Efficiently stream DML events on AWS

14 April 2023 - 8 min. read

Amazon Kinesis Data Streams | AWS Database Migration Service

In the past, many workloads came with their own big and monolithic database, where not only the application but also reporting tools and technical support connected to it and performed queries.

As this is still true today, companies are moving towards having single information stored on multiple data sources and servers. Only the core application should be able to access the database, reporting tools should use data that is stored on a separate instance, and monitoring and data analytics should be done by aggregating data that comes from different sources.

To do this we need to stream the changes that are occurring on our database to one or more destinations. Today we are going to take a look at how to do this on AWS.

AWS Database Migration Service (DMS) is a powerful tool for migrating data between various database platforms. One of the standout features of AWS DMS is its Change Data Capture (CDC) functionality, which allows for real-time streaming of changes made to a source database to a target database.

When using AWS DMS, you have the option to attach a target database directly as an endpoint or use Amazon Kinesis Data Streams to capture and process the streaming data.

Here are some differences between the two approaches:

1. Latency: when streaming data directly to a target database, there may be some latency involved in the processing and writing the data. With Kinesis Data Streams, the data is captured and processed in real-time, so there is no delay in processing.

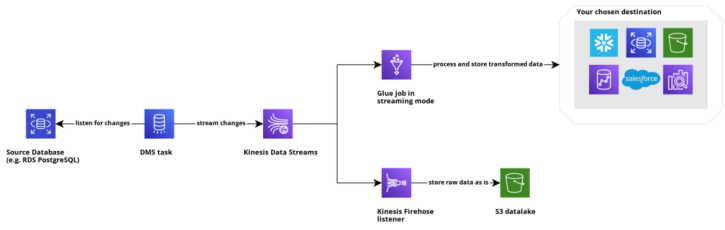
2. Scalability: Kinesis Data Streams is designed to handle large volumes of streaming data, and can automatically scale to accommodate increased traffic. When streaming data directly to a target database, you may need to manually scale the database to handle increased traffic.
3. Flexibility: with Kinesis Data Streams, you can easily process and analyze the streaming data using various AWS services, such as AWS Glue or AWS Lambda. When streaming data directly to a target database, you may have limited options for processing and analyzing the data.
4. Cost: using Kinesis Data Streams may incur additional costs for processing and storing the streaming data, as well as any associated AWS services used for processing and analysis. Streaming data directly to a target database may not have any additional costs, but you may need to consider the cost of scaling the database to handle increased traffic.

Overall, both approaches have their advantages and disadvantages, and the best choice depends on your specific use case and requirements. In this article, we are going to explore the possibility to process insert/update/delete events on flight with the help of Amazon Kinesis Data Streams.

Setup efficient DML events streaming on AWS

Now, let's build a proof of concept to test out the CDC streaming solution with DMS and Kinesis Data Streams. The idea is to have an automated process that gives us an easy way to replicate changes that happen on a source database to one or more destination engines.

This is a diagram of what we're going to build:



The ingestion

The first thing we need to do, if we want to enable CDC, is **configure our source database** to make all the information needed by DMS available to trap new events. For many engines,

this means running a bunch of queries that are well-described in the official AWS documentation.

After the source database has been configured, let's **create our Kinesis Data Stream**.

This step is pretty straightforward as we don't need to provide many parameters. We only need to decide what's our data stream's capacity mode:

