec of read throughput per shard. This throughput gets shared across all the consumers that are reading from a given shard. In other words, the default 2 MB/sec of throughput per shard is fixed, even if there are multiple consumers that are reading from the shard. To use this default throughput of shards see, Developing Custom Consumers with Shared Throughput (p. 125).

The following table compares default throughput to enhanced fan-out. Message propagation delay is defined as the time taken in milliseconds for a payload sent using the payload-dispatching APIs (like PutRecord and PutRecords) to reach the consumer application through the payload-consuming APIs (like GetRecords and SubscribeToShard).

Characteristics	Unregistered Consumers without Enhanced Fan-Out	Registered Consumers with Enhanced Fan-Out
Shard Read Throughput	Fixed at a total of 2 MB/sec per shard. If there are multiple consumers reading from the same shard, they all share this throughput. The sum of the throughputs they receive from the shard doesn't exceed 2 MB/sec.	Scales as consumers register to use enhanced fan-out. Each consumer registered to use enhanced fan-out receives its own read throughput per shard, up to 2 MB/sec, independently of other consumers.
Message propagation delay	An average of around 200 ms if you have one consumer reading from the stream. This average goes up to around 1000 ms if you have five consumers.	Typically an average of 70 ms whether you have one consumer or five consumers.
Cost	N/A	There is a data retrieval cost and a consumer-shard hour cost. For more information, see Amazon Kinesis Data Streams Pricing.
Record delivery model	Pull model over HTTP using GetRecords.	Kinesis Data Streams pushes the records to you over HTTP/2 using SubscribeToShard.

Topics

Developing Consumers Using AWS Lambda

You can use an AWS Lambda function to process records in a data stream. AWS Lambda is a compute service that lets you run code without provisioning or managing servers. It executes your code only when needed and scales automatically, from a few requests per day to thousands per second. You pay only for the compute time you consume. There is no charge when your code is not running. With AWS Lambda, you can run code for virtually any type of application or backend service, all with zero administration. It runs your code on a high-availability compute infrastructure and performs all of the administration of the compute resources, including server and operating system maintenance, capacity provisioning and automatic scaling, code monitoring and logging. For more information, see Using AWS Lambda with Amazon Kinesis.

For troubleshooting information, see Why is Kinesis Data Streams trigger unable to invoke my Lambda function?

Developing Consumers Using Amazon Kinesis Data Analytics

You can use an Amazon Kinesis Data Analytics application to process and analyze data in a Kinesis stream using SQL, Java, or Scala. Kinesis Data Analytics applications can enrich data using reference sources, aggregate data over time, or use machine learning to find data anomalies. Then you can write the analysis results to another Kinesis stream, a Kinesis Data Firehose delivery stream, or a Lambda function. For more information, see the Kinesis Data Analytics Developer Guide for SQL Applications or the Kinesis Data Analytics Developer Guide for Flink Applications.

Developing Consumers Using Amazon Kinesis Data Firehose

You can use a Kinesis Data Firehose to read and process records from a Kinesis stream. Kinesis Data Firehose is a fully managed service for delivering real-time streaming data to destinations such as Amazon S3, Amazon Redshift, Amazon OpenSearch Service, and Splunk. Kinesis Data Firehose also supports any custom HTTP endpoint or HTTP endpoints owned by supported third-party service providers, including Datadog, MongoDB, and New Relic. You can also configure Kinesis Data Firehose to transform your data records and to convert the record format before delivering your data to its destination. For more information, see Writing to Kinesis Data Firehose Using Kinesis Data Streams.

Using the Kinesis Client Library

One of the methods of developing custom consumer applications that can process data from KDS data streams is to use the Kinesis Client Library (KCL).

Topics

> **Note**
> For both KCL 1.x and KCL 2.x, it is recommended that you upgrade to the latest KCL 1.x version or KCL 2.x version, depending on your usage scenario. Both KCL 1.x and KCL 2.x are regularly updated with newer releases that include the latest dependency and security patches, bug fixes, and backward-compatible new features. For more information, see https://github.com/awslabs/amazon-kinesis-client/releases.

What is the Kinesis Client Library?

KCL helps you consume and process data from a Kinesis data stream by taking care of many of the complex tasks associated with distributed computing. These include load balancing across multiple consumer application instances, responding to consumer application instance failures, checkpointing processed records, and reacting to resharding. The KCL takes care of all of these subtasks so that you can focus your efforts on writing your custom record-processing logic.

The KCL is different from the Kinesis Data Streams APIs that are available in the AWS SDKs. The Kinesis Data Streams APIs help you manage many aspects of Kinesis Data Streams, including creating streams, resharding, and putting and getting records. The KCL provides a layer of abstraction around all these subtasks, specifically so that you can focus on your consumer application's custom data processing logic. For information about the Kinesis Data Streams API, see the Amazon Kinesis API Reference.

> **Important**
> The KCL is a Java library. Support for languages other than Java is provided using a multi-language interface called the MultiLangDaemon. This daemon is Java-based and runs in the background when you are using a KCL language other than Java. For example, if you install the KCL for Python and write your consumer application entirely in Python, you still need Java installed on your system because of the MultiLangDaemon. Further, MultiLangDaemon has some default settings that you might need to customize for your use case, for example, the AWS region that it connects to. For more information about the MultiLangDaemon on GitHub, see KCL MultiLangDaemon project.

The KCL acts as an intermediary between your record processing logic and Kinesis Data Streams. The KCL performs the following tasks:

- Connects to the data stream
- Enumerates the shards within the data stream
- Uses leases to coordinates shard associations with its workers
- Instantiates a record processor for every shard it manages
- Pulls data records from the data stream
- Pushes the records to the corresponding record processor
- Checkpoints processed records
- Balances shard-worker associations (leases) when the worker instance count changes or when the data stream is resharded (shards are split or merged)

KCL Available Versions

Currently, you can use either of the following supported versions of KCL to build your custom consumer applications:

- **KCL 1.x**

  For more information, see Developing KCL 1.x Consumers (p. 125)
- **KCL 2.x**

  For more information, see Developing KCL 2.x Consumers (p. 138)

You can use either KCL 1.x or KCL 2.x to build consumer applications that use shared throughput. For more information, see Developing Custom Consumers with Shared Throughput Using KCL (p. 125).

To build consumer applications that use dedicated throughput (enhanced fan-out consumers), you can only use KCL 2.x. For more information, see Developing Custom Consumers with Dedicated Throughput (Enhanced Fan-Out) (p. 154).

For information about the differences between KCL 1.x and KCL 2.x, and instructions on how to migrate from KCL 1.x to KCL 2.x, see Migrating Consumers from KCL 1.x to KCL 2.x (p. 162).

KCL Concepts

- **KCL consumer application** – an application that is custom-built using KCL and designed to read and process records from data streams.
- **Consumer application instance** - KCL consumer applications are typically distributed, with one or more application instances running simultaneously in order to coordinate on failures and dynamically load balance data record processing.
- **Worker** – a high level class that a KCL consumer application instance uses to start processing data.

  Important
  Each KCL consumer application instance has one worker.

  The worker initializes and oversees various tasks, including syncing shard and lease information, tracking shard assignments, and processing data from the shards. A worker provides KCL with the configuration information for the consumer application, such as the name of the data stream whose data records this KCL consumer application is going to process and the AWS credentials that are needed to access this data stream. The worker also kick starts that specific KCL consumer application instance to deliver data records from the data stream to the record processors.

  Important
  In KCL 1.x this class is called **Worker**. For more information, (these are the Java KCL repositories), see https://github.com/awslabs/amazon-kinesis-client/blob/v1.x/src/main/java/com/amazonaws/services/kinesis/clientlibrary/lib/worker/Worker.java. In KCL 2.x, this class is called **Scheduler**. Scheduler's purpose in KCL 2.x is identical to Worker's purpose in KCL 1.x. For more information about the Scheduler class in KCL 2.x, see https://github.com/awslabs/amazon-kinesis-client/blob/master/amazon-kinesis-client/src/main/java/software/amazon/kinesis/coordinator/Scheduler.java.
- **Lease** – data that defines the binding between a worker and a shard. Distributed KCL consumer applications use leases to partition data record processing across a fleet of workers. At any given time, each shard of data records is bound to a particular worker by a lease identified by the **leaseKey** variable.

  By default, a worker can hold one or more leases (subject to the value of the **maxLeasesForWorker** variable) at the same time.

Important
Every worker will contend to hold all available leases for all available shards in a data stream. But only one worker will successfully hold each lease at any one time.

For example, if you have a consumer application instance A with worker A that is processing a data stream with 4 shards, worker A can hold leases to shards 1, 2, 3, and 4 at the same time. But if you have two consumer application instances: A and B with worker A and worker B, and these instances are processing a data stream with 4 shards, worker A and worker B cannot both hold the lease to shard 1 at the same time. One worker holds the lease to a particular shard until it is ready to stop processing this shard's data records or until it fails. When one worker stops holding the lease, another worker takes up and holds the lease.

For more information, (these are the Java KCL repositories), see https://github.com/awslabs/amazon-kinesis-client/blob/v1.x/src/main/java/com/amazonaws/services/kinesis/leases/impl/Lease.java for KCL 1.x and https://github.com/awslabs/amazon-kinesis-client/blob/master/amazon-kinesis-client/src/main/java/software/amazon/kinesis/leases/Lease.java for KCL 2.x.

- **Lease table** - a unique Amazon DynamoDB table that is used to keep track of the shards in a KDS data stream that are being leased and processed by the workers of the KCL consumer application. The lease table must remain in sync (within a worker and across all workers) with the latest shard information from the data stream while the KCL consumer application is running. For more information, see Using a Lease Table to Track the Shards Processed by the KCL Consumer Application (p. 116).
- **Record processor** – the logic that defines how your KCL consumer application processes the data that it gets from the data streams. At runtime, a KCL consumer application instance instantiates a worker, and this worker instantiates one record processor for every shard to which it holds a lease.

Using a Lease Table to Track the Shards Processed by the KCL Consumer Application

Topics

What Is a Lease Table

For each Amazon Kinesis Data Streams application, KCL uses a unique lease table (stored in a Amazon DynamoDB table) to keep track of the shards in a KDS data stream that are being leased and processed by the workers of the KCL consumer application.

Important
KCL uses the name of the consumer application to create the name of the lease table that this consumer application uses, therefore each consumer application name must be unique.

You can view the lease table using the Amazon DynamoDB console while the consumer application is running.

If the lease table for your KCL consumer application does not exist when the application starts up, one of the workers creates the lease table for this application.

Important
Your account is charged for the costs associated with the DynamoDB table, in addition to the costs associated with Kinesis Data Streams itself.

Each row in the lease table represents a shard that is being processed by the workers of your consumer application. If your KCL consumer application processes only one data stream, then `leaseKey` which is the hash key for the lease table is the shard ID. If you are Processing Multiple Data Streams with the same KCL 2.x for Java Consumer Application (p. 122), then the structure of the leaseKey looks like this: `account-id:StreamName:streamCreationTimestamp:ShardId`. For example, `111111111:multiStreamTest-1:12345:shardId-000000000336`.

In addition to the shard ID, each row also includes the following data:

- **checkpoint:** The most recent checkpoint sequence number for the shard. This value is unique across all shards in the data stream.
- **checkpointSubSequenceNumber:** When using the Kinesis Producer Library's aggregation feature, this is an extension to **checkpoint** that tracks individual user records within the Kinesis record.
- **leaseCounter:** Used for lease versioning so that workers can detect that their lease has been taken by another worker.
- **leaseKey:** A unique identifier for a lease. Each lease is particular to a shard in the data stream and is held by one worker at a time.
- **leaseOwner:** The worker that is holding this lease.
- **ownerSwitchesSinceCheckpoint:** How many times this lease has changed workers since the last time a checkpoint was written.
- **parentShardId:** Used to ensure that the parent shard is fully processed before processing starts on the child shards. This ensures that records are processed in the same order they were put into the stream.
- **hashrange:** Used by the `PeriodicShardSyncManager` to run periodic syncs to find missing shards in the lease table and create leases for them if required.

  Note
  This data is present in the lease table for every shard starting with KCL 1.14 and KCL 2.3. For more information about `PeriodicShardSyncManager` and periodic synchronization between leases and shards, see How a Lease Table Is Synchronized with the Shards in a KDS Data Stream (p. 118).
- **childshards:** Used by the `LeaseCleanupManager` to review the child shard's processing status and decide whether the parent shard can be deleted from the lease table.

  Note
  This data is present in the lease table for every shard starting with KCL 1.14 and KCL 2.3.
- **shardID:** The ID of the shard.

  Note
  This data is only present in the lease table if you are Processing Multiple Data Streams with the same KCL 2.x for Java Consumer Application (p. 122). This is only supported in KCL 2.x for Java, starting with KCL 2.3 for Java and beyond.
- **stream name** The identifier of the data stream in the following format: `account-id:StreamName:streamCreationTimestamp`.

  Note
  This data is only present in the lease table if you are Processing Multiple Data Streams with the same KCL 2.x for Java Consumer Application (p. 122). This is only supported in KCL 2.x for Java, starting with KCL 2.3 for Java and beyond.

Throughput

If your Amazon Kinesis Data Streams application receives provisioned-throughput exceptions, you should increase the provisioned throughput for the DynamoDB table. The KCL creates the table with a provisioned throughput of 10 reads per second and 10 writes per second, but this might not be sufficient for your application. For example, if your Amazon Kinesis Data Streams application does

frequent checkpointing or operates on a stream that is composed of many shards, you might need more throughput.

For information about provisioned throughput in DynamoDB, see Read/Write Capacity Mode and Working with Tables and Data in the *Amazon DynamoDB Developer Guide*.

How a Lease Table Is Synchronized with the Shards in a KDS Data Stream

Workers in KCL consumer applications use leases to process shards from a given data stream. The information on what worker is leasing what shard at any given time is stored in a lease table. The lease table must remain in sync with the latest shard information from the data stream while the KCL consumer application is running. KCL synchronizes the lease table with the shards information acquired from the Kinesis Data Streams service during the consumer application bootstraping (either when the consumer application is initialized or restarted) and also whenever a shard that is being processed reaches an end (resharding). In other words, the workers or a KCL consumer application are synchronized with the data stream that they are processing during the initial consumer application bootstrap and whenever the consumer application encounters a data stream reshard event.

Topics

Synchronization in KCL 1.0 - 1.13 and KCL 2.0 - 2.2

In KCL 1.0 - 1.13 and KCL 2.0 - 2.2, during consumer application's bootstraping and also during each data stream reshard event, KCL synchronizes the lease table with the shards information acquired from the Kinesis Data Streams service by invoking the `ListShards` or the `DescribeStream` discovery APIs. In all the KCL versions listed above, each worker of a KCL consumer application completes the following steps to perform the lease/shard synchronization process during the consumer application's bootstrapping and at each stream reshard event:

- Fetches all the shards for data the stream that is being processed
- Fetches all the shard leases from the lease table
- Filters out each open shard that does not have a lease in the lease table
- Iterates over all found open shards and for each open shard with no open parent:
  - Traverses the hierarchy tree through its ancestors path to determine if the shard is a descendant. A shard is considered a descendant, if an ancestor shard is being processed (lease entry for ancestor shard exists in the lease table) or if an ancestor shard should be processed (for example, if the initial position is `TRIM_HORIZON` or `AT_TIMESTAMP`)
  - If the open shard in context is a descendant, KCL checkpoints the shard based on initial position and creates leases for its parents, if required

Synchronization in KCL 2.x, Starting with KCL 2.3 and Beyond

Starting with the latest supported versions of KCL 2.x (KCL 2.3) and beyond, the library now supports the following changes to the synchronization process. These lease/shard synchronization changes significantly reduce the number of API calls made by KCL consumer applications to the Kinesis Data Streams service and optimize the lease management in your KCL consumer application.

- During application's bootstraping, if the lease table is empty, KCL utilizes the `ListShard` API's filtering option (the `ShardFilter` optional request parameter) to retrieve and create leases only for

a snapshot of shards open at the time specified by the `ShardFilter` parameter. The `ShardFilter` parameter enables you to filter out the response of the `ListShards` API. The only required property of the `ShardFilter` parameter is `Type`. KCL uses the `Type` filter property and the following of its valid values to identify and return a snapshot of open shards that might require new leases:

- `AT_TRIM_HORIZON` - the response includes all the shards that were open at `TRIM_HORIZON`.
- `AT_LATEST` - the response includes only the currently open shards of the data stream.
- `AT_TIMESTAMP` - the response includes all shards whose start timestamp is less than or equal to the given timestamp and end timestamp is greater than or equal to the given timestamp or still open.

`ShardFilter` is used when creating leases for an empty lease table to initialize leases for a snapshot of shards specified at `RetrievalConfig#initialPositionInStreamExtended`.

For more information about `ShardFilter`, see https://docs.aws.amazon.com/kinesis/latest/APIReference/API_ShardFilter.html.

- Instead of all workers performing the lease/shard synchronization to keep the lease table up to date with the latest shards in the data stream, a single elected worker leader performs the lease/shard synchronization.
- KCL 2.3 uses the `ChildShards` return parameter of the `GetRecords` and the `SubscribeToShard` APIs to perform lease/shard synchronization that happens at `SHARD_END` for closed shards, allowing a KCL worker to only create leases for the child shards of the shard it finished processing. For shared throughout consumer applications, this optimization of the lease/shard synchronization uses the `ChildShards` parameter of the `GetRecords` API. For the dedicated throughput (enhanced fan-out) consumer applications, this optimization of the lease/shard synchronization uses the `ChildShards` parameter of the `SubscribeToShard` API. For more information, see GetRecords, SubscribeToShards, and ChildShard.
- With the above changes, the behavior of KCL is moving from the model of all workers learning about all existing shards to the model of workers learning only about the children shards of the shards that each worker owns. Therefore, in addition to the synchronization that happens during consumer application bootstraping and reshard events, KCL now also performs additional periodic shard/lease scans in order to identify any potential holes in the lease table (in other words, to learn about all new shards) to ensure the complete hash range of the data stream is being processed and create leases for them if required. `PeriodicShardSyncManager` is the component that is responsible for running periodic lease/shard scans.

  For more information about `PeriodicShardSyncManager` in KCL 2.3, see https://github.com/awslabs/amazon-kinesis-client/blob/master/amazon-kinesis-client/src/main/java/software/amazon/kinesis/leases/LeaseManagementConfig.java#L201-L213.

  In KCL 2.3, new configuration options are available to configure `PeriodicShardSyncManager` in `LeaseManagementConfig`:

Name	Default value	Description
leasesRecoveryAuditorExecutionFrequencyMillis	120000 (2 minutes)	Frequency (in millis) of the auditor job to scan for partial leases in the lease table. If the auditor detects any hole in the leases for a stream, then it would trigger shard

Name	Default value	Description
		synchronization based on `leasesRecoveryAuditorInco`
leasesRecoveryAuditorInconsistencyConfidenceThreshold	3	Confidence threshold for the periodic auditor job to determine if leases for a data stream in the lease table are inconsistent. If the auditor finds same set of inconsistencies consecutively for a data stream for this many times, then it would trigger a shard synchronization.

New CloudWatch metrics are also now emitted to monitor the health of the `PeriodicShardSyncManager`. For more information, see PeriodicShardSyncManager (p. 199).

- Including an optimization to `HierarchicalShardSyncer` to only create leases for one layer of shards.

Synchronization in KCL 1.x, Starting with KCL 1.14 and Beyond

Starting with the latest supported versions of KCL 1.x (KCL 1.14) and beyond, the library now supports the following changes to the synchronization process. These lease/shard synchronization changes significantly reduce the number of API calls made by KCL consumer applications to the Kinesis Data Streams service and optimize the lease management in your KCL consumer application.

- During application's bootstraping, if the lease table is empty, KCL utilizes the `ListShard` API's filtering option (the `ShardFilter` optional request parameter) to retrieve and create leases only for a snapshot of shards open at the time specified by the `ShardFilter` parameter. The `ShardFilter` parameter enables you to filter out the response of the `ListShards` API. The only required property of the `ShardFilter` parameter is `Type`. KCL uses the `Type` filter property and the following of its valid values to identify and return a snapshot of open shards that might require new leases:
  - `AT_TRIM_HORIZON` - the response includes all the shards that were open at `TRIM_HORIZON`.
  - `AT_LATEST` - the response includes only the currently open shards of the data stream.
  - `AT_TIMESTAMP` - the response includes all shards whose start timestamp is less than or equal to the given timestamp and end timestamp is greater than or equal to the given timestamp or still open.

  `ShardFilter` is used when creating leases for an empty lease table to initialize leases for a snapshot of shards specified at `KinesisClientLibConfiguration#initialPositionInStreamExtended`.

  For more information about `ShardFilter`, see https://docs.aws.amazon.com/kinesis/latest/APIReference/API_ShardFilter.html.

- Instead of all workers performing the lease/shard synchronization to keep the lease table up to date with the latest shards in the data stream, a single elected worker leader performs the lease/shard synchronization.
- KCL 1.14 uses the `ChildShards` return parameter of the `GetRecords` and the `SubscribeToShard` APIs to perform lease/shard synchronization that happens at `SHARD_END` for closed shards, allowing a KCL worker to only create leases for the child shards of the shard it finished processing. For more information, see GetRecords and ChildShard.
- With the above changes, the behavior of KCL is moving from the model of all workers learning about all existing shards to the model of workers learning only about the children shards of the shards that each worker owns. Therefore, in addition to the synchronization that happens during consumer application bootstraping and reshard events, KCL now also performs additional periodic shard/lease scans in order to identify any potential holes in the lease table (in other words, to learn about all new shards) to ensure the complete hash range of the data stream is being processed and create leases for them if required. `PeriodicShardSyncManager` is the component that is responsible for running periodic lease/shard scans.

  When `KinesisClientLibConfiguration#shardSyncStrategyType` is set to `ShardSyncStrategyType.SHARD_END`, `PeriodicShardSync` `leasesRecoveryAuditorInconsistencyConfidenceThreshold` is used to determine the threshold for number of consecutive scans containing holes in the lease table after which to enforce a shard synchronization. When `KinesisClientLibConfiguration#shardSyncStrategyType` is set to `ShardSyncStrategyType.PERIODIC`, `leasesRecoveryAuditorInconsistencyConfidenceThreshold` is ignored.

  For more information about `PeriodicShardSyncManager` in KCL 1.14, see https://github.com/awslabs/amazon-kinesis-client/blob/v1.x/src/main/java/com/amazonaws/services/kinesis/clientlibrary/lib/worker/KinesisClientLibConfiguration.java#L987-L999.

  In KCL 1.14, new configuration option is available to configure `PeriodicShardSyncManager` in `LeaseManagementConfig`:

Name	Default value	Description
leasesRecoveryAuditorInconsistencyConfidenceThreshold	3	Confidence threshold for the periodic auditor job to determine if leases for a data stream in the lease table are inconsistent. If the auditor finds same set of inconsistencies consecutively for a data stream for this many times, then it would trigger a shard synchronization.

  New CloudWatch metrics are also now emitted to monitor the health of the `PeriodicShardSyncManager`. For more information, see PeriodicShardSyncManager (p. 199).

- KCL 1.14 now also supports deferred lease cleanup. Leases are deleted asynchronously by `LeaseCleanupManager` upon reaching `SHARD_END`, when a shard has either expired past the data stream's retention period or been closed as the result of a resharding operation.

  New configuration options are available to configure `LeaseCleanupManager`.

Name	Default value	Description
leaseCleanupIntervalMillis	1 minute	Interval at which to run lease cleanup thread.
completedLeaseCleanupIntervalMillis	5 minutes	Interval at which to check if a lease is completed or not.
garbageLeaseCleanupIntervalMillis	30 minutes	Interval at which to check if a lease is garbage (i.e trimmed past the data stream's retention period) or not.

- Including an optimization to `KinesisShardSyncer` to only create leases for one layer of shards.

Processing Multiple Data Streams with the same KCL 2.x for Java Consumer Application

This section describes the following changes in KCL 2.x for Java that enable you to create KCL consumer applicaitons that can process more than one data stream at the same time.

> **Important**
> Multistream processing is only supported in KCL 2.x for Java, starting with KCL 2.3 for Java and beyond.
> Multistream processing is NOT supported for any other languages in which KCL 2.x can be implemented.
> Multistream processing is NOT supported in any versions of KCL 1.x.

- **MultistreamTracker interface**

  To build a consumer application that can process multiple streams at the same time, you must implement a new interface called MultistreamTracker. This interface includes the `streamConfigList` method that returns the list of data streams and their configurations to be processed by the KCL consumer application. Notice that the data streams that are being processed can be changed during the consumer application runtime. `streamConfigList` is called periodically by the KCL to learn about the changes in data streams to process.

  The `streamConfigList` method populates the StreamConfig list.

```
package software.amazon.kinesis.common;

import lombok.Data;
import lombok.experimental.Accessors;
```

```
@Data
@Accessors(fluent = true)
public class StreamConfig {
    private final StreamIdentifier streamIdentifier;
    private final InitialPositionInStreamExtended initialPositionInStreamExtended;
    private String consumerArn;
}
```

Note that the `StreamIdentifier` and `InitialPositionInStreamExtended` are required fields, while `consumerArn` is optional. You must provide the `consumerArn` only if you are using KCL 2.x to implement an enhanced fan-out consumer application.

