Database Migration Service â€” Case Study #2

Author: Jon Ross

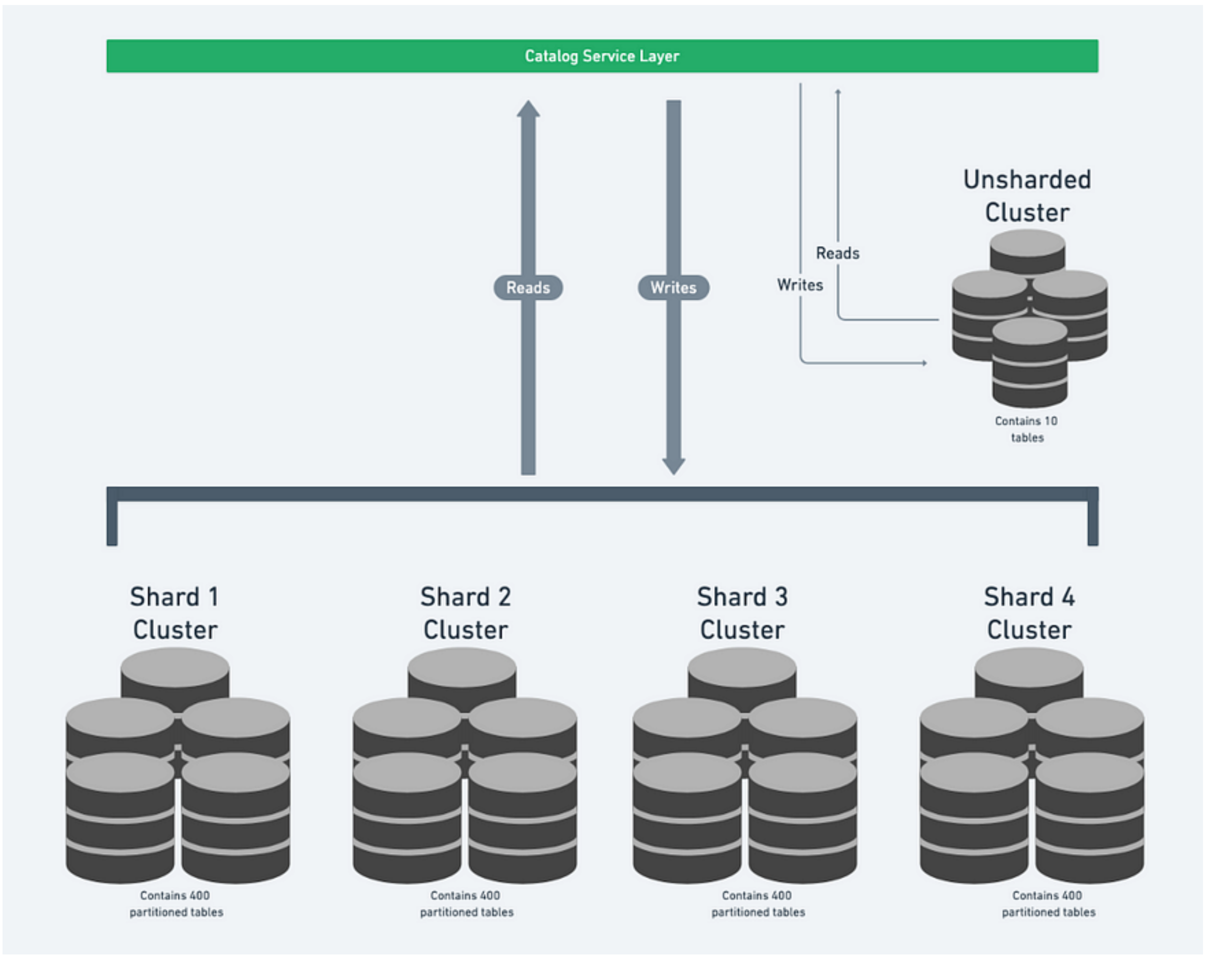
Claps: 207

Date: Aug 29

In a previous blog post, we reported on our use of AWS Database Migration Service (DMS) to migrate data from one database to another with little to no issues. My team is responsible for the storage of catalog data and we also recently used DMS. Due to the size of our databases and the volume of writes, we had to overcome a few bumps. Keeping the stack current is essential and challenging for any system operating at scale, and weâ€™re always getting better at it here at Klaviyo. I hope this case study will be helpful to engineers planning for large database upgrades.

There are over 130,000 companies using Klaviyo to communicate with their customers. Many of those companies have distinct product catalogs that can range anywhere from a single product to tens of millions of products. In order for Klaviyo users to reference those products in their messaging to their customers, we need to store the catalog data somewhere.

We store this catalog data across 400 partitioned tables which are also sharded across four different Postgres database clusters (that is 4 tables each split between 100 partitions per each database shard) along with a few other tables that live in an unsharded Postgres cluster. With this system weâ€™re able to service 479k writes/minute (690M writes/day) and service 551k reads/minute (794M reads/day) to support the different catalog data our users want to include in their messaging to their customers. For the past two years this data has lived on Postgres 11 databases, and seemed to be doing fine with that setup.