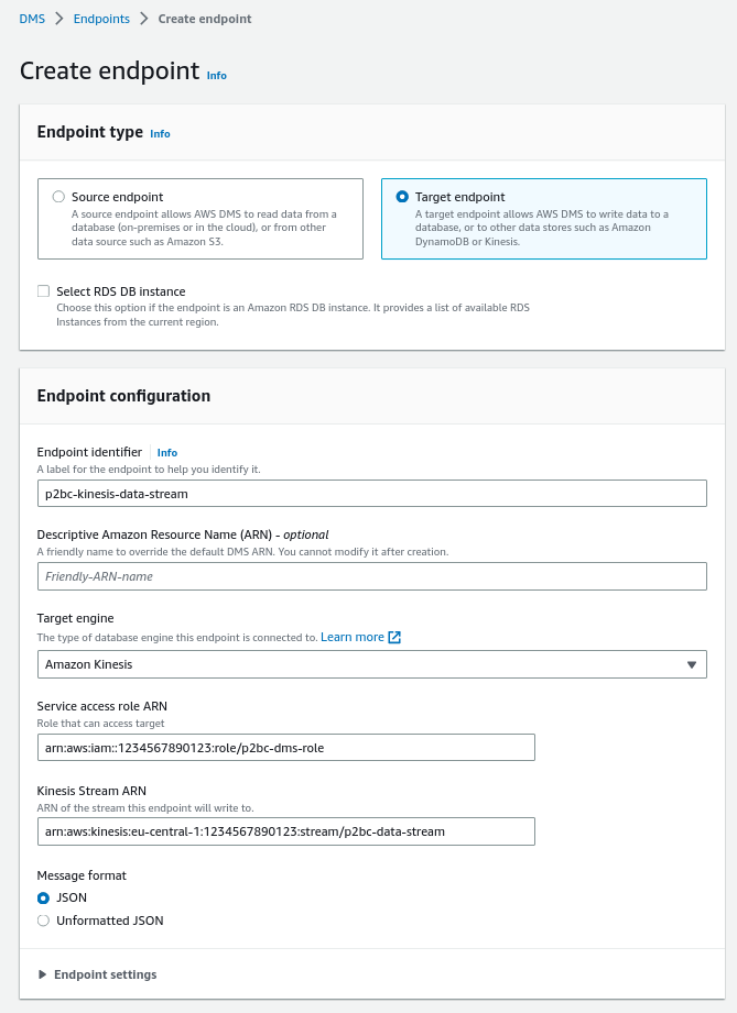
- With **on-demand** we delegate the scaling operations to AWS.
- With **provisioned** we need to provide the number of shards (i.e. the read and write throughput) which represents the size of our stream.

In our case, we chose the on-demand capacity mode. Keep in mind the **concept of sharding**, it'll come back later on in this article.

Now that we have the source and target systems configured, we need to **create the AWS DMS instance and endpoints,** which are basically a set of configurations that DMS uses to interact with the various systems.

Configuring a DMS endpoint is really straightforward: first, you need to choose whether you are creating a source or target endpoint, then you specify the type of engine that the endpoint is connected to, and finally, depending on the selected engine, you need to provide a few parameters to configure the connection.

Here's an example of the creation form for the Kinesis Data Stream endpoint:



AWS DMS requires an IAM role to be able to interact with Kinesis. This role should have at least these permissions:

```
{
        "Version": "2012-10-17",
        "Statement": [
                {
                        "Action": [
                                "kinesis:PutRecord",
                                        "kinesis:PutRecords",
                                        "kinesis:DescribeStream"
                                ],
                                "Resource": [
                                        "<your delivery stream arn>"
                                ],
                                "Effect": "Allow"
                }
        ]
}
```

Now the last resource to **create** on the AWS DMS console is the **actual task that will be taking care of copying data from our source to Kinesis**.

Choose the DMS replication instance where you want to deploy your task, select the endpoints you previously created and configure the type of migration you want to perform: you can have a migration that takes a snapshot of the data that is stored on the source engine, or you can have change data capture enabled so that new events are automatically replicated to the target engine, or you can have both.

Replication tasks have tons of configuration parameters you can tweak to increase the efficiency, stability, and speed, but I'm not going to deep dive into them; just know that, for this POC, I didn't change anything. Everything is set to its default value.

The task creation will take a few minutes, in the meantime we can focus on creating the resources for...

The processing

On the architecture diagram, I put **AWS Glue** as the processing service because of its native integration with many data warehouses and for the possibility to ingest and process big datasets thanks to Spark. Furthermore, AWS Glue Jobs can be configured in streaming mode, meaning that they are always on and processes incoming data as it comes.

Here you can find a small code snippet that can help you kickstart a processing job:

```
import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from awsglue.context import GlueContext
from awsglue.job import Job
from pyspark.context import SparkContext
from pyspark.sql.functions import from_json, col, lit, current_timestamp
from pyspark.sql import DataFrame

args = getResolvedOptions(sys.argv, ["JOB_NAME", "kinesis_stream_arn"])
job_run_id = args['JOB_RUN_ID']
sc = SparkContext()
glue_context = GlueContext(sc)
spark = glue_context.spark_session
job = Job(glue_context)

def process_batch(data_frame: DataFrame, _batch_id: int):
    job.init(args["JOB_NAME"], args)
    if data_frame.count() <= 0:
        job.commit()
        return
    data_frame.printSchema()
    data_frame.show()
    # Do your processing here
    job.commit()


amazon_kinesis_dataframe = glue_context.create_data_frame.from_options(
    connection_type="kinesis",
    connection_options={
        "typeOfData": "kinesis",
        "streamARN": args["kinesis_stream_arn"],
        "classification": "json",
        "startingPosition": "TRIM_HORIZON",
        "inferSchema": "false",
        "avoidEmptyBatches": "true",
        "schema": "`data` string, `metadata` STRUCT<`timestamp`: TIMESTAMP, `record-type`: STRING,
`operation`: STRING, `partition-key-type`: STRING, `schema-name`: STRING, `table-name`: STRING,
`transaction-id`: BIGINT> NOT NULL"
    },
    transformation_ctx="amazon_kinesis_dataframe",
)

glue_context.forEachBatch(
    frame=amazon_kinesis_dataframe,
    batch_function=process_batch,
    options={
        "windowSize": "100 seconds",
        "checkpointLocation": args["TempDir"] + "/" + args["JOB_NAME"] + "/checkpoint/",
    },
)
```

Check the GitHub Gist to copy the code.