For more information about `StreamIdentifier`, see https://github.com/awslabs/amazon-kinesis-client/blob/0c5042dadf794fe988438436252a5a8fe70b6b0b/amazon-kinesis-client/src/main/java/software/amazon/kinesis/common/StreamIdentifier.java#L29. You can create a multistream instance for the `StreamIdentifier` from the serialized stream identifier. The serialized stream identifier should be of the following format: `account-id:StreamName:streamCreationTimestamp`.

```
 * @param streamIdentifierSer
    * @return StreamIdentifier
    */
   public static StreamIdentifier multiStreamInstance(String streamIdentifierSer) {
       if (PATTERN.matcher(streamIdentifierSer).matches()) {
           final String[] split = streamIdentifierSer.split(DELIMITER);
           return new StreamIdentifier(split[0], split[1], Long.parseLong(split[2]));
       } else {
           throw new IllegalArgumentException("Unable to deserialize StreamIdentifier
from " + streamIdentifierSer);
       }
   }
```

`MultistreamTracker` also includes a strategy for deleting leases of old streams in the lease table (`formerStreamsLeasesDeletionStrategy`). Notice that the strategy CANNOT be changed during the consumer application runtime. For more information, see https://github.com/awslabs/amazon-kinesis-client/blob/0c5042dadf794fe988438436252a5a8fe70b6b0b/amazon-kinesis-client/src/main/java/software/amazon/kinesis/processor/FormerStreamsLeasesDeletionStrategy.java

- ConfigsBuilder is a an application-wide class that you can use to specify all of the KCL 2.x configuration settings to be used when building your KCL consumer application. `ConfigsBuilder` class now has support for the `MultistreamTracker` interface. You can initialize ConfigsBuilder either with the name of the one data stream to consume records from:

```
/**
    * Constructor to initialize ConfigsBuilder with StreamName
    * @param streamName
    * @param applicationName
    * @param kinesisClient
    * @param dynamoDBClient
    * @param cloudWatchClient
    * @param workerIdentifier
    * @param shardRecordProcessorFactory
    */
   public ConfigsBuilder(@NonNull String streamName, @NonNull String applicationName,
           @NonNull KinesisAsyncClient kinesisClient, @NonNull DynamoDbAsyncClient
dynamoDBClient,
           @NonNull CloudWatchAsyncClient cloudWatchClient, @NonNull String
workerIdentifier,
```

```
            @NonNull ShardRecordProcessorFactory shardRecordProcessorFactory) {
        this.appStreamTracker = Either.right(streamName);
        this.applicationName = applicationName;
        this.kinesisClient = kinesisClient;
        this.dynamoDBClient = dynamoDBClient;
        this.cloudWatchClient = cloudWatchClient;
        this.workerIdentifier = workerIdentifier;
        this.shardRecordProcessorFactory = shardRecordProcessorFactory;
    }
```

Or you can initialize ConfigsBuilder with `MultiStreamTracker` if you want to implement a KCL consumer application that processes multiple streams at the same time.

```
* Constructor to initialize ConfigsBuilder with MultiStreamTracker
     * @param multiStreamTracker
     * @param applicationName
     * @param kinesisClient
     * @param dynamoDBClient
     * @param cloudWatchClient
     * @param workerIdentifier
     * @param shardRecordProcessorFactory
     */
    public ConfigsBuilder(@NonNull MultiStreamTracker multiStreamTracker, @NonNull String
 applicationName,
            @NonNull KinesisAsyncClient kinesisClient, @NonNull DynamoDbAsyncClient
 dynamoDBClient,
            @NonNull CloudWatchAsyncClient cloudWatchClient, @NonNull String
 workerIdentifier,
            @NonNull ShardRecordProcessorFactory shardRecordProcessorFactory) {
        this.appStreamTracker = Either.left(multiStreamTracker);
        this.applicationName = applicationName;
        this.kinesisClient = kinesisClient;
        this.dynamoDBClient = dynamoDBClient;
        this.cloudWatchClient = cloudWatchClient;
        this.workerIdentifier = workerIdentifier;
        this.shardRecordProcessorFactory = shardRecordProcessorFactory;
    }
```

- With multistream support implemented for your KCL consumer application, each row of the application's lease table now contains the shard ID and the stream name of the multiple data streams that this application processes.
- When multistream support for your KCL consumer application is implemented, the leaseKey takes the following structure: `account-id:StreamName:streamCreationTimestamp:ShardId`. For example, `111111111:multiStreamTest-1:12345:shardId-000000000336`.

  Important
  When your existing KCL consumer application is configured to process only one data stream, the leaseKey (which is the hash key for the lease table) is the shard ID. If you reconfigure this existing KCL consumer application to process multiple data streams, it breaks your lease table, because with multistream support, the leaseKey structure must be as follows: `account-id:StreamName:StreamCreationTimestamp:ShardId`.

Using the Kinesis Client Library with the AWS Glue Schema Registry

You can integrate your Kinesis data streams with the AWS Glue schema registry. The AWS Glue schema registry allows you to centrally discover, control, and evolve schemas, while ensuring data produced is

continuously validated by a registered schema. A schema defines the structure and format of a data record. A schema is a versioned specification for reliable data publication, consumption, or storage. The AWS Glue Schema Registry enables you to improve end-to-end data quality and data governance within your streaming applications. For more information, see AWS Glue Schema Registry. One of the ways to set up this integration is through the KCL in Java.

> **Important**
> Currently, Kinesis Data Streams and AWS Glue schema registry integration is only supported for the Kinesis data streams that use KCL 2.3 consumers implemented in Java. Multi-language support is not provided. KCL 1.0 consumers are not supported. KCL 2.x consumers prior to KCL 2.3 are not supported.

For detailed instructions on how to set up integration of Kinesis Data Streams with Schema Registry using the KCL, see the "Interacting with Data Using the KPL/KCL Libraries" section in Use Case: Integrating Amazon Kinesis Data Streams with the AWS Glue Schema Registry.

Developing Custom Consumers with Shared Throughput

If you don't need dedicated throughput when receiving data from Kinesis Data Streams, and if you don't need read propagation delays under 200 ms, you can build consumer applications as described in the following topics.

Topics

For information about building consumers that can receive records from Kinesis data streams with dedicated throughput, see Developing Custom Consumers with Dedicated Throughput (Enhanced Fan-Out) (p. 154).

Developing Custom Consumers with Shared Throughput Using KCL

One of the methods of developing a custom consumer application with shared throughput is to use the Kinesis Client Library (KCL).

Topics

Developing KCL 1.x Consumers

You can develop a consumer application for Amazon Kinesis Data Streams using the Kinesis Client Library (KCL). For more information about KCL, see What is the Kinesis Client Library? (p. 114).

Contents

Developing a Kinesis Client Library Consumer in Java

You can use the Kinesis Client Library (KCL) to build applications that process data from your Kinesis data streams. The Kinesis Client Library is available in multiple languages. This topic discusses Java. To view the Javadoc reference, see the AWS Javadoc topic for Class AmazonKinesisClient.

To download the Java KCL from GitHub, go to Kinesis Client Library (Java). To locate the Java KCL on Apache Maven, go to the KCL search results page. To download sample code for a Java KCL consumer application from GitHub, go to the KCL for Java sample project page on GitHub.

The sample application uses Apache Commons Logging. You can change the logging configuration in the static `configure` method defined in the `AmazonKinesisApplicationSample.java` file. For more information about how to use Apache Commons Logging with Log4j and AWS Java applications, see Logging with Log4j in the *AWS SDK for Java Developer Guide*.

You must complete the following tasks when implementing a KCL consumer application in Java:

Tasks

Implement the IRecordProcessor Methods

The KCL currently supports two versions of the `IRecordProcessor` interface:The original interface is available with the first version of the KCL, and version 2 is available starting with KCL version 1.5.0. Both interfaces are fully supported. Your choice depends on your specific scenario requirements. Refer to your locally built Javadocs or the source code to see all the differences. The following sections outline the minimal implementation for getting started.

IRecordProcessor Versions

Original Interface (Version 1)

The original `IRecordProcessor` interface (`package com.amazonaws.services.kinesis.clientlibrary.interfaces`) exposes the following record processor methods that your consumer must implement. The sample provides implementations that you can use as a starting point (see `AmazonKinesisApplicationSampleRecordProcessor.java`).

```
public void initialize(String shardId)
public void processRecords(List<Record> records, IRecordProcessorCheckpointer checkpointer)
public void shutdown(IRecordProcessorCheckpointer checkpointer, ShutdownReason reason)
```

initialize

The KCL calls the `initialize` method when the record processor is instantiated, passing a specific shard ID as a parameter. This record processor processes only this shard and typically, the reverse is also

true (this shard is processed only by this record processor). However, your consumer should account for the possibility that a data record might be processed more than one time. Kinesis Data Streams has *at least once* semantics, meaning that every data record from a shard is processed at least one time by a worker in your consumer. For more information about cases in which a particular shard may be processed by more than one worker, see Resharding, Scaling, and Parallel Processing (p. 175).

```
public void initialize(String shardId)
```

processRecords

The KCL calls the `processRecords` method, passing a list of data record from the shard specified by the `initialize(shardId)` method. The record processor processes the data in these records according to the semantics of the consumer. For example, the worker might perform a transformation on the data and then store the result in an Amazon Simple Storage Service (Amazon S3) bucket.

```
public void processRecords(List<Record> records, IRecordProcessorCheckpointer
 checkpointer)
```

In addition to the data itself, the record also contains a sequence number and partition key. The worker can use these values when processing the data. For example, the worker could choose the S3 bucket in which to store the data based on the value of the partition key. The `Record` class exposes the following methods that provide access to the record's data, sequence number, and partition key.

```
record.getData()
record.getSequenceNumber()
record.getPartitionKey()
```

In the sample, the private method `processRecordsWithRetries` has code that shows how a worker can access the record's data, sequence number, and partition key.

Kinesis Data Streams requires the record processor to keep track of the records that have already been processed in a shard. The KCL takes care of this tracking for you by passing a checkpointer (`IRecordProcessorCheckpointer`) to `processRecords`. The record processor calls the `checkpoint` method on this interface to inform the KCL of how far it has progressed in processing the records in the shard. If the worker fails, the KCL uses this information to restart the processing of the shard at the last known processed record.

For a split or merge operation, the KCL won't start processing the new shards until the processors for the original shards have called `checkpoint` to signal that all processing on the original shards is complete.

If you don't pass a parameter, the KCL assumes that the call to `checkpoint` means that all records have been processed, up to the last record that was passed to the record processor. Therefore, the record processor should call `checkpoint` only after it has processed all the records in the list that was passed to it. Record processors do not need to call `checkpoint` on each call to `processRecords`. A processor could, for example, call `checkpoint` on every third call to `processRecords`. You can optionally specify the exact sequence number of a record as a parameter to `checkpoint`. In this case, the KCL assumes that all records have been processed up to that record only.

In the sample, the private method `checkpoint` shows how to call `IRecordProcessorCheckpointer.checkpoint` using the appropriate exception handling and retry logic.

The KCL relies on `processRecords` to handle any exceptions that arise from processing the data records. If an exception is thrown from `processRecords`, the KCL skips over the data records that were passed before the exception. That is, these records are not re-sent to the record processor that threw the exception or to any other record processor in the consumer.

shutdown

The KCL calls the `shutdown` method either when processing ends (the shutdown reason is `TERMINATE`) or the worker is no longer responding (the shutdown reason is `ZOMBIE`).

```
public void shutdown(IRecordProcessorCheckpointer checkpointer, ShutdownReason reason)
```

Processing ends when the record processor does not receive any further records from the shard, because either the shard was split or merged, or the stream was deleted.

The KCL also passes a `IRecordProcessorCheckpointer` interface to `shutdown`. If the shutdown reason is `TERMINATE`, the record processor should finish processing any data records, and then call the `checkpoint` method on this interface.

Updated Interface (Version 2)

The updated `IRecordProcessor` interface (`package com.amazonaws.services.kinesis.clientlibrary.interfaces.v2`) exposes the following record processor methods that your consumer must implement:

```
void initialize(InitializationInput initializationInput)
void processRecords(ProcessRecordsInput processRecordsInput)
void shutdown(ShutdownInput shutdownInput)
```

All of the arguments from the original version of the interface are accessible through get methods on the container objects. For example, to retrieve the list of records in `processRecords()`, you can use `processRecordsInput.getRecords()`.

As of version 2 of this interface (KCL 1.5.0 and later), the following new inputs are available in addition to the inputs provided by the original interface:

starting sequence number

> In the `InitializationInput` object passed to the `initialize()` operation, the starting sequence number from which records would be provided to the record processor instance. This is the sequence number that was last checkpointed by the record processor instance previously processing the same shard. This is provided in case your application needs this information.

pending checkpoint sequence number

> In the `InitializationInput` object passed to the `initialize()` operation, the pending checkpoint sequence number (if any) that could not be committed before the previous record processor instance stopped.

Implement a Class Factory for the IRecordProcessor Interface

You also need to implement a factory for the class that implements the record processor methods. When your consumer instantiates the worker, it passes a reference to this factory.

The sample implements the factory class in the file `AmazonKinesisApplicationSampleRecordProcessorFactory.java` using the original record processor interface. If you want the class factory to create version 2 record processors, use the package name `com.amazonaws.services.kinesis.clientlibrary.interfaces.v2`.

```
public class SampleRecordProcessorFactory implements IRecordProcessorFactory {
    /**
    * Constructor.
    */
    public SampleRecordProcessorFactory() {
        super();
```

```
    }
    /**
    * {@inheritDoc}
    */
    @Override
    public IRecordProcessor createProcessor() {
        return new SampleRecordProcessor();
    }
}
```

Create a Worker

As discussed in Implement the IRecordProcessor Methods (p. 126), there are two versions of the KCL record processor interface to choose from, which affects how you would create a worker. The original record processor interface uses the following code structure to create a worker:

```
final KinesisClientLibConfiguration config = new KinesisClientLibConfiguration(...)
final IRecordProcessorFactory recordProcessorFactory = new RecordProcessorFactory();
final Worker worker = new Worker(recordProcessorFactory, config);
```

With version 2 of the record processor interface, you can use `Worker.Builder` to create a worker without needing to worry about which constructor to use and the order of the arguments. The updated record processor interface uses the following code structure to create a worker:

```
final KinesisClientLibConfiguration config = new KinesisClientLibConfiguration(...)
final IRecordProcessorFactory recordProcessorFactory = new RecordProcessorFactory();
final Worker worker = new Worker.Builder()
    .recordProcessorFactory(recordProcessorFactory)
    .config(config)
    .build();
```

Modify the Configuration Properties

The sample provides default values for configuration properties. This configuration data for the worker is then consolidated in a `KinesisClientLibConfiguration` object. This object and a reference to the class factory for `IRecordProcessor` are passed in the call that instantiates the worker. You can override any of these properties with your own values using a Java properties file (see `AmazonKinesisApplicationSample.java`).

Application Name

The KCL requires an application name that is unique across your applications, and across Amazon DynamoDB tables in the same Region. It uses the application name configuration value in the following ways:

- All workers associated with this application name are assumed to be working together on the same stream. These workers may be distributed on multiple instances. If you run an additional instance of the same application code, but with a different application name, the KCL treats the second instance as an entirely separate application that is also operating on the same stream.
- The KCL creates a DynamoDB table with the application name and uses the table to maintain state information (such as checkpoints and worker-shard mapping) for the application. Each application has its own DynamoDB table. For more information, see Using a Lease Table to Track the Shards Processed by the KCL Consumer Application (p. 116).

Set Up Credentials

You must make your AWS credentials available to one of the credential providers in the default credential providers chain. For example, if you are running your consumer on an EC2 instance, we

recommend that you launch the instance with an IAM role. AWS credentials that reflect the permissions associated with this IAM role are made available to applications on the instance through its instance metadata. This is the most secure way to manage credentials for a consumer running on an EC2 instance.

The sample application first attempts to retrieve IAM credentials from instance metadata:

```
credentialsProvider = new InstanceProfileCredentialsProvider();
```

If the sample application cannot obtain credentials from the instance metadata, it attempts to retrieve credentials from a properties file:

```
credentialsProvider = new ClasspathPropertiesFileCredentialsProvider();
```

For more information about instance metadata, see Instance Metadata in the *Amazon EC2 User Guide for Linux Instances*.

Use Worker ID for Multiple Instances

The sample initialization code creates an ID for the worker, `workerId`, using the name of the local computer and appending a globally unique identifier as shown in the following code snippet. This approach supports the scenario of multiple instances of the consumer application running on a single computer.

```
String workerId = InetAddress.getLocalHost().getCanonicalHostName() + ":" +
 UUID.randomUUID();
```

Migrating to Version 2 of the Record Processor Interface

If you want to migrate code that uses the original interface, in addition to the steps described previously, the following steps are required:

1. Change your record processor class to import the version 2 record processor interface:

   ```
   import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
   ```

2. Change the references to inputs to use `get` methods on the container objects. For example, in the `shutdown()` operation, change "`checkpointer`" to "`shutdownInput.getCheckpointer()`".
3. Change your record processor factory class to import the version 2 record processor factory interface:

   ```
   import
    com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory;
   ```

4. Change the construction of the worker to use `Worker.Builder`. For example:

   ```
   final Worker worker = new Worker.Builder()
       .recordProcessorFactory(recordProcessorFactory)
       .config(config)
       .build();
   ```

Developing a Kinesis Client Library Consumer in Node.js

You can use the Kinesis Client Library (KCL) to build applications that process data from your Kinesis data streams. The Kinesis Client Library is available in multiple languages. This topic discusses Node.js.

The KCL is a Java library; support for languages other than Java is provided using a multi-language interface called the *MultiLangDaemon*. This daemon is Java-based and runs in the background when you are using a KCL language other than Java. Therefore, if you install the KCL for Node.js and write your consumer app entirely in Node.js, you still need Java installed on your system because of the MultiLangDaemon. Further, MultiLangDaemon has some default settings you may need to customize for your use case, for example, the AWS Region that it connects to. For more information about the MultiLangDaemon on GitHub, go to the KCL MultiLangDaemon project page.

To download the Node.js KCL from GitHub, go to Kinesis Client Library (Node.js).

Sample Code Downloads

There are two code samples available for KCL in Node.js:

- basic-sample

  Used in the following sections to illustrate the fundamentals of building a KCL consumer application in Node.js.
- click-stream-sample

  Slightly more advanced and uses a real-world scenario, after you have familiarized yourself with the basic sample code. This sample is not discussed here but has a README file with more information.

You must complete the following tasks when implementing a KCL consumer application in Node.js:

Tasks

- Implement the Record Processor (p. 131)
- Modify the Configuration Properties (p. 132)

Implement the Record Processor

The simplest possible consumer using the KCL for Node.js must implement a `recordProcessor` function, which in turn contains the functions `initialize`, `processRecords`, and `shutdown`. The sample provides an implementation that you can use as a starting point (see `sample_kcl_app.js`).

```
function recordProcessor() {
  // return an object that implements initialize, processRecords and shutdown functions.}
```

initialize

The KCL calls the `initialize` function when the record processor starts. This record processor processes only the shard ID passed as `initializeInput.shardId`, and typically, the reverse is also true (this shard is processed only by this record processor). However, your consumer should account for the possibility that a data record might be processed more than one time. This is because Kinesis Data Streams has *at least once* semantics, meaning that every data record from a shard is processed at least one time by a worker in your consumer. For more information about cases in which a particular shard might be processed by more than one worker, see Resharding, Scaling, and Parallel Processing (p. 175).

```
initialize: function(initializeInput, completeCallback)
```

processRecords

The KCL calls this function with input that contains a list of data records from the shard specified to the `initialize` function. The record processor that you implement processes the data in these records according to the semantics of your consumer. For example, the worker might perform a transformation on the data and then store the result in an Amazon Simple Storage Service (Amazon S3) bucket.

```
processRecords: function(processRecordsInput, completeCallback)
```

In addition to the data itself, the record also contains a sequence number and partition key, which the worker can use when processing the data. For example, the worker could choose the S3 bucket in which to store the data based on the value of the partition key. The `record` dictionary exposes the following key-value pairs to access the record's data, sequence number, and partition key:

```
record.data
record.sequenceNumber
record.partitionKey
```

Note that the data is Base64-encoded.

In the basic sample, the function `processRecords` has code that shows how a worker can access the record's data, sequence number, and partition key.

Kinesis Data Streams requires the record processor to keep track of the records that have already been processed in a shard. The KCL takes care of this tracking for with a `checkpointer` object passed as `processRecordsInput.checkpointer`. Your record processor calls the `checkpointer.checkpoint` function to inform the KCL how far it has progressed in processing the records in the shard. In the event that the worker fails, the KCL uses this information when you restart the processing of the shard so that it continues from the last known processed record.

For a split or merge operation, the KCL doesn't start processing the new shards until the processors for the original shards have called `checkpoint` to signal that all processing on the original shards is complete.

If you don't pass the sequence number to the `checkpoint` function, the KCL assumes that the call to `checkpoint` means that all records have been processed, up to the last record that was passed to the record processor. Therefore, the record processor should call `checkpoint` **only** after it has processed all the records in the list that was passed to it. Record processors do not need to call `checkpoint` on each call to `processRecords`. A processor could, for example, call `checkpoint` on every third call, or some event external to your record processor, such as a custom verification/validation service you've implemented.

You can optionally specify the exact sequence number of a record as a parameter to `checkpoint`. In this case, the KCL assumes that all records have been processed up to that record only.

The basic sample application shows the simplest possible call to the `checkpointer.checkpoint` function. You can add other checkpointing logic you need for your consumer at this point in the function.

shutdown

The KCL calls the `shutdown` function either when processing ends (`shutdownInput.reason` is `TERMINATE`) or the worker is no longer responding (`shutdownInput.reason` is `ZOMBIE`).

```
shutdown: function(shutdownInput, completeCallback)
```

Processing ends when the record processor does not receive any further records from the shard, because either the shard was split or merged, or the stream was deleted.

The KCL also passes a `shutdownInput.checkpointer` object to `shutdown`. If the shutdown reason is `TERMINATE`, you should make sure that the record processor has finished processing any data records, and then call the `checkpoint` function on this interface.

Modify the Configuration Properties

The sample provides default values for the configuration properties. You can override any of these properties with your own values (see `sample.properties` in the basic sample).

Application Name

The KCL requires an application that this is unique among your applications, and among Amazon DynamoDB tables in the same Region. It uses the application name configuration value in the following ways:

- All workers associated with this application name are assumed to be working together on the same stream. These workers may be distributed on multiple instances. If you run an additional instance of the same application code, but with a different application name, the KCL treats the second instance as an entirely separate application that is also operating on the same stream.
- The KCL creates a DynamoDB table with the application name and uses the table to maintain state information (such as checkpoints and worker-shard mapping) for the application. Each application has its own DynamoDB table. For more information, see Using a Lease Table to Track the Shards Processed by the KCL Consumer Application (p. 116).

Set Up Credentials

You must make your AWS credentials available to one of the credential providers in the default credential providers chain. You can you use the `AWSCredentialsProvider` property to set a credentials provider. The `sample.properties` file must make your credentials available to one of the credentials providers in the default credential providers chain. If you are running your consumer on an Amazon EC2 instance, we recommend that you configure the instance with an IAM role. AWS credentials that reflect the permissions associated with this IAM role are made available to applications on the instance through its instance metadata. This is the most secure way to manage credentials for a consumer application running on an EC2 instance.

The following example configures KCL to process a Kinesis data stream named `kclnodejssample` using the record processor supplied in `sample_kcl_app.js`:

```
# The Node.js executable script
executableName = node sample_kcl_app.js
# The name of an Amazon Kinesis stream to process
streamName = kclnodejssample
# Unique KCL application name
applicationName = kclnodejssample
# Use default AWS credentials provider chain
AWSCredentialsProvider = DefaultAWSCredentialsProviderChain
# Read from the beginning of the stream
initialPositionInStream = TRIM_HORIZON
```

Developing a Kinesis Client Library Consumer in .NET

You can use the Kinesis Client Library (KCL) to build applications that process data from your Kinesis data streams. The Kinesis Client Library is available in multiple languages. This topic discusses .NET.

The KCL is a Java library; support for languages other than Java is provided using a multi-language interface called the *MultiLangDaemon*. This daemon is Java-based and runs in the background when you are using a KCL language other than Java. Therefore, if you install the KCL for .NET and write your consumer app entirely in .NET, you still need Java installed on your system because of the MultiLangDaemon. Further, MultiLangDaemon has some default settings you may need to customize for your use case, for example, the AWS Region that it connects to. For more information about the MultiLangDaemon on GitHub, go to the KCL MultiLangDaemon project page.

To download the .NET KCL from GitHub, go to Kinesis Client Library (.NET). To download sample code for a .NET KCL consumer application, go to the KCL for .NET sample consumer project page on GitHub.

You must complete the following tasks when implementing a KCL consumer application in .NET:

Tasks

- Implement the IRecordProcessor Class Methods (p. 134)
- Modify the Configuration Properties (p. 135)

Implement the IRecordProcessor Class Methods

The consumer must implement the following methods for `IRecordProcessor`. The sample consumer provides implementations that you can use as a starting point (see the `SampleRecordProcessor` class in `SampleConsumer/AmazonKinesisSampleConsumer.cs`).

```
public void Initialize(InitializationInput input)
public void ProcessRecords(ProcessRecordsInput input)
public void Shutdown(ShutdownInput input)
```

Initialize

The KCL calls this method when the record processor is instantiated, passing a specific shard ID in the `input` parameter (`input.ShardId`). This record processor processes only this shard, and typically, the reverse is also true (this shard is processed only by this record processor). However, your consumer should account for the possibility that a data record might be processed more than one time. This is because Kinesis Data Streams has *at least once* semantics, meaning that every data record from a shard is processed at least one time by a worker in your consumer. For more information about cases in which a particular shard might be processed by more than one worker, see Resharding, Scaling, and Parallel Processing (p. 175).

```
public void Initialize(InitializationInput input)
```

ProcessRecords

The KCL calls this method, passing a list of data records in the `input` parameter (`input.Records`) from the shard specified by the `Initialize` method. The record processor that you implement processes the data in these records according to the semantics of your consumer. For example, the worker might perform a transformation on the data and then store the result in an Amazon Simple Storage Service (Amazon S3) bucket.

```
public void ProcessRecords(ProcessRecordsInput input)
```

In addition to the data itself, the record also contains a sequence number and partition key. The worker can use these values when processing the data. For example, the worker could choose the S3 bucket in which to store the data based on the value of the partition key. The `Record` class exposes the following to access the record's data, sequence number, and partition key:

```
byte[] Record.Data
string Record.SequenceNumber
string Record.PartitionKey
```

In the sample, the method `ProcessRecordsWithRetries` has code that shows how a worker can access the record's data, sequence number, and partition key.

Kinesis Data Streams requires the record processor to keep track of the records that have already been processed in a shard. The KCL takes care of this tracking for you by passing a `Checkpointer` object to `ProcessRecords` (`input.Checkpointer`). The record processor calls the `Checkpointer.Checkpoint` method to inform the KCL of how far it has progressed in processing the records in the shard. If the worker fails, the KCL uses this information to restart the processing of the shard at the last known processed record.

For a split or merge operation, the KCL doesn't start processing the new shards until the processors for the original shards have called `Checkpointer.Checkpoint` to signal that all processing on the original shards is complete.

If you don't pass a parameter, the KCL assumes that the call to `Checkpointer.Checkpoint` signifies that all records have been processed, up to the last record that was passed to the record processor. Therefore, the record processor should call `Checkpointer.Checkpoint` only after it has processed all the records in the list that was passed to it. Record processors do not need to call `Checkpointer.Checkpoint` on each call to `ProcessRecords`. A processor could, for example, call `Checkpointer.Checkpoint` on every third or fourth call. You can optionally specify the exact sequence number of a record as a parameter to `Checkpointer.Checkpoint`. In this case, the KCL assumes that records have been processed only up to that record.

In the sample, the private method `Checkpoint(Checkpointer checkpointer)` shows how to call the `Checkpointer.Checkpoint` method using appropriate exception handling and retry logic.

The KCL for .NET handles exceptions differently from other KCL language libraries in that it does not handle any exceptions that arise from processing the data records. Any uncaught exceptions from user code crashes the program.

Shutdown

The KCL calls the `Shutdown` method either when processing ends (the shutdown reason is `TERMINATE`) or the worker is no longer responding (the shutdown `input.Reason` value is `ZOMBIE`).

```
public void Shutdown(ShutdownInput input)
```

Processing ends when the record processor does not receive any further records from the shard, because the shard was split or merged, or the stream was deleted.

The KCL also passes a `Checkpointer` object to `shutdown`. If the shutdown reason is `TERMINATE`, the record processor should finish processing any data records, and then call the `checkpoint` method on this interface.

Modify the Configuration Properties

The sample consumer provides default values for the configuration properties. You can override any of these properties with your own values (see `SampleConsumer/kcl.properties`).

Application Name

The KCL requires an application that this is unique among your applications, and among Amazon DynamoDB tables in the same Region. It uses the application name configuration value in the following ways:

- All workers associated with this application name are assumed to be working together on the same stream. These workers may be distributed on multiple instances. If you run an additional instance of the same application code, but with a different application name, the KCL treats the second instance as an entirely separate application that is also operating on the same stream.
- The KCL creates a DynamoDB table with the application name and uses the table to maintain state information (such as checkpoints and worker-shard mapping) for the application. Each application has its own DynamoDB table. For more information, see Using a Lease Table to Track the Shards Processed by the KCL Consumer Application (p. 116).

