.

Table of Contents

[Overview 3](#_Toc535493219)

[Scope by Release 3](#_Toc535493220)

[Architecture Diagram 4](#_Toc535493221)

[Individual Components 5](#_Toc535493222)

[Assumptions 6](#_Toc535493223)

[Source File S3 Events 6](#_Toc535493224)

[DynamoDB Events Table 7](#_Toc535493225)

[DynamoDB Events Table Sample record 7](#_Toc535493226)

[DynamoDB Lookup Table 8](#_Toc535493227)

[DynamoDB Lookup Table Sample record 9](#_Toc535493228)

[Additional Notes 10](#_Toc535493229)

[Snowflake-specific Components 10](#_Toc535493230)

[Operation Types 11](#_Toc535493231)

[General Assumptions 11](#_Toc535493232)

[Supported / Unsupported Load Types (as of December 2018) 11](#_Toc535493233)

[INSERT 11](#_Toc535493234)

[UPSERT 13](#_Toc535493235)

[DELETE 15](#_Toc535493236)

[REPLACEALL 16](#_Toc535493237)

[Edge Cases & Error Handling 18](#_Toc535493238)

[Error Handling with Python connector 18](#_Toc535493239)

[Archiving 18](#_Toc535493240)

[Operational Reporting 18](#_Toc535493241)

[AWS Monitoring 18](#_Toc535493242)

[DynamoDB 18](#_Toc535493243)

[SQS 19](#_Toc535493244)

[Snowflake Monitoring 20](#_Toc535493245)

[Data Ops Troubleshooting 21](#_Toc535493246)

[No Metadata on S3 file 21](#_Toc535493247)

[File has metadata, but came in later (after file event was processed) 21](#_Toc535493248)

[Invalid Load Type 22](#_Toc535493249)

[New column in Avro file 22](#_Toc535493250)

[Changed data type on existing column in Avro file without changing corresponding data type in target (Snowflake) 22](#_Toc535493251)

[Missing information in Lookup Table 23](#_Toc535493252)

[Snowflake Staging/Target Schema or Staging/Target Table does not exist 23](#_Toc535493253)

[Snowflake Python Connector Error (connection to Snowflake cannot be established) 23](#_Toc535493254)

[Snowflake Operation Takes Too Long and Lambda Dies 24](#_Toc535493255)

[Data Ops Common Tasks 24](#_Toc535493256)

[Verify data 24](#_Toc535493257)

[Cleanup single File or Partition 24](#_Toc535493258)

[Cleanup entire Table 24](#_Toc535493259)

[Onboarding New non-Teradata Source Table (e.g., Facebook) 24](#_Toc535493260)

[Runbook: Catchup Utility 25](#_Toc535493261)

[Overview 25](#_Toc535493262)

[CSV File Setup 25](#_Toc535493263)

[Psuedo Code (CSV Approach) 25](#_Toc535493264)

[Psuedo Code (Direct Approach) 25](#_Toc535493265)

[Usage (CSV Approach) 25](#_Toc535493266)

[Usage (Direct Approach) 25](#_Toc535493267)

[Git Hub Link 26](#_Toc535493268)

[Runbook: Lookup Controller Utility 26](#_Toc535493269)

[Overview 26](#_Toc535493270)

[Psuedo Code 26](#_Toc535493271)

[Usage 26](#_Toc535493272)

[Git Hub Link 26](#_Toc535493273)

[Appendix 27](#_Toc535493274)

[How to pull metadata from S3 file via AWS CLI 27](#_Toc535493275)

[S3 Bucket Configuration for spawning Events 27](#_Toc535493276)

[Explaining S3 SQS Fifo Queues 28](#_Toc535493277)

[Steps for On-boarding New Target (e.g., non-Snowflake) 28](#_Toc535493278)

[Points of Contact by Team 28](#_Toc535493279)

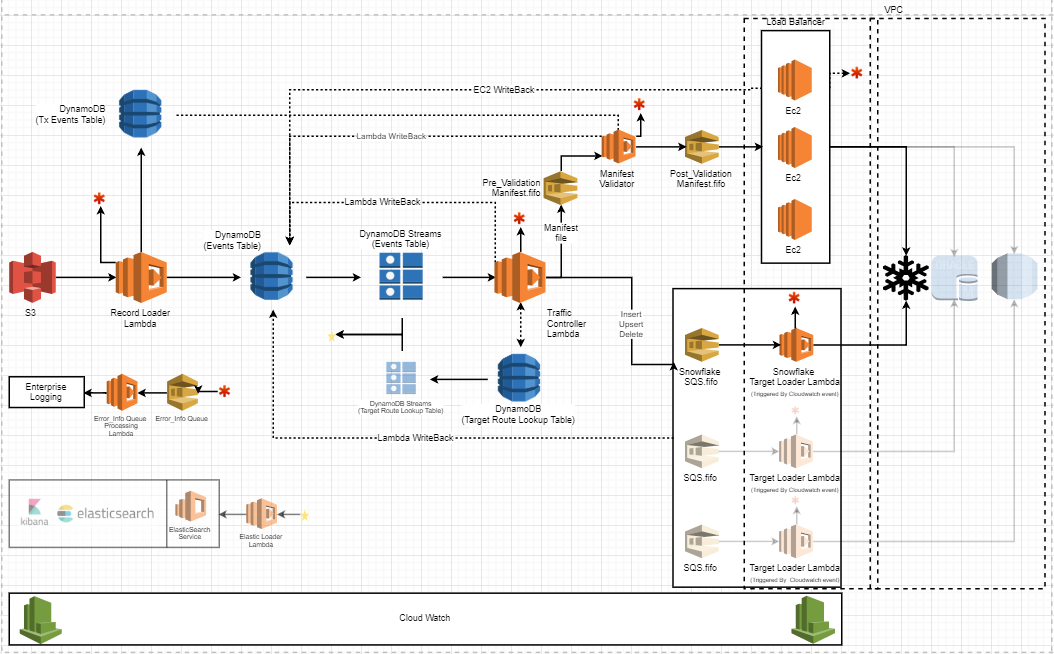
# Overview

The scope of this document is to cover the design and approach to the Universal Data Loader (“UDL”) which imports data from S3 into Snowflake cloud data warehouse (and may be expanded to accommodate other targets beyond Snowflake in the future). In contrast to the Backfill Utility, the UDL loads data using an event-driven pattern initiated by files uploaded to S3 by upstream processes.

# Scope by Release

|  |  |  |
| --- | --- | --- |
| **Release** | **Headline** | **Scope** |
| 1.0 | Package Backfill Utility for data ops team use | * Package backfill scripts into easily runnable utility for Data Ops to use for migrating Insert only tables into Snowflake. Includes documentation * Utility allows users to load data into snowflake by specifying which table they want to migrate. Error handling and status checking instructions available with documentation * Does not include advanced parametrization (including parallelization) |
| 1.5 | Happy Path Ongoing Pattern Design ready to go into Production for **inserts only** | * **Happy Path** Event Driven pattern that can detect S3 files for **inserts** and migrate them into Snowflake only going into Production * Includes detecting file event in S3 and migrating file content into Snowflake, recording success/failure of load in Snowflake for each file * Does not include optimized queuing, retry-ability or alerting |
| 2.0 | Ongoing Pattern ready to go into Production for **inserts only** | * Event Driven pattern that can detect S3 files for **inserts** and migrate them into Snowflake only going into production * Includes detecting new insert files landing in S3, picking up file and queueing for loading into Snowflake, recording success/failure of load in Snowflake for each file * Does not include operational dashboards/reporting, creation of schemas/DDLs in Snowflake programmatically (new table creation handled by DFS teams), error alerting system for support teams * The release is dependent on successful migration of 50 **insert** only tables using the backfill pattern to then enable ongoing insert only pattern for them so that the data is up to date in Snowflake * Backfill pattern can be used by Data Ops team to allow them to migrate more tables available in S3 |
| 3.0 | Ongoing Pattern including **upserts** and **deletes** | * Event Driven pattern that can detect S3 **upserts / deletes** files and migrate them into Snowflake only going into production. * Includes detecting new upserts / deletes files landing in S3, picking up file and queueing for loading into Snowflake, loading into Snowflake, recording success/failure of load in Snowflake for each file. * Includes files that are already loaded into S3 before this milestone that are upserts/deletes, requires triggering those files semi-manually and in order to ensure those files are migrated first and in order before enabling ongoing pattern for future files * Does not include operational dashboards/reporting, creation of schemas/DDLs in Snowflake programmatically (new table creation handled by DFS teams), error alerting system for support teams   \* Includes Ongoing Upserts/Deletes Data ingestion for individual Avro Files, but **not** Manifest files |