A simplified view of our service. The bulk of our data exists within the sharded databases and those databases experience the bulk of our reads/writes.

On February 3rd, we received this email from AWS:

> Hello,
>
> We are reaching out to let you know that newer versions of Aurora PostgreSQL are now available and that as of January 31, 2024, Amazon Aurora PostgreSQL-compatible edition will no longer support major version 11.x. Per the Aurora Version Policy [1], we are providing 12 months notice to give you time to upgrade your database cluster(s). We recommend that you proactively upgrade your databases running Amazon Aurora PostgreSQL major version 11.x to Amazon Aurora PostgreSQL 12 or higher at your convenience before January 31, 2024. We will send you updates and reminders throughout the year prior to the end of support deadline.
>
> Starting on or after 12:00 PM PST on January 31, 2024, if your Amazon Aurora PostgreSQL cluster has not been upgraded, Amazon Aurora will automatically upgrade your Amazon Aurora PostgreSQL 11.x databases to the appropriate Amazon Aurora PostgreSQL version during your next scheduled maintenance window.

If we didnâ€™t upgrade our databases by January 31, 2024, AWS would forcibly upgrade us to Postgres 12 or higher, resulting in an unknown amount of downtime. It would also potentially move us to a database version where patterns we rely on would no longer be supported or would behave differently than expected.

Making an Upgrade Plan

Weâ€™ve done database upgrades at Klaviyo in the past, and those have usually taken one of the following forms:

1. Accepting that downtime can occur, and just upgrading the database in place. This requires communicating with the proper up/downstream teams to ensure the downtime would only affect the expected product area and also requires communicating out to customers about the downtime, since it would impact their workflow.
2. Spinning up a fresh database running the newer version and writing bespoke backfill and dual read/write code. The backfill code would copy the existing database tables over to the new database. The dual read/write code would then copy over every update, insert, and delete operation as they occurred. This is known as a â€œdual read/writeâ€� approach, where the code is updated to write and read from both datastores simultaneously. If any inconsistencies between the data was noticed during a read it would be logged. Eventually the two data stores would become consistent with each other and the switch from the old database to the new could happen with no or minimal downtime.
3. Utilizing Amazonâ€™s Blue/Green Deployment. This is a tool we have begun using for upgrading our MySQL databases with great success, but as of now is not available for Postgres upgrades.

#1 seems like the â€œsimplestâ€� approach since the upgrade process itself is just clicking a button in the AWS UI. The downside is that systems within Klaviyo would be unable to talk to the database for an indeterminate amount of time. This could have severe customer impact â€” messages relating to catalog data would fail to send, updates to the catalog data in Klaviyo wouldnâ€™t process properly, and subscriptions to Back in Stock would not be persisted (nor would those notifications trigger). This type of outage wouldnâ€™t be acceptable.

#2 is how we did our previous database upgrade for the catalog databases. Before we sharded our data, all of this catalog data was stored in a single database. This database was very large and was not able to service our catalog related queries as we needed it to at the time, which led us to switching over to the partitioned and sharded model we have now. Dual Read/Write was a solid solution at the time, but that upgrade occurred more than two years ago, and our codebase had gone through a lot of changes since then. Trying to use this pattern again would take a good amount of time re-coding the solution, and invariably the next time we needed to upgrade these databases, we would need to rewrite the code again. Since we were trying to come up with repeatable patterns for our database upgrades to make them as simple as possible, we didnâ€™t want to write a ton of custom catalog service-specific code.

#3 would have been ideal. Klaviyo is currently using the Blue/Green deployment tool to upgrade all of our MySQL 5.7 databases to MySQL 8. This would check all of the boxes we were looking for: a simple, repeatable, known and proven pattern for upgrading our databases. Unfortunately, AWS only supports Aurora MySQL for the Blue/Green deployments so this was out of the question.

So none of the previous methods we were familiar with for database upgrades seemed like they were going to work for this sort of â€œrepeatable, near zero downtimeâ€� type upgrade. We had read a little bit about AWS Database Migration Service, and had seen some other teams at Klaviyo use it successfully, but we werenâ€™t too familiar with how it worked or if it would fit our use case.

To test DMS out to see if we could use it to facilitate this database upgrade, we cloned one of our production clusters into our dev account and then spun up a fresh Postgres 12 cluster and the requisite DMS infrastructure alongside it. We manually defined tasks (a DMS concept), kicked them off, and determined that the data was replicated over successfully.

An important detail is that some of our larger partitioned tables have JSON columns which store additional arbitrarily-large metadata. LOB columns can really slow down the DMS migration process and AWS has helpful information about the different settings you can apply to your task to speed them up. After testing out each one, we opted to go with the â€œLimited LOB modeâ€� setting. To figure out the max size of the JSON columns to set for the LobMaxSize, we ran queries similar to `SELECT max(pg_column_size(to_jsonb(metadata))) FROM â€¦;` for each partitioned table and specified it within each of the task settings.