Set Up Credentials

You must make your AWS credentials available to one of the credential providers in the default credential providers chain. You can you use the `AWSCredentialsProvider` property to set a credentials provider. The sample.properties must make your credentials available to one of the

credentials providers in the default credential providers chain. If you are running your consumer application on an EC2 instance, we recommend that you configure the instance with an IAM role. AWS credentials that reflect the permissions associated with this IAM role are made available to applications on the instance through its instance metadata. This is the most secure way to manage credentials for a consumer running on an EC2 instance.

The sample's properties file configures KCL to process a Kinesis data stream called "words" using the record processor supplied in `AmazonKinesisSampleConsumer.cs`.

Developing a Kinesis Client Library Consumer in Python

You can use the Kinesis Client Library (KCL) to build applications that process data from your Kinesis data streams. The Kinesis Client Library is available in multiple languages. This topic discusses Python.

The KCL is a Java library; support for languages other than Java is provided using a multi-language interface called the *MultiLangDaemon*. This daemon is Java-based and runs in the background when you are using a KCL language other than Java. Therefore, if you install the KCL for Python and write your consumer app entirely in Python, you still need Java installed on your system because of the MultiLangDaemon. Further, MultiLangDaemon has some default settings you may need to customize for your use case, for example, the AWS Region that it connects to. For more information about the MultiLangDaemon on GitHub, go to the KCL MultiLangDaemon project page.

To download the Python KCL from GitHub, go to Kinesis Client Library (Python). To download sample code for a Python KCL consumer application, go to the KCL for Python sample project page on GitHub.

You must complete the following tasks when implementing a KCL consumer application in Python:

Tasks

- Implement the RecordProcessor Class Methods (p. 136)
- Modify the Configuration Properties (p. 138)

Implement the RecordProcessor Class Methods

The `RecordProcess` class must extend the `RecordProcessorBase` to implement the following methods. The sample provides implementations that you can use as a starting point (see `sample_kclpy_app.py`).

```
def initialize(self, shard_id)
def process_records(self, records, checkpointer)
def shutdown(self, checkpointer, reason)
```

initialize

The KCL calls the `initialize` method when the record processor is instantiated, passing a specific shard ID as a parameter. This record processor processes only this shard, and typically, the reverse is also true (this shard is processed only by this record processor). However, your consumer should account for the possibility that a data record might be processed more than one time. This is because Kinesis Data Streams has *at least once* semantics, meaning that every data record from a shard is processed at least one time by a worker in your consumer. For more information about cases in which a particular shard may be processed by more than one worker, see Resharding, Scaling, and Parallel Processing (p. 175).

```
def initialize(self, shard_id)
```

process_records

The KCL calls this method, passing a list of data record from the shard specified by the `initialize` method. The record processor that you implement processes the data in these records according to the

semantics of your consumer. For example, the worker might perform a transformation on the data and then store the result in an Amazon Simple Storage Service (Amazon S3) bucket.

```
def process_records(self, records, checkpointer)
```

In addition to the data itself, the record also contains a sequence number and partition key. The worker can use these values when processing the data. For example, the worker could choose the S3 bucket in which to store the data based on the value of the partition key. The `record` dictionary exposes the following key-value pairs to access the record's data, sequence number, and partition key:

```
record.get('data')
record.get('sequenceNumber')
record.get('partitionKey')
```

Note that the data is Base64-encoded.

In the sample, the method `process_records` has code that shows how a worker can access the record's data, sequence number, and partition key.

Kinesis Data Streams requires the record processor to keep track of the records that have already been processed in a shard. The KCL takes care of this tracking for you by passing a `Checkpointer` object to `process_records`. The record processor calls the `checkpoint` method on this object to inform the KCL of how far it has progressed in processing the records in the shard. If the worker fails, the KCL uses this information to restart the processing of the shard at the last known processed record.

For a split or merge operation, the KCL doesn't start processing the new shards until the processors for the original shards have called `checkpoint` to signal that all processing on the original shards is complete.

If you don't pass a parameter, the KCL assumes that the call to `checkpoint` means that all records have been processed, up to the last record that was passed to the record processor. Therefore, the record processor should call `checkpoint` only after it has processed all the records in the list that was passed to it. Record processors do not need to call `checkpoint` on each call to `process_records`. A processor could, for example, call `checkpoint` on every third call. You can optionally specify the exact sequence number of a record as a parameter to `checkpoint`. In this case, the KCL assumes that all records have been processed up to that record only.

In the sample, the private method `checkpoint` shows how to call the `Checkpointer.checkpoint` method using appropriate exception handling and retry logic.

The KCL relies on `process_records` to handle any exceptions that arise from processing the data records. If an exception is thrown from `process_records`, the KCL skips over the data records that were passed to `process_records` before the exception. That is, these records are not re-sent to the record processor that threw the exception or to any other record processor in the consumer.

shutdown

The KCL calls the `shutdown` method either when processing ends (the shutdown reason is `TERMINATE`) or the worker is no longer responding (the shutdown `reason` is `ZOMBIE`).

```
def shutdown(self, checkpointer, reason)
```

Processing ends when the record processor does not receive any further records from the shard, because either the shard was split or merged, or the stream was deleted.

The KCL also passes a `Checkpointer` object to `shutdown`. If the shutdown `reason` is `TERMINATE`, the record processor should finish processing any data records, and then call the `checkpoint` method on this interface.

Modify the Configuration Properties

The sample provides default values for the configuration properties. You can override any of these properties with your own values (see `sample.properties`).

Application Name

The KCL requires an application name that is unique among your applications, and among Amazon DynamoDB tables in the same Region. It uses the application name configuration value in the following ways:

- All workers that are associated with this application name are assumed to be working together on the same stream. These workers can be distributed on multiple instances. If you run an additional instance of the same application code, but with a different application name, the KCL treats the second instance as an entirely separate application that is also operating on the same stream.
- The KCL creates a DynamoDB table with the application name and uses the table to maintain state information (such as checkpoints and worker-shard mapping) for the application. Each application has its own DynamoDB table. For more information, see Using a Lease Table to Track the Shards Processed by the KCL Consumer Application (p. 116).

Set Up Credentials

You must make your AWS credentials available to one of the credential providers in the default credential providers chain. You can you use the `AWSCredentialsProvider` property to set a credentials provider. The sample.properties must make your credentials available to one of the credentials providers in the default credential providers chain. If you are running your consumer application on an Amazon EC2 instance, we recommend that you configure the instance with an IAM role. AWS credentials that reflect the permissions associated with this IAM role are made available to applications on the instance through its instance metadata. This is the most secure way to manage credentials for a consumer application running on an EC2 instance.

The sample's properties file configures KCL to process a Kinesis data stream called "words" using the record processor supplied in `sample_kclpy_app.py`.

Developing a Kinesis Client Library Consumer in Ruby

You can use the Kinesis Client Library (KCL) to build applications that process data from your Kinesis data streams. The Kinesis Client Library is available in multiple languages. This topic discusses Ruby.

The KCL is a Java library; support for languages other than Java is provided using a multi-language interface called the *MultiLangDaemon*. This daemon is Java-based and runs in the background when you are using a KCL language other than Java. Therefore, if you install the KCL for Ruby and write your consumer app entirely in Ruby, you still need Java installed on your system because of the MultiLangDaemon. Further, MultiLangDaemon has some default settings you may need to customize for your use case, for example, the AWS Region that it connects to. For more information about the MultiLangDaemon on GitHub, go to the KCL MultiLangDaemon project page.

To download the Ruby KCL from GitHub, go to Kinesis Client Library (Ruby). To download sample code for a Ruby KCL consumer application, go to the KCL for Ruby sample project page on GitHub.

For more information about the KCL Ruby support library, see KCL Ruby Gems Documentation.

Developing KCL 2.x Consumers

This topic shows you how to use version 2.0 of the Kinesis Client Library (KCL). For more information about the KCL, see the overview provided in Developing Consumers Using the Kinesis Client Library 1.x.

Contents

Developing a Kinesis Client Library Consumer in Java

The following code shows an example implementation in Java of `ProcessorFactory` and `RecordProcessor`. If you want to take advantage of the enhanced fan-out feature, see Using Consumers with Enhanced Fan-Out .

```
/*
 *  Copyright 2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 *  Licensed under the Amazon Software License (the "License").
 *  You may not use this file except in compliance with the License.
 *  A copy of the License is located at
 *
 *  http://aws.amazon.com/asl/
 *
 *  or in the "license" file accompanying this file. This file is distributed
 *  on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 *  express or implied. See the License for the specific language governing
 *  permissions and limitations under the License.
 */


/*
 * Copyright 2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License.
 * A copy of the License is located at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * or in the "license" file accompanying this file. This file is distributed
 * on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 * express or implied. See the License for the specific language governing
 * permissions and limitations under the License.
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.UUID;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

import org.apache.commons.lang3.ObjectUtils;
import org.apache.commons.lang3.RandomStringUtils;
import org.apache.commons.lang3.RandomUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.slf4j.MDC;

import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.cloudwatch.CloudWatchAsyncClient;
```

```
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
import software.amazon.awssdk.services.kinesis.model.PutRecordRequest;
import software.amazon.kinesis.common.ConfigsBuilder;
import software.amazon.kinesis.common.KinesisClientUtil;
import software.amazon.kinesis.coordinator.Scheduler;
import software.amazon.kinesis.exceptions.InvalidStateException;
import software.amazon.kinesis.exceptions.ShutdownException;
import software.amazon.kinesis.lifecycle.events.InitializationInput;
import software.amazon.kinesis.lifecycle.events.LeaseLostInput;
import software.amazon.kinesis.lifecycle.events.ProcessRecordsInput;
import software.amazon.kinesis.lifecycle.events.ShardEndedInput;
import software.amazon.kinesis.lifecycle.events.ShutdownRequestedInput;

import software.amazon.kinesis.processor.ShardRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;
import software.amazon.kinesis.retrieval.polling.PollingConfig;

/**
 * This class will run a simple app that uses the KCL to read data and uses the AWS SDK to
 publish data.
 * Before running this program you must first create a Kinesis stream through the AWS
 console or AWS SDK.
 */
public class SampleSingle {

    private static final Logger log = LoggerFactory.getLogger(SampleSingle.class);

    /**
     * Invoke the main method with 2 args: the stream name and (optionally) the region.
     * Verifies valid inputs and then starts running the app.
     */
    public static void main(String... args) {
        if (args.length < 1) {
            log.error("At a minimum, the stream name is required as the first argument. The
 Region may be specified as the second argument.");
            System.exit(1);
        }

        String streamName = args[0];
        String region = null;
        if (args.length > 1) {
            region = args[1];
        }

        new SampleSingle(streamName, region).run();
    }

    private final String streamName;
    private final Region region;
    private final KinesisAsyncClient kinesisClient;

    /**
     * Constructor sets streamName and region. It also creates a KinesisClient object to
 send data to Kinesis.
     * This KinesisClient is used to send dummy data so that the consumer has something to
 read; it is also used
     * indirectly by the KCL to handle the consumption of the data.
     */
    private SampleSingle(String streamName, String region) {
        this.streamName = streamName;
        this.region = Region.of(ObjectUtils.firstNonNull(region, "us-east-2"));
        this.kinesisClient =
 KinesisClientUtil.createKinesisAsyncClient(KinesisAsyncClient.builder().region(this.region));
    }
```

```
   private void run() {

       /**
        * Sends dummy data to Kinesis. Not relevant to consuming the data with the KCL
        */
       ScheduledExecutorService producerExecutor =
Executors.newSingleThreadScheduledExecutor();
       ScheduledFuture<?> producerFuture =
producerExecutor.scheduleAtFixedRate(this::publishRecord, 10, 1, TimeUnit.SECONDS);

       /**
        * Sets up configuration for the KCL, including DynamoDB and CloudWatch
dependencies. The final argument, a
        * ShardRecordProcessorFactory, is where the logic for record processing lives, and
is located in a private
        * class below.
        */
       DynamoDbAsyncClient dynamoClient =
DynamoDbAsyncClient.builder().region(region).build();
       CloudWatchAsyncClient cloudWatchClient =
CloudWatchAsyncClient.builder().region(region).build();
       ConfigsBuilder configsBuilder = new ConfigsBuilder(streamName, streamName,
kinesisClient, dynamoClient, cloudWatchClient, UUID.randomUUID().toString(), new
SampleRecordProcessorFactory());

       /**
        * The Scheduler (also called Worker in earlier versions of the KCL) is the entry
point to the KCL. This
        * instance is configured with defaults provided by the ConfigsBuilder.
        */
       Scheduler scheduler = new Scheduler(
               configsBuilder.checkpointConfig(),
               configsBuilder.coordinatorConfig(),
               configsBuilder.leaseManagementConfig(),
               configsBuilder.lifecycleConfig(),
               configsBuilder.metricsConfig(),
               configsBuilder.processorConfig(),
               configsBuilder.retrievalConfig().retrievalSpecificConfig(new
PollingConfig(streamName, kinesisClient))
       );

       /**
        * Kickoff the Scheduler. Record processing of the stream of dummy data will
continue indefinitely
        * until an exit is triggered.
        */
       Thread schedulerThread = new Thread(scheduler);
       schedulerThread.setDaemon(true);
       schedulerThread.start();

       /**
        * Allows termination of app by pressing Enter.
        */
       System.out.println("Press enter to shutdown");
       BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
       try {
           reader.readLine();
       } catch (IOException ioex) {
           log.error("Caught exception while waiting for confirm. Shutting down.", ioex);
       }

       /**
        * Stops sending dummy data.
        */
       log.info("Cancelling producer and shutting down executor.");
       producerFuture.cancel(true);
```

```
        producerExecutor.shutdownNow();

        /**
         * Stops consuming data. Finishes processing the current batch of data already
received from Kinesis
         * before shutting down.
         */
        Future<Boolean> gracefulShutdownFuture = scheduler.startGracefulShutdown();
        log.info("Waiting up to 20 seconds for shutdown to complete.");
        try {
            gracefulShutdownFuture.get(20, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            log.info("Interrupted while waiting for graceful shutdown. Continuing.");
        } catch (ExecutionException e) {
            log.error("Exception while executing graceful shutdown.", e);
        } catch (TimeoutException e) {
            log.error("Timeout while waiting for shutdown.  Scheduler may not have
exited.");
        }
        log.info("Completed, shutting down now.");
    }

    /**
     * Sends a single record of dummy data to Kinesis.
     */
    private void publishRecord() {
        PutRecordRequest request = PutRecordRequest.builder()
                .partitionKey(RandomStringUtils.randomAlphabetic(5, 20))
                .streamName(streamName)
                .data(SdkBytes.fromByteArray(RandomUtils.nextBytes(10)))
                .build();
        try {
            kinesisClient.putRecord(request).get();
        } catch (InterruptedException e) {
            log.info("Interrupted, assuming shutdown.");
        } catch (ExecutionException e) {
            log.error("Exception while sending data to Kinesis. Will try again next
cycle.", e);
        }
    }

    private static class SampleRecordProcessorFactory implements
ShardRecordProcessorFactory {
        public ShardRecordProcessor shardRecordProcessor() {
            return new SampleRecordProcessor();
        }
    }

    /**
     * The implementation of the ShardRecordProcessor interface is where the heart of the
record processing logic lives.
     * In this example all we do to 'process' is log info about the records.
     */
    private static class SampleRecordProcessor implements ShardRecordProcessor {

        private static final String SHARD_ID_MDC_KEY = "ShardId";

        private static final Logger log =
LoggerFactory.getLogger(SampleRecordProcessor.class);

        private String shardId;

        /**
         * Invoked by the KCL before data records are delivered to the ShardRecordProcessor
instance (via
         * processRecords). In this example we do nothing except some logging.
```

```
         *
         * @param initializationInput Provides information related to initialization.
         */
        public void initialize(InitializationInput initializationInput) {
            shardId = initializationInput.shardId();
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Initializing @ Sequence: {}",
initializationInput.extendedSequenceNumber());
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        /**
         * Handles record processing logic. The Amazon Kinesis Client Library will invoke
this method to deliver
         * data records to the application. In this example we simply log our records.
         *
         * @param processRecordsInput Provides the records to be processed as well as
information and capabilities
         *                            related to them (e.g. checkpointing).
         */
        public void processRecords(ProcessRecordsInput processRecordsInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Processing {} record(s)", processRecordsInput.records().size());
                processRecordsInput.records().forEach(r -> log.info("Processing record pk:
{} -- Seq: {}", r.partitionKey(), r.sequenceNumber()));
            } catch (Throwable t) {
                log.error("Caught throwable while processing records. Aborting.");
                Runtime.getRuntime().halt(1);
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        /** Called when the lease tied to this record processor has been lost. Once the
lease has been lost,
         * the record processor can no longer checkpoint.
         *
         * @param leaseLostInput Provides access to functions and data related to the loss
of the lease.
         */
        public void leaseLost(LeaseLostInput leaseLostInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Lost lease, so terminating.");
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        /**
         * Called when all data on this shard has been processed. Checkpointing must occur
in the method for record
         * processing to be considered complete; an exception will be thrown otherwise.
         *
         * @param shardEndedInput Provides access to a checkpointer method for completing
processing of the shard.
         */
        public void shardEnded(ShardEndedInput shardEndedInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Reached shard end checkpointing.");
                shardEndedInput.checkpointer().checkpoint();
```

```
            } catch (ShutdownException | InvalidStateException e) {
                log.error("Exception while checkpointing at shard end. Giving up.", e);
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        /**
         * Invoked when Scheduler has been requested to shut down (i.e. we decide to stop
 running the app by pressing
         * Enter). Checkpoints and logs the data a final time.
         *
         * @param shutdownRequestedInput Provides access to a checkpointer, allowing a
 record processor to checkpoint
         *                               before the shutdown is completed.
         */
        public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Scheduler is shutting down, checkpointing.");
                shutdownRequestedInput.checkpointer().checkpoint();
            } catch (ShutdownException | InvalidStateException e) {
                log.error("Exception while checkpointing at requested shutdown. Giving
 up.", e);
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }
    }

}
```

Developing a Kinesis Client Library Consumer in Python

You can use the Kinesis Client Library (KCL) to build applications that process data from your Kinesis data streams. The Kinesis Client Library is available in multiple languages. This topic discusses Python.

The KCL is a Java library; support for languages other than Java is provided using a multi-language interface called the *MultiLangDaemon*. This daemon is Java-based and runs in the background when you are using a KCL language other than Java. Therefore, if you install the KCL for Python and write your consumer app entirely in Python, you still need Java installed on your system because of the MultiLangDaemon. Further, MultiLangDaemon has some default settings you may need to customize for your use case, for example, the AWS Region that it connects to. For more information about the MultiLangDaemon on GitHub, go to the KCL MultiLangDaemon project page.

To download the Python KCL from GitHub, go to Kinesis Client Library (Python). To download sample code for a Python KCL consumer application, go to the KCL for Python sample project page on GitHub.

You must complete the following tasks when implementing a KCL consumer application in Python:

Tasks

- Implement the RecordProcessor Class Methods (p. 136)
- Modify the Configuration Properties (p. 138)

Implement the RecordProcessor Class Methods

The `RecordProcess` class must extend the `RecordProcessorBase` class to implement the following methods:

```
initialize
```

```
process_records
shutdown_requested
```

This sample provides implementations that you can use as a starting point.

```
#!/usr/bin/env python

# Copyright 2014-2015 Amazon.com, Inc. or its affiliates. All Rights Reserved.
#
# Licensed under the Amazon Software License (the "License").
# You may not use this file except in compliance with the License.
# A copy of the License is located at
#
# http://aws.amazon.com/asl/
#
# or in the "license" file accompanying this file. This file is distributed
# on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied. See the License for the specific language governing
# permissions and limitations under the License.

from __future__ import print_function

import sys
import time

from amazon_kclpy import kcl
from amazon_kclpy.v3 import processor


class RecordProcessor(processor.RecordProcessorBase):
    """
    A RecordProcessor processes data from a shard in a stream. Its methods will be called
 with this pattern:

    * initialize will be called once
    * process_records will be called zero or more times
    * shutdown will be called if this MultiLangDaemon instance loses the lease to this
 shard, or the shard ends due
        a scaling change.
    """
    def __init__(self):
        self._SLEEP_SECONDS = 5
        self._CHECKPOINT_RETRIES = 5
        self._CHECKPOINT_FREQ_SECONDS = 60
        self._largest_seq = (None, None)
        self._largest_sub_seq = None
        self._last_checkpoint_time = None

    def log(self, message):
        sys.stderr.write(message)

    def initialize(self, initialize_input):
        """
        Called once by a KCLProcess before any calls to process_records

        :param amazon_kclpy.messages.InitializeInput initialize_input: Information about
 the lease that this record
            processor has been assigned.
        """
        self._largest_seq = (None, None)
        self._last_checkpoint_time = time.time()

    def checkpoint(self, checkpointer, sequence_number=None, sub_sequence_number=None):
        """
        Checkpoints with retries on retryable exceptions.
```

```
       :param amazon_kclpy.kcl.Checkpointer checkpointer: the checkpointer provided to
either process_records
           or shutdown
       :param str or None sequence_number: the sequence number to checkpoint at.
       :param int or None sub_sequence_number: the sub sequence number to checkpoint at.
       """
       for n in range(0, self._CHECKPOINT_RETRIES):
           try:
               checkpointer.checkpoint(sequence_number, sub_sequence_number)
               return
           except kcl.CheckpointError as e:
               if 'ShutdownException' == e.value:
                   #
                   # A ShutdownException indicates that this record processor should be
shutdown. This is due to
                   # some failover event, e.g. another MultiLangDaemon has taken the lease
for this shard.
                   #
                   print('Encountered shutdown exception, skipping checkpoint')
                   return
               elif 'ThrottlingException' == e.value:
                   #
                   # A ThrottlingException indicates that one of our dependencies is is
over burdened, e.g. too many
                   # dynamo writes. We will sleep temporarily to let it recover.
                   #
                   if self._CHECKPOINT_RETRIES - 1 == n:
                       sys.stderr.write('Failed to checkpoint after {n} attempts, giving
up.\n'.format(n=n))
                       return
                   else:
                       print('Was throttled while checkpointing, will attempt again in {s}
seconds'
                             .format(s=self._SLEEP_SECONDS))
               elif 'InvalidStateException' == e.value:
                   sys.stderr.write('MultiLangDaemon reported an invalid state while
checkpointing.\n')
               else:  # Some other error
                   sys.stderr.write('Encountered an error while checkpointing, error was
{e}.\n'.format(e=e))
           time.sleep(self._SLEEP_SECONDS)

   def process_record(self, data, partition_key, sequence_number, sub_sequence_number):
       """
       Called for each record that is passed to process_records.

       :param str data: The blob of data that was contained in the record.
       :param str partition_key: The key associated with this recod.
       :param int sequence_number: The sequence number associated with this record.
       :param int sub_sequence_number: the sub sequence number associated with this
record.
       """
       ####################################
       # Insert your processing logic here
       ####################################
       self.log("Record (Partition Key: {pk}, Sequence Number: {seq}, Subsequence Number:
{sseq}, Data Size: {ds}"
                .format(pk=partition_key, seq=sequence_number, sseq=sub_sequence_number,
ds=len(data)))

   def should_update_sequence(self, sequence_number, sub_sequence_number):
       """
       Determines whether a new larger sequence number is available

       :param int sequence_number: the sequence number from the current record
```

```
        :param int sub_sequence_number: the sub sequence number from the current record
        :return boolean: true if the largest sequence should be updated, false otherwise
        """
        return self._largest_seq == (None, None) or sequence_number > self._largest_seq[0]
 or \
            (sequence_number == self._largest_seq[0] and sub_sequence_number >
 self._largest_seq[1])

    def process_records(self, process_records_input):
        """
        Called by a KCLProcess with a list of records to be processed and a checkpointer
 which accepts sequence numbers
        from the records to indicate where in the stream to checkpoint.

        :param amazon_kclpy.messages.ProcessRecordsInput process_records_input: the
 records, and metadata about the
            records.
        """
        try:
            for record in process_records_input.records:
                data = record.binary_data
                seq = int(record.sequence_number)
                sub_seq = record.sub_sequence_number
                key = record.partition_key
                self.process_record(data, key, seq, sub_seq)
                if self.should_update_sequence(seq, sub_seq):
                    self._largest_seq = (seq, sub_seq)

            #
            # Checkpoints every self._CHECKPOINT_FREQ_SECONDS seconds
            #
            if time.time() - self._last_checkpoint_time > self._CHECKPOINT_FREQ_SECONDS:
                self.checkpoint(process_records_input.checkpointer,
 str(self._largest_seq[0]), self._largest_seq[1])
                self._last_checkpoint_time = time.time()

        except Exception as e:
            self.log("Encountered an exception while processing records. Exception was
 {e}\n".format(e=e))

    def lease_lost(self, lease_lost_input):
        self.log("Lease has been lost")

    def shard_ended(self, shard_ended_input):
        self.log("Shard has ended checkpointing")
        shard_ended_input.checkpointer.checkpoint()

    def shutdown_requested(self, shutdown_requested_input):
        self.log("Shutdown has been requested, checkpointing.")
        shutdown_requested_input.checkpointer.checkpoint()


if __name__ == "__main__":
    kcl_process = kcl.KCLProcess(RecordProcessor())
    kcl_process.run()
```

Modify the Configuration Properties

The sample provides default values for the configuration properties, as shown in the following script. You can override any of these properties with your own values.

```
# The script that abides by the multi-language protocol. This script will
# be executed by the MultiLangDaemon, which will communicate with this script
# over STDIN and STDOUT according to the multi-language protocol.
```

```
executableName = sample_kclpy_app.py

# The name of an Amazon Kinesis stream to process.
streamName = words

# Used by the KCL as the name of this application. Will be used as the name
# of an Amazon DynamoDB table which will store the lease and checkpoint
# information for workers with this application name
applicationName = PythonKCLSample

# Users can change the credentials provider the KCL will use to retrieve credentials.
# The DefaultAWSCredentialsProviderChain checks several other providers, which is
# described here:
# http://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/com/amazonaws/auth/
DefaultAWSCredentialsProviderChain.html
AWSCredentialsProvider = DefaultAWSCredentialsProviderChain

# Appended to the user agent of the KCL. Does not impact the functionality of the
# KCL in any other way.
processingLanguage = python/2.7

# Valid options at TRIM_HORIZON or LATEST.
# See http://docs.aws.amazon.com/kinesis/latest/APIReference/
API_GetShardIterator.html#API_GetShardIterator_RequestSyntax
initialPositionInStream = TRIM_HORIZON

# The following properties are also available for configuring the KCL Worker that is
 created
# by the MultiLangDaemon.

# The KCL defaults to us-east-1
#regionName = us-east-1

# Fail over time in milliseconds. A worker which does not renew it's lease within this time
 interval
# will be regarded as having problems and it's shards will be assigned to other workers.
# For applications that have a large number of shards, this msy be set to a higher number
 to reduce
# the number of DynamoDB IOPS required for tracking leases
#failoverTimeMillis = 10000

# A worker id that uniquely identifies this worker among all workers using the same
 applicationName
# If this isn't provided a MultiLangDaemon instance will assign a unique workerId to
 itself.
#workerId =

# Shard sync interval in milliseconds - e.g. wait for this long between shard sync tasks.
#shardSyncIntervalMillis = 60000

# Max records to fetch from Kinesis in a single GetRecords call.
#maxRecords = 10000

# Idle time between record reads in milliseconds.
#idleTimeBetweenReadsInMillis = 1000

# Enables applications flush/checkpoint (if they have some data "in progress", but don't
 get new data for while)
#callProcessRecordsEvenForEmptyRecordList = false

# Interval in milliseconds between polling to check for parent shard completion.
# Polling frequently will take up more DynamoDB IOPS (when there are leases for shards
 waiting on
# completion of parent shards).
#parentShardPollIntervalMillis = 10000
```

```
# Cleanup leases upon shards completion (don't wait until they expire in Kinesis).
# Keeping leases takes some tracking/resources (e.g. they need to be renewed, assigned), so
 by default we try
# to delete the ones we don't need any longer.
#cleanupLeasesUponShardCompletion = true

# Backoff time in milliseconds for Amazon Kinesis Client Library tasks (in the event of
 failures).
#taskBackoffTimeMillis = 500

# Buffer metrics for at most this long before publishing to CloudWatch.
#metricsBufferTimeMillis = 10000

# Buffer at most this many metrics before publishing to CloudWatch.
#metricsMaxQueueSize = 10000

# KCL will validate client provided sequence numbers with a call to Amazon Kinesis before
 checkpointing for calls
# to RecordProcessorCheckpointer#checkpoint(String) by default.
#validateSequenceNumberBeforeCheckpointing = true

# The maximum number of active threads for the MultiLangDaemon to permit.
# If a value is provided then a FixedThreadPool is used with the maximum
# active threads set to the provided value. If a non-positive integer or no
# value is provided a CachedThreadPool is used.
#maxActiveThreads = 0
```

Application Name

The KCL requires an application name that is unique among your applications and among Amazon DynamoDB tables in the same Region. It uses the application name configuration value in the following ways:

- All workers that are associated with this application name are assumed to be working together on the same stream. These workers can be distributed across multiple instances. If you run an additional instance of the same application code, but with a different application name, the KCL treats the second instance as an entirely separate application that is also operating on the same stream.
- The KCL creates a DynamoDB table with the application name and uses the table to maintain state information (such as checkpoints and worker-shard mapping) for the application. Each application has its own DynamoDB table. For more information, see Using a Lease Table to Track the Shards Processed by the KCL Consumer Application (p. 116).