# Architecture Diagram



## Individual Components

|  |  |
| --- | --- |
| **Component** | **Purpose** |
| **S3** | * Location where the files land. * S3 event is configured to trigger a call to the S3 Dynamo Loader Lambda function for PUTs and COPYs. |
| **S3 Dynamo Loader Lambda** | * Lambda function fires from the S3 event (either PUT or COPY) * Calls load function   + If event is blank, then raise error so lambda fails   + Else, proceed     - Extract contents (metadata and tags) from event message     - Try to get metadata from S3 file     - If no metadata found       * Raise WARNING and ignore message     - Else       * Generate keys for event passed       * Write to DynamoDB Events and Tx Events tables |
| **DynamoDB**  (Tx Events Table) | * DynamoDB table holds the file information of all the files that land in S3 as part of a transaction (specific to Manifest file loads). |
| **DynamoDB**  (Events Table) | * DynamoDB table holds the information of all the files that land in S3. |
| **DynamoDB Streams**  (Events Table) | * DynamoDB Stream maintains the order in which records were inserted into the DynamoDB Events table. * Passes those messages to the Traffic Controller Lambda Function. |
| **Traffic Controller Lambda** | * Triggered by DynanoDB Streams Event. * Lambda function responsible for reading messages of the stream and routing them to the appropriate path for the message/record/file to be processed. * Lambda takes values from the DynamoDB Stream and does the following.  1. Checks for valid Load Type (e.g., **Insert, Upsert, Delete**) for file event. Note: type **manifest** is ignored for release 3.0. 2. Looks up information for file in lookup DynamoDB table. 3. Check if targets in DynamoDB lookup table are green 4. If no targets are green, then no work to be done and move on to next event 5. Else (targets exist in green status)    * Reads Avro schema from file and generates schema hash value    * Compares schema hash retrieved from file to schema hash stored in the lookup DynamoDB table. If schema hash values match, then update the record in the DynamoDB Events Table setting schema\_validated = ‘Y’. Else, update Events table schema\_validated = ‘N’, log INFO and continue. *Note: for future use, an environment variable will be set to cause schema\_validation = ‘N’ to raise error and exit (hard stop). Currently, the default is to continue.*    * For each target with status = green      + From the data fetched in the Lookup Table, route the message to the appropriate FIFO queue and trigger the appropriate lambda      + For FUTURE USE ONLY, if the file is a manifest file then route it to the manifest.fifo queue and append the targets that file needs to be loaded into the message |
| **DynamoDB**  (Lookup Table) | * DynamoDB table used by the Traffic Controller Lambda function to instruct on which queues to route the message to. |
| **DynamoDB Streams**  (Lookup Table) | * DynamoDB Stream maintains the order in which records were inserted into the DynamoDB Lookup table. * Passes those messages to Lambda to be sent to ElasticSearch. |
| **Pre\_Validation\_Manifest.fifo** | * Fifo queue to hold the message containing the information about the manifest file before being validated. Traffic controller will send those messages here based on the type of file received. |
| **Post\_Validation\_Manifest.fifo** | * Fifo queue to hold the validated manifest file messages. Messages will be landed here by the Manifest Validator Lambda. |
| **EC2**  (Behind Load Balancer) | * Used to process Manifest files * Always running, but triggered when messages land in the post\_validation\_manifest.fifo queue. * Holds the python script which reads messages from the post\_validation\_manifest.fifo queue to get the manifest file from S3. * Parses manifest file to pull files and target systems referenced therein. * Based on the instructions in the message, calls the appropriate classes to load files from the manifest into the target locations in the order defined in the manifest file. |
| **Snowflake SQS.fifo** | * Fifo queue to hold the messages to be processed synchronously. * Only for insert/update/delete operation types. |
| **Snowflake Target Loader Lambda**  (temporarily Triggered By Cloudwatch, lambda will call itself when this blocker is lifted) | * Lambda triggered by Traffic Controller Lambda to call the Snowflake loader to process messages from the queue. * Snowflake Target Loader lambda then calls itself to process any additional messages in the queue until all messages are processed. * Lambda function takes the message off the fifo queue and loads the file into the target system (currently Snowflake) in the sequence specified. |
| **SQS error\_info Queue** | * SQS queue that accepts messages (e.g., information, warnings, errors). * Lambdas either send messages directly to it, or the asynchronous SQS queues sends messages there directly after a certain number of retries. |
| **error\_info Queue Processing Lambda** | * Triggered by messages in SQS error\_info queue. * Lambda reads messages from the error\_info queue and sends to Enterprise Logging |
| **Elastic Loader Lambda** | * Triggered by DyanmoDB Streams. * Lambda sends to Elastic Search |
| **Elastic Search Service** | * Service holds copies of the DynamoDB tables for operational reporting in realtime. |

## Assumptions

1. Modifications to the future state architecture may require updates to this document.
2. Each partition will have only one operation type (e.g. Insert, Upsert, Delete). Stated another way, there would never be files with different load types in the same partition.
3. Data is delivered to S3 by upstream data providers (e.g., Ab Initio) in the order it is intended to be loaded into Snowflake. Based on the eventTime of the record in Event table, the files are processed in sequence from oldest to newest time.
4. Teradata tables load into Snowflake retaining the same Schema (e.g., T\_SOT\_REFDATA) and Table name (e.g. INDST\_CLSS) as in Teradata. Tables from non-Teradata sources will never be loaded into designated Teradata schemas. Note that this is not enforced programmatically, however could violate data governance / best practices.
5. Snowflake targets are per database, so if more than one Snowflake database is targeted then a new target will need to be created (e.g., SQS queue, lambda, etc.).
6. Multiple messages are sent to the Traffic Controller Lambda at once and messages are processed in sequence.
7. There is no Load Type of "Replace"… only “ReplaceAll”.
8. ReplaceAll Load Type will always be delivered to S3 as part of a Manifest file. You would never process a single file of type ReplaceAll (outside of a manifest file).

## Source File S3 Events

* + Any PUT or COPY event is accepted
  + If file has metadata at that point in time, the file is attempted to be processed, otherwise the file is ignored (metadata may be written later, which would create a new COPY event)
  + PUT / COPY event time in S3 dictates the order in which the files should be loaded
  + INSERT operations may be processed in parallel
  + UPSERT and DELETE operations must maintain sequence

## DynamoDB Events Table

|  |  |
| --- | --- |
| **Field** | **Purpose** |
| file\_uuid | * Partition key for the dynamoDB table, the folder path up to the partition level of the S3 file (extracted from the S3 event message). |
| eventTime | * Sort key for the dynamoDB table, the time the file landed in S3 (extracted from the S3 event message). |
| bucketName | * Name of the bucket (extracted from the s3 event message). |
| dataAssetName | * Name of the file (extracted from the S3 event message). |
| event | * Actual S3 event. |
| eventSource | * Where the event originated from (extracted from the S3 event message, the configurationId from the actual S3 event). |
| lastUpdated | * Column storing time (as a string) for anytime something is updated for this record by any of the lambda functions. |
| metadata | * Metadata of the file. * Fetched by doing a head object on the S3 file. |
| objectKey | * S3 key to the file (extracted from the S3 event message). |
| status | * Status of the file, whether it processed successfully (Success), is ready to be processed (Ready), is currently in process (Processing) in which case no other lambda can acquire the lock, or failed to process (Failed). |
| tableName | * Name of the table that the file belongs to (extracted from the key fetched from the S3 event). |
| tags | * Tags of the file. Fetched by doing a head object on the S3 file. |
| uuid | * Unique identifier generated by the DynamoDB S3 loader lambda. |

### DynamoDB Events Table Sample record

{

"bucketName": "dfs-analytics-dlt-use1-ent-etl",

"dataAssetName": "Edcims\_acaps\_pn\_notes.20181218.load.insert.1545283133.part\_006.avro",

"event": {

"Records": [

{

"awsRegion": "us-east-1",

"eventName": "ObjectCreated:Put",

"eventSource": "aws:s3",

"eventTime": "2018-12-20T10:19:08.707Z",

"eventVersion": "2.1",

"requestParameters": {

"sourceIPAddress": "10.52.31.47"

},

"responseElements": {

"x-amz-id-2": "yLJf3PiMGq3yBI2Sxu7S6U/iuEWU8kB5l0/xZBPImU2TeJzitnTtwNzzuI5ESuDHs5hXvCAJ/WA=",

"x-amz-request-id": "676DDDDBD1A84CF4"

},

"s3": {

"bucket": {

"arn": "arn:aws:s3:::dfs-analytics-dlt-use1-ent-etl",

"name": "dfs-analytics-dlt-use1-ent-etl",

"ownerIdentity": {

"principalId": "A2HREGGS08DHHO"

}

},

"configurationId": "s3-lambda-acaps\_pn\_notes",

"object": {

"eTag": "5e332365d969788624b9e00cb2c0b5d9",

"key": "dirbnk/card/acaps\_pn\_notes/src\_dt\_pc%3D20181218T000000Z/Edcims\_acaps\_pn\_notes.20181218.load.insert.1545283133.part\_006.avro",

"sequencer": "005C1B6C9C94F7542D",

"size": 362451,

"versionId": "8TL\_ZgWzl\_k5ZavlCSRF3EDQeXu.6spi"

},

"s3SchemaVersion": "1.0"

},

"userIdentity": {

"principalId": "AWS:AIDAJPQ76ZPXSWNCLQMSQ"

}

}

]

},

"eventSource": "MetaParser",

"eventTime": "2018-12-20T10:19:08.707Z",

"file\_uuid": "dirbnk/card/acaps\_pn\_notes/src\_dt\_pc=20181218T000000Z",

"lastUpdated": "2018-12-20T10:19:08.707Z",

"metadata": {

"load-type": "Insert",

"record-count": "37443",

"source": "Edcims\_acaps\_pn\_notes.20181218.load.gz",

"source-type": "File"

},

"objectKey": "dirbnk/card/acaps\_pn\_notes/src\_dt\_pc=20181218T000000Z/Edcims\_acaps\_pn\_notes.20181218.load.insert.1545283133.part\_006.avro",

"status": "Ready",

"tableName": "acaps\_pn\_notes",

"tags": [

{

"Key": "org-code",

"Value": "ACQT"

}

],

"uuid": "c6da0752-b9fa-4524-a4c4-2f9a2f6cfe99"

}

## DynamoDB Lookup Table

|  |  |
| --- | --- |
| **Field** | **Purpose** |
| data\_asset\_name | * Primary / hash key. |
| version | * Sort key. |
| avro\_file\_schema\_hash | * Table schema used by the traffic controller lambda function to compare the values to the schema hash value from the file. |
| target\_queue\_arns.**non\_manifest**.{target}.queue\_arn | * Used by the traffic controller lambda function to see what queue it should send the message to for non manifest files. |
| target\_queue\_arns.**non\_manifest**.{target}.lambda\_arn | * Used by the traffic controller lambda function to see what lambda function it needs to call after adding message to the queue. |
| target\_queue\_arns.**non\_manifest**.{target}.group\_status | * Used by the target loader lambda function to see if it should process the message for that specific group belonging to the non manifest file type. |
| targets.**non\_manifest**.{target}.status | * Used by is used by traffic controller lambda function to see if it should route the message to that specific non manifest file queue. |
| targets.**non\_manifest**.{target}.subscribers.error | * Used by the traffic controller or the target loader to notify who to alert in case of issues. |
| targets.**non\_manifest**.{target}.subscribers.info | * Used by the traffic controller or the target loader to send out messages to interested parties. |
| target\_queue\_arns.manifest.{target}.queue\_arn | * Used by the traffic controller lambda function to see what queue it should send the message to for manifest files. |
| target\_queue\_arns.manifest.{target}.group\_status | * Used by the target loader lambda function to see if it should process the message for that specific group belonging to the manifest file type. |
| targets.manifest.{target}.status | * Used by traffic controller lambda function to see if it should route the message to that specific manifest file queue. |
| targets.manifest.{target}.subscribers.error | * Used by the traffic controller or the target loader to notify who to alert in case of issues. |
| targets.manifest.{target}.subscribers.info | * Used by the traffic controller or the target loader to send out messages to interested parties. |

### DynamoDB Lookup Table Sample record

{

  "target\_queue\_arns": {

    "non\_manifest": {

      "mysql": {

        "lambda\_arn": "full\_arn\_of\_the\_mysql\_lambda",

        "queue\_arn": "full\_arn\_of\_the\_mysql\_fifo\_queue",

        "group\_status": "red"

      },

      "redshift": {

        "lambda\_arn": "full\_arn\_of\_the\_mysql\_lambda",

        "queue\_arn": "full\_arn\_of\_the\_redshift\_fifo\_queue",

        "group\_status": "red"

      },

      "snowflake": {

        "lambda\_arn": "full\_arn\_of\_the\_mysql\_lambda",

        "queue\_arn": "full\_arn\_of\_the\_snowflake\_fifo\_queue",

        "group\_status": "green"

      }

    },

    "manifest": {

      "mysql": {

        "queue\_arn": "full\_arn\_of\_the\_fifo\_manifest\_queue",

        "group\_status": "red"

      },

      "redshift": {

        "queue\_arn": "full\_arn\_of\_the\_fifo\_manifest\_queue",

        "group\_status": "red"