After testing queries and confirming that upgrading to Postgres 12 wouldnâ€™t break anything, we upgraded the test database up to Postgres 14 to see if we could make the jump from Postgres 11 -> 14 instead of going to 12, then 13, then 14. Our tests were successful, so we wrote up an upgrade plan and presented it to the team.

The plan was to spin up DMS infrastructure for each of our database shards and then â€œupgradeâ€� each one by running ten DMS tasks (with each one in charge of replicating 40 tables) to replicate the data to a new Postgres 14 database. And then once replication caught up, we would hot switch over from the old database to the new one.

Setting up the replication

We wanted this process to be repeatable. For this specific upgrade, we needed to upgrade five different database clusters. For each of these clusters, we wanted the process to be as easy as possible â€” just spin up the infrastructure and start the tasks. In our testing, we manually defined the DMS tasks through the UI. This was a) time consuming and b) error prone as you had to do a few clicks in the UI to get the tasks created. As such, we decided to define the DMS infrastructure (and more specifically, the DMS tasks) through Terraform.

We created a â€œsharded_catalogs_dms_upgradeâ€� module, which took in the following arguments:

```
module "dms_sharded_catalogs_upgrade_node_1" {
  source                        = "../sharded_catalogs_dms_upgrade"
  shard                         = 1
  availability_zone             = "..."
  target_db_endpoint            = ...
  source_db_endpoint            = ...
}
```

Under the hood, this module initialized all of the infrastructure needed to run the DMS replication. Since the naming of the 400 different tables was consistent across each of the database shards (shard 1 contains partitions 1000â€“1099, shard 2 has 2000â€“2099, etc.) this was easy to configure:

```
# Create the replication subnet group
resource "aws_dms_replication_subnet_group" "sharded_catalog_replication_s
  replication_subnet_group_description = "Replication subnet group for the
  ...
}

# Create the replication instance
```

```
resource "aws_dms_replication_instance" "catalog_dms_upgrade_replication_i
  replication_instance_id      = "catalogs-postgres-upgrade-shard-${var.sh
  replication_subnet_group_id  = aws_dms_replication_subnet_group.sharded_
  availability_zone            = var.availability_zone
  ...
}

# Create the source endpoint
resource "aws_dms_endpoint" "catalogs_source_db" {
  server_name                 = var.source_db_endpoint
  ...
}

# Create the target endpoint
resource "aws_dms_endpoint" "catalogs_target_db" {
  server_name                 = var.target_db_endpoint
  ...
}

# Create the replication tasks
resource "aws_dms_replication_task" "catalog_dms_upgrde_replication_tasks"
  for_each = {for instance in local.sharded_replication_task_data.task_lis

  migration_type            = "full-load-and-cdc"
  replication_instance_arn  = aws_dms_replication_instance.ccc_replication
  replication_task_id       = "c3-postgres-upgrade-shard-${var.shard}-${ea
  replication_task_settings = file("${path.module}/task_settings.json")
  source_endpoint_arn       = aws_dms_endpoint.catalogs_source_db.endpoint
  target_endpoint_arn       = aws_dms_endpoint.catalogs_target_db.endpoint
  table_mappings            = "{\"rules\":[{\"rule-type\":\"selection\",\"
}
```

While coding the Terraform and getting our systems ready for the database migration, we noticed three important details:

1. DMS does not play entirely nicely with partitioned tables.

From the AWS docs:

> To replicate partitioned tables from a PostgreSQL source to a PostgreSQL target, first manually create the parent and child tables on the target. Then define a separate task to replicate to those tables. In such a case, set the task configuration to Truncate before loading.

We would have to define each of the 400 tables on each database cluster manually before kicking off the replication. Thankfully Postgres supplies pg_dump, which we used to dump out the schema from the source database which we then applied to the target database to initialize our partitioned table definitions (and associated indices, etc).

2. The data transfer in DMS is free, if configured properly.

From the docs:

All data transfer into AWS DMS is free, and data transferred between AWS DMS and Amazon RDS, Amazon Redshift and Amazon EC2 instances in the same Availability Zone also is free.

If the replication instance is initialized in an availability zone different from the AZ of the target database, AWS will charge you for that data transfer. In our migration (see above Terraform), we made sure that the replication instance was being spun up in the same AZ as the target cluster to avoid being charged additional fees.