Credentials

You must make your AWS credentials available to one of the credential providers in the default credential providers chain. You can you use the `AWSCredentialsProvider` property to set a credentials provider. If you run your consumer application on an Amazon EC2 instance, we recommend that you configure the instance with an IAM role. AWS credentials that reflect the permissions associated with this IAM role are made available to applications on the instance through its instance metadata. This is the most secure way to manage credentials for a consumer application running on an EC2 instance.

Developing Custom Consumers with Shared Throughput Using the AWS SDK for Java

One of the methods for developing custom Kinesis Data Streams consumers with shared throughout is to use the Amazon Kinesis Data Streams APIs. This section describes using the Kinesis Data Streams APIs with the AWS SDK for Java. The Java sample code in this section demonstrates how to perform basic KDS API operations, and is divided up logically by operation type.

These examples don't represent production-ready code. They don't check for all possible exceptions or account for all possible security or performance considerations.

You can call the Kinesis Data Streams APIs using other different programming languages. For more information about all available AWS SDKs, see Start Developing with Amazon Web Services.

> **Important**
> The recommended method for developing custom Kinesis Data Streams consumers with shared throughout is to use the Kinesis Client Library (KCL). KCL helps you consume and process data from a Kinesis data stream by taking care of many of the complex tasks associated with distributed computing. For more information, see Developing Custom Consumers with Shared Throughput Using KCL.

Topics

Getting Data from a Stream

The Kinesis Data Streams APIs include the `getShardIterator` and `getRecords` methods that you can invoke to retrieve records from a data stream. This is the pull model, where your code draws data records directly from the shards of the data stream.

> **Important**
> We recommend that you use the record processor support provided by KCL to retrieve records from your data streams. This is the push model, where you implement the code that processes the data. The KCL retrieves data records from the data stream and delivers them to your application code. In addition, the KCL provides failover, recovery, and load balancing functionality. For more information, see Developing Custom Consumers with Shared Throughput Using KCL.

However, in some cases you might prefer to use the Kinesis Data Streams APIs. For example, to implement custom tools for monitoring or debugging your data streams.

> **Important**
> Kinesis Data Streams supports changes to the data record retention period of your data stream. For more information, see Changing the Data Retention Period (p. 80).

Using Shard Iterators

You retrieve records from the stream on a per-shard basis. For each shard, and for each batch of records that you retrieve from that shard, you must obtain a *shard iterator*. The shard iterator is used in the `getRecordsRequest` object to specify the shard from which records are to be retrieved. The type associated with the shard iterator determines the point in the shard from which the records should be retrieved (see later in this section for more details). Before you can work with the shard iterator, you need to retrieve the shard, as discussed in DescribeStream API - Deprecated (p. 74).

Obtain the initial shard iterator using the `getShardIterator` method. Obtain shard iterators for additional batches of records using the `getNextShardIterator` method of the `getRecordsResult` object returned by the `getRecords` method. A shard iterator is valid for 5 minutes. If you use a shard iterator while it is valid, you get a new one. Each shard iterator remains valid for 5 minutes, even after it is used.

To obtain the initial shard iterator, instantiate `GetShardIteratorRequest` and pass it to the `getShardIterator` method. To configure the request, specify the stream and the shard ID. For

information about how to obtain the streams in your AWS account, see Listing Streams (p. 71). For information about how to obtain the shards in a stream, see DescribeStream API - Deprecated (p. 74).

```
String shardIterator;
GetShardIteratorRequest getShardIteratorRequest = new GetShardIteratorRequest();
getShardIteratorRequest.setStreamName(myStreamName);
getShardIteratorRequest.setShardId(shard.getShardId());
getShardIteratorRequest.setShardIteratorType("TRIM_HORIZON");

GetShardIteratorResult getShardIteratorResult =
 client.getShardIterator(getShardIteratorRequest);
shardIterator = getShardIteratorResult.getShardIterator();
```

This sample code specifies `TRIM_HORIZON` as the iterator type when obtaining the initial shard iterator. This iterator type means that records should be returned beginning with the first record added to the shard—rather than beginning with the most recently added record, also known as the *tip*. The following are possible iterator types:

- `AT_SEQUENCE_NUMBER`
- `AFTER_SEQUENCE_NUMBER`
- `AT_TIMESTAMP`
- `TRIM_HORIZON`
- `LATEST`

For more information, see ShardIteratorType.

Some iterator types require that you specify a sequence number in addition to the type; for example:

```
getShardIteratorRequest.setShardIteratorType("AT_SEQUENCE_NUMBER");
getShardIteratorRequest.setStartingSequenceNumber(specialSequenceNumber);
```

After you obtain a record using `getRecords`, you can get the sequence number for the record by calling the record's `getSequenceNumber` method.

```
record.getSequenceNumber()
```

In addition, the code that adds records to the data stream can get the sequence number for an added record by calling `getSequenceNumber` on the result of `putRecord`.

```
lastSequenceNumber = putRecordResult.getSequenceNumber();
```

You can use sequence numbers to guarantee strictly increasing ordering of records. For more information, see the code example in PutRecord Example (p. 99).

Using GetRecords

After you obtain the shard iterator, instantiate a `GetRecordsRequest` object. Specify the iterator for the request using the `setShardIterator` method.

Optionally, you can also set the number of records to retrieve using the `setLimit` method. The number of records returned by `getRecords` is always equal to or less than this limit. If you do not specify this limit, `getRecords` returns 10 MB of retrieved records. The sample code below sets this limit to 25 records.

If no records are returned, that means no data records are currently available from this shard at the sequence number referenced by the shard iterator. In this situation, your application should wait for an amount of time that's appropriate for the data sources for the stream, but at least 1 second. Then try to get data from the shard again using the shard iterator returned by the preceding call to `getRecords`. There is about a 3-second latency from the time that a record is added to the stream to the time that it is available from `getRecords`.

Pass the `getRecordsRequest` to the `getRecords` method, and capture the returned value as a `getRecordsResult` object. To get the data records, call the `getRecords` method on the `getRecordsResult` object.

```
GetRecordsRequest getRecordsRequest = new GetRecordsRequest();
getRecordsRequest.setShardIterator(shardIterator);
getRecordsRequest.setLimit(25);

GetRecordsResult getRecordsResult = client.getRecords(getRecordsRequest);
List<Record> records = getRecordsResult.getRecords();
```

To prepare for another call to `getRecords`, obtain the next shard iterator from `getRecordsResult`.

```
shardIterator = getRecordsResult.getNextShardIterator();
```

For best results, sleep for at least 1 second (1,000 milliseconds) between calls to `getRecords` to avoid exceeding the limit on `getRecords` frequency.

```
try {
  Thread.sleep(1000);
}
catch (InterruptedException e) {}
```

Typically, you should call `getRecords` in a loop, even when you're retrieving a single record in a test scenario. A single call to `getRecords` might return an empty record list, even when the shard contains more records at later sequence numbers. When this occurs, the `NextShardIterator` returned along with the empty record list references a later sequence number in the shard, and successive `getRecords` calls eventually returns the records. The following sample demonstrates the use of a loop.

Example: getRecords

The following code example reflects the `getRecords` tips in this section, including making calls in a loop.

```
// Continuously read data records from a shard
List<Record> records;

while (true) {

  // Create a new getRecordsRequest with an existing shardIterator
  // Set the maximum records to return to 25

  GetRecordsRequest getRecordsRequest = new GetRecordsRequest();
  getRecordsRequest.setShardIterator(shardIterator);
```

```
  getRecordsRequest.setLimit(25);

  GetRecordsResult result = client.getRecords(getRecordsRequest);

  // Put the result into record list. The result can be empty.
  records = result.getRecords();

  try {
    Thread.sleep(1000);
  }
  catch (InterruptedException exception) {
    throw new RuntimeException(exception);
  }

  shardIterator = result.getNextShardIterator();
}
```

If you are using the Kinesis Client Library, it might make multiple calls before returning data. This behavior is by design and does not indicate a problem with the KCL or your data.

Adapting to a Reshard

If `getRecordsResult.getNextShardIterator` returns `null`, it indicates that a shard split or merge has occurred that involved this shard. This shard is now in a `CLOSED` state and you have read all available data records from this shard.

In this scenario, you can use `getRecordsResult.childShards` to learn about the new child shards of the shard that is being processed that were created by the split or merge. For more information, see ChildShard.

In the case of a split, the two new shards both have `parentShardId` equal to the shard ID of the shard that you were processing previously. The value of `adjacentParentShardId` for both of these shards is `null`.

In the case of a merge, the single new shard created by the merge has `parentShardId` equal to shard ID of one of the parent shards and `adjacentParentShardId` equal to the shard ID of the other parent shard. Your application has already read all the data from one of these shards. This is the shard for which `getRecordsResult.getNextShardIterator` returned `null`. If the order of the data is important to your application, ensure that it also reads all the data from the other parent shard before reading any new data from the child shard created by the merge.

If you are using multiple processors to retrieve data from the stream (say, one processor per shard), and a shard split or merge occurs, adjust the number of processors up or down to adapt to the change in the number of shards.

For more information about resharding, including a discussion of shards states—such as `CLOSED`—see Resharding a Stream (p. 75).

Interacting with Data Using the AWS Glue Schema Registry

You can integrate your Kinesis data streams with the AWS Glue schema registry. The AWS Glue schema registry allows you to centrally discover, control, and evolve schemas, while ensuring data produced is continuously validated by a registered schema. A schema defines the structure and format of a data record. A schema is a versioned specification for reliable data publication, consumption, or storage. The AWS Glue Schema Registry enables you to improve end-to-end data quality and data governance within your streaming applications. For more information, see AWS Glue Schema Registry. One of the ways to set up this integration is through the `GetRecords` Kinesis Data Streams API available in the AWS Java SDK.

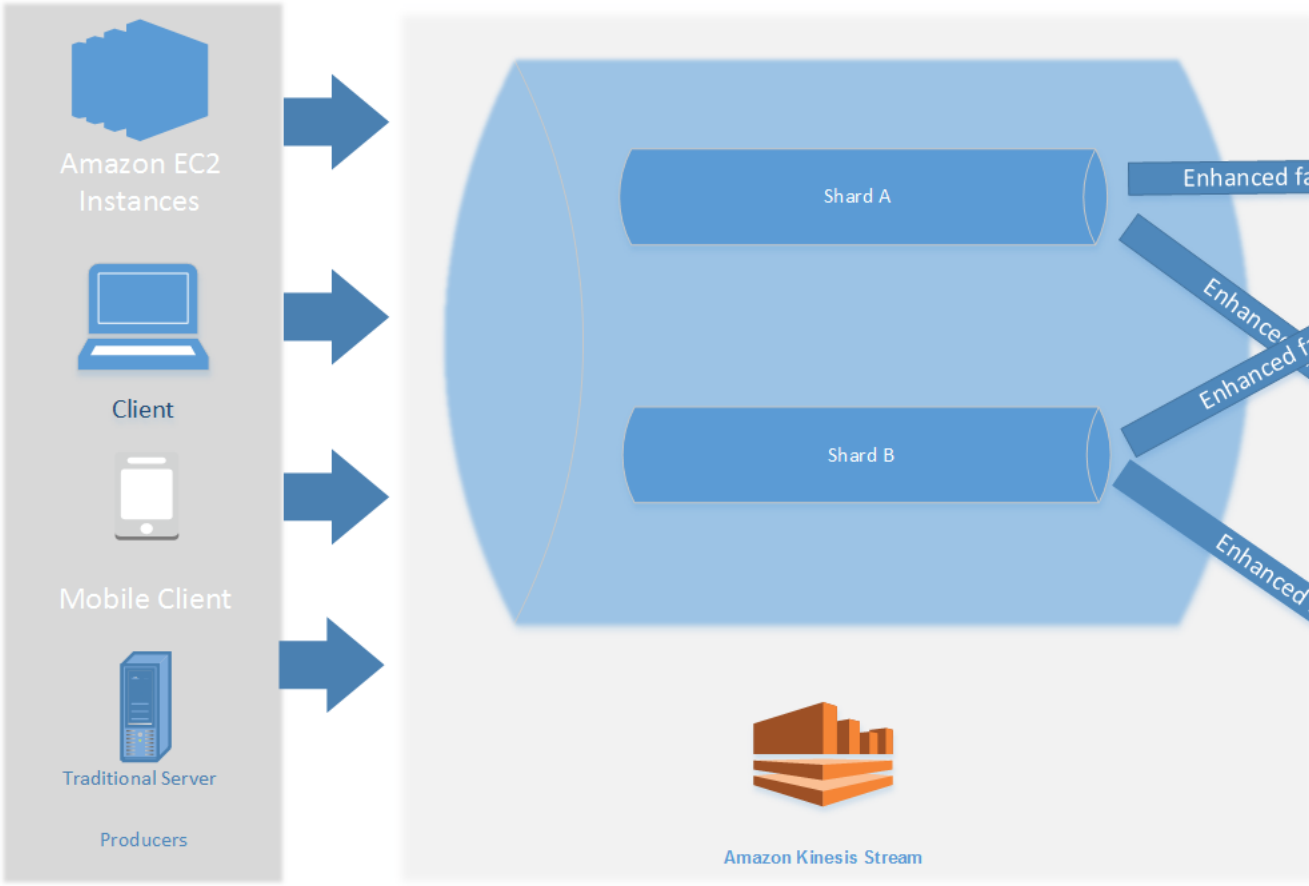
For detailed instructions on how to set up integration of Kinesis Data Streams with Schema Registry using the `GetRecords` Kinesis Data Streams APIs, see the "Interacting with Data Using the Kinesis Data Streams APIs" section in Use Case: Integrating Amazon Kinesis Data Streams with the AWS Glue Schema Registry.

Developing Custom Consumers with Dedicated Throughput (Enhanced Fan-Out)

In Amazon Kinesis Data Streams, you can build consumers that use a feature called *enhanced fan-out*. This feature enables consumers to receive records from a stream with throughput of up to 2 MB of data per second per shard. This throughput is dedicated, which means that consumers that use enhanced fan-out don't have to contend with other consumers that are receiving data from the stream. Kinesis Data Streams pushes data records from the stream to consumers that use enhanced fan-out. Therefore, these consumers don't need to poll for data.

Important
You can register up to twenty consumers per stream to use enhanced fan-out.

The following diagram shows the enhanced fan-out architecture. If you use version 2.0 or later of the Amazon Kinesis Client Library (KCL) to build a consumer, the KCL sets up the consumer to use enhanced fan-out to receive data from all the shards of the stream. If you use the API to build a consumer that uses enhanced fan-out, then you can subscribe to individual shards.



The diagram shows the following:

- A stream with two shards.
- Two consumers that are using enhanced fan-out to receive data from the stream: Consumer X and Consumer Y. Each of the two consumers is subscribed to all of the shards and all of the records of the stream. If you use version 2.0 or later of the KCL to build a consumer, the KCL automatically subscribes that consumer to all the shards of the stream. On the other hand, if you use the API to build a consumer, you can subscribe to individual shards.
- Arrows representing the enhanced fan-out pipes that the consumers use to receive data from the stream. An enhanced fan-out pipe provides up to 2 MB/sec of data per shard, independently of any other pipes or of the total number of consumers.

Topics

Developing Enhanced Fan-Out Consumers with KCL 2.x

Consumers that use *enhanced fan-out* in Amazon Kinesis Data Streams can receive records from a data stream with dedicated throughput of up to 2 MB of data per second per shard. This type of consumer doesn't have to contend with other consumers that are receiving data from the stream. For more information, see Developing Custom Consumers with Dedicated Throughput (Enhanced Fan-Out) (p. 154).

You can use version 2.0 or later of the Kinesis Client Library (KCL) to develop applications that use enhanced fan-out to receive data from streams. The KCL automatically subscribes your application to all the shards of a stream, and ensures that your consumer application can read with a throughput value of 2 MB/sec per shard. If you want to use the KCL without turning on enhanced fan-out, see Developing Consumers Using the Kinesis Client Library 2.0.

Topics

Developing Enhanced Fan-Out Consumers Using KCL 2.x in Java

You can use version 2.0 or later of the Kinesis Client Library (KCL) to develop applications in Amazon Kinesis Data Streams to receive data from streams using enhanced fan-out. The following code shows an example implementation in Java of `ProcessorFactory` and `RecordProcessor`.

It is recommended that you use `KinesisClientUtil` to create `KinesisAsyncClient` and to configure `maxConcurrency` in `KinesisAsyncClient`.

> **Important**
> The Amazon Kinesis Client might see significantly increased latency, unless you configure `KinesisAsyncClient` to have a `maxConcurrency` high enough to allow all leases plus additional usages of `KinesisAsyncClient`.

```
/*
 *  Copyright 2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 *  Licensed under the Amazon Software License (the "License").
 *  You may not use this file except in compliance with the License.
 *  A copy of the License is located at
```

```
 *
 *  http://aws.amazon.com/asl/
 *
 *  or in the "license" file accompanying this file. This file is distributed
 *  on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 *  express or implied. See the License for the specific language governing
 *  permissions and limitations under the License.
 */

/*
 * Copyright 2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License.
 * A copy of the License is located at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * or in the "license" file accompanying this file. This file is distributed
 * on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 * express or implied. See the License for the specific language governing
 * permissions and limitations under the License.
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.UUID;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

import org.apache.commons.lang3.ObjectUtils;
import org.apache.commons.lang3.RandomStringUtils;
import org.apache.commons.lang3.RandomUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.slf4j.MDC;

import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.cloudwatch.CloudWatchAsyncClient;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
import software.amazon.awssdk.services.kinesis.model.PutRecordRequest;
import software.amazon.kinesis.common.ConfigsBuilder;
import software.amazon.kinesis.common.KinesisClientUtil;
import software.amazon.kinesis.coordinator.Scheduler;
import software.amazon.kinesis.exceptions.InvalidStateException;
import software.amazon.kinesis.exceptions.ShutdownException;
import software.amazon.kinesis.lifecycle.events.InitializationInput;
import software.amazon.kinesis.lifecycle.events.LeaseLostInput;
import software.amazon.kinesis.lifecycle.events.ProcessRecordsInput;
import software.amazon.kinesis.lifecycle.events.ShardEndedInput;
import software.amazon.kinesis.lifecycle.events.ShutdownRequestedInput;
import software.amazon.kinesis.processor.ShardRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;

public class SampleSingle {

    private static final Logger log = LoggerFactory.getLogger(SampleSingle.class);
```

```
   public static void main(String... args) {
       if (args.length < 1) {
           log.error("At a minimum, the stream name is required as the first argument. The
Region may be specified as the second argument.");
           System.exit(1);
       }

       String streamName = args[0];
       String region = null;
       if (args.length > 1) {
           region = args[1];
       }

       new SampleSingle(streamName, region).run();
   }

   private final String streamName;
   private final Region region;
   private final KinesisAsyncClient kinesisClient;

   private SampleSingle(String streamName, String region) {
       this.streamName = streamName;
       this.region = Region.of(ObjectUtils.firstNonNull(region, "us-east-2"));
       this.kinesisClient =
KinesisClientUtil.createKinesisAsyncClient(KinesisAsyncClient.builder().region(this.region));
   }

   private void run() {
       ScheduledExecutorService producerExecutor =
Executors.newSingleThreadScheduledExecutor();
       ScheduledFuture<?> producerFuture =
producerExecutor.scheduleAtFixedRate(this::publishRecord, 10, 1, TimeUnit.SECONDS);

       DynamoDbAsyncClient dynamoClient =
DynamoDbAsyncClient.builder().region(region).build();
       CloudWatchAsyncClient cloudWatchClient =
CloudWatchAsyncClient.builder().region(region).build();
       ConfigsBuilder configsBuilder = new ConfigsBuilder(streamName, streamName,
kinesisClient, dynamoClient, cloudWatchClient, UUID.randomUUID().toString(), new
SampleRecordProcessorFactory());

       Scheduler scheduler = new Scheduler(
               configsBuilder.checkpointConfig(),
               configsBuilder.coordinatorConfig(),
               configsBuilder.leaseManagementConfig(),
               configsBuilder.lifecycleConfig(),
               configsBuilder.metricsConfig(),
               configsBuilder.processorConfig(),
               configsBuilder.retrievalConfig()
       );

       Thread schedulerThread = new Thread(scheduler);
       schedulerThread.setDaemon(true);
       schedulerThread.start();

       System.out.println("Press enter to shutdown");
       BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
       try {
           reader.readLine();
       } catch (IOException ioex) {
           log.error("Caught exception while waiting for confirm. Shutting down.", ioex);
       }

       log.info("Cancelling producer, and shutting down executor.");
       producerFuture.cancel(true);
       producerExecutor.shutdownNow();
```

```
        Future<Boolean> gracefulShutdownFuture = scheduler.startGracefulShutdown();
        log.info("Waiting up to 20 seconds for shutdown to complete.");
        try {
            gracefulShutdownFuture.get(20, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            log.info("Interrupted while waiting for graceful shutdown. Continuing.");
        } catch (ExecutionException e) {
            log.error("Exception while executing graceful shutdown.", e);
        } catch (TimeoutException e) {
            log.error("Timeout while waiting for shutdown. Scheduler may not have
exited.");
        }
        log.info("Completed, shutting down now.");
    }

    private void publishRecord() {
        PutRecordRequest request = PutRecordRequest.builder()
                .partitionKey(RandomStringUtils.randomAlphabetic(5, 20))
                .streamName(streamName)
                .data(SdkBytes.fromByteArray(RandomUtils.nextBytes(10)))
                .build();
        try {
            kinesisClient.putRecord(request).get();
        } catch (InterruptedException e) {
            log.info("Interrupted, assuming shutdown.");
        } catch (ExecutionException e) {
            log.error("Exception while sending data to Kinesis. Will try again next
cycle.", e);
        }
    }

    private static class SampleRecordProcessorFactory implements
ShardRecordProcessorFactory {
        public ShardRecordProcessor shardRecordProcessor() {
            return new SampleRecordProcessor();
        }
    }


    private static class SampleRecordProcessor implements ShardRecordProcessor {

        private static final String SHARD_ID_MDC_KEY = "ShardId";

        private static final Logger log =
LoggerFactory.getLogger(SampleRecordProcessor.class);

        private String shardId;

        public void initialize(InitializationInput initializationInput) {
            shardId = initializationInput.shardId();
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Initializing @ Sequence: {}",
initializationInput.extendedSequenceNumber());
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        public void processRecords(ProcessRecordsInput processRecordsInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Processing {} record(s)", processRecordsInput.records().size());
                processRecordsInput.records().forEach(r -> log.info("Processing record pk:
{} -- Seq: {}", r.partitionKey(), r.sequenceNumber()));
```

```
            } catch (Throwable t) {
                log.error("Caught throwable while processing records. Aborting.");
                Runtime.getRuntime().halt(1);
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        public void leaseLost(LeaseLostInput leaseLostInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Lost lease, so terminating.");
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        public void shardEnded(ShardEndedInput shardEndedInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Reached shard end checkpointing.");
                shardEndedInput.checkpointer().checkpoint();
            } catch (ShutdownException | InvalidStateException e) {
                log.error("Exception while checkpointing at shard end. Giving up.", e);
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }

        public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
            MDC.put(SHARD_ID_MDC_KEY, shardId);
            try {
                log.info("Scheduler is shutting down, checkpointing.");
                shutdownRequestedInput.checkpointer().checkpoint();
            } catch (ShutdownException | InvalidStateException e) {
                log.error("Exception while checkpointing at requested shutdown. Giving
 up.", e);
            } finally {
                MDC.remove(SHARD_ID_MDC_KEY);
            }
        }
    }

}
```

Developing Enhanced Fan-Out Consumers with the Kinesis Data Streams API

Enhanced fan-out is an Amazon Kinesis Data Streams feature that enables consumers to receive records from a data stream with dedicated throughput of up to 2 MB of data per second per shard. A consumer that uses enhanced fan-out doesn't have to contend with other consumers that are receiving data from the stream. For more information, see Developing Custom Consumers with Dedicated Throughput (Enhanced Fan-Out) (p. 154).

You can use API operations to build a consumer that uses enhanced fan-out in Kinesis Data Streams.

To register a consumer with enhanced fan-out using the Kinesis Data Streams API

1. Call RegisterStreamConsumer to register your application as a consumer that uses enhanced fan-out. Kinesis Data Streams generates an Amazon Resource Name (ARN) for the consumer and returns it in the response.

2. To start listening to a specific shard, pass the consumer ARN in a call to SubscribeToShard. Kinesis Data Streams then starts pushing the records from that shard to you, in the form of events of type SubscribeToShardEvent over an HTTP/2 connection. The connection remains open for up to 5 minutes. Call SubscribeToShard again if you want to continue receiving records from the shard after the `future` that is returned by the call to SubscribeToShard completes normally or exceptionally.

   Note
   `SubscribeToShard` API also returns the list of the child shards of the current shard when the end of the current shard is reached.