      },

      "snowflake": {

        "queue\_arn": "full\_arn\_of\_the\_fifo\_manifest\_queue",

        "group\_status": "red"

      }

    }

  },

  "targets": {

    "non\_manifest": {

      "mysql": { "status": "red", "subscribers": { "error": "org\_id\_of\_people\_who\_want\_error\_notifications\_for\_this\_target", "info": "org\_id\_of\_people\_who\_want\_info\_notifications\_for\_this\_target" } },

      "redshift": { "status": "red", "subscribers": { "error": "org\_id\_of\_people\_who\_want\_error\_notifications\_for\_this\_target", "info": "org\_id\_of\_people\_who\_want\_info\_notifications\_for\_this\_target" } },

      "snowflake": { "status": "green", "subscribers": { "error": "org\_id\_of\_people\_who\_want\_error\_notifications\_for\_this\_target", "info": "org\_id\_of\_people\_who\_want\_info\_notifications\_for\_this\_target" } }

    },

    "manifest": {

      "mysql": { "status": "red", "subscribers": { "error": "org\_id\_of\_people\_who\_want\_error\_notifications\_for\_this\_target", "info": "org\_id\_of\_people\_who\_want\_info\_notifications\_for\_this\_target" } },

      "redshift": { "status": "red", "subscribers": { "error": "org\_id\_of\_people\_who\_want\_error\_notifications\_for\_this\_target", "info": "org\_id\_of\_people\_who\_want\_info\_notifications\_for\_this\_target" } },

      "snowflake": { "status": "green", "subscribers": { "error": "org\_id\_of\_people\_who\_want\_error\_notifications\_for\_this\_target", "info": "org\_id\_of\_people\_who\_want\_info\_notifications\_for\_this\_target" } }

    }

  },

  "avro\_file\_schema\_hash": "alsjfis435q3cfwt5y",

  "data\_asset\_name": "line\_of\_business/product/sub\_product/acaps\_pn",

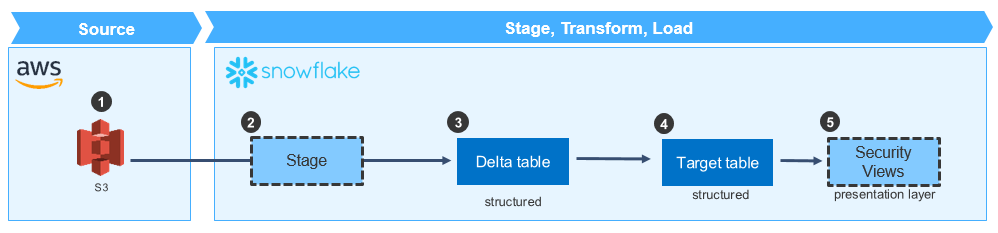
  "version": 1

}

## Additional Notes

* Currently, the Target Loader Lambda cannot call itself, so the temporary workaround is to have CloudWatch invoke the Lambda every 5 minutes (microbatching)
* Tentatively, we will set the Traffic Controller Lambda to handle a maximum of **1,000** messages at the same time. *Note: setting must be revisited to adjust as needed after testing with real data (and normal volumes).*

# Snowflake-specific Components



|  |  |
| --- | --- |
| **Component** | **Description** |
| 1. S3 | * Source data stored in S3 buckets in Avro 1.8 format * S3 folder structure (prefixes) organized by:   + Line of Business, Product, Sub-Product, Table, Partition Date * Avro files land in the Partition Date directory, split into multiple parts with similar filenames and a sequence number |
| 1. Stage | * Stage objects in Snowflake contain references to source location in S3, specifying:   + S3 bucket name   + Prefix (sub-folder path)   + IAM Role, if applicable   + Encryption (e.g., client-side, server-side KMS), if applicable   + File Format (e.g., Avro)   + Regex Pattern to identify multiple files * Data is NOT persisted in the Stage. It serves only to provide information to the COPY command about the location of the source data and expected format * A single external Stage object in Snowflake points to the S3 **bucket dfs-analytics-dlt-use1-ent-etl** |
| 1. Delta table | * For all operations (Insert, Upsert, Delete, Replace), data is bulk-loaded to the Delta table from the S3 source via the COPY command * Both Delta and Target tables in Snowflake store data in a **structured** format * Data from the S3 file is parsed and data types are transformed from Avro Logical Data Types to the data types expected by the Delta table * Naming convention of the Delta table is *<target\_table\_name>*\_STG. For example, ACAPS\_10 would have Delta table ACAPS\_10\_STG |
| 1. Target table | * Data from Delta Table is synchronized with the Target Table using SQL statements |
| 1. Security views | * Security views are built on the Target table to restrict access to columns based on the sensitivity of the data (e.g., Critical Risk Data (CRD), High Risk Data (HRD), etc.) |

# Operation Types

Regardless of Operation Type, data first lands in Snowflake Delta tables from S3 via the COPY statement. Once landed in the Delta table, there are four (4) potential operation types supported by the Universal Data Loader ongoing load pattern to manipulate data in the Target table:

1. **Insert**: data from Delta table is sorted by the Target table’s Cluster Keys (if applicable), then inserted into the Target table.
2. **Upsert**: data from Delta table is compared to data in Target table (based on defined join key). If a match is found, the record from the Delta table updates the record in the Target table. If a match is not found, the record from the Delta table inserts into the Target table.
3. **Delete**: data from Delta table is compared to data in Target table (based on defined join key). If a match is found, the record from the Delta table deletes the record in the Target table.
4. **Replace:** All records in Target table are deleted and all records from the Delta table are then inserted into the Target table.

## General Assumptions

* Regardless of Operation Type, each statement will be wrapped in a transaction (BEGIN, then either ROLLBACK on error, or COMMIT if no errors)
* If no metadata exists on a source file in S3, then the file will not be processed
* Query Id will be captured by Python and stored in Cloudwatch for traceability

## Supported / Unsupported Load Types (as of December 2018)

* + Supported
    - Insert
    - Upsert
    - Delete – *Note: functionality has been built but cannot be tested as we are dependent on sample file of Load Type DELETE being provided Ab Initio team*
  + Unsupported
    - ReplaceAll – to be delivered with the Manifest solution

## INSERT

* Specified in the S3 file metadata (x-amz-meta-load-type=’Insert’)
* Data from Delta table is sorted by the Target table’s Cluster Keys (if applicable), then inserted into the Target table
* Python script will dynamically generate and execute the **INSERT INTO SELECT** statement to perform the operation

|  |  |
| --- | --- |
| **Variable** | **Logic to populate** |
| Delta Table Schema Name | * Sourced from Avro file * Derived from file prefix, replacing forward slash with underscore. * Prepending “T\_SOT” * Appending “\_STG” * For example, /dirbnk/card becomes T\_SOT\_DIRBNK\_CARD\_STG |
| Delta Table Name | * Sourced from Avro file * Derived as <Table Name from Avro file> + “\_STG” * For example, Table Name ACAPS\_30 means Delta Table Name is ACAPS\_30\_STG |
| Target Table Schema Name | * Sourced from Avro file * Derived from file prefix, replacing forward slash with underscore. * Prepending “T\_SOT” * For example, /dirbnk/card becomes T\_SOT\_DIRBNK\_CARD |
| Target Table Name | * Sourced from Avro file * Derived as <Table Name from Avro file> |
| Column Names | * Sourced from Avro file |
| Order By | * Sourced from Cluster Keys defined for the table * Lookup into INFORMATION\_SCHEMA to see Cluster Key information and pass as parameter to ORDER BY clause of the INSERT statement |

* For reference, SQL below used to prove out the desired behavior. Statement in BOLD performs the operation. Statements before and after are needed to setup the test and confirm results are as-expected.

--INSERT Scenarios

create or replace table tbl\_schma\_stg.insert\_target\_stg (id int, state\_abrv varchar(2), state\_dscr varchar(30));

create or replace table tbl\_schma.insert\_target (id int, state\_abrv varchar(2), state\_dscr varchar(30));

-- add cluster keys for sorting

alter table tbl\_schma.insert\_target cluster by (id, state\_abrv);

--Data inserted in insert\_target table

insert into tbl\_schma.insert\_target values

(1,'IL','ILLINOIS'),

(2,'NY','NEY YORK'),

(3,'TX','TEXAS'),

(9,'MI','MICHIGAN');

--Data inserted in insert\_target\_stg table (in practice, this will be bulk-loaded via COPY command)

insert into tbl\_schma\_stg.insert\_target\_stg values

(11,'CA','CALIFORNIA'),

(17,'CO','COLORADO');

--Confirm data in target and delta tables as expected

select \* from tbl\_schma.insert\_target;

select \* from tbl\_schma\_stg.insert\_target\_stg;

-- Determine cluster keys by looking up in Information Schema

select replace(replace(clustering\_key, 'LINEAR(', ''), ')', '') as order\_by from information\_schema.tables where table\_name = 'insert\_target';

--Insert records from delta table into target table

**begin name insert\_trnsctn;**

**insert into tbl\_schma.insert\_target**

**select \* from tbl\_schma\_stg.insert\_target\_stg order by id, state\_abrv;**

**--if no error**

**commit;**

**--if error**

**rollback;**

--Confirm data in delta table inserted into target, as expected

select \* from tbl\_schma insert\_target;

select \* from tbl\_schma\_stg insert\_target\_stg;

## UPSERT

* Specified in the S3 file metadata (x-amz-meta-load-type=’Upsert’)
* Data from Delta table is compared to data in Target table (based on defined join key),
  + If a match is found, the record from the Delta table updates the record in the Target table
  + If a match is NOT found, the record from the Delta table inserts into the Target table
* Upsert Key will be sourced from the S3 metadata (x-amz-meta-upsert-key field)
* All fields will be sent from the source to the delta table, and all fields will be updated from the values in the delta table.
* Python script will dynamically generate and execute the **MERGE** statement to perform the operation

|  |  |
| --- | --- |
| **Variable** | **Logic to populate** |
| Delta Table Schema Name | * Sourced from Avro file * Derived from file prefix, replacing forward slash with underscore * Prepending “T\_SOT” * Appending “\_STG” * For example, /dirbnk/card becomes T\_SOT\_DIRBNK\_CARD\_STG |
| Delta Table Name | * Sourced from Avro file * Derived as <Table Name from Avro file> + “\_STG” * For example, Table Name ACAPS\_30 means Delta Table Name is ACAPS\_30\_STG |
| Target Table Schema Name | * Sourced from Avro file * Derived from file prefix, replacing forward slash with underscore * Prepending “T\_SOT” * For example, /dirbnk/card becomes T\_SOT\_DIRBNK\_CARD |
| Target Table Name | * Sourced from Avro file * Derived as <Table Name from Avro file> |
| Column Names | * Sourced from Avro file |
| Order By | * Sourced from Cluster Keys defined for the table * Lookup into INFORMATION\_SCHEMA to see Cluster Key information and pass as parameter to ORDER BY clause of the INSERT statement |
| Match on | * Sourced from S3 file metadata Upsert Key (x-amz-meta-upsert-key) |

* For reference, SQL below used to prove out the desired behavior. Statements in BOLD perform the operations. Statements before and after are needed to setup the test and confirm results are as-expected.