3. Auto incrementing sequences will not increment on the target.

This wasnâ€™t 100% surprising, but was still something we were hoping we wouldnâ€™t have to deal with. If we had tried to switch over to the target database without accounting for this, the newly created rows would have ids starting at 0 and would clash with already existing ids. To work around this issue, we created a script to copy the sequence from the source db to the target db, adding in some buffer. This script was planned to be run as one of the cutover steps taken by the engineer in charge of the upgrade for that specific shard.

```
def reset_sequence_on_target():
    """
    This function will reset the sequences on the target to be in
    line with the sequences that we have on the source database.
    We're adding in a little bit of buffer room just to be extra safe
    """

    old_db_conn = psycopg2.connect(
        dbname=OLD_DB_NAME,
        user=DB_USER,
        password=DB_PASSWORD,
        host=OLD_WRITER,
        port=DB_PORT,
    )

    new_db_conn = psycopg2.connect(
        dbname=NEW_DB_NAME,
        user=NEW_DB_USER,
        password=NEW_DB_PASSWORD,
        host=NEW_WRITER,
        port=DB_PORT,
    )

    new_db_cursor = new_db_conn.cursor()
    old_db_cursor = old_db_conn.cursor()

    for sequence in SEQUENCES:
        new_db_cursor.execute(f"select last_value from {sequence}")
        target_seq_value = int(new_db_cursor.fetchone()[0])
        print(f"Target db has {sequence} == {target_seq_value}")
        old_db_cursor.execute(f"select last_value from {sequence}")
        seq_value = int(old_db_cursor.fetchone()[0])
        new_seq_value = seq_value + SEQUENCE_BUFFER
        print(
            f"Source sequence value is {seq_value}, going to update the ta
        )
        new_db_cursor.execute("SELECT setval(%s, %s)", (sequence, new_seq_

    old_db_conn.close()
    new_db_conn.close()
```

Optimistic that weâ€™d accounted for everything, and following our upgrade plan, we kicked off the DMS tasks for our shard 1 database cluster. Unfortunately this is where the fun began and we learned, once again, that things rarely work as advertised at scale!

Troubleshooting the Replication

Initially it looked like this was going to go off without a hitch! The tasks were progressing and migrating the data over without any issues. That was until we started to get to our tables that were a bit larger (due to having more rows in them and also the rows including LOB data in the form of JSON columns).

Weâ€™d check in on a task and see that it had moved over 1M+ rows of one table and then weâ€™d check in on the task again a little bit later and see it was still processing the same table, but somehow had only moved over 500k rows. This was slightly confusing, but we figured the task had hit some kind of blip, so we let them run overnight with the hopes that come the next morning it would have finished processing all of the tables as expected.

Instead, when we logged on the next morning, we saw that that task (and all others) had seemingly got stuck when processing these similarly large tables. By â€œstuckâ€� I mean they would begin processing the table, but before the task was able to finish processing the table (since it was a larger table) the task itself would restart. In addition to being the largest tables, these tables were also the ones that experienced the most writes when compared to the previous tables which had replicated over with no issue (~1k writes/minute vs ~20 writes/minute). With this in mind, we decided to start pulling some levers by turning off certain write heavy workloads that we had control over. This ended up helping a bit, the tasks werenâ€™t restarting as quickly as they had been, so some tables that had been stuck were finally able to finish replicating over to the target database.

This wasnâ€™t a full solve, though. Ideally we wanted this database migration process to have little to no impact on our production workloads, so having to stop some processes to get it to work was out of the question. Additionally, since this change didnâ€™t stop the restarting issue, we were still having trouble progressing through our last few tables which were our largest.

We opened a ticket up with AWS support to see if they could help us debug. We now suspected that the error we were experiencing had *something* to do with the write load on the source database. Support advised us to look into (and helped us parse through) the various logs from our DMS tasks, our target database, and our source database.

In both our source and target database logs, we were seeing the following error logs right around the times of each of the task restarts:

```
LOG:  could not receive data from client: Connection reset by peer
LOG:  unexpected EOF on client connection with an open transaction
LOG:  could not receive data from client: Connection reset by peer
LOG:  could not receive data from client: Connection reset by peer
```

AWS support was able to supply more in-depth DMS task logs, which were consistently logging these errors around the same times:

```
[TASK_MANAGER    ]I:  Task - JRU2JX7YSETTPAWHILLVSQLM6CK5VPC9BQYTBPI is in
[TASK_MANAGER    ]I:  Task 'JRU2JX7YSETTPAWHILLVSQLM6CK5VPC9BQYTBPI' encou
[SERVER          ]I:  Task process for 'JRU2JX7YSETTPAWHILLVSQLM6CK5VPC9BQ
[SERVER          ]I:  Resume task 'JRU2JX7YSETTPAWHILLVSQLM6CK5VPC9BQYTBPI
```

As mentioned in the DMS logs, this was due to some error occurring, DMS deciding it was a â€œrecoverable error,â€� and then restarting the table replication from the beginning. From the database logs it can be assumed that the error the task was running into was due to some sort of disconnection from the database.