3. To deregister a consumer that is using enhanced fan-out, call DeregisterStreamConsumer.

The following code is an example of how you can subscribe your consumer to a shard, renew the subscription periodically, and handle the events.

```
    import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
    import software.amazon.awssdk.services.kinesis.model.ShardIteratorType;
    import software.amazon.awssdk.services.kinesis.model.SubscribeToShardEvent;
    import software.amazon.awssdk.services.kinesis.model.SubscribeToShardRequest;
    import software.amazon.awssdk.services.kinesis.model.SubscribeToShardResponseHandler;

    import java.util.concurrent.CompletableFuture;

    /**
     * See https://github.com/awsdocs/aws-doc-sdk-examples/blob/master/javav2/example_code/
kinesis/src/main/java/com/example/kinesis/KinesisStreamEx.java
     * for complete code and more examples.
     */
    public class SubscribeToShardSimpleImpl {

        private static final String CONSUMER_ARN = "arn:aws:kinesis:us-
east-1:123456789123:stream/foobar/consumer/test-consumer:1525898737";
        private static final String SHARD_ID = "shardId-000000000000";

        public static void main(String[] args) {

            KinesisAsyncClient client = KinesisAsyncClient.create();

            SubscribeToShardRequest request = SubscribeToShardRequest.builder()
                    .consumerARN(CONSUMER_ARN)
                    .shardId(SHARD_ID)
                    .startingPosition(s -> s.type(ShardIteratorType.LATEST)).build();

            // Call SubscribeToShard iteratively to renew the subscription periodically.
            while(true) {
                // Wait for the CompletableFuture to complete normally or exceptionally.
                callSubscribeToShardWithVisitor(client, request).join();
            }

            // Close the connection before exiting.
            // client.close();
        }


        /**
         * Subscribes to the stream of events by implementing the
 SubscribeToShardResponseHandler.Visitor interface.
         */
        private static CompletableFuture<Void>
 callSubscribeToShardWithVisitor(KinesisAsyncClient client, SubscribeToShardRequest
 request) {
            SubscribeToShardResponseHandler.Visitor visitor = new
 SubscribeToShardResponseHandler.Visitor() {
                @Override
```

```
                public void visit(SubscribeToShardEvent event) {
                    System.out.println("Received subscribe to shard event " + event);
                }
            };
            SubscribeToShardResponseHandler responseHandler =
SubscribeToShardResponseHandler
                    .builder()
                    .onError(t -> System.err.println("Error during stream - " +
t.getMessage()))
                    .subscriber(visitor)
                    .build();
            return client.subscribeToShard(request, responseHandler);
        }
    }
```

If `event.ContinuationSequenceNumber` returns `null`, it indicates that a shard split or merge has occurred that involved this shard. This shard is now in a `CLOSED` state, and you have read all available data records from this shard. In this scenario, per example above, you can use `event.childShards` to learn about the new child shards of the shard that is being processed that were created by the split or merge. For more information, see ChildShard.

Interacting with Data Using the AWS Glue Schema Registry

You can integrate your Kinesis data streams with the AWS Glue schema registry. The AWS Glue schema registry allows you to centrally discover, control, and evolve schemas, while ensuring data produced is continuously validated by a registered schema. A schema defines the structure and format of a data record. A schema is a versioned specification for reliable data publication, consumption, or storage. The AWS Glue Schema Registry enables you to improve end-to-end data quality and data governance within your streaming applications. For more information, see AWS Glue Schema Registry. One of the ways to set up this integration is through the `GetRecords` Kinesis Data Streams API available in the AWS Java SDK.

For detailed instructions on how to set up integration of Kinesis Data Streams with Schema Registry using the `GetRecords` Kinesis Data Streams APIs, see the "Interacting with Data Using the Kinesis Data Streams APIs" section in Use Case: Integrating Amazon Kinesis Data Streams with the AWS Glue Schema Registry.

Managing Enhanced Fan-Out Consumers with the AWS Management Console

Consumers that use *enhanced fan-out* in Amazon Kinesis Data Streams can receive records from a data stream with dedicated throughput of up to 2 MB of data per second per shard. For more information, see Developing Custom Consumers with Dedicated Throughput (Enhanced Fan-Out) (p. 154).

You can use the AWS Management Console to see a list of all the consumers that are registered to use enhanced fan-out with a specific stream. For each such consumer, you can see details such as ARN, status, creation date, and monitoring metrics.

To view consumers that are registered to use enhanced fan-out, their status, creation date, and metrics on the console

1. Sign in to the AWS Management Console and open the Kinesis console at https://console.aws.amazon.com/kinesis.
2. Choose **Data Streams** in the navigation pane.
3. Choose a Kinesis data stream to view its details.
4. On the details page for the stream, choose the **Enhanced fan-out** tab.
5. Choose a consumer to see its name, status, and date of registration.

To deregister a consumer

1. Open the Kinesis console at https://console.aws.amazon.com/kinesis.
2. Choose **Data Streams** in the navigation pane.
3. Choose a Kinesis data stream to view its details.
4. On the details page for the stream, choose the **Enhanced fan-out** tab.
5. Select the check box to the left of the name of every consumer that you want to deregister.
6. Choose **Deregister consumer**.

Migrating Consumers from KCL 1.x to KCL 2.x

This topic explains the differences between versions 1.x and 2.x of the Kinesis Client Library (KCL). It also shows you how to migrate your consumer from version 1.x to version 2.x of the KCL. After you migrate your client, it will start processing records from the last checkpointed location.

Version 2.0 of the KCL introduces the following interface changes:

KCL Interface Changes

KCL 1.x Interface	KCL 2.0 Interface
com.amazonaws.services.kinesis.clientlibrary.interfaces.IRecordProcessor	software.amazon.kinesis.processor.ShardRecordProc
com.amazonaws.services.kinesis.clientlibrary.interfaces.IRecordProcessorFactory	software.amazon.kinesis.processor.ShardRecordProc
com.amazonaws.services.kinesis.clientlibrary.lib.worker.InitialPositionInStream	Folded into software.amazon.kinesis.processor.ShardRecordProc

Topics

Migrating the Record Processor

The following example shows a record processor implemented for KCL 1.x:

```
package com.amazonaws.kcl;

import com.amazonaws.services.kinesis.clientlibrary.exceptions.InvalidStateException;
import com.amazonaws.services.kinesis.clientlibrary.exceptions.ShutdownException;
import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.IRecordProcessorCheckpointer;
import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotificationAware;
import com.amazonaws.services.kinesis.clientlibrary.lib.worker.ShutdownReason;
import com.amazonaws.services.kinesis.clientlibrary.types.InitializationInput;
import com.amazonaws.services.kinesis.clientlibrary.types.ProcessRecordsInput;
import com.amazonaws.services.kinesis.clientlibrary.types.ShutdownInput;
```

```
public class TestRecordProcessor implements IRecordProcessor, IShutdownNotificationAware {
    @Override
    public void initialize(InitializationInput initializationInput) {
        //
        // Setup record processor
        //
    }

    @Override
    public void processRecords(ProcessRecordsInput processRecordsInput) {
        //
        // Process records, and possibly checkpoint
        //
    }

    @Override
    public void shutdown(ShutdownInput shutdownInput) {
        if (shutdownInput.getShutdownReason() == ShutdownReason.TERMINATE) {
            try {
                shutdownInput.getCheckpointer().checkpoint();
            } catch (ShutdownException | InvalidStateException e) {
                throw new RuntimeException(e);
            }
        }
    }

    @Override
    public void shutdownRequested(IRecordProcessorCheckpointer checkpointer) {
        try {
            checkpointer.checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow exception
            //
            e.printStackTrace();
        }
    }
}
```

To migrate the record processor class

1. Change the interfaces from `com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor` and `com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotificationAware` to `software.amazon.kinesis.processor.ShardRecordProcessor`, as follows:

```
// import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
// import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IShutdownNotificationAware;
import software.amazon.kinesis.processor.ShardRecordProcessor;

// public class TestRecordProcessor implements IRecordProcessor,
 IShutdownNotificationAware {
public class TestRecordProcessor implements ShardRecordProcessor {
```

2. Update the `import` statements for the `initialize` and `processRecords` methods.

```
// import com.amazonaws.services.kinesis.clientlibrary.types.InitializationInput;
import software.amazon.kinesis.lifecycle.events.InitializationInput;

//import com.amazonaws.services.kinesis.clientlibrary.types.ProcessRecordsInput;
```

```
import software.amazon.kinesis.lifecycle.events.ProcessRecordsInput;
```

3. Replace the `shutdown` method with the following new methods: `leaseLost`, `shardEnded`, and `shutdownRequested`.

```
//    @Override
//    public void shutdownRequested(IRecordProcessorCheckpointer checkpointer) {
//        //
//        // This is moved to shardEnded(...)
//        //
//        try {
//            checkpointer.checkpoint();
//        } catch (ShutdownException | InvalidStateException e) {
//            //
//            // Swallow exception
//            //
//            e.printStackTrace();
//        }
//    }

    @Override
    public void leaseLost(LeaseLostInput leaseLostInput) {

    }

    @Override
    public void shardEnded(ShardEndedInput shardEndedInput) {
        try {
            shardEndedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow the exception
            //
            e.printStackTrace();
        }
    }

//    @Override
//    public void shutdownRequested(IRecordProcessorCheckpointer checkpointer) {
//        //
//        // This is moved to shutdownRequested(ShutdownReauestedInput)
//        //
//        try {
//            checkpointer.checkpoint();
//        } catch (ShutdownException | InvalidStateException e) {
//            //
//            // Swallow exception
//            //
//            e.printStackTrace();
//        }
//    }

    @Override
    public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
        try {
            shutdownRequestedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow the exception
            //
            e.printStackTrace();
        }
    }
```

The following is the updated version of the record processor class.

```
package com.amazonaws.kcl;

import software.amazon.kinesis.exceptions.InvalidStateException;
import software.amazon.kinesis.exceptions.ShutdownException;
import software.amazon.kinesis.lifecycle.events.InitializationInput;
import software.amazon.kinesis.lifecycle.events.LeaseLostInput;
import software.amazon.kinesis.lifecycle.events.ProcessRecordsInput;
import software.amazon.kinesis.lifecycle.events.ShardEndedInput;
import software.amazon.kinesis.lifecycle.events.ShutdownRequestedInput;
import software.amazon.kinesis.processor.ShardRecordProcessor;

public class TestRecordProcessor implements ShardRecordProcessor {
    @Override
    public void initialize(InitializationInput initializationInput) {

    }

    @Override
    public void processRecords(ProcessRecordsInput processRecordsInput) {

    }

    @Override
    public void leaseLost(LeaseLostInput leaseLostInput) {

    }

    @Override
    public void shardEnded(ShardEndedInput shardEndedInput) {
        try {
            shardEndedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow the exception
            //
            e.printStackTrace();
        }
    }

    @Override
    public void shutdownRequested(ShutdownRequestedInput shutdownRequestedInput) {
        try {
            shutdownRequestedInput.checkpointer().checkpoint();
        } catch (ShutdownException | InvalidStateException e) {
            //
            // Swallow the exception
            //
            e.printStackTrace();
        }
    }
}
```

Migrating the Record Processor Factory

The record processor factory is responsible for creating record processors when a lease is acquired. The following is an example of a KCL 1.x factory.

```
package com.amazonaws.kcl;

import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
```

```
import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory;

public class TestRecordProcessorFactory implements IRecordProcessorFactory {
    @Override
    public IRecordProcessor createProcessor() {
        return new TestRecordProcessor();
    }
}
```

To migrate the record processor factory

1. Change the implemented interface from `com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory` to `software.amazon.kinesis.processor.ShardRecordProcessorFactory`, as follows.

```
// import com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessor;

// import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.v2.IRecordProcessorFactory;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;

// public class TestRecordProcessorFactory implements IRecordProcessorFactory {
public class TestRecordProcessorFactory implements ShardRecordProcessorFactory {
```

2. Change the return signature for `createProcessor`.

```
// public IRecordProcessor createProcessor() {
public ShardRecordProcessor shardRecordProcessor() {
```

The following is an example of the record processor factory in 2.0:

```
package com.amazonaws.kcl;

import software.amazon.kinesis.processor.ShardRecordProcessor;
import software.amazon.kinesis.processor.ShardRecordProcessorFactory;

public class TestRecordProcessorFactory implements ShardRecordProcessorFactory {
    @Override
    public ShardRecordProcessor shardRecordProcessor() {
        return new TestRecordProcessor();
    }
}
```

Migrating the Worker

In version 2.0 of the KCL, a new class, called `Scheduler`, replaces the `Worker` class. The following is an example of a KCL 1.x worker.

```
final KinesisClientLibConfiguration config = new KinesisClientLibConfiguration(...)
final IRecordProcessorFactory recordProcessorFactory = new RecordProcessorFactory();
final Worker worker = new Worker.Builder()
    .recordProcessorFactory(recordProcessorFactory)
    .config(config)
    .build();
```

To migrate the worker

1. Change the `import` statement for the `Worker` class to the import statements for the `Scheduler` and `ConfigsBuilder` classes.

```
// import com.amazonaws.services.kinesis.clientlibrary.lib.worker.Worker;
import software.amazon.kinesis.coordinator.Scheduler;
import software.amazon.kinesis.common.ConfigsBuilder;
```

2. Create the `ConfigsBuilder` and a `Scheduler` as shown in the following example.

   It is recommended that you use `KinesisClientUtil` to create `KinesisAsyncClient` and to configure `maxConcurrency` in `KinesisAsyncClient`.

   Important
   The Amazon Kinesis Client might see significantly increased latency, unless you configure `KinesisAsyncClient` to have a `maxConcurrency` high enough to allow all leases plus additional usages of `KinesisAsyncClient`.

```
import java.util.UUID;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;
import software.amazon.awssdk.services.cloudwatch.CloudWatchAsyncClient;
import software.amazon.awssdk.services.kinesis.KinesisAsyncClient;
import software.amazon.kinesis.common.ConfigsBuilder;
import software.amazon.kinesis.common.KinesisClientUtil;
import software.amazon.kinesis.coordinator.Scheduler;

...

Region region = Region.AP_NORTHEAST_2;
KinesisAsyncClient kinesisClient =
 KinesisClientUtil.createKinesisAsyncClient(KinesisAsyncClient.builder().region(region));
DynamoDbAsyncClient dynamoClient =
 DynamoDbAsyncClient.builder().region(region).build();
CloudWatchAsyncClient cloudWatchClient =
 CloudWatchAsyncClient.builder().region(region).build();

ConfigsBuilder configsBuilder = new ConfigsBuilder(streamName, applicationName,
 kinesisClient, dynamoClient, cloudWatchClient, UUID.randomUUID().toString(), new
 SampleRecordProcessorFactory());

Scheduler scheduler = new Scheduler(
    configsBuilder.checkpointConfig(),
    configsBuilder.coordinatorConfig(),
    configsBuilder.leaseManagementConfig(),
    configsBuilder.lifecycleConfig(),
    configsBuilder.metricsConfig(),
    configsBuilder.processorConfig(),
    configsBuilder.retrievalConfig()
    );
```

Configuring the Amazon Kinesis Client

With the 2.0 release of the Kinesis Client Library, the configuration of the client moved from a single configuration class (`KinesisClientLibConfiguration`) to six configuration classes. The following table describes the migration.

Configuration Fields and Their New Classes

Original Field	New Configuration Class	Description
`applicationName`	`ConfigsBuilder`	The name for this the KCL application. Used as the default for the `tableName` and `consumerName`.
`tableName`	`ConfigsBuilder`	Allows overriding the table name used for the Amazon DynamoDB lease table.
`streamName`	`ConfigsBuilder`	The name of the stream that this application processes records from.
`kinesisEndpoint`	`ConfigsBuilder`	This option has been removed. See Client Configuration Removals.
`dynamoDBEndpoint`	`ConfigsBuilder`	This option has been removed. See Client Configuration Removals.
`initialPositionInStreamExtended`	`RetrievalConfig`	The location in the shard from which the KCL begins fetching records, starting with the application's initial run.
`kinesisCredentialsProvider`	`ConfigsBuilder`	This option has been removed. See Client Configuration Removals.
`dynamoDBCredentialsProvider`	`ConfigsBuilder`	This option has been removed. See Client Configuration Removals.
`cloudWatchCredentialsProvider`	`ConfigsBuilder`	This option has been removed. See Client Configuration Removals.
`failoverTimeMillis`	`LeaseManagementConfig`	The number of milliseconds that must pass before you can consider a lease owner to have failed.
`workerIdentifier`	`ConfigsBuilder`	A unique identifier that represents this instantiation of the application processor. This must be unique.
`shardSyncIntervalMillis`	`LeaseManagementConfig`	The time between shard sync calls.
`maxRecords`	`PollingConfig`	Allows setting the maximum number of records that Kinesis returns.
`idleTimeBetweenReadsInMillis`	`CoordinatorConfig`	This option has been removed. See Idle Time Removal.
`callProcessRecordsEvenForEmptyRecordList`	`ProcessorConfig`	When set, the record processor is called even when no records were provided from Kinesis.
`parentShardPollIntervalMillis`	`CoordinatorConfig`	How often a record processor should poll to see if the parent shard has been completed.
`cleanupLeasesUponShardCompletion`	`LeaseManagementConfig`	When set, leases are removed as soon as the child leases have started processing.
`ignoreUnexpectedChildShards`	`LeaseManagementConfig`	When set, child shards that have an open shard are ignored. This is primarily for DynamoDB Streams.
`kinesisClientConfig`	`ConfigsBuilder`	This option has been removed. See Client Configuration Removals.

Original Field	New Configuration Class	Description
dynamoDBClientConfig	ConfigsBuilder	This option has been removed. See Client Configuration Removals.
cloudWatchClientConfig	ConfigsBuilder	This option has been removed. See Client Configuration Removals.
taskBackoffTimeMillis	LifecycleConfig	The time to wait to retry failed tasks.
metricsBufferTimeMillis	MetricsConfig	Controls CloudWatch metric publishing.
metricsMaxQueueSize	MetricsConfig	Controls CloudWatch metric publishing.
metricsLevel	MetricsConfig	Controls CloudWatch metric publishing.
metricsEnabledDimensions	MetricsConfig	Controls CloudWatch metric publishing.
validateSequenceNumberBeforeCheckpointing	CheckpointConfig	This option has been removed. See Checkpoint Sequence Number Validation.
regionName	ConfigsBuilder	This option has been removed. See Client Configuration Removal.
maxLeasesForWorker	LeaseManagementConfig	The maximum number of leases a single instance of the application should accept.
maxLeasesToStealAtOneTime	LeaseManagementConfig	The maximum number of leases an application should attempt to steal at one time.
initialLeaseTableReadCapacity	LeaseManagementConfig	The DynamoDB read IOPs that is used if the Kinesis Client Library needs to create a new DynamoDB lease table.
initialLeaseTableWriteCapacity	LeaseManagementConfig	The DynamoDB read IOPs that is used if the Kinesis Client Library needs to create a new DynamoDB lease table.
initialPositionInStream	LeaseManagementConfig	The initial position in the stream that the application should start at. This is only used during initial lease creation.
skipShardSyncAtWorkerInitializationIfLeasesExist	CoordinatorConfig	Disables synchronizing shard data if the lease table contains existing leases. TODO: KinesisEco-438
shardPrioritization	CoordinatorConfig	Which shard prioritization to use.
shutdownGraceMillis	N/A	This option has been removed. See MultiLang Removals.
timeoutInSeconds	N/A	This option has been removed. See MultiLang Removals.
retryGetRecordsInSeconds	PollingConfig	Configures the delay between GetRecords attempts for failures.
maxGetRecordsThreadPool	PollingConfig	The thread pool size used for GetRecords.
maxLeaseRenewalThreads	LeaseManagementConfig	Controls the size of the lease renewer thread pool. The more leases that your application could take, the larger this pool should be.
recordsFetcherFactory	PollingConfig	Allows replacing the factory used to create fetchers that retrieve from streams.

Original Field	New Configuration Class	Description
`logWarningForTaskAfterMillis`	`RetrievalConfig`	How long to wait before a warning is logged if a task hasn't completed.
`listShardsBackoffTimeInMillis`	`RetrievalConfig`	The number of milliseconds to wait between calls to `ListShards` when failures occur.
`maxListShardsRetryAttempts`	`RetrievalConfig`	The maximum number of times that `ListShards` retries before giving up.

Idle Time Removal

In the 1.x version of the KCL, the `idleTimeBetweenReadsInMillis` corresponded to two quantities:

- The amount of time between task dispatching checks. You can now configure this time between tasks by setting `CoordinatorConfig#shardConsumerDispatchPollIntervalMillis`.
- The amount of time to sleep when no records were returned from Kinesis Data Streams. In version 2.0, in enhanced fan-out records are pushed from their respective retriever. Activity on the shard consumer only occurs when a pushed request arrives.

Client Configuration Removals

In version 2.0, the KCL no longer creates clients. It depends on the user to supply a valid client. With this change, all configuration parameters that controlled client creation have been removed. If you need these parameters, you can set them on the clients before providing the clients to `ConfigsBuilder`.

Removed Field	Equivalent Configuration
`kinesisEndpoint`	Configure the SDK `KinesisAsyncClient` with preferred endpoint: `KinesisAsyncClient.builder().endpointOverride(URI.create("https://<kinesis endpoint>")).build()`.
`dynamoDBEndpoint`	Configure the SDK `DynamoDbAsyncClient` with preferred endpoint: `DynamoDbAsyncClient.builder().endpointOverride(URI.create("https://<dynamodb endpoint>")).build()`.
`kinesisClientConfig`	Configure the SDK `KinesisAsyncClient` with the needed configuration: `KinesisAsyncClient.builder().overrideConfiguration(<your configuration>).build()`.
`dynamoDBClientConfig`	Configure the SDK `DynamoDbAsyncClient` with the needed configuration: `DynamoDbAsyncClient.builder().overrideConfiguration(<your configuration>).build()`.
`cloudWatchClientConfig`	Configure the SDK `CloudWatchAsyncClient` with the needed configuration: `CloudWatchAsyncClient.builder().overrideConfiguration(<your configuration>).build()`.
`regionName`	Configure the SDK with the preferred Region. This is the same for all SDK clients. For example, `KinesisAsyncClient.builder().region(Region.US_WEST_2).build()`.

Troubleshooting Kinesis Data Streams Consumers

The following sections offer solutions to some common problems you may find while working with Amazon Kinesis Data Streams consumers.

Some Kinesis Data Streams Records are Skipped When Using the Kinesis Client Library

The most common cause of skipped records is an unhandled exception thrown from `processRecords`. The Kinesis Client Library (KCL) relies on your `processRecords` code to handle any exceptions that arise from processing the data records. Any exception thrown from `processRecords` is absorbed by the KCL. To avoid infinite retries on a recurring failure, the KCL does not resend the batch of records processed at the time of the exception. The KCL then calls `processRecords` for the next batch of data records without restarting the record processor. This effectively results in consumer applications observing skipped records. To prevent skipped records, handle all exceptions within `processRecords` appropriately.

Records Belonging to the Same Shard are Processed by Different Record Processors at the Same Time

For any running Kinesis Client Library (KCL) application, a shard only has one owner. However, multiple record processors may temporarily process the same shard. In the case of a worker instance that loses network connectivity, the KCL assumes that the unreachable worker is no longer processing records, after the failover time expires, and directs other worker instances to take over. For a brief period, new record processors and record processors from the unreachable worker may process data from the same shard.

You should set a failover time that is appropriate for your application. For low-latency applications, the 10-second default may represent the maximum time you want to wait. However, in cases where you expect connectivity issues such as making calls across geographical areas where connectivity could be lost more frequently, this number may be too low.

Your application should anticipate and handle this scenario, especially because network connectivity is usually restored to the previously unreachable worker. If a record processor has its shards taken by another record processor, it must handle the following two cases to perform graceful shutdown:

1. After the current call to `processRecords` is completed, the KCL invokes the shutdown method on the record processor with shutdown reason 'ZOMBIE'. Your record processors are expected to clean up any resources as appropriate and then exit.
2. When you attempt to checkpoint from a 'zombie' worker, the KCL throws `ShutdownException`. After receiving this exception, your code is expected to exit the current method cleanly.

For more information, see Handling Duplicate Records (p. 176).

Consumer Application is Reading at a Slower Rate Than Expected

The most common reasons for read throughput being slower than expected are as follows:

1. Multiple consumer applications have total reads exceeding the per-shard limits. For more information, see Quotas and Limits (p. 7). In this case, increase the number of shards in the Kinesis data stream.
2. The limit that specifies the maximum number of **GetRecords** per call may have been configured with a low value. If you are using the KCL, you may have configured the worker with a low value for the `maxRecords` property. In general, we recommend using the system defaults for this property.
3. The logic inside your `processRecords` call may be taking longer than expected for a number of possible reasons; the logic may be CPU intensive, I/O blocking, or bottlenecked on synchronization. To test if this is true, test run empty record processors and compare the read throughput. For information about how to keep up with the incoming data, see Resharding, Scaling, and Parallel Processing (p. 175).

If you have only one consumer application, it is always possible to read at least two times faster than the put rate. That's because you can write up to 1,000 records per second for writes, up to a maximum total data write rate of 1 MB per second (including partition keys). Each open shard can support up to 5 transactions per second for reads, up to a maximum total data read rate of 2 MB per second. Note that each read (**GetRecords** call) gets a batch of records. The size of the data returned by **GetRecords** varies depending on the utilization of the shard. The maximum size of data that **GetRecords** can return is 10 MB. If a call returns that limit, subsequent calls made within the next 5 seconds throw `ProvisionedThroughputExceededException`.

GetRecords Returns Empty Records Array Even When There is Data in the Stream

Consuming, or getting records is a pull model. Developers are expected to call GetRecords in a continuous loop with no back-offs. Every call to **GetRecords** also returns a `ShardIterator` value, which must be used in the next iteration of the loop.

The **GetRecords** operation does not block. Instead, it returns immediately; with either relevant data records or with an empty `Records` element. An empty `Records` element is returned under two conditions:

1. There is no more data currently in the shard.
2. There is no data near the part of the shard pointed to by the `ShardIterator`.

The latter condition is subtle, but is a necessary design tradeoff to avoid unbounded seek time (latency) when retrieving records. Thus, the stream-consuming application should loop and call **GetRecords**, handling empty records as a matter of course.

In a production scenario, the only time the continuous loop should be exited is when the `NextShardIterator` value is `NULL`. When `NextShardIterator` is `NULL`, it means that the current shard has been closed and the `ShardIterator`value would otherwise point past the last record. If the consuming application never calls **SplitShard** or **MergeShards**, the shard remains open and the calls to **GetRecords** never return a `NextShardIterator` value that is `NULL`.

If you use the Kinesis Client Library (KCL), the above consumption pattern is abstracted for you. This includes automatic handling of a set of shards that dynamically change. With the KCL, the developer

only supplies the logic to process incoming records. This is possible because the library makes continuous calls to **GetRecords** for you.

Shard Iterator Expires Unexpectedly

A new shard iterator is returned by every **GetRecords** request (as `NextShardIterator`), which you then use in the next **GetRecords** request (as `ShardIterator`). Typically, this shard iterator does not expire before you use it. However, you may find that shard iterators expire because you have not called **GetRecords** for more than 5 minutes, or because you've performed a restart of your consumer application.

If the shard iterator expires immediately, before you can use it, this might indicate that the DynamoDB table used by Kinesis does not have enough capacity to store the lease data. This situation is more likely to happen if you have a large number of shards. To solve this problem, increase the write capacity assigned to the shard table. For more information, see Using a Lease Table to Track the Shards Processed by the KCL Consumer Application (p. 116).