--UPSERT Scenarios:  
create or replace table tbl\_schma.merge\_target (id int, state\_abrv varchar(2), state\_dscr varchar(30));  
create or replace table tbl\_schma\_stg merge\_target\_stg (id int, state\_abrv varchar(2), state\_dscr varchar(30));

--Data loaded in random order into delta table (in practice, this will be bulk-loaded via COPY command)

insert into tbl\_schma\_stg merge\_target\_stg values

(1,'IL','ILLINOIS'),  
(3,'TX','TEXAS'),  
(2,'NY','NEY YORK'),  
(9,'MI','MICHIGAN');

select \* from tbl\_schma\_stg merge\_target\_stg;  
select \* from tbl\_schma.merge\_target;

--Insert into merge\_target where no matches in merge\_target\_stg

**begin name upsert\_trnsctn;**

**merge into tbl\_schma.merge\_target**

**using (select id, state\_abrv, state\_dscr from tbl\_schma\_stg merge\_target\_stg order by id) tbl\_schma\_stg.merge\_target\_stg  
on tbl\_schma.merge\_target.id = tbl\_schma\_stg.merge\_target\_stg.id**

**when not MATCHED then insert (id, state\_abrv, state\_dscr) values**

**(tbl\_schma\_stg.merge\_target\_stg.id, tbl\_schma\_stg.merge\_target\_stg.state\_abrv, tbl\_schma\_stg.merge\_target\_stg.state\_dscr)  
when MATCHED then update set**

**tbl\_schma.merge\_target.id = tbl\_schma\_stg.merge\_target\_stg.id,  
tbl\_schma.merge\_target.state\_abrv = tbl\_schma\_stg.merge\_target\_stg.state\_abrv,   
tbl\_schma.merge\_target.state\_dscr = tbl\_schma\_stg.merge\_target\_stg.state\_dscr;**

**--if no error**

**commit;**

**--if error**

**rollback;**

--Table merge\_target is now sorted, merge\_target\_stg remains unsorted  
select \* from tbl\_schma.merge\_target;  
select \* from tbl\_schma\_stg.merge\_target\_stg;

--Update records in merge\_target\_stg to test update functionality. Modify state abbreviation and state name for id=3  
truncate table tbl\_schma\_stg.merge\_target\_stg;  
insert into tbl\_schma\_stg.merge\_target\_stg values

(7, 'AZ', 'ARIZONA'),  
(3, 'WY', 'WYOMING');

--Confirm merge\_target\_stg was modified as expected  
select \* from tbl\_schma\_stg.merge\_target\_stg;

--Upsert merge\_target with new, updated records in merge\_target\_stg

**begin name upsert\_trnsctn;**

**merge into tbl\_schma.merge\_target**

**using (select id, state\_abrv, state\_dscr from tbl\_schma\_stg.merge\_target\_stg order by id) tbl\_schma\_stg.merge\_target\_stg  
on tbl\_schma.merge\_target.id = tbl\_schma\_stg.merge\_target\_stg.id**

**when not MATCHED then**

**insert (id, state\_abrv, state\_dscr) values (tbl\_schma\_stg.merge\_target\_stg.id, tbl\_schma\_stg.merge\_target\_stg.state\_abrv, tbl\_schma\_stg.merge\_target\_stg.state\_dscr)  
when MATCHED then**

**update set**

**tbl\_schma.merge\_target.id = tbl\_schma\_stg.merge\_target\_stg.id,**

**tbl\_schma.merge\_target.state\_abrv = tbl\_schma\_stg.merge\_target\_stg.state\_abrv,**

**tbl\_schma.merge\_target.state\_dscr = tbl\_schma\_stg.merge\_target\_stg.state\_dscr;**

**--if no error**

**commit;**

**--if error**

**rollback;**

--Table merge\_target is now updated with new/modified records from merge\_target\_stg  
select \* from tbl\_schma.merge\_target;  
select \* from tbl\_schma\_stg.merge\_target\_stg;

## DELETE

* Specified in the S3 file metadata (x-amz-meta-load-type=’Delete’)
* Data from Delta table is compared to data in Target table (based on upsert key from S3 file metadata).
  + If a match is found, the record from the Delta table deletes the record from the Target table.
  + If a match is NOT found, the record is ignored.
    - *Note: future enhancement must be designed to capture a Warning or Error when no match is found on a given record for which a DELETE is attempted.*
* All fields will be sent from the source to the delta table
* Python script will dynamically generate and execute the **MERGE** statement to perform the operation

|  |  |
| --- | --- |
| **Variable** | **Logic to populate** |
| Delta Table Schema Name | * Sourced from Avro file * Derived from file prefix, replacing forward slash with underscore * Prepending “T\_SOT” * Appending “\_STG” * For example, /dirbnk/card becomes T\_SOT\_DIRBNK\_CARD\_STG |
| Delta Table Name | * Sourced from Avro file * Derived as <Table Name from Avro file> + “\_STG” * For example, Table Name ACAPS\_30 means Delta Table Name is ACAPS\_30\_STG |
| Target Table Schema Name | * Sourced from Avro file * Derived from file prefix, replacing forward slash with underscore * Prepending “T\_SOT” * For example, /dirbnk/card becomes T\_SOT\_DIRBNK\_CARD |
| Target Table Name | * Sourced from Avro file * Derived as <Table Name from Avro file> |
| Column Names | * Sourced from Avro file |
| Match on | * Sourced from S3 file metadata Upsert Key (x-amz-meta-upsert-key) |

* For reference, SQL below used to prove out the desired behavior. Statement in BOLD performs the operation. Statements before and after are needed to setup the test and confirm results are as-expected.

--DELETE Scenarios  
create or replace table tbl\_schma\_stg.merge\_delete\_target\_stg (id int, state\_abrv varchar(2), state\_dscr varchar(30));  
create or replace table tbl\_schma.merge\_delete\_target (id int, state\_abrv varchar(2), state\_dscr varchar(30));

--Data inserted in merge\_delete\_target table   
insert into tbl\_schma.merge\_delete\_target values

(1,'IL','ILLINOIS'),  
(2,'NY','NEY YORK'),  
(3,'TX','TEXAS'),   
(9,'MI','MICHIGAN');

--Data inserted in merge\_delete\_target\_stg table (in practice, this will be bulk-loaded via COPY command)  
insert into tbl\_schma\_stg.merge\_delete\_target\_stg values

(2,'NY','NEY YORK'),  
(3,'TX','TEXAS');

--Confirm data in target and delta tables as expected  
select \* from tbl\_schma.merge\_delete\_target;  
select \* from tbl\_schma\_stg.merge\_delete\_target\_stg;

--Delete records from target table where found in delta table

**begin name delete\_trnsctn;**

**merge into tbl\_schma.merge\_delete\_target**

**using tbl\_schma\_stg.merge\_delete\_target\_stg   
on tbl\_schma.merge\_delete\_target.id = tbl\_schma\_stg.merge\_delete\_target\_stg.id**

**when MATCHED then delete;**

**--if no error**

**commit;**

**--if error**

**rollback;**

--Confirm data deleted from target, as expected  
select \* from tbl\_schma.merge\_delete\_target;  
select \* from tbl\_schma\_stg.merge\_delete\_target\_stg;

## REPLACEALL

* Specified in the S3 file metadata (x-amz-meta-load-type=’ReplaceAll’)
* All records in Target table are deleted and all records from the Delta table are then inserted into the Target table
* Python script will dynamically generate and execute the **INSERT OVERWRITE INTO SELECT** statement to perform the operation

|  |  |
| --- | --- |
| **Variable** | **Logic to populate** |
| Delta Table Schema Name | * Sourced from Avro file * Derived from file prefix, replacing forward slash with underscore * Prepending “T\_SOT” * Appending “\_STG” * For example, /dirbnk/card becomes T\_SOT\_DIRBNK\_CARD\_STG |
| Delta Table Name | * Sourced from Avro file * Derived as <Table Name from Avro file> + “\_STG” * For example, Table Name ACAPS\_30 means Delta Table Name is ACAPS\_30\_STG |
| Target Table Schema Name | * Sourced from Avro file * Derived from file prefix, replacing forward slash with underscore * Prepending “T\_SOT” * For example, /dirbnk/card becomes T\_SOT\_DIRBNK\_CARD |
| Target Table Name | * Sourced from Avro file * Derived as <Table Name from Avro file> |
| Column Names | * Sourced from Avro file |
| Order By | * Sourced from Cluster Keys defined for the table * Lookup into INFORMATION\_SCHEMA to see Cluster Key information and pass as parameter to ORDER BY clause of the INSERT statement |

* For reference, SQL below used to prove out the desired behavior. Statement in BOLD performs the operation. Statements before and after are needed to setup the test and confirm results are as-expected.

--REPLACEALL Scenarios  
create or replace table tbl\_schma\_stg.replace\_target\_stg (id int, state\_abrv varchar(2), state\_dscr varchar(30));  
create or replace table tbl\_schma.replace\_target (id int, state\_abrv varchar(2), state\_dscr varchar(30));

-- add cluster keys for sorting

alter table tbl\_schma.replace\_target cluster by (id, state\_abrv);

--Data inserted in replace\_target table   
insert into tbl\_schma.replace\_target values

(1,'IL','ILLINOIS'),  
(2,'NY','NEY YORK'),  
(3,'TX','TEXAS'),   
(9,'MI','MICHIGAN');

--Data inserted in replace\_target\_stg table (in practice, this will be bulk-loaded via COPY command)  
insert into tbl\_schma\_stg.replace\_target\_stg values

(11,'CA','CALIFORNIA'),  
(17,'CO','COLORADO');

--Confirm data in target and delta tables as expected  
select \* from tbl\_schma.replace\_target;  
select \* from tbl\_schma\_stg.replace\_target\_stg;

-- Determine cluster keys by looking up in Information Schema

select replace(replace(clustering\_key, 'LINEAR(', ''), ')', '') as order\_by from information\_schema.tables where table\_name = 'replace\_target';

--Replace records in target table with records in delta table  
**begin name replace\_trnsctn;**

**insert overwrite into tbl\_schma.replace\_target**

**select \* from tbl\_schma\_stg.replace\_target\_stg order by id, state\_abrv;**

**--if no error**

**commit;**

**--if error**

**rollback;**

--Confirm data in target replaced by delta, as expected  
select \* from tbl\_schma.replace\_target;  
select \* from tbl\_schma\_stg.replace\_target\_stg;

# Edge Cases & Error Handling

Potential Points of Failure

1. Errors executing COPY statement to pull data from S3 into Delta table
2. Errors executing SQL statements (INSERT, MERGE, etc) to reconcile data from Delta table with Target table must be captured and logged via Python