We began to look a little deeper into our source database logs to see if there were any more specific errors logged before the client disconnect error. Sure enough, a little further up in our logs, hidden between other production specific database logging, we were able to see:

```
LOG:  could not truncate directory "pg_commit_ts": apparent wraparound
LOG:  could not truncate directory "pg_commit_ts": apparent wraparound
LOG:  could not truncate directory "pg_commit_ts": apparent wraparound
LOG:  could not truncate directory "pg_commit_ts": apparent wraparound
LOG:  could not truncate directory "pg_commit_ts": apparent wraparound
```

As part of the preparation for using `pglogical` for DMS, we enabled the `track_commit_timestamp` parameter on our databases. This parameter does exactly what it sounds like it does, which is if it is set to True it will cause the database to track the commit time of transactions. Apparently this couldnâ€™t keep up with the load of commits the database was experiencing, would hit wraparound and then would result in the task disconnecting and restarting. This definitely seemed to be the smoking gun, especially since decreasing the number of commits our database was experiencing had led to positive results. After conferring with AWS, they guided us to use their `test_decoding` plugin instead of `pglogical` for our migration and to turn `track_commit_timestamp` off.

Time to pop the champagne and celebrate a successful migration, right? Wrong. The tasks did begin to make further progress than before, but were still restarting every 5â€“6 hours. Overnight we were able to get almost all tables replicated over to the target database, but due to the restarts we were stuck with one very large table never completing.

AWS sent us with the following logs from our DMS tasks:

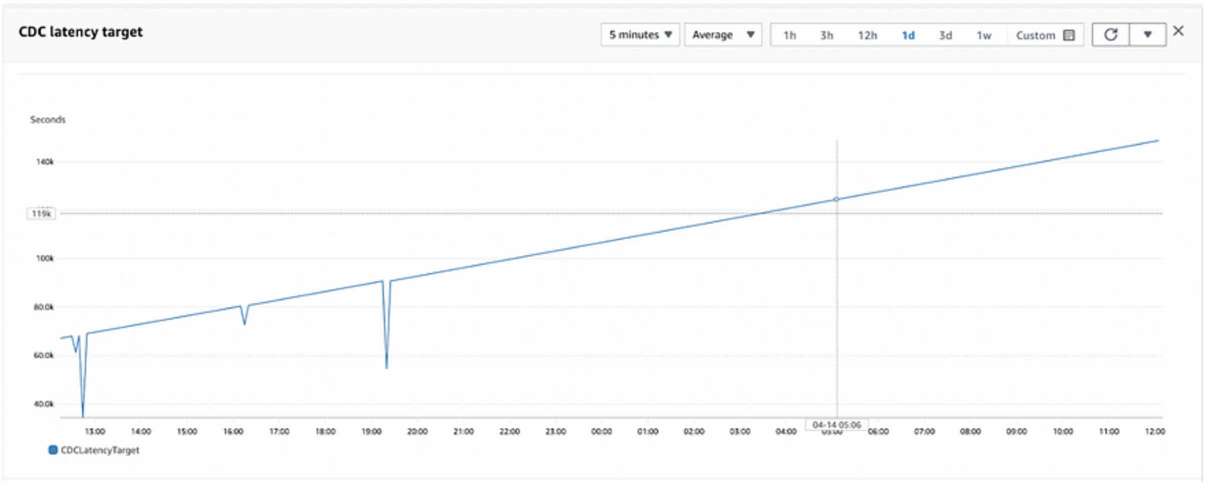
```
[SERVER ]I:  Task 'JRU2JX7YSETTPAWHILLVSQLM6CK5VPC9BQYTBPI' stopped with r
[TASK_MANAGER ]I:  Task - JRU2JX7YSETTPAWHILLVSQLM6CK5VPC9BQYTBPI is in ER

[SOURCE_CAPTURE  ]D:  Could not receive data from WAL stream: SSL SYSCALL
[SOURCE_CAPTURE  ]D:  WAL stream loop ended abnormally. (STATUS_RECOVERABL
[SOURCE_CAPTURE  ]I:  End of source capture. LSN-s for slot 'jru2jx7ysettp
[SOURCE_CAPTURE  ]D:  WAL reader terminated with broken connection / recov
```

This seemed to point towards one final issue with our WAL (write-ahead log) sender timeout. We had previously increased this from 60 seconds to 5 minutes, but that had clearly not been enough for our needs. We set wal_sender_timeout to 0 (which turns off the timeout as a whole) and restarted the task.