Consumer Record Processing Falling Behind

For most use cases, consumer applications are reading the latest data from the stream. In certain circumstances, consumer reads may fall behind, which may not be desired. After you identify how far behind your consumers are reading, look at the most common reasons why consumers fall behind.

Start with the `GetRecords.IteratorAgeMilliseconds` metric, which tracks the read position across all shards and consumers in the stream. Note that if an iterator's age passes 50% of the retention period (by default, 24 hours, configurable up to 365 days), there is risk for data loss due to record expiration. A quick stopgap solution is to increase the retention period. This stops the loss of important data while you troubleshoot the issue further. For more information, see Monitoring the Amazon Kinesis Data Streams Service with Amazon CloudWatch (p. 180). Next, identify how far behind your consumer application is reading from each shard using a custom CloudWatch metric emitted by the Kinesis Client Library (KCL), `MillisBehindLatest`. For more information, see Monitoring the Kinesis Client Library with Amazon CloudWatch (p. 195).

Here are the most common reasons consumers can fall behind:

- Sudden large increases to `GetRecords.IteratorAgeMilliseconds` or `MillisBehindLatest` usually indicate a transient problem, such as API operation failures to a downstream application. You should investigate these sudden increases if either of the metrics consistently display this behavior.
- A gradual increase to these metrics indicates that a consumer is not keeping up with the stream because it is not processing records fast enough. The most common root causes for this behavior are insufficient physical resources or record processing logic that has not scaled with an increase in stream throughput. You can verify this behavior by looking at the other custom CloudWatch metrics that the KCL emits associated with the `processTask` operation, including `RecordProcessor.processRecords.Time`, `Success`, and `RecordsProcessed`.
  - If you see an increase in the `processRecords.Time` metric that correlates with increased throughput, you should analyze your record processing logic to identify why it is not scaling with the increased throughput.
  - If you see an increase to the `processRecords.Time` values that are not correlated with increased throughput, check to see if you are making any blocking calls in the critical path, which are often the cause of slowdowns in record processing. An alternative approach is to increase your parallelism by increasing the number of shards. Finally, confirm you have an adequate amount of physical resources (memory, CPU utilization, etc.) on the underlying processing nodes during peak demand.

Unauthorized KMS master key permission error

This error occurs when a consumer application reads from an encrypted stream without permissions on the KMS master key. To assign permissions to an application to access a KMS key, see Using Key Policies in AWS KMS and Using IAM Policies with AWS KMS.

Common issues, questions, and troubleshooting ideas for consumers

- Why is Kinesis Data Streams trigger unable to invoke my Lambda function?
- How do I detect and troubleshoot ReadProvisionedThroughputExceeded exceptions in Kinesis Data Streams?
- Why am I experiencing high latency issues with Kinesis Data Streams?
- Why is my Kinesis data stream returning a 500 Internal Server Error?
- How do I troubleshoot a blocked or stuck KCL application for Kinesis Data Streams?
- Can I use different Amazon Kinesis Client Library applications with the same Amazon DynamoDB table?

Advanced Topics for Amazon Kinesis Data Streams Consumers

Learn how to optimize your Amazon Kinesis Data Streams consumer.

Contents

Low-Latency Processing

Propagation delay is defined as the end-to-end latency from the moment a record is written to the stream until it is read by a consumer application. This delay varies depending upon a number of factors, but it is primarily affected by the polling interval of consumer applications.

For most applications, we recommend polling each shard one time per second per application. This enables you to have multiple consumer applications processing a stream concurrently without hitting Amazon Kinesis Data Streams limits of 5 `GetRecords` calls per second. Additionally, processing larger batches of data tends to be more efficient at reducing network and other downstream latencies in your application.

The KCL defaults are set to follow the best practice of polling every 1 second. This default results in average propagation delays that are typically below 1 second.

Kinesis Data Streams records are available to be read immediately after they are written. There are some use cases that need to take advantage of this and require consuming data from the stream as soon as it

is available. You can significantly reduce the propagation delay by overriding the KCL default settings to poll more frequently, as shown in the following examples.

Java KCL configuration code:

```
kinesisClientLibConfiguration = new
        KinesisClientLibConfiguration(applicationName,
        streamName,
        credentialsProvider,

 workerId).withInitialPositionInStream(initialPositionInStream).withIdleTimeBetweenReadsInMillis(250);
```

Property file setting for Python and Ruby KCL:

```
idleTimeBetweenReadsInMillis = 250
```

Note
Because Kinesis Data Streams has a limit of 5 `GetRecords` calls per second, per shard, setting the `idleTimeBetweenReadsInMillis` property lower than 200ms may result in your application observing the `ProvisionedThroughputExceededException` exception. Too many of these exceptions can result in exponential back-offs and thereby cause significant unexpected latencies in processing. If you set this property to be at or just above 200 ms and have more than one processing application, you will experience similar throttling.

Using AWS Lambda with the Kinesis Producer Library

The Kinesis Producer Library (KPL) aggregates small user-formatted records into larger records up to 1 MB to make better use of Amazon Kinesis Data Streams throughput. While the KCL for Java supports deaggregating these records, you need to use a special module to deaggregate records when using AWS Lambda as the consumer of your streams. You can obtain the necessary project code and instructions from GitHub at Kinesis Producer Library Deaggregation Modules for AWS Lambda. The components in this project give you the ability to process KPL serialized data within AWS Lambda, in Java, Node.js and Python. These components can also be used as part of a multi-lang KCL application.

Resharding, Scaling, and Parallel Processing

Resharding enables you to increase or decrease the number of shards in a stream in order to adapt to changes in the rate of data flowing through the stream. Resharding is typically performed by an administrative application that monitors shard data-handling metrics. Although the KCL itself doesn't initiate resharding operations, it is designed to adapt to changes in the number of shards that result from resharding.

As noted in Using a Lease Table to Track the Shards Processed by the KCL Consumer Application (p. 116), the KCL tracks the shards in the stream using an Amazon DynamoDB table. When new shards are created as a result of resharding, the KCL discovers the new shards and populates new rows in the table. The workers automatically discover the new shards and create processors to handle the data from them. The KCL also distributes the shards in the stream across all the available workers and record processors.

The KCL ensures that any data that existed in shards prior to the resharding is processed first. After that data has been processed, data from the new shards is sent to record processors. In this way, the KCL preserves the order in which data records were added to the stream for a particular partition key.

Example: Resharding, Scaling, and Parallel Processing

The following example illustrates how the KCL helps you handle scaling and resharding:

- For example, if your application is running on one EC2 instance, and is processing one Kinesis data stream that has four shards. This one instance has one KCL worker and four record processors (one record processor for every shard). These four record processors run in parallel within the same process.
- Next, if you scale the application to use another instance, you have two instances processing one stream that has four shards. When the KCL worker starts up on the second instance, it load-balances with the first instance, so that each instance now processes two shards.
- If you then decide to split the four shards into five shards. The KCL again coordinates the processing across instances: one instance processes three shards, and the other processes two shards. A similar coordination occurs when you merge shards.

Typically, when you use the KCL, you should ensure that the number of instances does not exceed the number of shards (except for failure standby purposes). Each shard is processed by exactly one KCL worker and has exactly one corresponding record processor, so you never need multiple instances to process one shard. However, one worker can process any number of shards, so it's fine if the number of shards exceeds the number of instances.

To scale up processing in your application, you should test a combination of these approaches:

- Increasing the instance size (because all record processors run in parallel within a process)
- Increasing the number of instances up to the maximum number of open shards (because shards can be processed independently)
- Increasing the number of shards (which increases the level of parallelism)

Note that you can use Auto Scaling to automatically scale your instances based on appropriate metrics. For more information, see the Amazon EC2 Auto Scaling User Guide.

When resharding increases the number of shards in the stream, the corresponding increase in the number of record processors increases the load on the EC2 instances that are hosting them. If the instances are part of an Auto Scaling group, and the load increases sufficiently, the Auto Scaling group adds more instances to handle the increased load. You should configure your instances to launch your Amazon Kinesis Data Streams application at startup, so that additional workers and record processors become active on the new instance right away.

For more information about resharding, see Resharding a Stream (p. 75).

Handling Duplicate Records

There are two primary reasons why records may be delivered more than one time to your Amazon Kinesis Data Streams application: producer retries and consumer retries. Your application must anticipate and appropriately handle processing individual records multiple times.

Producer Retries

Consider a producer that experiences a network-related timeout after it makes a call to `PutRecord`, but before it can receive an acknowledgement from Amazon Kinesis Data Streams. The producer cannot be sure if the record was delivered to Kinesis Data Streams. Assuming that every record is important to the application, the producer would have been written to retry the call with the same data. If both `PutRecord` calls on that same data were successfully committed to Kinesis Data Streams, then there will be two Kinesis Data Streams records. Although the two records have identical data, they also have unique sequence numbers. Applications that need strict guarantees should embed a primary key within the record to remove duplicates later when processing. Note that the number of duplicates due to producer retries is usually low compared to the number of duplicates due to consumer retries.

Note
If you use the AWS SDK `PutRecord`, the default configuration retries a failed `PutRecord` call up to three times.

Consumer Retries

Consumer (data processing application) retries happen when record processors restart. Record processors for the same shard restart in the following cases:

1. A worker terminates unexpectedly
2. Worker instances are added or removed
3. Shards are merged or split
4. The application is deployed

In all these cases, the shards-to-worker-to-record-processor mapping is continuously updated to load balance processing. Shard processors that were migrated to other instances restart processing records from the last checkpoint. This results in duplicated record processing as shown in the example below. For more information about load-balancing, see Resharding, Scaling, and Parallel Processing (p. 175).

Example: Consumer Retries Resulting in Redelivered Records

In this example, you have an application that continuously reads records from a stream, aggregates records into a local file, and uploads the file to Amazon S3. For simplicity, assume there is only 1 shard and 1 worker processing the shard. Consider the following example sequence of events, assuming that the last checkpoint was at record number 10000:

1. A worker reads the next batch of records from the shard, records 10001 to 20000.
2. The worker then passes the batch of records to the associated record processor.
3. The record processor aggregates the data, creates an Amazon S3 file, and uploads the file to Amazon S3 successfully.
4. Worker terminates unexpectedly before a new checkpoint can occur.
5. Application, worker, and record processor restart.
6. Worker now begins reading from the last successful checkpoint, in this case 10001.

Thus, records 10001-20000 are consumed more than one time.

Being Resilient to Consumer Retries

Even though records may be processed more than one time, your application may want to present the side effects as if records were processed only one time (idempotent processing). Solutions to this problem vary in complexity and accuracy. If the destination of the final data can handle duplicates well, we recommend relying on the final destination to achieve idempotent processing. For example, with Opensearch you can use a combination of versioning and unique IDs to prevent duplicated processing.

In the example application in the previous section, it continuously reads records from a stream, aggregates records into a local file, and uploads the file to Amazon S3. As illustrated, records 10001 -20000 are consumed more than one time resulting in multiple Amazon S3 files with the same data. One way to mitigate duplicates from this example is to ensure that step 3 uses the following scheme:

1. Record Processor uses a fixed number of records per Amazon S3 file, such as 5000.
2. The file name uses this schema: Amazon S3 prefix, shard-id, and `First-Sequence-Num`. In this case, it could be something like `sample-shard000001-10001`.
3. After you upload the Amazon S3 file, checkpoint by specifying `Last-Sequence-Num`. In this case, you would checkpoint at record number 15000.

With this scheme, even if records are processed more than one time, the resulting Amazon S3 file has the same name and has the same data. The retries only result in writing the same data to the same file more than one time.

In the case of a reshard operation, the number of records left in the shard may be less than your desired fixed number needed. In this case, your `shutdown()` method has to flush the file to Amazon S3 and checkpoint on the last sequence number. The above scheme is compatible with reshard operations as well.

Handling Startup, Shutdown, and Throttling

Here are some additional considerations to incorporate into the design of your Amazon Kinesis Data Streams application.

Contents

Starting Up Data Producers and Data Consumers

By default, the KCL begins reading records from the tip of the stream, which is the most recently added record. In this configuration, if a data-producing application adds records to the stream before any receiving record processors are running, the records are not read by the record processors after they start up.

To change the behavior of the record processors so that it always reads data from the beginning of the stream, set the following value in the properties file for your Amazon Kinesis Data Streams application:

```
initialPositionInStream = TRIM_HORIZON
```

By default, Amazon Kinesis Data Streams stores all data for 24 hours. It also supports extended retention of up to 7 days and the long-term retention of up to 365 days. This time frame is called the *retention period*. Setting the starting position to the `TRIM_HORIZON` will start the record processor with the oldest data in the stream, as defined by the retention period. Even with the `TRIM_HORIZON` setting, if a record processor were to start after a greater time has passed than the retention period, then some of the records in the stream will no longer be available. For this reason, you should always have consumer applications reading from the stream and use the CloudWatch metric `GetRecords.IteratorAgeMilliseconds` to monitor that applications are keeping up with incoming data.

In some scenarios, it may be fine for record processors to miss the first few records in the stream. For example, you might run some initial records through the stream to test that the stream is working end-to-end as expected. After doing this initial verification, you would then start your workers and begin to put production data into the stream.

For more information about the `TRIM_HORIZON` setting, see Using Shard Iterators (p. 150).

Shutting Down an Amazon Kinesis Data Streams Application

When your Amazon Kinesis Data Streams application has completed its intended task, you should shut it down by terminating the EC2 instances on which it is running. You can terminate the instances using the AWS Management Console or the AWS CLI.

After shutting down your Amazon Kinesis Data Streams application, you should delete the Amazon DynamoDB table that the KCL used to track the application's state.

Read Throttling

The throughput of a stream is provisioned at the shard level. Each shard has a read throughput of up to 5 transactions per second for reads, up to a maximum total data read rate of 2 MB per second. If an application (or a group of applications operating on the same stream) attempts to get data from a shard at a faster rate, Kinesis Data Streams throttles the corresponding Get operations.

In an Amazon Kinesis Data Streams application, if a record processor is processing data faster than the limit — such as in the case of a failover — throttling occurs. Because KCL manages the interactions between the application and Kinesis Data Streams, throttling exceptions occur in the KCL code rather than in the application code. However, because the KCL logs these exceptions, you see them in the logs.

If you find that your application is throttled consistently, you should consider increasing the number of shards for the stream.

Monitoring Amazon Kinesis Data Streams

You can monitor your data streams in Amazon Kinesis Data Streams using the following features:

- CloudWatch metrics (p. 180)— Kinesis Data Streams sends Amazon CloudWatch custom metrics with detailed monitoring for each stream.
- Kinesis Agent (p. 190)— The Kinesis Agent publishes custom CloudWatch metrics to help assess if the agent is working as expected.
- API logging (p. 191)— Kinesis Data Streams uses AWS CloudTrail to log API calls and store the data in an Amazon S3 bucket.
- Kinesis Client Library (p. 195)— Kinesis Client Library (KCL) provides metrics per shard, worker, and KCL application.
- Kinesis Producer Library (p. 204)— Kinesis Producer Library (KPL) provides metrics per shard, worker, and KPL application.

For more information about common monitoring issues, questions, and troubleshooting, see the following:

- Which metrics should I use to monitor and troubleshoot Kinesis Data Streams issues?
- Why does the IteratorAgeMilliseconds value in Kinesis Data Streams keep increasing?

Monitoring the Amazon Kinesis Data Streams Service with Amazon CloudWatch

Amazon Kinesis Data Streams and Amazon CloudWatch are integrated so that you can collect, view, and analyze CloudWatch metrics for your Kinesis data streams. For example, to track shard usage, you can monitor the `IncomingBytes` and `OutgoingBytes` metrics and compare them to the number of shards in the stream.

The metrics that you configure for your streams are automatically collected and pushed to CloudWatch every minute. Metrics are archived for two weeks; after that period, the data is discarded.

The following table describes basic stream-level and enhanced shard-level monitoring for Kinesis data streams.

Type	Description
Basic (stream-level)	Stream-level data is sent automatically every minute at no charge.
Enhanced (shard-level)	Shard-level data is sent every minute for an additional cost. To get this level of data, you must specifically enable it for the stream using the EnableEnhancedMonitoring operation. For information about pricing, see the Amazon CloudWatch product page.

Amazon Kinesis Data Streams Dimensions and Metrics

Kinesis Data Streams sends metrics to CloudWatch at two levels: the stream level and, optionally, the shard level. Stream-level metrics are for most common monitoring use cases in normal conditions. Shard-level metrics are for specific monitoring tasks, usually related to troubleshooting, and are enabled using the EnableEnhancedMonitoring operation.

For an explanation of the statistics gathered from CloudWatch metrics, see CloudWatch Statistics in the *Amazon CloudWatch User Guide*.

Topics

Basic Stream-level Metrics

The `AWS/Kinesis` namespace includes the following stream-level metrics.

Kinesis Data Streams sends these stream-level metrics to CloudWatch every minute. These metrics are always available.

Metric	Description
`GetRecords.Bytes`	The number of bytes retrieved from the Kinesis stream, measured over the specified time period. Minimum, Maximum, and Average statistics represent the bytes in a single `GetRecords` operation for the stream in the specified time period. Shard-level metric name: `OutgoingBytes` Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Bytes
`GetRecords.IteratorAge`	This metric is deprecated. Use `GetRecords.IteratorAgeMilliseconds`.
`GetRecords.IteratorAgeMilliseconds`	The age of the last record in all `GetRecords` calls made against a Kinesis stream, measured over the specified time period. Age is the difference between the current time and when the last record of the `GetRecords` call was written to the stream. The Minimum and Maximum statistics can be used to track the progress of Kinesis consumer applications. A value of zero indicates that the records being read are completely caught up with the stream. Shard-level metric name: `IteratorAgeMilliseconds` Dimensions: StreamName

Metric	Description
	Statistics: Minimum, Maximum, Average, Samples Units: Milliseconds
`GetRecords.Latency`	The time taken per `GetRecords` operation, measured over the specified time period. Dimensions: StreamName Statistics: Minimum, Maximum, Average Units: Milliseconds
`GetRecords.Records`	The number of records retrieved from the shard, measured over the specified time period. Minimum, Maximum, and Average statistics represent the records in a single `GetRecords` operation for the stream in the specified time period. Shard-level metric name: `OutgoingRecords` Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count
`GetRecords.Success`	The number of successful `GetRecords` operations per stream, measured over the specified time period. Dimensions: StreamName Statistics: Average, Sum, Samples Units: Count
`IncomingBytes`	The number of bytes successfully put to the Kinesis stream over the specified time period. This metric includes bytes from `PutRecord` and `PutRecords` operations. Minimum, Maximum, and Average statistics represent the bytes in a single put operation for the stream in the specified time period. Shard-level metric name: `IncomingBytes` Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Bytes

Metric	Description
IncomingRecords	The number of records successfully put to the Kinesis stream over the specified time period. This metric includes record counts from PutRecord and PutRecords operations. Minimum, Maximum, and Average statistics represent the records in a single put operation for the stream in the specified time period. Shard-level metric name: IncomingRecords Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count
PutRecord.Bytes	The number of bytes put to the Kinesis stream using the PutRecord operation over the specified time period. Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Bytes
PutRecord.Latency	The time taken per PutRecord operation, measured over the specified time period. Dimensions: StreamName Statistics: Minimum, Maximum, Average Units: Milliseconds
PutRecord.Success	The number of successful PutRecord operations per Kinesis stream, measured over the specified time period. Average reflects the percentage of successful writes to a stream. Dimensions: StreamName Statistics: Average, Sum, Samples Units: Count
PutRecords.Bytes	The number of bytes put to the Kinesis stream using the PutRecords operation over the specified time period. Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Bytes

Metric	Description
PutRecords.Latency	The time taken per PutRecords operation, measured over the specified time period. Dimensions: StreamName Statistics: Minimum, Maximum, Average Units: Milliseconds
PutRecords.Records	This metric is deprecated. Use PutRecords.SuccessfulRecords. Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count
PutRecords.Success	The number of PutRecords operations where at least one record succeeded, per Kinesis stream, measured over the specified time period. Dimensions: StreamName Statistics: Average, Sum, Samples Units: Count
PutRecords.TotalRecords	The total number of records sent in a PutRecords operation per Kinesis data stream, measured over the specified time period. Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count
PutRecords.SuccessfulRecords	The number of successful records in a PutRecords operation per Kinesis data stream, measured over the specified time period. Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count
PutRecords.FailedRecords	The number of records rejected due to internal failures in a PutRecords operation per Kinesis data stream, measured over the specified time period. Occasional internal failures are to be expected and should be retried. Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count

Metric	Description
PutRecords.ThrottledRecords	The number of records rejected due to throttling in a PutRecords operation per Kinesis data stream, measured over the specified time period. Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count
ReadProvisionedThroughputExceeded	The number of GetRecords calls throttled for the stream over the specified time period. The most commonly used statistic for this metric is Average. When the Minimum statistic has a value of 1, all records were throttled for the stream during the specified time period. When the Maximum statistic has a value of 0 (zero), no records were throttled for the stream during the specified time period. Shard-level metric name: ReadProvisionedThroughputExceeded Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count
SubscribeToShard.RateExceeded	This metric is emitted when a new subscription attempt fails because there already is an active subscription by the same consumer or if you exceed the number of calls per second allowed for this operation. Dimensions: StreamName, ConsumerName
SubscribeToShard.Success	This metric records whether the SubscribeToShard subscription was successfully established. The subscription only lives for at most 5 minutes. Therefore, this metric gets emitted at least once every 5 minutes. Dimensions: StreamName, ConsumerName
SubscribeToShardEvent.Bytes	The number of bytes received from the shard, measured over the specified time period. Minimum, Maximum, and Average statistics represent the bytes published in a single event for the specified time period. Shard-level metric name: OutgoingBytes Dimensions: StreamName, ConsumerName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Bytes

Metric	Description
SubscribeToShardEvent.MillisBehindLatest	The difference between the current time and when the last record of the SubscribeToShard event was written to the stream. Dimensions: StreamName, ConsumerName Statistics: Minimum, Maximum, Average, Samples Units: Milliseconds
SubscribeToShardEvent.Records	The number of records received from the shard, measured over the specified time period. Minimum, Maximum, and Average statistics represent the records in a single event for the specified time period. Shard-level metric name: OutgoingRecords Dimensions: StreamName, ConsumerName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count
SubscribeToShardEvent.Success	This metric is emitted every time an event is published successfully. It is only emitted when there's an active subscription. Dimensions: StreamName, ConsumerName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count
WriteProvisionedThroughputExceeded	The number of records rejected due to throttling for the stream over the specified time period. This metric includes throttling from PutRecord and PutRecords operations. The most commonly used statistic for this metric is Average. When the Minimum statistic has a non-zero value, records were being throttled for the stream during the specified time period. When the Maximum statistic has a value of 0 (zero), no records were being throttled for the stream during the specified time period. Shard-level metric name: WriteProvisionedThroughputExceeded Dimensions: StreamName Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count

Enhanced Shard-level Metrics

The `AWS/Kinesis` namespace includes the following shard-level metrics.

Kinesis sends the following shard-level metrics to CloudWatch every minute. Each metric dimension creates 1 CloudWatch metric and makes approximately 43,200 `PutMetricData` API calls per month. These metrics are not enabled by default. There is a charge for enhanced metrics emitted from Kinesis. For more information, see Amazon CloudWatch Pricing under the heading *Amazon CloudWatch Custom Metrics*. The charges are given per shard per metric per month.

Metric	Description
`IncomingBytes`	The number of bytes successfully put to the shard over the specified time period. This metric includes bytes from `PutRecord` and `PutRecords` operations. Minimum, Maximum, and Average statistics represent the bytes in a single put operation for the shard in the specified time period. Stream-level metric name: `IncomingBytes` Dimensions: StreamName, ShardId Statistics: Minimum, Maximum, Average, Sum, Samples Units: Bytes
`IncomingRecords`	The number of records successfully put to the shard over the specified time period. This metric includes record counts from `PutRecord` and `PutRecords` operations. Minimum, Maximum, and Average statistics represent the records in a single put operation for the shard in the specified time period. Stream-level metric name: `IncomingRecords` Dimensions: StreamName, ShardId Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count
`IteratorAgeMilliseconds`	The age of the last record in all `GetRecords` calls made against a shard, measured over the specified time period. Age is the difference between the current time and when the last record of the `GetRecords` call was written to the stream. The Minimum and Maximum statistics can be used to track the progress of Kinesis consumer applications. A value of 0 (zero) indicates that the records being read are completely caught up with the stream. Stream-level metric name: `GetRecords.IteratorAgeMilliseconds` Dimensions: StreamName, ShardId Statistics: Minimum, Maximum, Average, Samples Units: Milliseconds

Metric	Description
`OutgoingBytes`	The number of bytes retrieved from the shard, measured over the specified time period. Minimum, Maximum, and Average statistics represent the bytes returned in a single `GetRecords` operation or published in a single `SubscribeToShard` event for the shard in the specified time period. Stream-level metric name: `GetRecords.Bytes` Dimensions: StreamName, ShardId Statistics: Minimum, Maximum, Average, Sum, Samples Units: Bytes
`OutgoingRecords`	The number of records retrieved from the shard, measured over the specified time period. Minimum, Maximum, and Average statistics represent the records returned in a single `GetRecords` operation or published in a single `SubscribeToShard` event for the shard in the specified time period. Stream-level metric name: `GetRecords.Records` Dimensions: StreamName, ShardId Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count
`ReadProvisionedThroughputExceeded`	The number of `GetRecords` calls throttled for the shard over the specified time period. This exception count covers all dimensions of the following limits: 5 reads per shard per second or 2 MB per second per shard. The most commonly used statistic for this metric is Average. When the Minimum statistic has a value of 1, all records were throttled for the shard during the specified time period. When the Maximum statistic has a value of 0 (zero), no records were throttled for the shard during the specified time period. Stream-level metric name: `ReadProvisionedThroughputExceeded` Dimensions: StreamName, ShardId Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count

Metric	Description
`WriteProvisionedThroughputExceeded`	The number of records rejected due to throttling for the shard over the specified time period. This metric includes throttling from `PutRecord` and `PutRecords` operations and covers all dimensions of the following limits: 1,000 records per second per shard or 1 MB per second per shard. The most commonly used statistic for this metric is Average. When the Minimum statistic has a non-zero value, records were being throttled for the shard during the specified time period. When the Maximum statistic has a value of 0 (zero), no records were being throttled for the shard during the specified time period. Stream-level metric name: `WriteProvisionedThroughputExceeded` Dimensions: StreamName, ShardId Statistics: Minimum, Maximum, Average, Sum, Samples Units: Count

Dimensions for Amazon Kinesis Data Streams Metrics

Dimension	Description
`StreamName`	The name of the Kinesis stream. All available statistics are filtered by `StreamName`.

Recommended Amazon Kinesis Data Streams Metrics

Several Amazon Kinesis Data Streams metrics might be of particular interest to Kinesis Data Streams customers. The following list provides recommended metrics and their uses.