## Error Handling with Python connector

1. See documentation: [https://docs.snowflake.net/manuals/user-guide/python-connector-example.html#handling-errors](https://docs.snowflake.net/manuals/user-guide/python-connector-example.html%23handling-errors)
2. Try / Except / Finally code blocks
3. On error, the sqlstate attribute contains success / error code
4. Close connection in finally block

# Archiving

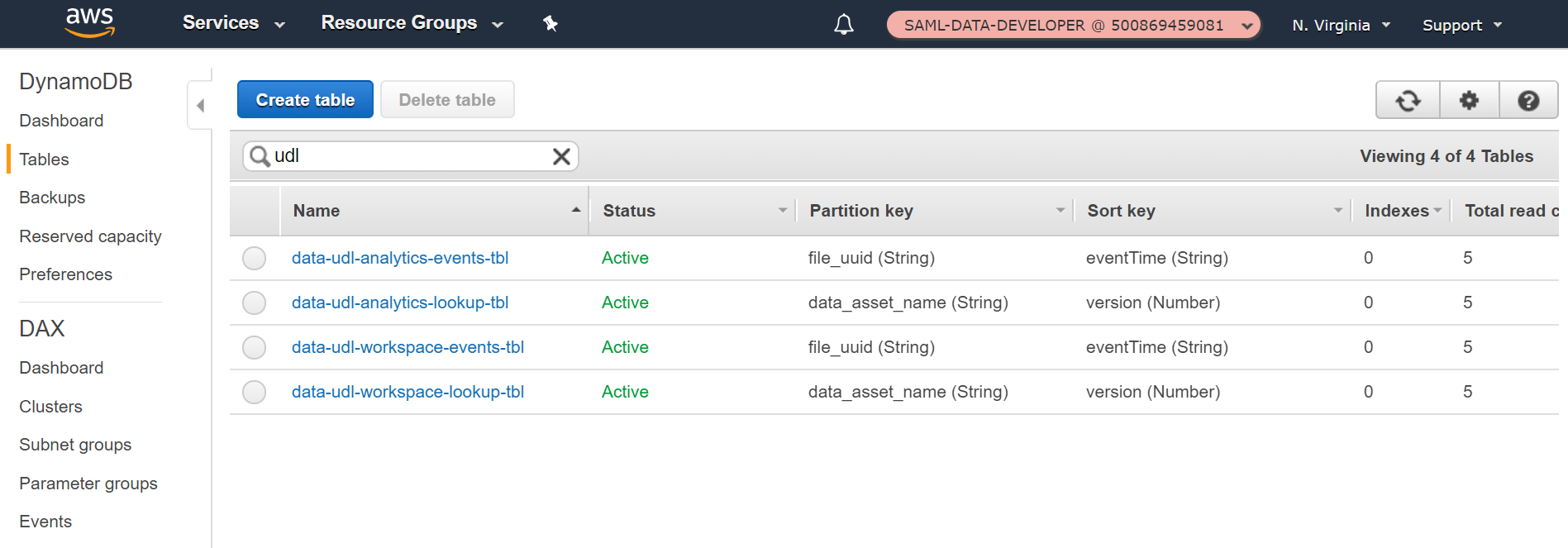
1. No purging of data is anticipated from the "ETL" S3 bucket (**dfs-analytics-dlt-use1-ent-etl**) for at least one year from the time of implementation (December 2018). Data files (Avro) will remain in bucket throughout the backfill, catchup, and ongoing loads.

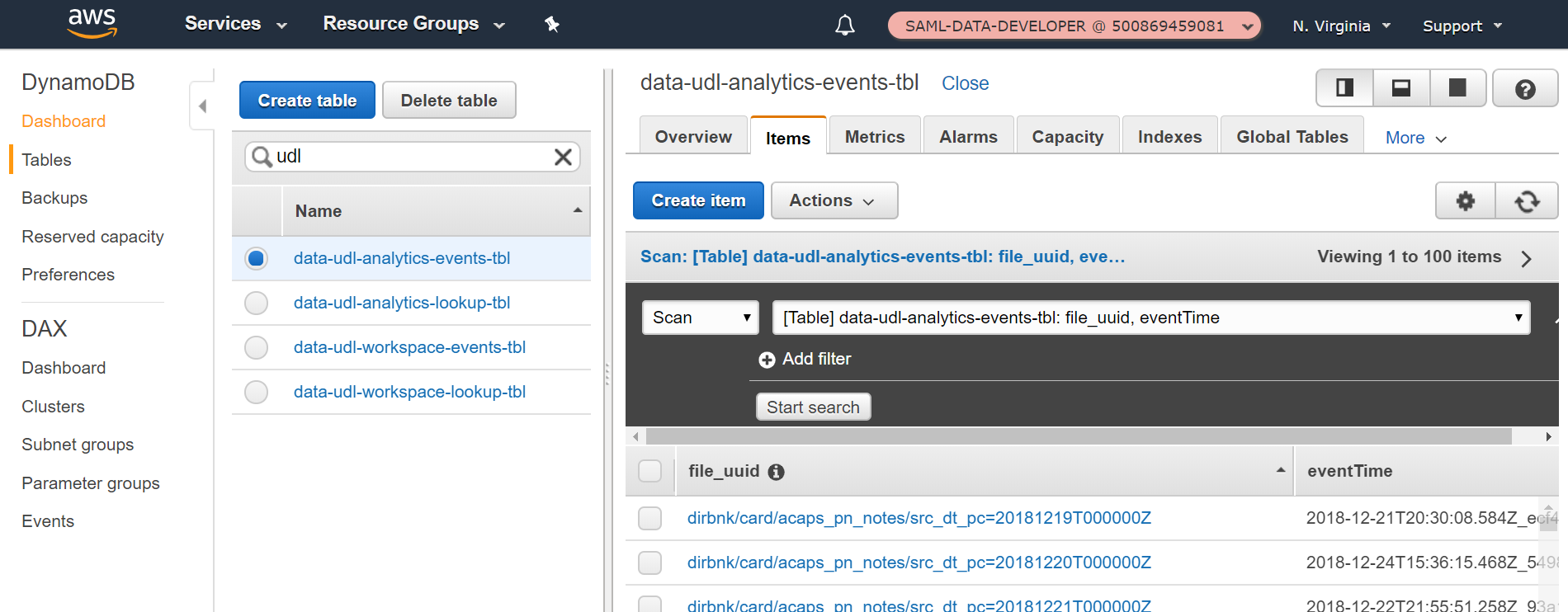
# Operational Reporting

* PLACEHOLDER – design in progress to provide Ops Dashboard to provide visibility into completed and in-flight loads. Tentative high-level solution is anticipated to be sourced from DynamoDB, landed in a query-able data store, then visualized in Tableau.