This code basically creates a link between the Glue Job and a Data Frame that has Kinesis Data Stream as the source. This Data Frame will be refreshed every 100 seconds, and every new version of it will be passed to the *process_batch* function. This function takes care of the processing of new data.

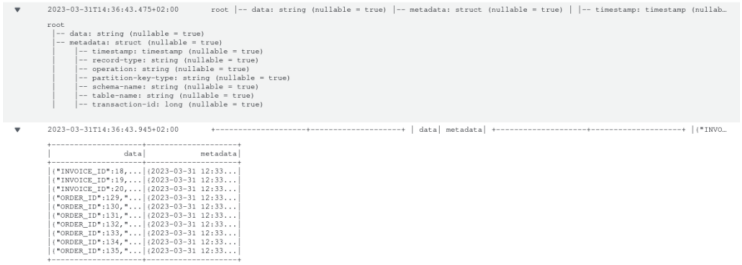
When creating the Kinesis Data Frame, we can provide a **schema format for the incoming data**.

This schema must match the format of the data that DMS provides us, which is a JSON that contains two keys:

- data, another JSON that contains the values of the inserted/updated/deleted record;
- metadata, another JSON that contains information about the event, such as the schema and table name where the event occurred, the timestamp of the event, etc.

As you can see, I only provided a "structured" schema for the metadata part. This is because we can't possibly know the format of the "data" key before knowing the name of the schema and table.

After starting our new Glue job, we should see some logs on CloudWatch similar to these:



Finally, I wanted to **dump the events on Amazon S3** as they come from DMS, so I can always decide to go back and reprocess everything up to a certain point in time. We already have our Data Stream on Kinesis, and we can attach multiple consumers to it, so we can easily create a Kinesis Firehose Delivery Stream to aggregate events and dump them on an S3 bucket.

Possible issues

When we tried for the first time this solution, we bumped into a performance problem related to the scalability of our Data Stream: as we discussed before, Kinesis uses a concept of shards as the unit to measure the scaling. When the producer produces a new event, it has to provide the shard where the event has to go or use a random ID to make Kinesis decide.

DMS always uses the same value.

By doing so, it always uses the same shard, so even if you configured your Data Stream's capacity mode as on-demand, it won't scale because of DMS.

The DMS Kinesis target endpoint has a setting that you can enable called **PartitionIncludeSchemaTable** which makes DMS use the schema and table name as the partition key (thus the shard). This is still not enough for scaling in my opinion: if you are migrating x tables, you'll have x shards, each one dedicated to its own table. If one of your tables sees an increase in events, Kinesis still won't scale to manage the spike because DMS will keep sending data to the same one shard.

Fortunately, this might be a problem only if your source tables are very big (we are talking about millions of events every day). If your workload is more standard, even with this DMS issue you probably won't see your performance degraded.

Hopefully, AWS will put a patch on it :)

Conclusion

This infrastructure is pretty simple to set up and it surely gives you many benefits: you can process your data as it comes while maintaining a raw copy of it on a separate storage, and it is scalable and flexible: Since Glue Jobs configured in streaming mode are not cheap, you can always replace them with a Lambda function if you want to save some money and if you don't have many data manipulation events happening.

Did you try this or a similar solution? Tell us about your experience!

See you in 14 days here on **Proud2beCloud**!

About Proud2beCloud

Proud2beCloud is a blog by beSharp, an Italian APN Premier Consulting Partner expert in designing, implementing, and managing complex Cloud infrastructures and advanced services on AWS. Before being writers, we are Cloud Experts working daily with AWS services since 2007. We are hungry readers, innovative builders, and gem-seekers. On Proud2beCloud, we regularly share our best AWS pro tips, configuration insights, in-depth news, tips&tricks, how-tos, and many other resources. Take part in the discussion!

Mattia Costamagna

DevOps engineer and cloud-native developer @ beSharp. I love spending my free time reading novels and listening to 70s rock and blues music. Always in search of new technologies and frameworks to test and use. Craft beer is my fuel!