Metric	Usage Notes
`GetRecords.IteratorAgeMilliseconds`	Tracks the read position across all shards and consumers in the stream. If an iterator's age passes 50% of the retention period (by default, 24 hours, configurable up to 7 days), there is risk for data loss due to record expiration. We recommend that you use CloudWatch alarms on the Maximum statistic to alert you before this loss is a risk. For an example scenario that uses this metric, see Consumer Record Processing Falling Behind (p. 173).
`ReadProvisionedThroughputExceeded`	When your consumer-side record processing is falling behind, it is sometimes difficult to know where the bottleneck is. Use this metric to determine if your reads are being throttled due to exceeding your read throughput limits. The most commonly used statistic for this metric is Average.
`WriteProvisionedThroughputExceeded`	This is for the same purpose as the `ReadProvisionedThroughputExceeded` metric, but for the producer

Metric	Usage Notes
	(put) side of the stream. The most commonly used statistic for this metric is Average.
`PutRecord.Success`, `PutRecords.Success`	We recommend using CloudWatch alarms on the Average statistic to indicate when records are failing to the stream. Choose one or both put types depending on what your producer uses. If using the Kinesis Producer Library (KPL), use `PutRecords.Success`.
`GetRecords.Success`	We recommend using CloudWatch alarms on the Average statistic to indicate when records are failing from the stream.

Accessing Amazon CloudWatch Metrics for Kinesis Data Streams

You can monitor metrics for Kinesis Data Streams using the CloudWatch console, the command line, or the CloudWatch API. The following procedures show you how to access metrics using these different methods.

To access metrics using the CloudWatch console

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. On the navigation bar, choose a Region.
3. In the navigation pane, choose **Metrics**.
4. In the **CloudWatch Metrics by Category** pane, choose **Kinesis Metrics**.
5. Click the relevant row to view the statistics for the specified **MetricName** and **StreamName**.

   Note: Most console statistic names match the corresponding CloudWatch metric names listed above, except for **Read Throughput** and **Write Throughput**. These statistics are calculated over 5-minute intervals: **Write Throughput** monitors the `IncomingBytes` CloudWatch metric, and **Read Throughput** monitors `GetRecords.Bytes`.
6. (Optional) In the graph pane, select a statistic and a time period, and then create a CloudWatch alarm using these settings.

To access metrics using the AWS CLI

Use the list-metrics and get-metric-statistics commands.

To access metrics using the CloudWatch CLI

Use the mon-list-metrics and mon-get-stats commands.

To access metrics using the CloudWatch API

Use the ListMetrics and GetMetricStatistics operations.

Monitoring Kinesis Data Streams Agent Health with Amazon CloudWatch

The agent publishes custom CloudWatch metrics with a namespace of **AWSKinesisAgent**. These metrics help you assess whether the agent is submitting data into Kinesis Data Streams as specified, and whether

it is healthy and consuming the appropriate amount of CPU and memory resources on the data producer. Metrics such as number of records and bytes sent are useful to understand the rate at which the agent is submitting data to the stream. When these metrics fall below expected thresholds by some percentage or drop to zero, it could indicate configuration issues, network errors, or agent health issues. Metrics such as on-host CPU and memory consumption and agent error counters indicate data producer resource usage, and provide insights into potential configuration or host errors. Finally, the agent also logs service exceptions to help investigate agent issues. These metrics are reported in the Region specified in the agent configuration setting `cloudwatch.endpoint`. Cloudwatch metrics published from multiple Kinesis agents are aggregated or combined. For more information about agent configuration, see Agent Configuration Settings (p. 102).

Monitoring with CloudWatch

The Kinesis Data Streams agent sends the following metrics to CloudWatch.

Metric	Description
`BytesSent`	The number of bytes sent to Kinesis Data Streams over the specified time period. Units: Bytes
`RecordSendAttempts`	The number of records attempted (either first time, or as a retry) in a call to `PutRecords` over the specified time period. Units: Count
`RecordSendErrors`	The number of records that returned failure status in a call to `PutRecords`, including retries, over the specified time period. Units: Count
`ServiceErrors`	The number of calls to `PutRecords` that resulted in a service error (other than a throttling error) over the specified time period. Units: Count

Logging Amazon Kinesis Data Streams API Calls with AWS CloudTrail

Amazon Kinesis Data Streams is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Kinesis Data Streams. CloudTrail captures all API calls for Kinesis Data Streams as events. The calls captured include calls from the Kinesis Data Streams console and code calls to the Kinesis Data Streams API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for Kinesis Data Streams. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to Kinesis Data Streams, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, including how to configure and enable it, see the AWS CloudTrail User Guide.

Kinesis Data Streams Information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When supported event activity occurs in Kinesis Data Streams, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see Viewing Events with CloudTrail Event History.

For an ongoing record of events in your AWS account, including events for Kinesis Data Streams, create a trail. A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following:

- Overview for Creating a Trail
- CloudTrail Supported Services and Integrations
- Configuring Amazon SNS Notifications for CloudTrail
- Receiving CloudTrail Log Files from Multiple Regions and Receiving CloudTrail Log Files from Multiple Accounts

Kinesis Data Streams supports logging the following actions as events in CloudTrail log files:

- AddTagsToStream
- CreateStream
- DecreaseStreamRetentionPeriod
- DeleteStream
- DeregisterStreamConsumer
- DescribeStream
- DescribeStreamConsumer
- DisableEnhancedMonitoring
- EnableEnhancedMonitoring
- IncreaseStreamRetentionPeriod
- ListStreamConsumers
- ListStreams
- ListTagsForStream
- MergeShards
- RegisterStreamConsumer
- RemoveTagsFromStream
- SplitShard
- StartStreamEncryption
- StopStreamEncryption
- UpdateShardCount

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or AWS Identity and Access Management (IAM) user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the CloudTrail userIdentity Element.

Example: Kinesis Data Streams Log File Entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so they don't appear in any specific order.

The following example shows a CloudTrail log entry that demonstrates the `CreateStream`, `DescribeStream`, `ListStreams`, `DeleteStream`, `SplitShard`, and `MergeShards` actions.


```
{
    "Records": [
        {
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
                "accessKeyId": "EXAMPLE_KEY_ID",
                "userName": "Alice"
            },
            "eventTime": "2014-04-19T00:16:31Z",
            "eventSource": "kinesis.amazonaws.com",
            "eventName": "CreateStream",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
            "requestParameters": {
                "shardCount": 1,
                "streamName": "GoodStream"
            },
            "responseElements": null,
            "requestID": "db6c59f8-c757-11e3-bc3b-57923b443c1c",
            "eventID": "b7acfcd0-6ca9-4ee1-a3d7-c4e8d420d99b"
        },
        {
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
                "accessKeyId": "EXAMPLE_KEY_ID",
                "userName": "Alice"
            },
            "eventTime": "2014-04-19T00:17:06Z",
            "eventSource": "kinesis.amazonaws.com",
            "eventName": "DescribeStream",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
            "requestParameters": {
                "streamName": "GoodStream"
            },
            "responseElements": null,
            "requestID": "f0944d86-c757-11e3-b4ae-25654b1d3136",
            "eventID": "0b2f1396-88af-4561-b16f-398f8eaea596"
        },
        {
```

```
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
                "accessKeyId": "EXAMPLE_KEY_ID",
                "userName": "Alice"
            },
            "eventTime": "2014-04-19T00:15:02Z",
            "eventSource": "kinesis.amazonaws.com",
            "eventName": "ListStreams",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
            "requestParameters": {
                "limit": 10
            },
            "responseElements": null,
            "requestID": "a68541ca-c757-11e3-901b-cbcfe5b3677a",
            "eventID": "22a5fb8f-4e61-4bee-a8ad-3b72046b4c4d"
        },
        {
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
                "accessKeyId": "EXAMPLE_KEY_ID",
                "userName": "Alice"
            },
            "eventTime": "2014-04-19T00:17:07Z",
            "eventSource": "kinesis.amazonaws.com",
            "eventName": "DeleteStream",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
            "requestParameters": {
                "streamName": "GoodStream"
            },
            "responseElements": null,
            "requestID": "f10cd97c-c757-11e3-901b-cbcfe5b3677a",
            "eventID": "607e7217-311a-4a08-a904-ec02944596dd"
        },
        {
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
                "accessKeyId": "EXAMPLE_KEY_ID",
                "userName": "Alice"
            },
            "eventTime": "2014-04-19T00:15:03Z",
            "eventSource": "kinesis.amazonaws.com",
            "eventName": "SplitShard",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
            "requestParameters": {
                "shardToSplit": "shardId-000000000000",
                "streamName": "GoodStream",
                "newStartingHashKey": "11111111"
            },
```

```
            "responseElements": null,
            "requestID": "a6e6e9cd-c757-11e3-901b-cbcfe5b3677a",
            "eventID": "dcd2126f-c8d2-4186-b32a-192dd48d7e33"
        },
        {
            "eventVersion": "1.01",
            "userIdentity": {
                "type": "IAMUser",
                "principalId": "EX_PRINCIPAL_ID",
                "arn": "arn:aws:iam::012345678910:user/Alice",
                "accountId": "012345678910",
                "accessKeyId": "EXAMPLE_KEY_ID",
                "userName": "Alice"
            },
            "eventTime": "2014-04-19T00:16:56Z",
            "eventSource": "kinesis.amazonaws.com",
            "eventName": "MergeShards",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "127.0.0.1",
            "userAgent": "aws-sdk-java/unknown-version Linux/x.xx",
            "requestParameters": {
                "streamName": "GoodStream",
                "adjacentShardToMerge": "shardId-000000000002",
                "shardToMerge": "shardId-000000000001"
            },
            "responseElements": null,
            "requestID": "e9f9c8eb-c757-11e3-bf1d-6948db3cd570",
            "eventID": "77cf0d06-ce90-42da-9576-71986fec411f"
        }
    ]
}
```


Monitoring the Kinesis Client Library with Amazon CloudWatch

The Kinesis Client Library (KCL) for Amazon Kinesis Data Streams publishes custom Amazon CloudWatch metrics on your behalf, using the name of your KCL application as the namespace. You can view these metrics by navigating to the CloudWatch console and choosing **Custom Metrics**. For more information about custom metrics, see Publish Custom Metrics in the *Amazon CloudWatch User Guide*.

There is a nominal charge for the metrics uploaded to CloudWatch by the KCL; specifically, *Amazon CloudWatch Custom Metrics* and *Amazon CloudWatch API Requests* charges apply. For more information, see Amazon CloudWatch Pricing.

Topics

Metrics and Namespace

The namespace that is used to upload metrics is the application name that you specify when you launch the KCL.

Metric Levels and Dimensions

There are two options to control which metrics are uploaded to CloudWatch:

metric levels

Every metric is assigned an individual level. When you set a metrics reporting level, metrics with an individual level below the reporting level are not sent to CloudWatch. The levels are: `NONE`, `SUMMARY`, and `DETAILED`. The default setting is `DETAILED`; that is, all metrics are sent to CloudWatch. A reporting level of `NONE` means that no metrics are sent at all. For information about which levels are assigned to what metrics, see List of Metrics (p. 196).

enabled dimensions

Every KCL metric has associated dimensions that also get sent to CloudWatch. In KCL 2.x, if KCL is configured to process a single data stream, all the metrics dimensions (`Operation`, `ShardId`, and `WorkerIdentifier`) are enabled by default. Also, in KCL 2.x, if KCL is configured to process a single data stream, `Operation` dimension cannot be disabled. In KCL 2.x, if KCL is configured to process multiple data streams, all the metrics dimentions (`Operation`, `ShardId`, `StreamId`, and `WorkerIdentifier`) are enabled by default. Also, in KCL 2.x, if KCL is configured to process multiple data streams, the `Operation` and the `StreamId` dimensions cannot be disabled. `StreamId` dimention is available only for the per-shard metrics.

In KCL 1.x, only the `Operation` and the `ShardId` dimensions are enabled by default, and the `WorkerIdentifier` dimension is disabled. In KCL 1.x, the `Operation` dimension cannot be disabled.

For more information about CloudWatch metric dimensions, see the Dimensions section in the Amazon CloudWatch Concepts topic, in the *Amazon CloudWatch User Guide*.

When the `WorkerIdentifier` dimension is enabled, if a different value is used for the worker ID property every time a particular KCL worker restarts, new sets of metrics with new `WorkerIdentifier` dimension values are sent to CloudWatch. If you need the `WorkerIdentifier` dimension value to be the same across specific KCL worker restarts, you must explicitly specify the same worker ID value during initialization for each worker. Note that the worker ID value for each active KCL worker must be unique across all KCL workers.

Metric Configuration

Metric levels and enabled dimensions can be configured using the KinesisClientLibConfiguration instance, which is passed to Worker when launching the KCL application. In the MultiLangDaemon case, the `metricsLevel` and `metricsEnabledDimensions` properties can be specified in the .properties file used to launch the MultiLangDaemon KCL application.

Metric levels can be assigned one of three values: NONE, SUMMARY, or DETAILED. Enabled dimensions values must be comma-separated strings with the list of dimensions that are allowed for the CloudWatch metrics. The dimensions used by the KCL application are `Operation`, `ShardId`, and `WorkerIdentifier`.

List of Metrics

The following tables list the KCL metrics, grouped by scope and operation.

Topics

Per-KCL-Application Metrics

These metrics are aggregated across all KCL workers within the scope of the application, as defined by the Amazon CloudWatch namespace.

Topics

InitializeTask

The `InitializeTask` operation is responsible for initializing the record processor for the KCL application. The logic for this operation includes getting a shard iterator from Kinesis Data Streams and initializing the record processor.

Metric	Description
KinesisDataFetcher.getIterator.Success	Number of successful `GetShardIterator` operations per KCL application. Metric level: Detailed Units: Count
KinesisDataFetcher.getIterator.Time	Time taken per `GetShardIterator` operation for the given KCL application. Metric level: Detailed Units: Milliseconds
RecordProcessor.initialize.Time	Time taken by the record processor's initialize method. Metric level: Summary Units: Milliseconds
Success	Number of successful record processor initializations. Metric level: Summary Units: Count
Time	Time taken by the KCL worker for the record processor initialization. Metric level: Summary Units: Milliseconds

ShutdownTask

The `ShutdownTask` operation initiates the shutdown sequence for shard processing. This can occur because a shard is split or merged, or when the shard lease is lost from the worker. In both cases, the record processor `shutdown()` function is invoked. New shards are also discovered in the case where a shard was split or merged, resulting in the creation of one or two new shards.

Metric	Description
CreateLease.Success	Number of times that new child shards are successfully added into the KCL application DynamoDB table following parent shard shutdown. Metric level: Detailed Units: Count
CreateLease.Time	Time taken for adding new child shard information in the KCL application DynamoDB table. Metric level: Detailed Units: Milliseconds
UpdateLease.Success	Number of successful final checkpoints during the record processor shutdown. Metric level: Detailed Units: Count
UpdateLease.Time	Time taken by the checkpoint operation during the record processor shutdown. Metric level: Detailed Units: Milliseconds
RecordProcessor.shutdownTime	Time taken by the record processor's shutdown method. Metric level: Summary Units: Milliseconds
Success	Number of successful shutdown tasks. Metric level: Summary Units: Count
Time	Time taken by the KCL worker for the shutdown task. Metric level: Summary Units: Milliseconds

ShardSyncTask

The `ShardSyncTask` operation discovers changes to shard information for the Kinesis data stream, so new shards can be processed by the KCL application.

Metric	Description
CreateLease.Success	Number of successful attempts to add new shard information into the KCL application DynamoDB table. Metric level: Detailed Units: Count
CreateLease.Time	Time taken for adding new shard information in the KCL application DynamoDB table. Metric level: Detailed Units: Milliseconds
Success	Number of successful shard sync operations. Metric level: Summary Units: Count
Time	Time taken for the shard sync operation. Metric level: Summary Units: Milliseconds

BlockOnParentTask

If the shard is split or merged with other shards, then new child shards are created. The `BlockOnParentTask` operation ensures that record processing for the new shards does not start until the parent shards are completely processed by the KCL.

Metric	Description
Success	Number of successful checks for parent shard completion. Metric level: Summary Units: Count
Time	Time taken for parent shards completion. Metric level: Summary Unit: Milliseconds

PeriodicShardSyncManager

The `PeriodicShardSyncManager` is responsible for examining the data streams that are being processed by the KCL consumer application, identifying data streams with partial leases and handing them off for synchronization.

The following metrics are available when KCL is configured to process a single data stream (then the value of NumStreamsToSync and NumStreamsWithPartialLeases is set to 1) and also when KCL is configured to process multiple data streams.

Metric	Description
NumStreamsToSync	The number of data streams (per AWS account) being processed by the consumer application that contain partial leases and that must be handed off for synchronization. Metric level: Summary Units: Count
NumStreamsWithPartialLeases	The number of data streams (per AWS account) that the consumer application is processing that contain partial leases. Metric level: Summary Units: Count
Success	The number of times `PeriodicShardSyncManager` was able to successfully identify partial leases in the data streams that the consumer application is processing. Metric level: Summary Units: Count
Time	The amount of the time (in miliseconds) that the `PeriodicShardSyncManager` takes to examine the data streams that the consumer application is processing, in order to determine which data streams require shard synchronization. Metric level: Summary Units: Milliseconds

MultistreamTracker

The `MultistreamTracker` interface enables you to build KCL consumer applications that can process multiple data streams at the same time.

Metric	Description
DeletedStreams.Count	The number of data streams deleted at this time period. Metric level: Summary Units: Count
ActiveStreams.Count	The number of active data streams being processed. Metric level: Summary Units: Count
StreamsPendingDeletion.Count	The number of data streams that are pending deletion based on `FormerStreamsLeasesDeletionStrategy`. Metric level: Summary

Metric	Description
	Units: Count

Per-Worker Metrics

These metrics are aggregated across all record processors consuming data from a Kinesis data stream, such as an Amazon EC2 instance.

Topics

RenewAllLeases

The `RenewAllLeases` operation periodically renews shard leases owned by a particular worker instance.

Metric	Description
RenewLease.Success	Number of successful lease renewals by the worker. Metric level: Detailed Units: Count
RenewLease.Time	Time taken by the lease renewal operation. Metric level: Detailed Units: Milliseconds
CurrentLeases	Number of shard leases owned by the worker after all leases are renewed. Metric level: Summary Units: Count
LostLeases	Number of shard leases that were lost following an attempt to renew all leases owned by the worker. Metric level: Summary Units: Count
Success	Number of times lease renewal operation was successful for the worker. Metric level: Summary Units: Count
Time	Time taken for renewing all leases for the worker. Metric level: Summary Units: Milliseconds

TakeLeases

The `TakeLeases` operation balances record processing between all KCL workers. If the current KCL worker has fewer shard leases than required, it takes shard leases from another worker that is overloaded.

Metric	Description
ListLeases.Success	Number of times all shard leases were successfully retrieved from the KCL application DynamoDB table. Metric level: Detailed Units: Count
ListLeases.Time	Time taken to retrieve all shard leases from the KCL application DynamoDB table. Metric level: Detailed Units: Milliseconds
TakeLease.Success	Number of times the worker successfully took shard leases from other KCL workers. Metric level: Detailed Units: Count
TakeLease.Time	Time taken to update the lease table with leases taken by the worker. Metric level: Detailed Units: Milliseconds
NumWorkers	Total number of workers, as identified by a specific worker. Metric level: Summary Units: Count
NeededLeases	Number of shard leases that the current worker needs for a balanced shard-processing load. Metric level: Detailed Units: Count
LeasesToTake	Number of leases that the worker will attempt to take. Metric level: Detailed Units: Count
TakenLeases	Number of leases taken successfully by the worker. Metric level: Summary Units: Count
TotalLeases	Total number of shards that the KCL application is processing.

Metric	Description
	Metric level: Detailed Units: Count
ExpiredLeases	Total number of shards that are not being processed by any worker, as identified by the specific worker. Metric level: Summary Units: Count
Success	Number of times the `TakeLeases` operation successfully completed. Metric level: Summary Units: Count
Time	Time taken by the `TakeLeases` operation for a worker. Metric level: Summary Units: Milliseconds

Per-Shard Metrics

These metrics are aggregated across a single record processor.

ProcessTask

The `ProcessTask` operation calls GetRecords with the current iterator position to retrieve records from the stream and invokes the record processor `processRecords` function.

Metric	Description
KinesisDataFetcher.getRecords.Success	Number of successful `GetRecords` operations per Kinesis data stream shard. Metric level: Detailed Units: Count
KinesisDataFetcher.getRecords.Time	Time taken per `GetRecords` operation for the Kinesis data stream shard. Metric level: Detailed Units: Milliseconds
UpdateLease.Success	Number of successful checkpoints made by the record processor for the given shard. Metric level: Detailed Units: Count
UpdateLease.Time	Time taken for each checkpoint operation for the given shard. Metric level: Detailed

Metric	Description
	Units: Milliseconds
DataBytesProcessed	Total size of records processed in bytes on each `ProcessTask` invocation. Metric level: Summary Units: Byte
RecordsProcessed	Number of records processed on each `ProcessTask` invocation. Metric level: Summary Units: Count
ExpiredIterator	Number of ExpiredIteratorException received when calling `GetRecords`. Metric level: Summary Units: Count
MillisBehindLatest	Time that the current iterator is behind from the latest record (tip) in the shard. This value is less than or equal to the difference in time between the latest record in a response and the current time. This is a more accurate reflection of how far a shard is from the tip than comparing time stamps in the last response record. This value applies to the latest batch of records, not an average of all time stamps in each record. Metric level: Summary Units: Milliseconds
RecordProcessor.processRecords.Time	Time taken by the record processor's `processRecords` method. Metric level: Summary Units: Milliseconds
Success	Number of successful process task operations. Metric level: Summary Units: Count
Time	Time taken for the process task operation. Metric level: Summary Units: Milliseconds

Monitoring the Kinesis Producer Library with Amazon CloudWatch

The Kinesis Producer Library (KPL) for Amazon Kinesis Data Streams publishes custom Amazon CloudWatch metrics on your behalf. You can view these metrics by navigating to the CloudWatch console and choosing **Custom Metrics**. For more information about custom metrics, see Publish Custom Metrics in the *Amazon CloudWatch User Guide*.

There is a nominal charge for the metrics uploaded to CloudWatch by the KPL; specifically, Amazon CloudWatch Custom Metrics and Amazon CloudWatch API Requests charges apply. For more information, see Amazon CloudWatch Pricing. Local metrics gathering does not incur CloudWatch charges.

Topics

Metrics, Dimensions, and Namespaces

You can specify an application name when launching the KPL, which is then used as part of the namespace when uploading metrics. This is optional; the KPL provides a default value if an application name is not set.

You can also configure the KPL to add arbitrary additional dimensions to the metrics. This is useful if you want finer-grained data in your CloudWatch metrics. For example, you can add the hostname as a dimension, which then allows you to identify uneven load distributions across your fleet. All KPL configuration settings are immutable, so you can't change these additional dimensions after the KPL instance is initialized.

Metric Level and Granularity

There are two options to control the number of metrics uploaded to CloudWatch:

metric level

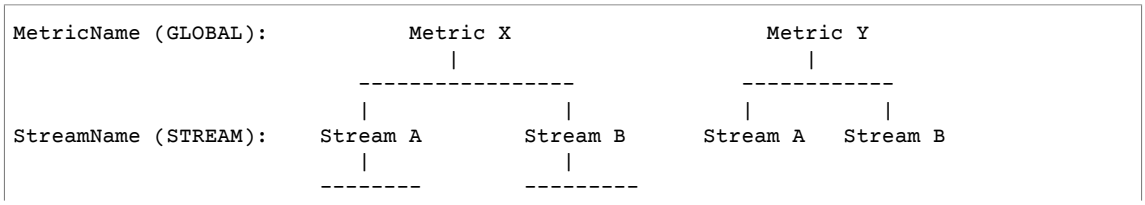
This is a rough gauge of how important a metric is. Every metric is assigned a level. When you set a level, metrics with levels below that are not sent to CloudWatch. The levels are `NONE`, `SUMMARY`, and `DETAILED`. The default setting is `DETAILED`; that is, all metrics. `NONE` means no metrics at all, so no metrics are actually assigned to that level.

granularity

This controls whether the same metric is emitted at additional levels of granularity. The levels are `GLOBAL`, `STREAM`, and `SHARD`. The default setting is `SHARD`, which contains the most granular metrics.

When `SHARD` is chosen, metrics are emitted with the stream name and shard ID as dimensions. In addition, the same metric is also emitted with only the stream name dimension, and the metric without the stream name. This means that, for a particular metric, two streams with two shards each will produce seven CloudWatch metrics: one for each shard, one for each stream, and one overall; all describing the same statistics but at different levels of granularity. For an illustration, see the following diagram.

The different granularity levels form a hierarchy, and all the metrics in the system form trees, rooted at the metric names:



```
                          |       |         |       |
ShardID (SHARD):      Shard 0 Shard 1   Shard 0 Shard 1
```

Not all metrics are available at the shard level; some are stream level or global by nature. These are not produced at the shard level, even if you have enabled shard-level metrics (`Metric Y` in the preceding diagram).

When you specify an additional dimension, you need to provide values for `tuple:<DimensionName, DimensionValue, Granularity>`. The granularity is used to determine where the custom dimension is inserted in the hierarchy: `GLOBAL` means that the additional dimension is inserted after the metric name, `STREAM` means it's inserted after the stream name, and `SHARD` means it's inserted after the shard ID. If multiple additional dimensions are given per granularity level, they are inserted in the order given.

Local Access and Amazon CloudWatch Upload

Metrics for the current KPL instance are available locally in real time; you can query the KPL at any time to get them. The KPL locally computes the sum, average, minimum, maximum, and count of every metric, as in CloudWatch.

You can get statistics that are cumulative from the start of the program to the present point in time, or using a rolling window over the past *N* seconds, where *N* is an integer between 1 and 60.

All metrics are available for upload to CloudWatch. This is especially useful for aggregating data across multiple hosts, monitoring, and alarming. This functionality is not available locally.

As described previously, you can select which metrics to upload with the *metric level* and *granularity* settings. Metrics that are not uploaded are available locally.

Uploading data points individually is untenable because it could produce millions of uploads per second, if traffic is high. For this reason, the KPL aggregates metrics locally into 1-minute buckets and uploads a statistics object to CloudWatch one time per minute, per enabled metric.

List of Metrics

Metric	Description
`UserRecordsReceived`	Count of how many logical user records were received by the KPL core for put operations. Not available at shard level. Metric level: Detailed Unit: Count
`UserRecordsPending`	Periodic sample of how many user records are currently pending. A record is pending if it is either currently buffered and waiting for to be sent, or sent and in-flight to the backend service. Not available at shard level. The KPL provides a dedicated method to retrieve this metric at the global level for customers to manage their put rate. Metric level: Detailed Unit: Count
`UserRecordsPut`	Count of how many logical user records were put successfully.