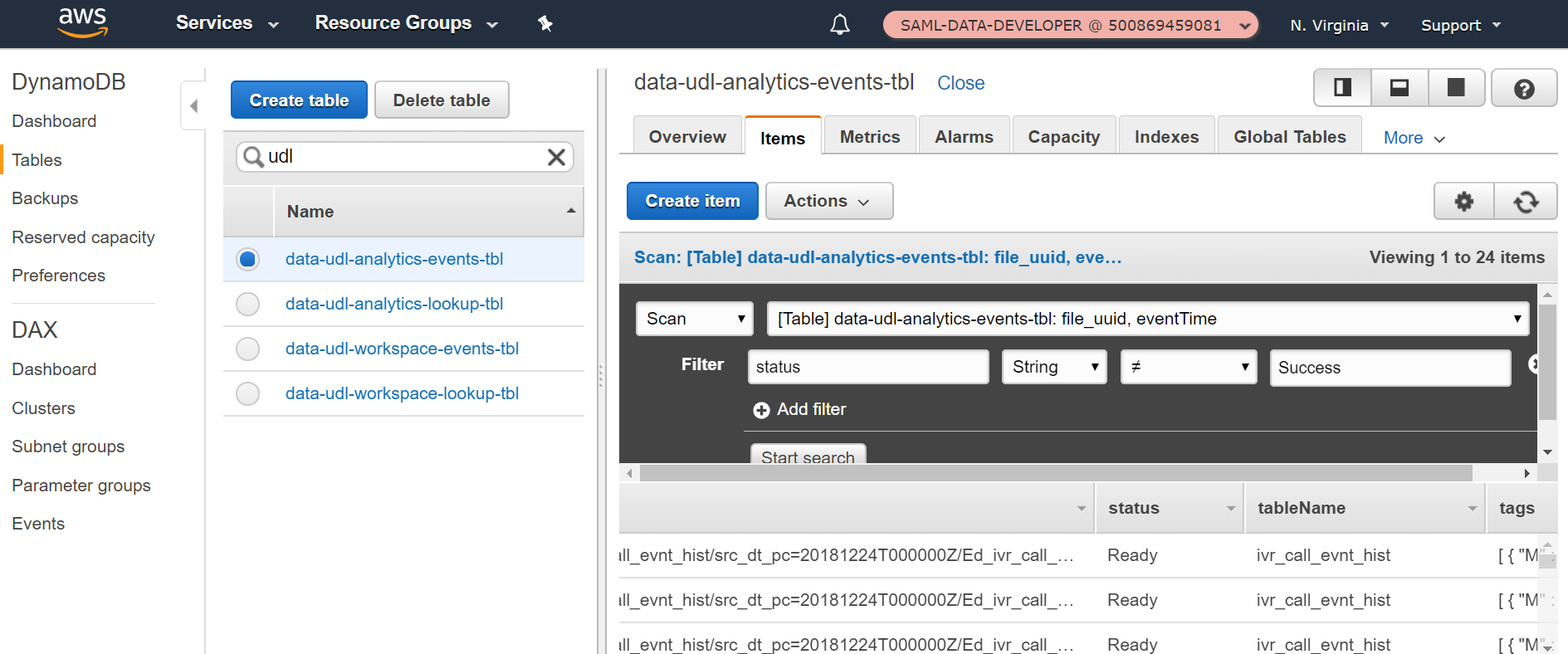
## AWS Monitoring

DynamoDB

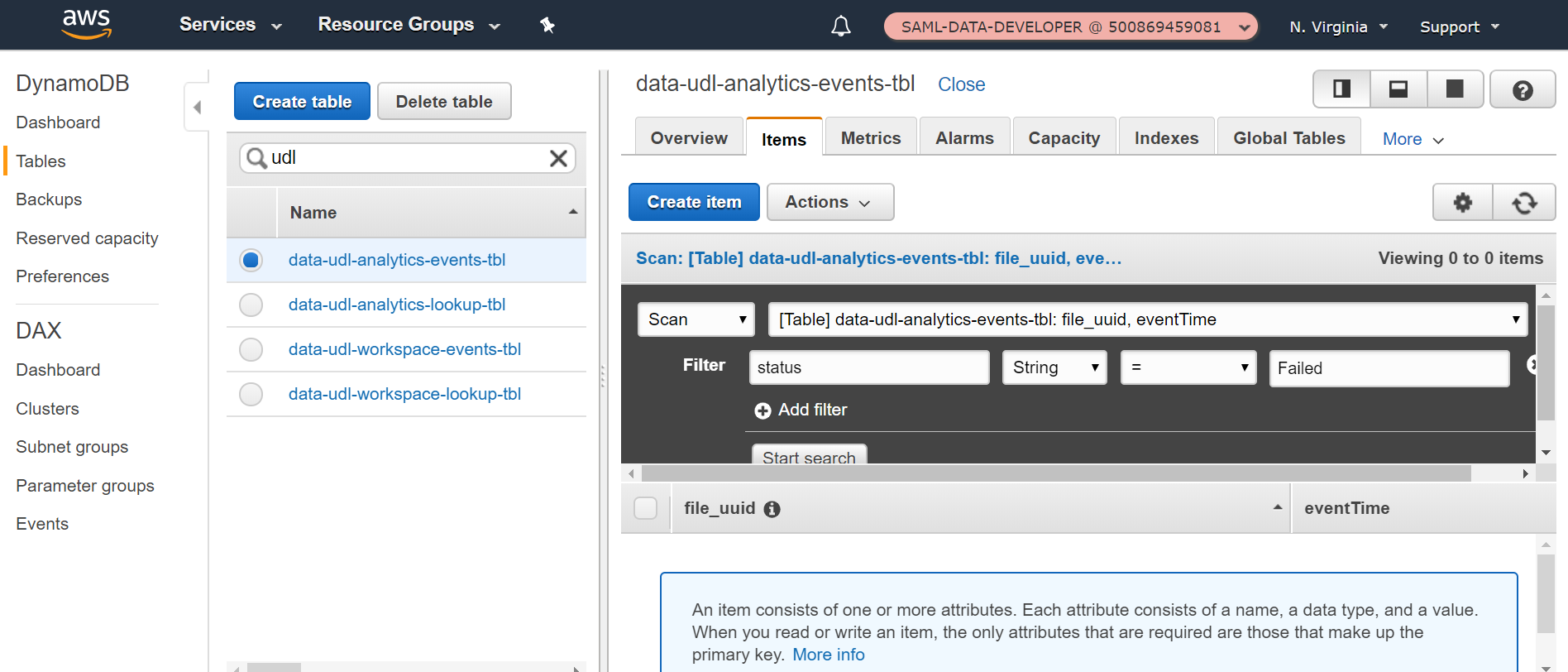
1. Navigate to AWS Console 🡪 DyanmoDB 🡪 Tables and search for “udl”
2. Click into the Production Events table (data-udl-analytics-events-tbl) and select Items tab



1. Scan table items by adding Filter with status not equal to Success, and click Start Search button. If any records are returned, they should show status of Ready

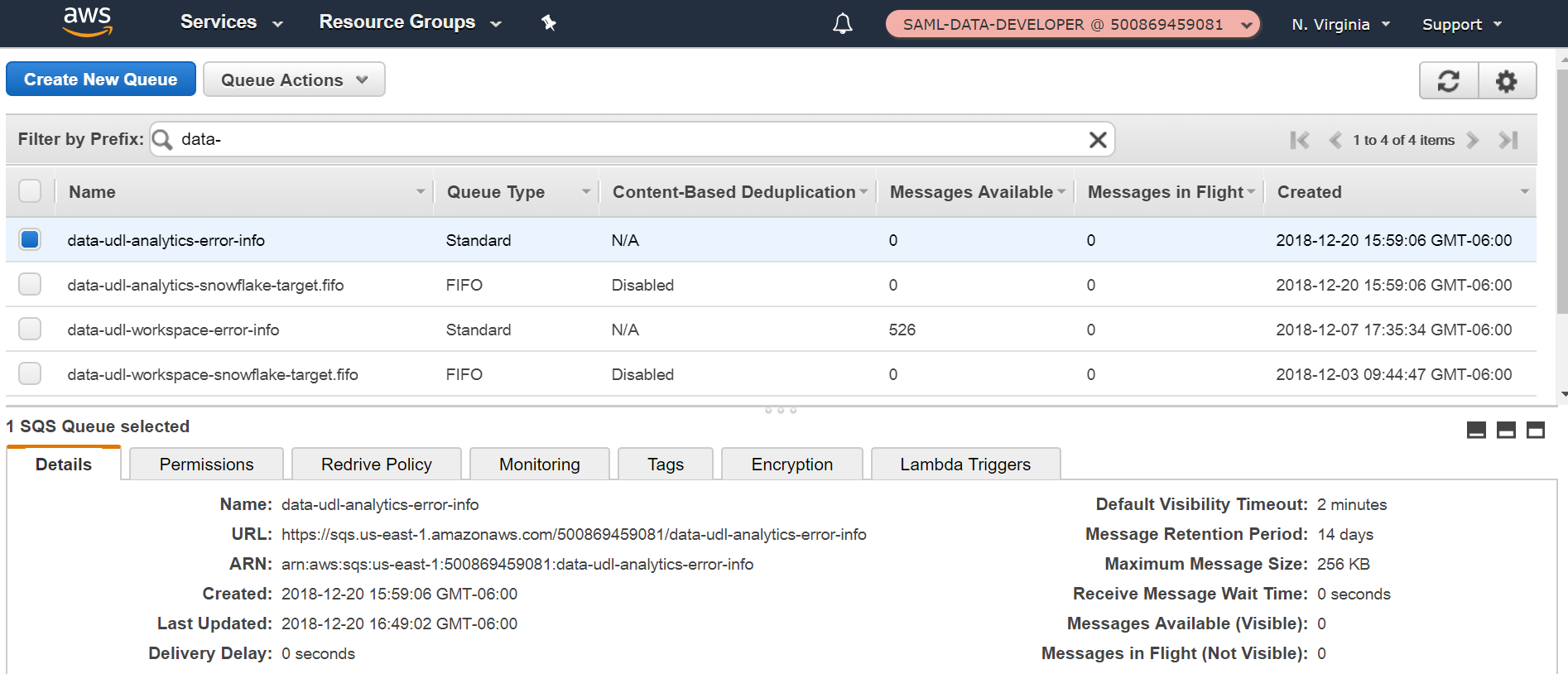


1. Query for status equal to Failed – no records should be returned



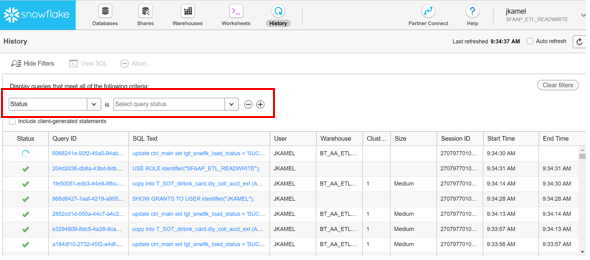
SQS

1. Navigate to AWS Console 🡪 Simple Queue Service and filter by prefix “data-”. Find Production Error Queue (data-udl-analytics-error-info) and confirm Messages Available and Messages in Flight are both set to zero (0).



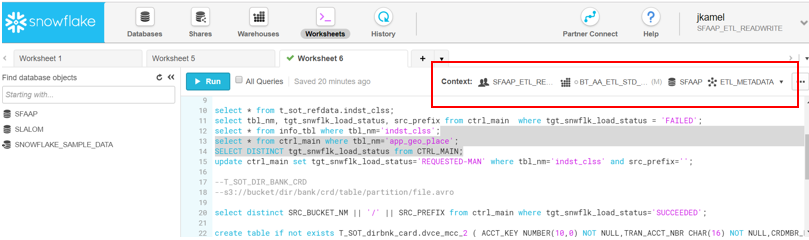
## Snowflake Monitoring

Query History Tab:  
  
The query [History tab](https://dfs.us-east-1.privatelink.snowflakecomputing.com/console#/monitoring/queries) in Snowflake displays currently running queries with the ability to filter and find specific queries of interested(Filter Red box in screenshot below) Most commonly used filters are “Status” to check on running/succeeded queries and specific “SQL Text” search ( In case looking for a particular table/partition or script)



In the screenshot above, the SQL Text column shows the exact SQL command running in Snowflake. Looking at this column helps identify the different running processes in Snowflake while the utility is running. Click on the SQL text to see the full query running to identify which partition/table is being processed.   
  
**“COPY INTO”:** Indicates the COPY command that is loading a partition in S3 into Snowflake

Snowflake Query to check on partition and file migration statuses:

To run a query in Snowflake, first go to the “Worksheet” tab then select your appropriate role, warehouse and database in the context section ( Red box below) to be able to run queries:   
  
  
  
Users can check on files/paritions in Snowflake that failed being copied using the query below:

**Role**: Role provisioned to you when your Snowflake account was setup at DFS to oversee migration  
**Warehouse**: Warehouse available to you for querying when your Snowflake account was setup at DFS  
**Database**: SFAAP  
**Schema:** ETL\_METADATA

select \*

from SFAAP.INFORMATION\_SCHEMA.load\_history

where status <> **'LOADED'**;

Status can be one of three values: “LOADED”,”LOAD FAILED”, or “PARTIALLY LOADED…

* **LOADED**: File loaded successfully in Snowflake
* **LOAD FAILED**: File did not load successfully in Snowflake
* **PARTIALLY LOADED**: File had some records load successfully, but not all. We should **never** see this, as the COPY statement is issued with parameter SKIP\_FILE, which ensures that if even a single record fails to load, all records for the file fail. Similarly, if all records succeed, only then would you see status set to LOADED.

<https://docs.snowflake.net/manuals/sql-reference/info-schema/load_history.html>

*Note: additional filters in the WHERE clause can be added to the SQL statement such as looking for a specific table, file or partition.*

# Data Ops Troubleshooting

## No Metadata on S3 file

1. Logged from S3 Dynamo Loader
2. WARNING logged to Enterprise Logging queue with the following information:
   * Event Source (name of Python module)
   * Message (for the error event) = “unable to fetch metadata for file *<<object key>>* in bucket <<bucket name>>... exiting…”
   * Event (copy of event that came from s3)
   * Process Status = “FAILED”
3. File will not be sent to the UDL ingestion architecture. Remains in S3 and events are ignored.
4. Fix: Work with upstream data providers (e.g., Ab Initio team) to place the file back into S3 with metadata.