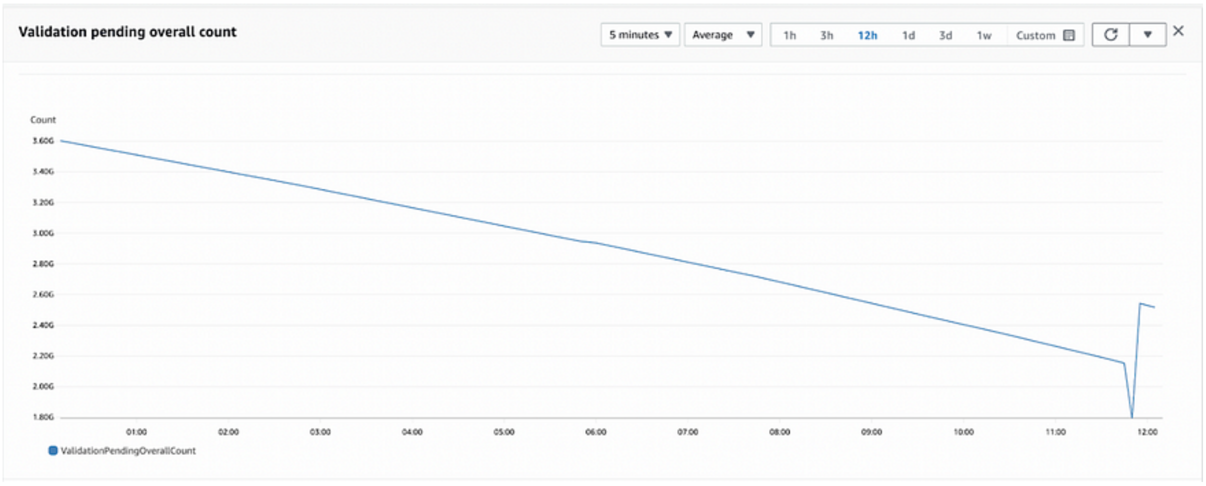
Time for that champagne? Not quite yet.

Our tasks had finally completed the Full Load phase of DMS, but they still had to catch up to the changes the source database saw during the replication, and they still had to validate that all of the data transferred over to the target database was the same as the data stored in the source database. This, apparently, can take some time.

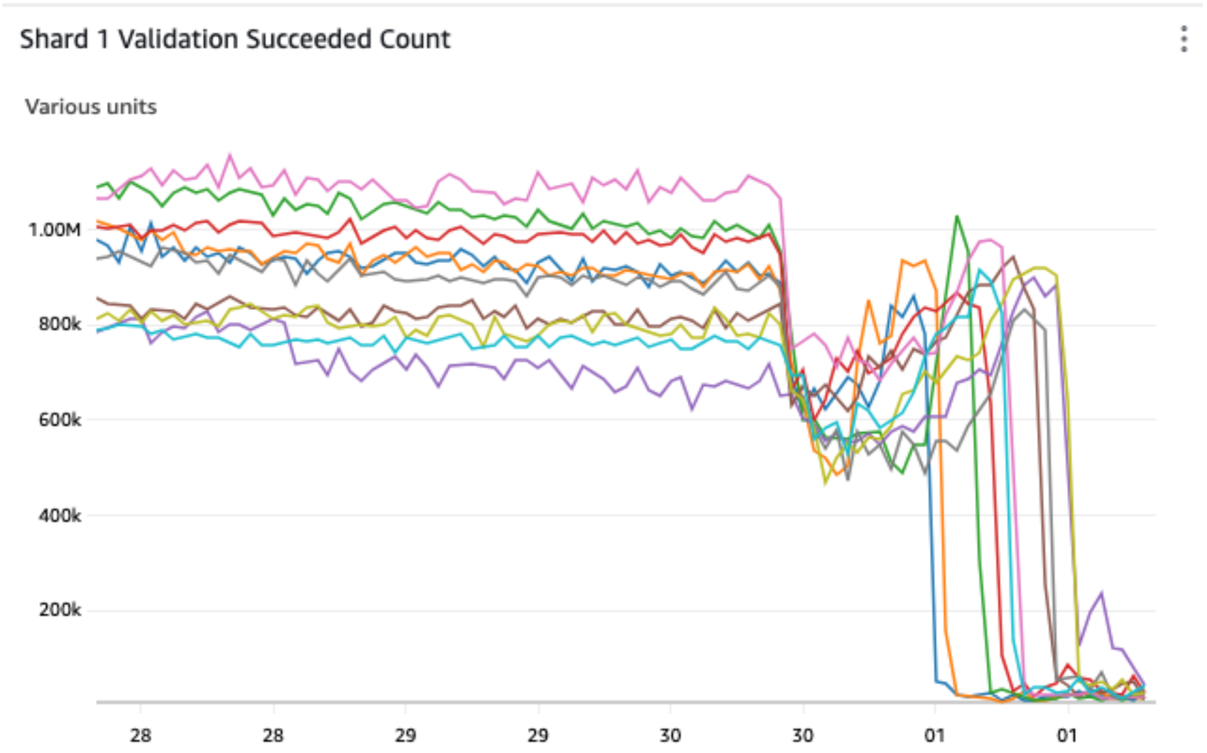
After the full load finished, we were expecting the CDC (change data capture) Target Latency values to drop immediately. This latency had been slowly building over time as we troubleshooted the different issues we were running into during Full Load.



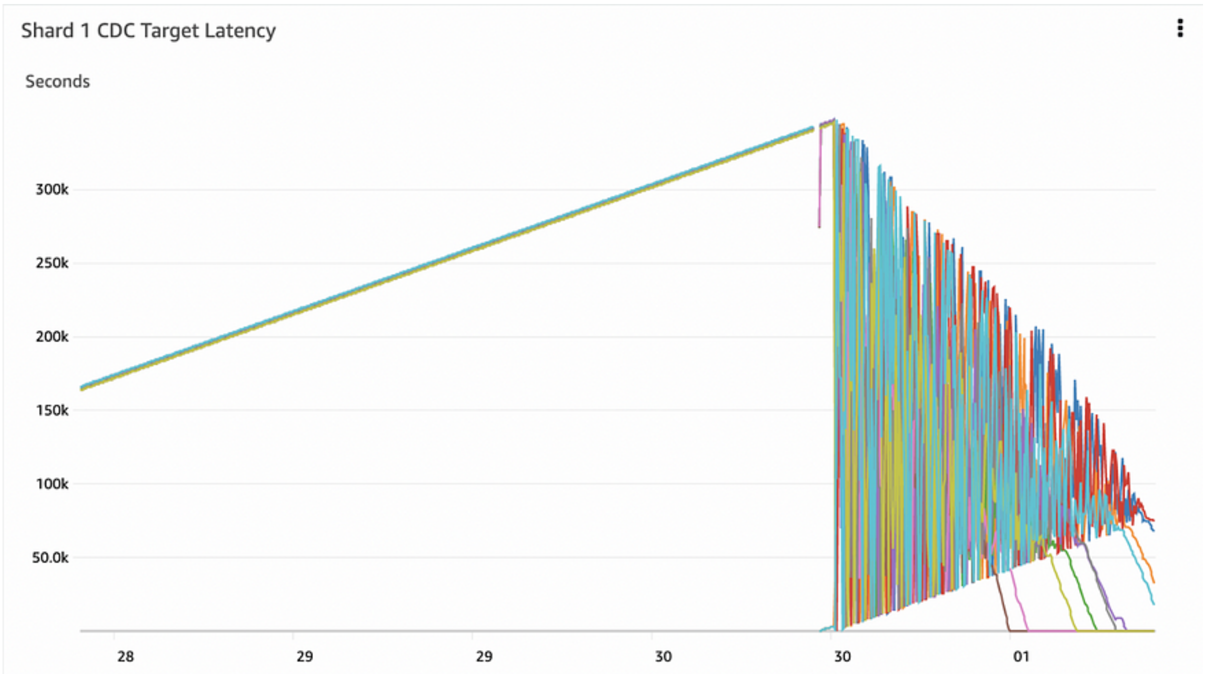
We reached out to AWS for clarity on this, and they explained to us that for tasks running in Full Load + CDC with validation enabled, the tasks donâ€™t begin replicating over the CDC changes until validation of the Full Load data. And boy were there a lot of records to validateâ€¦



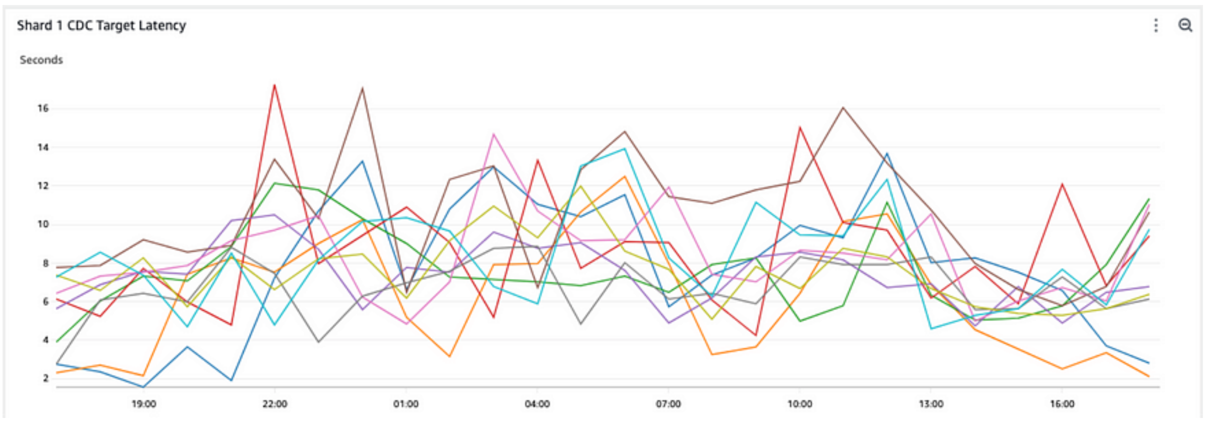
It took a few days, but eventually the validation completed and the CDC target latency began to drop as the CDC changes began processing. That also took a few days (as there were a lot of changes to get through) but eventually the CDC latency settled down to an average of ~8 seconds of latency. We would see brief spikes up to ~30 seconds occasionally (when our db was experiencing periods of high writes), but generally stayed between 2 and 16 seconds of latency.



Successful validation attempts on the shard 1 replication. The count dropped significantly as the CDC replication completed.



Latency for the changes being replicated to the target database. Around the 30th and 1st this dropped considerably once the validation began to complete.



A zoomed in graph of the CDC latency taken after the 1st, once the validation had completed and CDC replication had caught up.

This was great! It took a little longer than we wanted, and we had to overcome some issues along the way, but we had successfully migrated all of our data from the source Postgres 11 cluster to the target Postgres 14 cluster and were continually replicating that data with less than 10 seconds of latency between the two. Now that we were replicating data over to the target cluster in near real time, we could begin the work to switch over our application code to talk directly to that database.

Swapping over to our new database

We needed to be cautious. Remember, the main goal of this project was to upgrade to a new database version with little to no production impact. Swapping over to the new database and accidentally causing an outage within the catalogs product area would have made all of our prior work on the upgrade moot. As such, we had to come up with a clear and straightforward plan:

Step 1: Create the necessary replicas for the cluster, matching the config of our Postgres 11 cluster. We intentionally did not bring up any replicas for the target DB initially so as to not waste money on instances we werenâ€™t using.

Step 2: Add the Postgres 14 database creds to our secrets config and prepare a PR that would update the catalogs microservice to talk to the new database.

Step 3: Deploy this PR to an on-demand kubernetes pod to test out the connection to the database.

Step 4: After confirming that the database connection works as expected, roll out the change to our read-only pods. These are the pods that service our product feed queries, which are used to render catalog information in messages to customers. These pods never need to write to the database, so updating them to talk to the target db while the DMS replication was still ongoing was ok.

Step 5: Let the above change bake for ~10 minutes. Monitor the different queries going through those pods and ensure that they are performing as expected.

****This is where the actual cutover begins****

The following steps should be done in quick succession to ensure limited impact to our production systems.

Step 6: Revoke insert, update and delete access on the source database from the production user. This will error out with permissions errors on write calls in the application, but the system will catch them and raise them as retryable exceptions to its callers.

Step 7: Check the DMS replication tasks. CDC latency should drop to zero now that the source is no longer taking on any writes.

Step 8: Run the script (shown above) to update the sequences on the target database to match those on the source database plus some buffer.

Step 9: Deploy the database change out to the rest of the fleet. The write errors that were previously raised as retryable should now be processing successfully and all queries should be flowing through to the target database instead of the source.

This process feels a little bit like Indy swapping out the bag of sand for the idol. Luckily for us, since we had prepared and written out a detailed plan for this process (complete with rollback steps for each step of the plan) we didnâ€™t have to spend the next few moments escaping from an array of booby traps.

Repeating the process

We had now successfully cut over one of our Postgres 11 clusters to Postgres 14. But as mentioned previously, this cluster was just one of the five database clusters we use to power our catalogs product area. We needed to follow the above process again for each of these clusters to get them upgraded to Postgres 14.

This was relatively simple, we just needed to define the DMS infrastructure for the remaining shards (we also created a separate module for the fifth cluster, which is unsharded).

```
module "dms_sharded_catalogs_upgrade_node_2" {
  source                        = "../sharded_catalogs_dms_upgrade"
  shard                         = 2
  availability_zone             = "..."
  target_db_endpoint           = ...
  source_db_endpoint           = ...
}
```

Once the infrastructure was brought up successfully, we kicked off each of the tasks for the remaining clusters (after successfully getting one shard completed, we decided that we could likely do the rest of the upgrades in parallel â€” or at least have the DMS replication occur in parallel). Our unsharded cluster, which contains a lot less data, was able to complete replication within an hour or two, and was able to be cutover within the same day (again, following the exact same process as above).

Based on our experience with shard 1, the three remaining database shards were going to take a couple of days to complete. After checking back in on the tasks, we realized that something was off. A subset of the tasks for each shard had completed successfully, but a lot of tasks hadnâ€™t completed yet and the CDC latency was getting pretty high. We opened up (yet another) case with AWS to see if they had any insight into this issue. From our side, it looked like the tasks themselves were kind of â€œstuckâ€� and werenâ€™t processing as we expected.

After a bunch of back and forth with AWS while they triaged the issue, we also noticed that one of our source databases was starting to log the following:

```
WARNING:  oldest xmin is far in the past
HINT:  Close open transactions soon to avoid wraparound problems.
       You might also need to commit or roll back old prepared transaction
```

We have unfortunately had issues with hitting wraparound in the past, and luckily have alarms set for when the xmin reaches 1.2B, so we werenâ€™t *too* concerned when this log started to appear. But it did give us pause â€” if the DMS tasks continued to be stuck, and we hadnâ€™t had proper alerting to notify us of an impending wraparound, we could have easily gotten into a very rough state.

Luckily AWS was able to fix the issue on their end (they sadly did not go into detail on *what* exactly happened) and the remaining tasks were able to finish before we came close to a wraparound scenario.

Again â€” this process was highly repetitive. Once the databases had finished replicating and the CDC latency was able to drop down to the expected sub 10 second threshold, we followed the above cutover process for each of the clusters. That (unsurprisingly) went off without a hitch! We were now successfully running the catalog product area on Postgres 14 databases. And finally it was time to pour the champagne into Indyâ€™s grail!

A happy ending