Metric	Description
	The KPL does not count failed records for this metric. This allows the average to give the success rate, the count to give the total attempts, and the difference between the count and sum to give the failure count. Metric level: Summary Unit: Count
`UserRecordsDataPut`	Bytes in the logical user records successfully put. Metric level: Detailed Unit: Bytes
`KinesisRecordsPut`	Count of how many Kinesis Data Streams records were put successfully (each Kinesis Data Streams record can contain multiple user records). The KPL outputs a zero for failed records. This allows the average to give the success rate, the count to give the total attempts, and the difference between the count and sum to give the failure count. Metric level: Summary Unit: Count
`KinesisRecordsDataPut`	Bytes in the Kinesis Data Streams records. Metric level: Detailed Unit: Bytes
`ErrorsByCode`	Count of each type of error code. This introduces an additional dimension of `ErrorCode`, in addition to the normal dimensions such as `StreamName` and `ShardId`. Not every error can be traced to a shard. The errors that cannot be traced are only emitted at stream or global levels. This metric captures information about such things as throttling, shard map changes, internal failures, service unavailable, timeouts, and so on. Kinesis Data Streams API errors are counted one time per Kinesis Data Streams record. Multiple user records within a Kinesis Data Streams record do not generate multiple counts. Metric level: Summary Unit: Count
`AllErrors`	This is triggered by the same errors as **Errors by Code**, but does not distinguish between types. This is useful as a general monitor of the error rate without requiring a manual sum of the counts from all the different types of errors. Metric level: Summary Unit: Count

Metric	Description
`RetriesPerRecord`	Number of retries performed per user record. Zero is emitted for records that succeed in one try. Data is emitted at the moment a user record finishes (when it either succeeds or can no longer be retried). If record time-to-live is a large value, this metric may be significantly delayed. Metric level: Detailed Unit: Count
`BufferingTime`	The time between a user record arriving at the KPL and leaving for the backend. This information is transmitted back to the user on a per-record basis, but is also available as an aggregated statistic. Metric level: Summary Unit: Milliseconds
`Request Time`	The time it takes to perform `PutRecordsRequests`. Metric level: Detailed Unit: Milliseconds
`User Records per Kinesis Record`	The number of logical user records aggregated into a single Kinesis Data Streams record. Metric level: Detailed Unit: Count
`Amazon Kinesis Records per PutRecordsRequest`	The number of Kinesis Data Streams records aggregated into a single `PutRecordsRequest`. Not available at shard level. Metric level: Detailed Unit: Count
`User Records per PutRecordsRequest`	The total number of user records contained within a `PutRecordsRequest`. This is roughly equivalent to the product of the previous two metrics. Not available at shard level. Metric level: Detailed Unit: Count

Security in Amazon Kinesis Data Streams

Cloud security at AWS is the highest priority. As an AWS customer, you will benefit from a data center and network architecture built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. The effectiveness of our security is regularly tested and verified by third-party auditors as part of the AWS compliance programs. To learn about the compliance programs that apply to Kinesis Data Streams, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your organization's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Kinesis Data Streams. The following topics show you how to configure Kinesis Data Streams to meet your security and compliance objectives. You'll also learn how to use other AWS services that can help you to monitor and secure your Kinesis Data Streams resources.

Topics

Data Protection in Amazon Kinesis Data Streams

Server-side encryption using AWS Key Management Service (AWS KMS) keys makes it easy for you to meet strict data management requirements by encrypting your data at rest within Amazon Kinesis Data Streams.

> **Note**
> If you require FIPS 140-2 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-2.

Topics

What Is Server-Side Encryption for Kinesis Data Streams?

Server-side encryption is a feature in Amazon Kinesis Data Streams that automatically encrypts data before it's at rest by using an AWS KMS customer master key (CMK) you specify. Data is encrypted before it's written to the Kinesis stream storage layer, and decrypted after it's retrieved from storage. As a result, your data is encrypted at rest within the Kinesis Data Streams service. This allows you to meet strict regulatory requirements and enhance the security of your data.

With server-side encryption, your Kinesis stream producers and consumers don't need to manage master keys or cryptographic operations. Your data is automatically encrypted as it enters and leaves the Kinesis Data Streams service, so your data at rest is encrypted. AWS KMS provides all the master keys that are used by the server-side encryption feature. AWS KMS makes it easy to use a CMK for Kinesis that is managed by AWS, a user-specified AWS KMS CMK, or a master key imported into the AWS KMS service.

> **Note**
> Server-side encryption encrypts incoming data only after encryption is enabled. Preexisting data in an unencrypted stream is not encrypted after server-side encryption is enabled.

Costs, Regions, and Performance Considerations

When you apply server-side encryption, you are subject to AWS KMS API usage and key costs. Unlike custom KMS master keys, the `(Default) aws/kinesis` customer master key (CMK) is offered free of charge. However, you still must pay for the API usage costs that Amazon Kinesis Data Streams incurs on your behalf.

API usage costs apply for every CMK, including custom ones. Kinesis Data Streams calls AWS KMS approximately every five minutes when it is rotating the data key. In a 30-day month, the total cost of AWS KMS API calls that are initiated by a Kinesis stream should be less than a few dollars. This cost scales with the number of user credentials that you use on your data producers and consumers because each user credential requires a unique API call to AWS KMS. When you use an IAM role for authentication, each assume role call results in unique user credentials. To save KMS costs, you might want to cache user credentials that are returned by the assume role call.

The following describes the costs by resource:

Keys

- The CMK for Kinesis that's managed by AWS (alias = `aws/kinesis`) is free.
- User-generated KMS keys are subject to KMS key costs. For more information, see AWS Key Management Service Pricing.

API usage costs apply for every CMK, including custom ones. Kinesis Data Streams calls KMS approximately every 5 minutes when it is rotating the data key. In a 30-day month, the total cost of KMS API calls initiated by a Kinesis data stream should be less than a few dollars. Please note that this cost scales with the number of user credentials you use on your data producers and consumers because each user credential requires a unique API call to AWS KMS. When you use IAM role for authentication,

each assume-role-call will result in unique user credentials and you might want to cache user credentials returned by the assume-role-call to save KMS costs.

KMS API Usage

For every encrypted stream, when reading from TIP and using a single IAM account/user access key across readers and writers, Kinesis service calls the AWS KMS service approximately 12 times every 5 minutes. Not reading from TIP could lead to higher calls to AWS KMS service. API requests to generate new data encryption keys are subject to AWS KMS usage costs. For more information, see AWS Key Management Service Pricing: Usage.

Availability of Server-Side Encryption by Region

Currently, server-side encryption of Kinesis streams is available in all the regions supported for Kinesis Data Streams, including AWS GovCloud (US-West), and the China regions. For more information about supported regions for Kinesis Data Streams see https://docs.aws.amazon.com/general/latest/gr/ak.html.

Performance Considerations

Due to the service overhead of applying encryption, applying server-side encryption increases the typical latency of `PutRecord`, `PutRecords`, and `GetRecords` by less than 100µs.

How Do I Get Started with Server-Side Encryption?

The easiest way to get started with server-side encryption is to use the AWS Management Console and the Amazon Kinesis KMS Service Key, `aws/kinesis`.

The following procedure demonstrates how to enable server-side encryption for a Kinesis stream.

To enable server-side encryption for a Kinesis stream

1. Sign in to the AWS Management Console and open the Amazon Kinesis Data Streams console.
2. Create or select a Kinesis stream in the AWS Management Console.
3. Choose the **details** tab.
4. In **Server-side encryption**, choose **edit**.
5. Unless you want to use a user-generated KMS master key, ensure the **(Default) aws/kinesis** KMS master key is selected. This is the KMS master key generated by the Kinesis service. Choose **Enabled**, and then choose **Save**.

   Note
   The default Kinesis service master key is free, however, the API calls made by Kinesis to the AWS KMS service are subject to KMS usage costs.
6. The stream transitions through a **pending** state. After the stream returns to an **active** state with encryption enabled, all incoming data written to the stream is encrypted using the KMS master key you selected.
7. To disable server-side encryption, choose **Disabled** in **Server-side encryption** in the AWS Management Console, and then choose **Save**.

Creating and Using User-Generated KMS Master Keys

This section describes how to create and use your own KMS master keys, instead of using the master key administered by Amazon Kinesis.

Creating User-Generated KMS Master Keys

For instructions on creating your own master keys, see Creating Keys in the *AWS Key Management Service Developer Guide*. After you create keys for your account, the Kinesis Data Streams service returns these keys in the **KMS master key** list.

Using User-Generated KMS Master Keys

After the correct permissions are applied to your consumers, producers, and administrators, you can use custom KMS master keys in your own AWS account or another AWS account. All KMS master keys in your account appear in the **KMS Master Key** list within the AWS Management Console.

To use custom KMS master keys located in another account, you need permissions to use those keys. You must also specify the ARN of the KMS master key in the ARN input box in the AWS Management Console.

Permissions to Use User-Generated KMS Master Keys

Before you can use server-side encryption with a user-generated KMS master key, you must configure AWS KMS key policies to allow encryption of streams and encryption and decryption of stream records. For examples and more information about AWS KMS permissions, see AWS KMS API Permissions: Actions and Resources Reference.

> **Note**
> The use of the default service key for encryption does not require application of custom IAM permissions.

Before you use user-generated KMS master keys, ensure that your Kinesis stream producers and consumers (IAM principals) are users in the KMS master key policy. Otherwise, writes and reads from a stream will fail, which could ultimately result in data loss, delayed processing, or hung applications. You can manage permissions for KMS keys using IAM policies. For more information, see Using IAM Policies with AWS KMS.

Example Producer Permissions

Your Kinesis stream producers must have the `kms:GenerateDataKey` permission.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
        "Effect": "Allow",
        "Action": [
            "kms:GenerateDataKey"
        ],
        "Resource": "arn:aws:kms:us-
west-2:123456789012:key/1234abcd-12ab-34cd-56ef-1234567890ab"
    },
    {
        "Effect": "Allow",
        "Action": [
            "kinesis:PutRecord",
            "kinesis:PutRecords"
        ],
        "Resource": "arn:aws:kinesis:*:123456789012:MyStream"
    }
  ]
}
```

Example Consumer Permissions

Your Kinesis stream consumers must have the `kms:Decrypt` permission.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
        "Effect": "Allow",
        "Action": [
            "kms:Decrypt"
        ],
        "Resource": "arn:aws:kms:us-
west-2:123456789012:key/1234abcd-12ab-34cd-56ef-1234567890ab"
    },
    {
        "Effect": "Allow",
        "Action": [
            "kinesis:GetRecords",
            "kinesis:DescribeStream"
        ],
        "Resource": "arn:aws:kinesis:*:123456789012:MyStream"
    }
  ]
}
```

Amazon Kinesis Data Analytics and AWS Lambda use roles to consume Kinesis streams. Make sure to add the `kms:Decrypt` permission to the roles that these consumers use.

Stream Administrator Permissions

Kinesis stream administrators must have authorization to call `kms:List*` and `kms:DescribeKey*`.

Verifying and Troubleshooting KMS Key Permissions

After enabling encryption on a Kinesis stream, we recommend that you monitor the success of your `putRecord`, `putRecords`, and `getRecords` calls using the following Amazon CloudWatch metrics:

- `PutRecord.Success`
- `PutRecords.Success`
- `GetRecords.Success`

For more information, see Monitoring Amazon Kinesis Data Streams (p. 180)

Using Amazon Kinesis Data Streams with Interface VPC Endpoints

You can use an interface VPC endpoint to keep traffic between your Amazon VPC and Kinesis Data Streams from leaving the Amazon network. Interface VPC endpoints don't require an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. Interface VPC endpoints are powered by AWS PrivateLink, an AWS technology that enables private communication between AWS services using an elastic network interface with private IPs in your Amazon VPC. For more information, see Amazon Virtual Private Cloud and Interface VPC Endpoints (AWS PrivateLink).

Topics

- Using Interface VPC Endpoints for Kinesis Data Streams (p. 214)

Using Interface VPC Endpoints for Kinesis Data Streams

To get started you do not need to change the settings for your streams, producers, or consumers. Simply create an interface VPC endpoint in order for your Kinesis Data Streams traffic from and to your Amazon VPC resources to start flowing through the interface VPC endpoint. For more information, see Creating an Interface Endpoint.

The Kinesis Producer Library (KPL) and Kinesis Consumer Library (KCL) call AWS services like Amazon CloudWatch and Amazon DynamoDB using either public endpoints or private interface VPC endpoints, whichever are in use. For example, if your KCL application is running in a VPC with DynamoDB interface with VPC endpoints enabled, calls between DynamoDB and your KCL application are flowing through the interface VPC endpoint.

Controlling Access to VPCE Endpoints for Kinesis Data Streams

VPC endpoint policies enable you to control access by either attaching a policy to a VPC endpoint or by using additional fields in a policy that is attached to an IAM user, group, or role to restrict access to only occur via the specified VPC endpoint. These policies can be used to restrict access to specific streams to a specified VPC endpoint when used in conjunction with the IAM policies to only grant access to Kinesis data stream actions via the specified VPC endpoint.

The following are example endpoint policies for accessing Kinesis data streams.

- **VPC policy example: read-only access** - this sample policy can be attached to a VPC endpoint. (For more information, see Controlling Access to Amazon VPC Resources). It restricts actions to only listing and describing a Kinesis data stream through the VPC endpoint to which it is attached.

```
{
  "Statement": [
    {
      "Sid": "ReadOnly",
      "Principal": "*",
      "Action": [
        "kinesis:List*",
        "kinesis:Describe*"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

- **VPC policy example: restrict access to a specific Kinesis data stream** - this sample policy can be attached to a VPC endpoint. It restricts access to a specific data stream through the VPC endpoint to which it is attached.

```
{
  "Statement": [
    {
      "Sid": "AccessToSpecificDataStream",
      "Principal": "*",
      "Action": "kinesis:*",
      "Effect": "Allow",
      "Resource": "arn:aws:kinesis:us-east-1:123456789012:stream/MyStream"
```

```
    }
  ]
}
```

- **IAM policy example: restrict access to a specific Stream from a specific VPC endpoint only** - this sample policy can be attached to an IAM user, role, or group. It restricts access to a specified Kinesis data stream to occur only from a specified VPC endpoint.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "AccessFromSpecificEndpoint",
         "Action": "kinesis:*",
         "Effect": "Deny",
         "Resource": "arn:aws:kinesis:us-east-1:123456789012:stream/MyStream",
         "Condition": { "StringNotEquals" : { "aws:sourceVpce": "vpce-11aa22bb" } }
      }
   ]
}
```

Availability of VPC Endpoint Policies for Kinesis Data Streams

Kinesis Data Streams interface VPC endpoints with policies are supported in the following regions:

- Europe (Paris)
- Europe (Ireland)
- US East (N. Virginia)
- Europe (Stockholm)
- US East (Ohio)
- Europe (Frankfurt)
- South America (São Paulo)
- Europe (London)
- Asia Pacific (Tokyo)
- US West (N. California)
- Asia Pacific (Singapore)
- Asia Pacific (Sydney)
- China (Beijing)
- China (Ningxia)
- Asia Pacific (Hong Kong)
- Middle East (Bahrain)
- Europe (Milan)
- Africa (Cape Town)
- Asia Pacific (Mumbai)
- Asia Pacific (Seoul)
- Canada (Central)
- US West (Oregon) except usw2-az4
- AWS GovCloud (US-East)
- AWS GovCloud (US-West)

- Asia Pacific (Osaka)

Controlling Access to Amazon Kinesis Data Streams Resources Using IAM

AWS Identity and Access Management (IAM) enables you to do the following:

- Create users and groups under your AWS account
- Assign unique security credentials to each user under your AWS account
- Control each user's permissions to perform tasks using AWS resources
- Allow the users in another AWS account to share your AWS resources
- Create roles for your AWS account and define the users or services that can assume them
- Use existing identities for your enterprise to grant permissions to perform tasks using AWS resources

By using IAM with Kinesis Data Streams, you can control whether users in your organization can perform a task using specific Kinesis Data Streams API actions and whether they can use specific AWS resources.

If you are developing an application using the Kinesis Client Library (KCL), your policy must include permissions for Amazon DynamoDB and Amazon CloudWatch; the KCL uses DynamoDB to track state information for the application, and CloudWatch to send KCL metrics to CloudWatch on your behalf. For more information about the KCL, see Developing KCL 1.x Consumers (p. 125).

For more information about IAM, see the following:

- AWS Identity and Access Management (IAM)
- Getting started
- IAM User Guide

For more information about IAM and Amazon DynamoDB, see Using IAM to Control Access to Amazon DynamoDB Resources in the *Amazon DynamoDB Developer Guide*.

For more information about IAM and Amazon CloudWatch, see Controlling User Access to Your AWS Account in the *Amazon CloudWatch User Guide*.

Contents

Policy Syntax

An IAM policy is a JSON document that consists of one or more statements. Each statement is structured as follows:

```
{
  "Statement":[{
    "Effect":"effect",
    "Action":"action",
```

```
    "Resource":"arn",
    "Condition":{
      "condition":{
        "key":"value"
        }
      }
    }
  ]
}
```

There are various elements that make up a statement:

- **Effect:** The *effect* can be `Allow` or `Deny`. By default, IAM users don't have permission to use resources and API actions, so all requests are denied. An explicit allow overrides the default. An explicit deny overrides any allows.
- **Action**: The *action* is the specific API action for which you are granting or denying permission.
- **Resource**: The resource that's affected by the action. To specify a resource in the statement, you need to use its Amazon Resource Name (ARN).
- **Condition**: Conditions are optional. They can be used to control when your policy will be in effect.

As you create and manage IAM policies, you might want to use the IAM Policy Generator and the IAM Policy Simulator.

Actions for Kinesis Data Streams

In an IAM policy statement, you can specify any API action from any service that supports IAM. For Kinesis Data Streams, use the following prefix with the name of the API action: `kinesis:`. For example: `kinesis:CreateStream`, `kinesis:ListStreams`, and `kinesis:DescribeStreamSummary`.

To specify multiple actions in a single statement, separate them with commas as follows:

```
"Action": ["kinesis:action1", "kinesis:action2"]
```

You can also specify multiple actions using wildcards. For example, you can specify all actions whose name begins with the word "Get" as follows:

```
"Action": "kinesis:Get*"
```

To specify all Kinesis Data Streams operations, use the * wildcard as follows:

```
"Action": "kinesis:*"
```

For the complete list of Kinesis Data Streams API actions, see the Amazon Kinesis API Reference.

Amazon Resource Names (ARNs) for Kinesis Data Streams

Each IAM policy statement applies to the resources that you specify using their ARNs.

Use the following ARN resource format for Kinesis data streams:

```
arn:aws:kinesis:region:account-id:stream/stream-name
```

For example:

```
"Resource": arn:aws:kinesis:*:111122223333:stream/my-stream
```

Example Policies for Kinesis Data Streams

The following example policies demonstrate how you could control user access to your Kinesis data streams.

Example 1: Allow users to get data from a stream

This policy allows a user or group to perform the `DescribeStreamSummary`, `GetShardIterator`, and `GetRecords` operations on the specified stream and `ListStreams` on any stream. This policy could be applied to users who should be able to get data from a specific stream.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kinesis:Get*",
                "kinesis:DescribeStreamSummary"
            ],
            "Resource": [
                "arn:aws:kinesis:us-east-1:111122223333:stream/stream1"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "kinesis:ListStreams"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

Example 2: Allow users to add data to any stream in the account

This policy allows a user or group to use the `PutRecord` operation with any of the account's streams. This policy could be applied to users that should be able to add data records to all streams in an account.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kinesis:PutRecord"
            ],
            "Resource": [
                "arn:aws:kinesis:us-east-1:111122223333:stream/*"
            ]
        }
    ]
}
```

Example 3: Allow any Kinesis Data Streams action on a specific stream

This policy allows a user or group to use any Kinesis Data Streams operation on the specified stream. This policy could be applied to users that should have administrative control over a specific stream.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "kinesis:*",
            "Resource": [
                "arn:aws:kinesis:us-east-1:111122223333:stream/stream1"
            ]
        }
    ]
}
```

Example 4: Allow any Kinesis Data Streams action on any stream

This policy allows a user or group to use any Kinesis Data Streams operation on any stream in an account. Because this policy grants full access to all your streams, you should restrict it to administrators only.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "kinesis:*",
            "Resource": [
                "arn:aws:kinesis:*:111122223333:stream/*"
            ]
        }
    ]
}
```

Compliance Validation for Amazon Kinesis Data Streams

Third-party auditors assess the security and compliance of Amazon Kinesis Data Streams as part of multiple AWS compliance programs. These include SOC, PCI, FedRAMP, HIPAA, and others.

For a list of AWS services in scope of specific compliance programs, see AWS Services in Scope by Compliance Program. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using Kinesis Data Streams is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. If your use of Kinesis Data Streams is subject to compliance with standards such as HIPAA, PCI, or FedRAMP, AWS provides resources to help:

- Security and Compliance Quick Start Guides – These deployment guides discuss architectural considerations and provide steps for deploying security- and compliance-focused baseline environments on AWS.

- Architecting for HIPAA Security and Compliance Whitepaper – This whitepaper describes how companies can use AWS to create HIPAA-compliant applications.
- AWS Compliance Resources – This collection of workbooks and guides that might apply to your industry and location
- AWS Config – This AWS service that assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – This AWS service provides a comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.

Resilience in Amazon Kinesis Data Streams

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between Availability Zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

In addition to the AWS global infrastructure, Kinesis Data Streams offers several features to help support your data resiliency and backup needs.

Disaster Recovery in Amazon Kinesis Data Streams

Failure can occur at the following levels when you use an Amazon Kinesis Data Streams application to process data from a stream:

- A record processor could fail
- A worker could fail, or the instance of the application that instantiated the worker could fail
- An EC2 instance that is hosting one or more instances of the application could fail

Record Processor Failure

The worker invokes record processor methods using Java ExecutorService tasks. If a task fails, the worker retains control of the shard that the record processor was processing. The worker starts a new record processor task to process that shard. For more information, see Read Throttling (p. 179).

Worker or Application Failure

If a worker—or an instance of the Amazon Kinesis Data Streams application—fails, you should detect and handle the situation. For example, if the `Worker.run` method throws an exception, you should catch and handle it.

If the application itself fails, you should detect this and restart it. When the application starts up, it instantiates a new worker, which in turn instantiates new record processors that are automatically assigned shards to process. These could be the same shards that these record processors were processing before the failure, or shards that are new to these processors.

In a situation where the worker or application fails, the failure isn't detected, and there are other instances of the application running on other EC2 instances, workers on these other instances handle the failure. They create additional record processors to process the shards that are no longer being processed by the failed worker. The load on these other EC2 instances increases accordingly.

The scenario described here assumes that although the worker or application has failed, the hosting EC2 instance is still running and is therefore not restarted by an Auto Scaling group.

Amazon EC2 Instance Failure

We recommend that you run the EC2 instances for your application in an Auto Scaling group. This way, if one of the EC2 instances fails, the Auto Scaling group automatically launches a new instance to replace it. You should configure the instances to launch your Amazon Kinesis Data Streams application at startup.

Infrastructure Security in Kinesis Data Streams

As a managed service, Amazon Kinesis Data Streams is protected by the AWS global network security procedures that are described in the Amazon Web Services: Overview of Security Processes whitepaper.

You use AWS published API calls to access Kinesis Data Streams through the network. Clients must support Transport Layer Security (TLS) 1.0 or later. We recommend TLS 1.2 or later. Clients must also support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

Security Best Practices for Kinesis Data Streams

Amazon Kinesis Data Streams provides a number of security features to consider as you develop and implement your own security policies. The following best practices are general guidelines and don't represent a complete security solution. Because these best practices might not be appropriate or sufficient for your environment, treat them as helpful considerations rather than prescriptions.

Implement least privilege access

When granting permissions, you decide who is getting what permissions to which Kinesis Data Streams resources. You enable specific actions that you want to allow on those resources. Therefore you should grant only the permissions that are required to perform a task. Implementing least privilege access is fundamental in reducing security risk and the impact that could result from errors or malicious intent.

Use IAM roles

Producer and client applications must have valid credentials to access Kinesis data streams. You should not store AWS credentials directly in a client application or in an Amazon S3 bucket. These are long-term credentials that are not automatically rotated and could have a significant business impact if they are compromised.

Instead, you should use an IAM role to manage temporary credentials for your producer and client applications to access Kinesis data streams. When you use a role, you don't have to use long-term credentials (such as a user name and password or access keys) to access other resources.

For more information, see the following topics in the *IAM User Guide*:

- IAM Roles

- Common Scenarios for Roles: Users, Applications, and Services

Implement Server-Side Encryption in Dependent Resources

Data at rest and data in transit can be encrypted in Kinesis Data Streams. For more information, see Data Protection in Amazon Kinesis Data Streams (p. 209).

Use CloudTrail to Monitor API Calls

Kinesis Data Streams is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Kinesis Data Streams.

Using the information collected by CloudTrail, you can determine the request that was made to Kinesis Data Streams, the IP address from which the request was made, who made the request, when it was made, and additional details.

For more information, see the section called “Logging Amazon Kinesis Data Streams API Calls with AWS CloudTrail” (p. 191).

Document History

The following table describes the important changes to the Amazon Kinesis Data Streams documentation.

Change	Description	Date Changed
Added support for the on-demand and provisioned data stream capacity modes.	Added Choosing the Data Stream Capacity Mode (p. 65).	November 29, 2021
New content for server-side encryption.	Added Data Protection in Amazon Kinesis Data Streams (p. 209).	July 7, 2017
New content for enhanced CloudWatch metrics.	Updated Monitoring Amazon Kinesis Data Streams (p. 180).	April 19, 2016
New content for enhanced Kinesis agent.	Updated Writing to Amazon Kinesis Data Streams Using Kinesis Agent (p. 100).	April 11, 2016
New content for using Kinesis agents.	Added Writing to Amazon Kinesis Data Streams Using Kinesis Agent (p. 100).	October 2, 2015
Update KPL content for release 0.10.0.	Added Developing Producers Using the Amazon Kinesis Producer Library (p. 84).	July 15, 2015
Update KCL metrics topic for configurable metrics.	Added Monitoring the Kinesis Client Library with Amazon CloudWatch (p. 195).	July 9, 2015
Re-organized content.	Significantly re-organized content topics for more concise tree view and more logical grouping.	July 01, 2015
New KPL developer's guide topic.	Added Developing Producers Using the Amazon Kinesis Producer Library (p. 84).	June 02, 2015
New KCL metrics topic.	Added Monitoring the Kinesis Client Library with Amazon CloudWatch (p. 195).	May 19, 2015
Support for KCL .NET	Added Developing a Kinesis Client Library Consumer in .NET (p. 133).	May 1, 2015
Support for KCL Node.js	Added Developing a Kinesis Client Library Consumer in Node.js (p. 130).	March 26, 2015
Support for KCL Ruby	Added links to KCL Ruby library.	January 12, 2015
New API PutRecords	Added information about new PutRecords API to the section called “Adding Multiple Records with PutRecords” (p. 96).	December 15, 2014

Change	Description	Date Changed
Support for tagging	Added Tagging Your Streams in Amazon Kinesis Data Streams (p. 80).	September 11, 2014
New CloudWatch metric	Added the metric `GetRecords.IteratorAgeMilliseconds` to Amazon Kinesis Data Streams Dimensions and Metrics (p. 181).	September 3, 2014
New monitoring chapter	Added Monitoring Amazon Kinesis Data Streams (p. 180) and Monitoring the Amazon Kinesis Data Streams Service with Amazon CloudWatch (p. 180).	July 30, 2014
Default shard limit	Updated the Quotas and Limits (p. 7): the default shard limit has been raised from 5 to 10.	February 25, 2014
Default shard limit	Updated the Quotas and Limits (p. 7): the default shard limit has been raised from 2 to 5.	January 28, 2014
API version updates	Updates for version 2013-12-02 of the Kinesis Data Streams API.	December 12, 2013
Initial release	Initial release of the Amazon Kinesis Developer Guide.	November 14, 2013

AWS glossary

For the latest AWS terminology, see the AWS glossary in the *AWS General Reference*.