## File has metadata, but came in later (after file event was processed)

1. Logged from S3 Dynamo Loader
2. WARNING logged to Enterprise Logging queue with the following information:
   * Event Source (name of Python module)
   * Message (for the error event) = “already exists a row in event table for *<<file uuid>>*... will not write duplicate.”
   * Event (copy of event that came from s3)
3. File will not be sent to the UDL ingestion architecture. Remains in S3 and events are ignored.
4. Fix: Run catchup utility for the individual file (or entire partition if all files were not processed) per Catchup Utility Usage [(link)](#_Runbook:_Catchup_Utility)

## Invalid Load Type

1. Logged from Traffic Controller
2. WARNING logged to Enterprise Logging queue with the following information:
   * Event Source (name of Python module)
   * Message (for the error event) = “load type for this object <<object key>> in bucket <<bucket name>> is: <<load type>>… only handling <<supported types>> at this time.”
   * Event (copy of event that came from dynamo streams)
   * Process Status = “FAILED”
3. File will not be sent to the UDL ingestion architecture. Remains in S3 and events are ignored.
4. Fix: work with upstream data providers (e.g., Ab Initio team) to generate new file(s) with valid load type

Note: supported load types are specified in environment variables in Terraform. Please see readme file for additional information [(link)](https://github.discoverfinancial.com/eddm-edl-org/cloud-data-enablement/blob/snowflake_migration/Snowflake/udl/readme.md)

## New column in Avro file

1. Logged from Traffic Controller, which attempts to validate the schema. This does not stop the message from being sent to queue for processing.
2. INFO logged to Enterprise Logging queue with the following information:
   * Event Source (name of Python module)
   * Message (for the error event) = “validation: FALSE”
   * Event (copy of event that came from dynamo streams)
   * Process Status = “validation failed”
   * Table Name = <<lob>>/<<product>>/<<sub product>>/<<table name>>
3. COPY statement will be generated with the new column and attempt to execute COPY statement causing an error
4. Python connector will capture the error on load into Snowflake, and Table Status will automatically be set from Green to Red in Lookup Table
5. Fix: work with Data Model team to create new column in Snowflake table
6. Fix: change table status in Lookup table from Red to Green using the Lookup Controller Utility [(link)](#_Runbook:_Lookup_Table)

## Changed data type on existing column in Avro file without changing corresponding data type in target (Snowflake)

1. Logged from Traffic Controller, which attempts to validate the schema. This does not stop the message from being sent to queue for processing.
2. INFO logged to Enterprise Logging queue with the following information:
   * Event Source (name of Python module)
   * Message (for the error event) = “validation: FALSE”
   * Event (copy of event that came from dynamo streams)
   * Process Status = “validation failed”
   * Table Name = <<lob>>/<<product>>/<<sub product>>/<<table name>>
3. This will not cause an error if Snowflake implicit casting is successful (e.g., INT --> DOUBLE). However, it will fail if changing, for example, from INT to STRING
   * Logged from Target Loader
   * ERROR logged to Enterprise Logging queue with the following information:
     + Event Source (name of Python module)
     + Message (for the error event) = “snowflake target failed: <<error message from snowflake connector>>”
     + Load Target = SnowflakeLoader.target
   * COPY statement will be generated with the new column and attempt to execute COPY statement causing an error
   * Python connector will capture the error on load into Snowflake, and Table Status will automatically be set from Green to Red in Lookup Table
   * Fix: work with Data Model team to create new column in Snowflake table
   * Fix: change table status in Lookup table from Red to Green using the Lookup Controller Utility [(link)](#_Runbook:_Lookup_Table)

## Missing information in Lookup Table

1. Logged from Traffic Controller
2. WARNING logged to Enterprise Logging queue with the following information:
   * Event Source (name of Python module)
   * Message (for the error event) = “<<bucket name>>/<<object key>> cannot be found in the lookup table… continuing…”
   * Event (copy of event that came from dynamo streams)
3. Fix: insert record in Lookup Table with required values [(link)](#_DynamoDB_Target_Route)
4. Fix: Run catchup utility for the individual file (or entire partition if all files were not processed) per Catchup Utility Usage [(link)](#_Runbook:_Catchup_Utility)

## Snowflake Staging/Target Schema or Staging/Target Table does not exist

1. ASSUMPTION / SCENARIO: record in Lookup Table exists, but Snowflake table was dropped or was never created
2. Logged from Target Loader
3. ERROR logged to Enterprise Logging queue with the following information:
   * Event Source = “loaders.SnowflakeLoader.\_perform\_snowflake”
   * Message (for the error event) = “Snowflake Target Failed: <<exception returned from Snowflake Connector>>”
   * Load Target = “Snowflake”
4. Fix: work with Data Model team to create new Snowflake table
5. Fix: change table status in Lookup table from Red to Green using the Lookup Controller Utility [(link)](#_Runbook:_Lookup_Table)

## Snowflake Python Connector Error (connection to Snowflake cannot be established)

1. Logged from Target Loader
2. ERROR logged to Enterprise Logging queue with the following information:
   * Event Source = “loaders.SnowflakeLoader.\_perform\_snowflake”
   * Message (for the error event) = “Snowflake Target Failed: <<exception returned from Snowflake Connector>>”
   * Load Target = “Snowflake”
3. Fix: work with Cloud Data Enablement team to re-establish connectivity between UDL and the Snowflake target
4. Fix: change table status in Lookup table from Red to Green using the Lookup Controller Utility [(link)](#_Runbook:_Lookup_Table)

## Snowflake Operation Takes Too Long and Lambda Dies

1. Event message gets stuck in the Snowflake FIFO queue with status = "Processing". Retry logic will attempt to reprocess, however, it cannot acquire the lock. Table status in Lookup table is set from Green to Red. Manual intervention is required.
2. Check Cloudwatch to confirm Lambda timeout (message = “<<hash key of lambda request>> Task timed out after 480.00 seconds”)
3. Fix: Try to reprocess by setting Event status = "Ready" manually in the Console for DynamoDB, and change table status in Lookup Table from Red to Green
4. Fix: If fails again, clear message from Snowflake Target FIFO queue manually via AWS Console for DynamoDB and use Manifest Approach (which can run for over 15 min)

*Note: currently, the Lambda timeout is set to 8 min. Cloud Data Enablement team should adjust as needed, perhaps setting Lambda execution times to max of 15 min.*

# Data Ops Common Tasks

## Verify data

* Use Data Validation Utility [(link)](https://github.discoverfinancial.com/eddm-edl-org/cloud-data-enablement/wiki/Data-Validation)

## Cleanup single File or Partition

* + 1. OPTIONAL: Manually delete records from table in Snowflake (only if records loaded to Snowflake)
    2. See Data Ops Troubleshooting [(link)](#_Data_Ops_Troubleshooting)
    3. Work with upstream data providers (e.g., Ab Initio team) to generate new files (must have different filename(s) than the original file(s) that failed)
    4. If message made it to the Snowflake Target FIFO queue, change table status in Lookup table from Red to Green using the Lookup Controller Utility [(link)](#_Runbook:_Lookup_Table)
    5. If message did NOT make it to the Snowflake Target FIFO queue,
       1. Run catchup utility for the individual file or partition as specified in Catchup Utility Usage [(link)](#_Runbook:_Catchup_Utility)
       2. Change table status in Lookup table from Red to Green using the Lookup Controller Utility [(link)](#_Runbook:_Lookup_Table)

## Cleanup entire Table

1. Change table status in Lookup table from Green to Red using the Lookup Controller Utility [(link)](#_Runbook:_Lookup_Table)
2. Work with Cloud Data Enablement tem to truncate table in Snowflake
3. Run Backfill Utility to repopulate table [(link)](https://teams.microsoft.com/_#/docx/viewer/teams/https%3A~2F~2Fdiscoverfinancial.sharepoint.com~2Fsites~2FPlatformsCapabilities~2FShared%20Documents~2FSlalom~2FSnowflake%20Backfill%20Pattern%20Documentation.docx?threadId=19%3A428a9854e8a34927b651b459a569c98b%40thread.skype&baseUrl=htt)
4. Change table status in Lookup table from Red to Green using the Lookup Controller Utility [(link)](#_Runbook:_Lookup_Table)

## Onboarding New non-Teradata Source Table (e.g., Facebook)

1. Work with Data Model Team to create new Snowflake staging & target schema & table
2. Create record for new table in Lookup tablewith required values [(link)](#_DynamoDB_Target_Route)
3. Enable the S3 event (only if still configuring events on table-by-table basis, eventually will be applied to entire bucket)

# Runbook: Catchup Utility

## Overview

The purpose of the catchup utility is to manually generate and inject mock s3 events into the UDL architecture to trigger the processing of the files you would like to insert into the target tables. This allows us to reprocess files via the UDL that have already been landed in S3 without the need to land the files in S3 again.

There are two ways to run the catchup utility:

1. Provide a list of prefixes and suffixes in a csv file and the utility will iterate thru the csv and run the catchup on the provided files or partition.
2. Directly against a partition or file.

## CSV File Setup

The CSV should only have 2 columns. If it has more only the first two columns will be read.   
1- The first column should contain the prefix of the partition/file you want to go after in the order wanted.   
2- The second column should contain the suffix of the file/s you want to go after (Usually ‘.avro’. The CSV file will also ignore the first row as it is meant for the header of the file.

Note that if you want to catchup specific files within a partition you can choose to provide the full file prefix as opposed to just the prefix to the partition level.   
This is useful when a partition has two load types (i.e: Upserts and Inserts) and only the upserts need to be processed or if some files need to be processed before others in the partition.

## Psuedo Code (CSV Approach)

1. Open and read CSV file.
2. Ignore the first row
3. Loop thru the csv until the end is reached.
4. While looping call the execute\_main function in catchup\_util.py by passing in the contents of each row in the csv.

## Psuedo Code (Direct Approach)

1. Initiate connection to lambda
2. Initiate connection to s3
3. Get list of files from s3 using the prefix and suffix provided via parameters
4. Generate mock s3 event
5. Submit mocked s3 event to the dynamodb s3 loader lambda function in UDL.

## Usage (CSV Approach)

usage: udl-support-util catchup\_csv [-h] [--bucket BUCKET]

--dynamoDBEventsTblName

DYNAMODBEVENTSTBLNAME

--dynamoDBLookupTblName

DYNAMODBLOOKUPTBLNAME --csv CSV --runType

{catchup,replay}

[--sleepBetweenCSVEntries SLEEPBETWEENCSVENTRIES]

Arguments:

-h, --help show this help message and exit

--bucket BUCKET S3 bucket to look for data.

--dynamoDBEventsTblName DYNAMODBEVENTSTBLNAME

Name of the dynamodb events table that the catchup

records need to be inserted to.Ex: "data-udl-

workspace-events-tbl"

--dynamoDBLookupTblName DYNAMODBLOOKUPTBLNAME

Name of the dynamodb lookup table that the catchup

records need to verify if they can be inserted or not.

EX. "data-udl-workspace-lookup-tbl"

--csv CSV Relative path to the csv, look at the example csv in

the directory to figure out how you need to create the

csv

--runType {catchup,replay}

Type of run you want to perform. Values can be

[catchup, replay]

--sleepBetweenCSVEntries SLEEPBETWEENCSVENTRIES

How many seconds do you want to sleep between csv

entries?

Note that for the DynamoDBEventsTblName and DynamoDBLookupTblName argument the below are the test vs production lambda one could send messages to.   
  
Test Lookup Table: data-udl-workspace-lookup-tbl  
Production Lookup Table: data-udl-analytics-lookup-tbl

Test Events Table: data-udl-workspace-events-tbl  
Production Events Table: data-udl-analytics-events-tbl

## Usage (Direct Approach)

udl-support-util -V catchup\_util -h

usage: udl-support-util catchup\_util [-h] [--bucket BUCKET] --prefix PREFIX

[--suffix SUFFIX] --dynamoDBEventsTblName

DYNAMODBEVENTSTBLNAME

--dynamoDBLookupTblName

DYNAMODBLOOKUPTBLNAME --runType

{catchup,replay}

Arguments:

-h, --help show this help message and exit

--bucket BUCKET S3 bucket to look for data.

--prefix PREFIX The prefix of the partition you want to run catchup

on. Ex:

"dirbnk/card/pymt\_pgm\_stat/src\_dt\_pc=20091225T000000Z"

--suffix SUFFIX The suffix of the files within the partition you want

to run catchup on. Ex. ".avro"

--dynamoDBEventsTblName DYNAMODBEVENTSTBLNAME

Name of the dynamodb events table that the catchup

records need to be inserted to.Ex: "data-udl-

workspace-events-tbl"

--dynamoDBLookupTblName DYNAMODBLOOKUPTBLNAME

Name of the dynamodb lookup table that the catchup

records need to verify if they can be inserted or not.

EX. "data-udl-workspace-lookup-tbl"

--runType {catchup,replay}

Type of run you want to perform. Values can be

[catchup, replay]

Note that for the DynamoDBEventsTblName and DynamoDBLookupTblName argument the below are the test vs production lambda one could send messages to.

Test Lookup Table: data-udl-workspace-lookup-tbl  
Production Lookup Table: data-udl-analytics-lookup-tbl

Test Events Table: data-udl-workspace-events-tbl  
Production Events Table: data-udl-analytics-e

## Git Hub Link

* <https://github.discoverfinancial.com/eddm-edl-org/cloud-data-enablement/tree/snowflake_migration/Snowflake/udl_utils/catchup>

# Runbook: Lookup Controller Utility

## Overview

The purpose of the lookup controller utility is to control the interaction with the lookup dynamodb table in the UDL architecture. At the moment, the utility will only allow you to switch the status of the targets and groups/tables from green to red or red to green.

## Psuedo Code

1. Read parameters passed to util.
2. If parameter passed is to update status of whole target then update target in dynamodb
3. If parameter passed is to update status of a table within a target then update the status of the table within the target.

## Usage

usage: lookup\_controller.py [-h] [--lookupTableName LOOKUPTABLENAME]

                            --dataAssetName DATAASSETNAME --version VERSION

                            --updateType {table\_status,group\_status}

                            --targetType {manifest,non\_manifest} –targetName TARGETNAME

      --statusToSet {red,green}

Description: This utility controls the interaction with the lookup dynamodb table.

optional arguments:

  -h, --help            show this help message and exit

  --lookupTableName LOOKUPTABLENAME

                        Name of the lookup table you want to make changes to. Default will be Dev dynamoDB table "data-udl-workspace-lookup-tbl"

  --dataAssetName DATAASSETNAME

                        Name of the data\_asset you are trying to update the entries for.

  --version VERSION     Version of the data\_asset you are trying to update entries for.

  --updateType {table\_status,group\_status}

                        Type of update you want to make. Either table\_status or group\_status

  --targetType {manifest,non\_manifest}

                        Type of target you are updating. Options are manifest or non\_manifest

  --targetName TARGETNAME

                        Name of the target you are looking to update the value status for.

  --statusToSet {red,green}

                        The value you want to set for the status. Has to be red or green.

## Git Hub Link

* [https://github.discoverfinancial.com/eddm-edl-org/cloud-data-enablement/tree/snowflake\_migration/Snowflake/udl\_utils/lookup](https://urldefense.proofpoint.com/v2/url?u=https-3A__github.discoverfinancial.com_eddm-2Dedl-2Dorg_cloud-2Ddata-2Denablement_tree_snowflake-5Fmigration_Snowflake_udl-5Futils_lookup&d=DwMFAg&c=fa_WZs7nNMvOIDyLmzi2sMVHyyC4hN9WQl29lWJQ5Y4&r=gsnZL9cXrzyvTnjWma7vyYM68KEuSZzGm3_oOuZNgCM&m=D3W3elVyYjnSS3wTHPH-XamUy-pcYmr0nRhGvsZchaE&s=V1-BckOylgPf6yhtD-idWc3KNUbdyB1z43EbCajITAM&e=)

# Appendix

## How to pull metadata from S3 file via AWS CLI

**aws s3api head-object --bucket dfs-analytics-dlt-use1-ent-etl --key dirbnk/card/prt\_ind\_cr\_sgm\_cr\_rpt\_hist/src\_dt\_pc=20181231T000000Z/Ededw\_prt\_ind\_cr\_sgm\_cr\_rpt\_hist.20181129.load.1543501572.20181231.part\_000.avro**

{

"AcceptRanges": "bytes",

"ContentType": "binary/octet-stream",

"LastModified": "Tue, 04 Dec 2018 00:50:10 GMT",

"ContentLength": 684958131,

"VersionId": "UGBb5W7lnSwwaTilTWM75QstUtg8HwGo",

"ETag": "\"ed8206a2c5875c86764c589805ea336c\"",

"ServerSideEncryption": "AES256",

"Metadata": {

"load-type": "Upsert",

"source": "Ededw\_prt\_ind\_cr\_sgm\_cr\_rpt\_hist.20181129.load.gz",

"source-type": "File",

"upsert-key": "acct\_key;perf\_mthly\_dt;cr\_bur\_cde;cr\_rpt\_eff\_dt",

"record-count": "3633080"

}

}

**aws s3api get-object-tagging --bucket dfs-analytics-dlt-use1-ent-etl --key dirbnk/card/acaps\_pn/src\_dt\_pc=20181203T000000Z/Edcims\_acaps\_pn.20181203.load.1543908058.part\_006.avro**

{

"VersionId": "m9A8wO\_QZxpZRAKi00Sb8sK1neZF2vZr",

"TagSet": [

{

"Value": "ACQT",

"Key": "org-code"

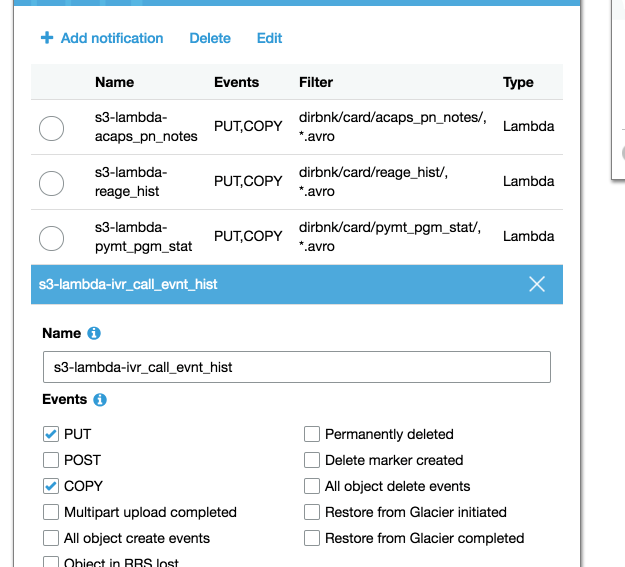
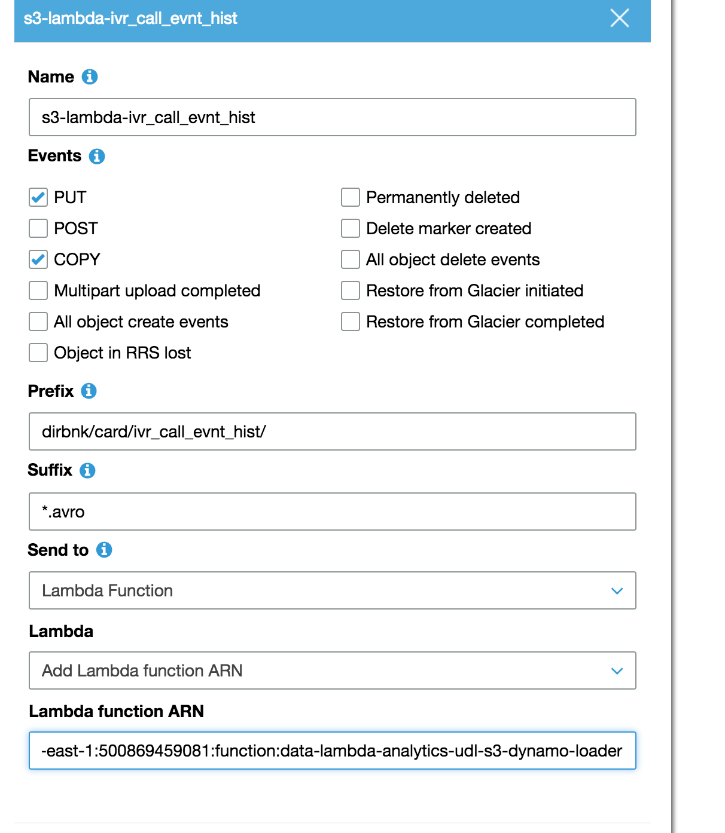
}

]

}

## S3 Bucket Configuration for spawning Events

Screenshots detailing setup and configuration of S3 bucket (or any sub-directory / prefix therein):



## Explaining S3 SQS Fifo Queues

* <https://aws.amazon.com/blogs/compute/solving-complex-ordering-challenges-with-amazon-sqs-fifo-queues/>

## Steps for On-boarding New Target (e.g., non-Snowflake)

Work with Cloud Data Enablement team to coordinate and execute the following…

1. Create new target loader lambda
2. Create new FIFO queue
3. Create new record in Lookup Table

* Non-Manifest: ARN for FIFO queue and ARN for target loader lambda
* Manifest: ARN for FIFO queue

1. Test all Load Types and Operations in a non-Production environment
2. Package Code via Terraform
3. Deploy to Production
4. Perform hyper-care until enhancements are stabilized

## Points of Contact by Team

|  |  |  |
| --- | --- | --- |
| **Team** | **Point of Contact (Primary)** | **Email** |
| Cloud Data Enablement | Will Clemons (Primary) | williamclemens@discover.com |
| Cloud Data Enablement | Hassan Faouaz | hassanfaouaz@discover.com |
| Cloud Data Enablement | Praveen Guduri | praveenguduri@discover.com |
| Cloud Data Enablement | Robert Giron | robertgiron@discover.com |
| Data Modeling | Harshada Pawar (Primary) | HarshadaPawar@discover.com |
| Data Modeling | Jon Collazo | jonathancollazo@discover.com |
| Data Modeling | Larry Tsang | larrytsang@discover.com |
| Snowflake Platform | Matt Lambert (Primary) | mattlambert@discover.com |
| Snowflake Platform | Krishna Govindan | krishnagovindan@discover.com |
| Snowflake Platform | Qin Huang | qinhuang@discover.com |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |