Cloud Native Data – Foundation Layer

Software Design Document

Version # : 1.0

Released on :

Author :

1. Introduction 4

1.1 Purpose 4

1.2 Scope 4

1.3 Out of Scope 4

1.4 Associated Business Requirement Documents (BRDs) 4

1.5 Acronyms and Definitions 4

1.6 References 5

2. SRDM (Snowflake Replication Data Manager) 5

2.1 High-Level Design 5

2.2 Infrastructure Design 6

3. SRDM Detailed Design 7

3.1 SRDM Main Module 7

3.2 Netezza Bulk Extract Module 9

3.3 Snowflake Sync Module 10

3.4 Configuration & Audit Module 11

3.5 Asset Deployment Module 11

3.6 Balancing & Reconciliation Module 12

3.7 Password Rotation 12

4. Snowflake - DB & Schema design 13

4.1 SyncControlAuditDB 14

4.1.1 SN\_DB\_SYNC\_CONTROL 15

4.1.2 SN\_TABLE\_SYNC\_CONTROL 15

4.1.3 SN\_BATCH\_AUDIT 17

4.1.4 SN\_BATCH\_DETAIL\_AUDIT 18

4.1.5 SN\_BATCH\_METRICS 18

4.1.6 PRC\_BATCH\_SYNC 18

4.2 Landing DB 19

5. Snowflake Environments 20

5.1 Environment Setup 20

5.2 Virtual Warehouse Management 21

5.2.1 Virtual Warehouse Size Based On Environment 22

5.2.2 Enable Auto-Suspend 22

5.2.3 Enable Auto-Resume 22

5.2.4 Enable Auto-Scaling 23

5.2.5 Disable Warehouse Auto-Suspension for Frequent Queries 23

5.2.6 Set Timeouts Appropriately for workloads 23

5.2.7 Set Account Statement Timeouts 23

5.2.8 Monitor Cloud Service Billing Threshold 24

5.2.9 Use Resource Monitors 24

6. RBAC Configurations 25

6.1 Role Based Column Level Security 25

6.2 Role Based Row Level Security 26

6.3 Role Based Hierarchy 26

7. General Design and Implementation 26

7.1 Interfaces with Other Systems 26

7.2 List of Impacted Inventory 26

7.3 Scheduling Impact 27

8. Database Design 27

9. Inputs and Outputs 27

9.1 Screen Layouts 27

9.2 Report Layouts 27

9.2.1 Input Structure 27

9.2.2 Output Structure 27

10. *Non-Development Services* Considerations 27

11. Constraints 27

12. Assumptions and Issues 27

13. Appendices 28

14. Amendment History 28

15. Approval of Final Document 28

# Introduction

## Purpose

BCBSRI has decided to migrate legacy systems to Cloud data platforms and chosen the Snowflake and Matillion to replace Netezza and DataStage respectively. Creating Foundation Layer is the first step of Cloud native data initiative project. As part of Foundation layer, we must create Netezza equivalent data model in Snowflake and should sync up data between Netezza and Snowflake.

This documents will explain the high level and low level design for Cloud Native Data – Foundation Layer project.

## Scope

## Out of Scope

## Associated Business Requirement Documents (BRDs)

**Intended Audience**

| Name | Project Role |
| --- | --- |
| Sunil Rangineni | Data Architect, Informatics/IT |
|  |  |

## Acronyms and Definitions

| Acronym | Description |
| --- | --- |
| ADM | Application Development & Maintenance |
| BRD | Business Requirement Document |
| BCBSRI | Blue Cross Blue Shield of Rhode Island |
| CMS | Center for Medicare & Medicaid Services |
| EDR | Enterprise Data Repository is a relational database which resides on a Microsoft SQL server and supports longitudinal and research-based reporting, without compromising the responsiveness of your LIVE system. The BCBSRI EDR houses claims data. |
| NCQA | National Committee for Quality Assurance |
| SRS | Software Requirements Specification |

## References

| Document Name | Version | Date |
| --- | --- | --- |
| Insert BRD |  |  |

# SRDM (Snowflake Replication Data Manager)

## High-Level Design



1. Extract the Netezza data into AIX server in the form of files (ex: csv, JSON etc) using python/AIX scripts.
2. Upload the files from the AIX server to Snowflake internal stage using PUT command.
3. Use COPY INTO to load data from Internal stage to the Snowflake Landing tables.
4. Call the Snowflake procedure to load data from Landing tables to main tables depends on load flag.

## Infrastructure Design

Diagram

Description automatically generated

[Azure Private Link](https://docs.microsoft.com/en-us/azure/private-link/private-link-overview) provides private connectivity to Snowflake by ensuring that access to Snowflake is through a private IP address. Traffic can only occur from the customer virtual network (VNet) to the Snowflake VNet using the Microsoft backbone and avoids the public Internet. This significantly simplifies the network configuration by keeping access rules private while providing secure and private communication.

The above diagram summarizes the Azure Private Link architecture with respect to the customer VNet and the Snowflake VNet.

From either a Azure VM (virtual machine) (2) or through VNet peering (1) between on-prem to Azure Cloud, you can connect to the Azure Private Link endpoint (3) in your virtual network. That endpoint then connects to the Private Link Service (4) and routes to Snowflake.

# SRDM Detailed Design

SRDM (Snowflake Replication Data Manager) is a deployable bundle that will be developed in python to migrate the initial data from Netezza to Snowflake as well as maintain the daily/weekly/monthly data sync between Netezza & Snowflake.

This deployment module will include sub modules like configuration, control & audit, deployment, snowflake sync, balancing etc.

A diagram of data management

Description automatically generated with medium confidence

## SRDM Main Module

SRDM Main is the parent module that gets invoked by the orchestration framework (i.e., by HWA & Matillion). This module takes inputs of Database name & Sync Schedule to trigger the Netezza to Snowflake Sync process. Runbooks & Schedules will be created separately out of this SRDM Main module for each database.

SRDM Main calls other dependent modules like control & audit, Netezza bulk extract, snowflake sync etc to implement necessary functionality.



## Netezza Bulk Extract Module

Netezza Bulk Extract module is responsible for extracting the data from Netezza based on the configuration definitions and creating necessary files in netapp storage location. This module also splits big files into 200MB chucks for efficient migration to snowflake.



## Snowflake Sync Module

Snowflake Sync module is responsible for copying the extract files into snowflake, invoking the snow pipes and loading the data into landing area & finally merging the data into snowflake core tables..



## Configuration & Audit Module

Configuration:

Configure/Registration of Control table is onetime activity(Initial activity). Create catalogue sheet which will have all the Databases, corresponding tables, Load frequency, Load type, extract sql and other required information which will feed the data into control tables (SN\_DB\_SYNC\_CONTROL and SN\_TABLE\_SYNC\_CONTROL). Below is the drafted catalogue sheet.

Audit:

This module will have python functions which should insert/update entries into Snowflake Audit tables.

Create python package to make entries into the Audit tables (SN\_BATCH\_AUDT & SN\_BATCH\_DETAIL\_AUDIT) for each step (external table creation, extraction, copy into stage, copy into landing and sync from Landing to target table) while doing Sync process from Netezza to Snowflake. Modularize the package such a way we can use this across the scripts. Below are the python functions we should create in this Control and Audit package.

Configuration & Audit Package name: sncontrolauditpkg.py

CreateSnowflakeConnection: Create function which makes connection to Snowflake from python. This function we will use across the process.

InserUpdateBatchAudit: Create a generic python function such a way to insert/update an entry into SN\_BATCH\_AUDIT table. Insert and Update will happen depends on indicator which we pass.

InsertBatchDetailAudit: Create a generic python function to log every step of sync process into log table called SN\_BATCH\_DETAIL\_AUDIT table.

SplitDirCreation: Create a generic python function, which should take AIX\_SERVER\_PATH and DBANME as arguments and which should check whether sub directory with passed DBNAME exits or not under AIX\_SERVER\_PATH path. Create if directory does not exist, else delete all the files under that directory.

SplitFiles: Create a generic python function, which should take extracted file and row size as parameter and split the extracted file into chunks of files (around 200MB) to get optimal performance.

## Asset Deployment Module

This module will have deployment functions for each snowflake objects (Table, Stage, Pipe etc). Create a python package which deploy the required Snowflake objects automatically. This package should be generic to construct the snowflake objects dynamically and deploy them. Below are the python functions which should be created for deployment.

Suggested Asset Deployment Package name: sndeploypkg.py

DeploySnSyncTable: Create a function, which should take source (Netezza) and target (Snowflake) database and table details. Extract corresponding DDL from Netezza database, run python script which converts Netezza DDL into Snowflake compatible and deploy converted DDL into Snowflake.

DeploySnLdTable: Create a function, which should take Snowflake landing and target database, table details and create the landing table with the structure of snowflake target table dynamically and deploy.

DeploySnStage: Create a function, which should take snowflake database name, schema name and stage object name. Construct the stage object DDL dynamically and deploy into Snowflake.

DeploySnPipe: Create a function, which should take Snowflake landing database, table, and other required details. Construct the pipe object DDL dynamically and deploy into Snowflake database.

## Balancing & Reconciliation Module

This module will create the balancing for each run/runbook and determine whether batch is successful or not.

Create a python package which runs at end of the srdmmain.py script. This script should perform couple of checks to determine whether batch is successful or not.

Table level reconciliation:

Get the sync table count which should be processed foe that batch and store in variable (use the result of getsynctablecount function).

Get the extraction successful table count for the batch and store in variable (write python function to get table count which extraction completed successfully for that batch - use SN\_BATCH\_DETAIL\_AUDIT table)

Get the sync successful table count for the batch and store in variable (write python function to get table count which sync completed successfully for that batch – use SN\_BATCH\_DETAIL\_AUDIT table)

Finally, compare above 3 variables, generate batch status as successful if all 3 variables are equal.

Record level reconciliation:

Write a python code to load the SN\_BATCH\_METRICS table from SN\_BATCH\_DETAIL\_AUDIT table.

Extract data from SN\_BATCH\_METRICS table and compare the record count for source and target for each table for that batch. If the record count is equal or within threshold we can mark batch as success, else mark as failed.

## Password Rotation

Password rotation is applicable for service accounts only. Normal users/developers login with SSO. We have written the customized python module which will generate the 2034 encrypted private and public key and publish the private in azure key vault and public key with Snowflake user through snowflake procedure.

All the applications which need to connect to snowflake should use the private key which is there in azure key vault and connects with Snowflake.

Below are the sequence of steps python module does:

1. We should pass the Snowflake username, Key vault name, Secret name, path and path2 as arguments to the python script like below.

Python3 snow\_key\_rotation.py TEST\_PUB\_KEY azkv-eus2-dev-pythn-01 DEMO-SECRET-KEY /apps/srdm/int/key/ /apps/srdm/int/key2/

2. When you executed above script

a. first it will generate an encrypted private and public key pair in /apps/srdm/int/key2/ path (rsa\_key.p8 and rsa\_key.pub).

b. Second step is, update the corresponding Snowflake user with generated public key through by triggering stored procedure( Attached the procedure).

c. Third step is, update the Azure key vault secret with generated private key.

d. Finally, Remove the private key file and move the public key file from /apps/srdm/int/key2/ to /apps/srdm/int/key/. This moved public key will be used while updating next time.

Note: For first time, we should generate the private and public key pair and update the public key in Snowflake and Private key in Azure key vault manually.

ALTER USER <SNOWFLAKE USERNAME> SET RSA\_PUBLIC\_KEY=’<Generated Public key>’;

Sudo code for password rotation is:



# Snowflake - DB & Schema design

A picture containing text, screenshot

Description automatically generated

Every database in Netezza will be replicated (i.e., copied & kept in sync) into Snowflake environment. In addition to the existing DB’s, new database will be created for saving control/audit data & Landing structures.

[New DB’s needs better naming conventions 😊]

## SyncControlAuditDB

Diagram

Description automatically generated with medium confidence

This database handles the control bundles and persists the audit log information for the day-to-day operational Netezza to Snowflake Sync process (along with the initial db migration).

This database will hold the below objects & procedures

### SN\_DB\_SYNC\_CONTROL

This table will hold the batch configuration for each database which we should sync between Netezza and Snowflake. Below is the table structure.

|  |  |  |
| --- | --- | --- |
| **TABLE\_NAME** | **COLUMN\_NAME** | **Column description** |
| SN\_DB\_SYNC\_CONTROL | BATCH\_DB\_ID | Unique Id for each database |
| SRC\_DATABASE | Source Netezza database name |
| TGT\_DATABASE | Target Snowflake database name |
| EXTRACT\_FLAG | Indicates whether extraction should do or not |
| SYNC\_FLAG | Indicates whether sync process should run or not |
| CREATED\_DATE | Date when the record got inserted |
| UPDATED\_DATE | Date when the record got updated in table |

### SN\_TABLE\_SYNC\_CONTROL

This table will have the batch detailed configuration for each table (extraction query, database objects, file paths, file patterns, snowflake objects etc.) of each database which is there in SN\_DB\_SYNC\_CONTROL table.

|  |  |  |
| --- | --- | --- |
| **TABLE\_NAME** | **COLUMN\_NAME** | **Column description** |
| SN\_TABLE\_SYNC\_CONTROL | BATCH\_DB\_ID | Batch DB id from SN\_DB\_SYNC\_CONTROL(FK) |
| BATCH\_TABLE\_ID | Individual id for each table |
| SRC\_TABLE | Source Netezza table name |
| TGT\_LOAD\_TYPE | Target Sync/load type (Full/Delta) |
| LOAD\_FREQUENCY | Load frequency (D/W/M/A) |
| PROCESS\_ID | Process id for set for set of tables in order to balance the load for multi-threading |
| SUBPROCESS\_ID | Sub process id for set for set of tables in order to balance the load for multi-threading |
| SOURCE\_SYNC\_SQL | Source Netezza Sql for bulk extraction |
| SOURCE\_COUNT\_SQL | Source Netezza SQL to get record count for balancing |
| AIX\_SERVER\_PATH | Source files Linux path |
| AIX\_FILEPATTERN | Extraction files pattern |
| SN\_INTERNALSTAGENAME | Snowflake stage name |
| SN\_FILEFORMAT | Snowflake File Format name |
| SNOWPIPENAME | Snow pipe name |
| LN\_DATABASE | Snowflake Landing database name |
| LN\_SCHEMA | Snowflake Landing schema name |
| LN\_TABLE | Snowflake Landing table name |
| TGT\_DATABASE | Snowflake target database name |
| TGT\_SCHEMA | Snowflake target schema name |
| TGT\_TABLE | Snowflake target table name |
| SYNC\_FLAG | Indicates whether table should be considered for sync process or not |
| OBJECTS\_BUILD\_FLAG | Flag which decides objects deployment/re build (nz external table, snowflake tables, stages and pipes) |
| TGT\_PRIMARY\_KEY\_1\_COL | Primary key column1 to make Upsert into Target table |
| TGT\_PRIMARY\_KEY\_2\_COL | Primary key column2 to make Upsert into Target table |
| TGT\_PRIMARY\_KEY\_3\_COL | Primary key column3 to make Upsert into Target table |
| TGT\_PRIMARY\_KEY\_4\_COL | Primary key column4 to make Upsert into Target table |
| TGT\_PRIMARY\_KEY\_5\_COL | Primary key column5 to make Upsert into Target table |
| TGT\_PRIMARY\_KEY\_6\_COL | Primary key column6 to make Upsert into Target table |
| TGT\_PRIMARY\_KEY\_7\_COL | Primary key column7 to make Upsert into Target table |
| TGT\_PRIMARY\_KEY\_8\_COL | Primary key column8 to make Upsert into Target table |
| TGT\_PRIMARY\_KEY\_9\_COL | Primary key column9 to make Upsert into Target table |
| TGT\_PRIMARY\_KEY\_10\_COL | Primary key column10 to make Upsert into Target table |
| WATERMARK\_1\_COL |  |
| WATERMARK\_2\_COL |  |
| WATERMARK\_3\_COL |  |
| CREATED\_DATE | Date when the record got inserted |
| UPDATED\_DATE | Date when the record got updated in table |

### SN\_BATCH\_AUDIT

This table is the top-level log table where we will have the log entry for each runbook. We will insert an entry into this table with the status of ‘STARTED’ at beginning of the batch, will update the batch status to FINISHED’ if it’s successful else will update with ABORTED at the end of the batch.

|  |  |  |
| --- | --- | --- |
| **TABLE\_NAME** | **COLUMN\_NAME** | **Column description** |
| SN\_BATCH\_AUDIT | BATCH\_RUN\_ID | Sequence/Unique value for each runbook |
| BATCH\_DB\_ID | Batch DB id from SN\_DB\_SYNC\_CONTROL(FK) |
| RUN\_DATE | Batch run date |
| BATCH\_START\_TIME | Start time of batch |
| BATCH\_END\_TIME | End time of batch |
| BATCH\_STATUS | Status of batch (STARTED/FINISHED/ABORTED) |

### SN\_BATCH\_DETAIL\_AUDIT

This table will have detailed level log entries of the extracting and sync process. For every step we will insert an entry into table with timestamp and status.

|  |  |  |
| --- | --- | --- |
| **TABLE\_NAME** | **COLUMN\_NAME** | **Column description** |
| SN\_BATCH\_DETAIL\_AUDIT | BATCH\_DETAIL\_RUN\_ID | Sequence/Unique value for event/step |
| BATCH\_RUN\_ID | Corresponding Batch Run Id from SN\_BATCH\_AUDIT table |
| BATCH\_TABLE\_ID | Corresponding Batch table Id from SN\_TABLE\_SYNC\_CONTROL table |
| PROCESSED\_OBJECT\_NAME | Processed object name |
| BATCH\_DETAIL\_STEP | Detailed step (EXTRACTION/COPY INTO STAGE, LOAD INTO LANDING etc) |
| BATCH\_DETAIL\_STATUS | Status of each step (STARTED/COMPLETED/ABORTED) |
| BATCH\_DETAIL\_DATETIME | Timestamp of each step |
| RECORD\_COUNT | Record count of each step |

### SN\_BATCH\_METRICS

This table will be used to capture the batch metrics for each table which is processed. This table data will be used to determine the reconciliation of records between source and target and derive the batch failed/successful.

|  |  |  |
| --- | --- | --- |
| **TABLE\_NAME** | **COLUMN\_NAME** | **Column description** |
| SN\_BATCH\_METRICS | BATCH\_RUN\_ID | Corresponding Batch Run Id from SN\_BATCH\_DETAIL\_AUDIT table |
| BATCH\_TABLE\_ID | Corresponding Batch Run Id from SN\_BATCH\_DETAIL\_AUDIT table |
| EXTRACTION\_REC\_COUNT | Extraction record count of Netezza table |
| LN\_TBL\_REC\_COUNT | Record count in Landing table |
| TGT\_TBL\_REC\_COUNT | Record count (no. of inserted records + no. of updated record) in Landing table |

### PRC\_BATCH\_SYNC

This is the common Snowflake procedure for all the tables to load the data from Landing table to Target table load. This procedure takes Landing table and Target table details including Load type and generates a dynamic sql queries (Merge statement for Delta load/Truncate and Load statement for Full load) and executes that.

## Landing DB

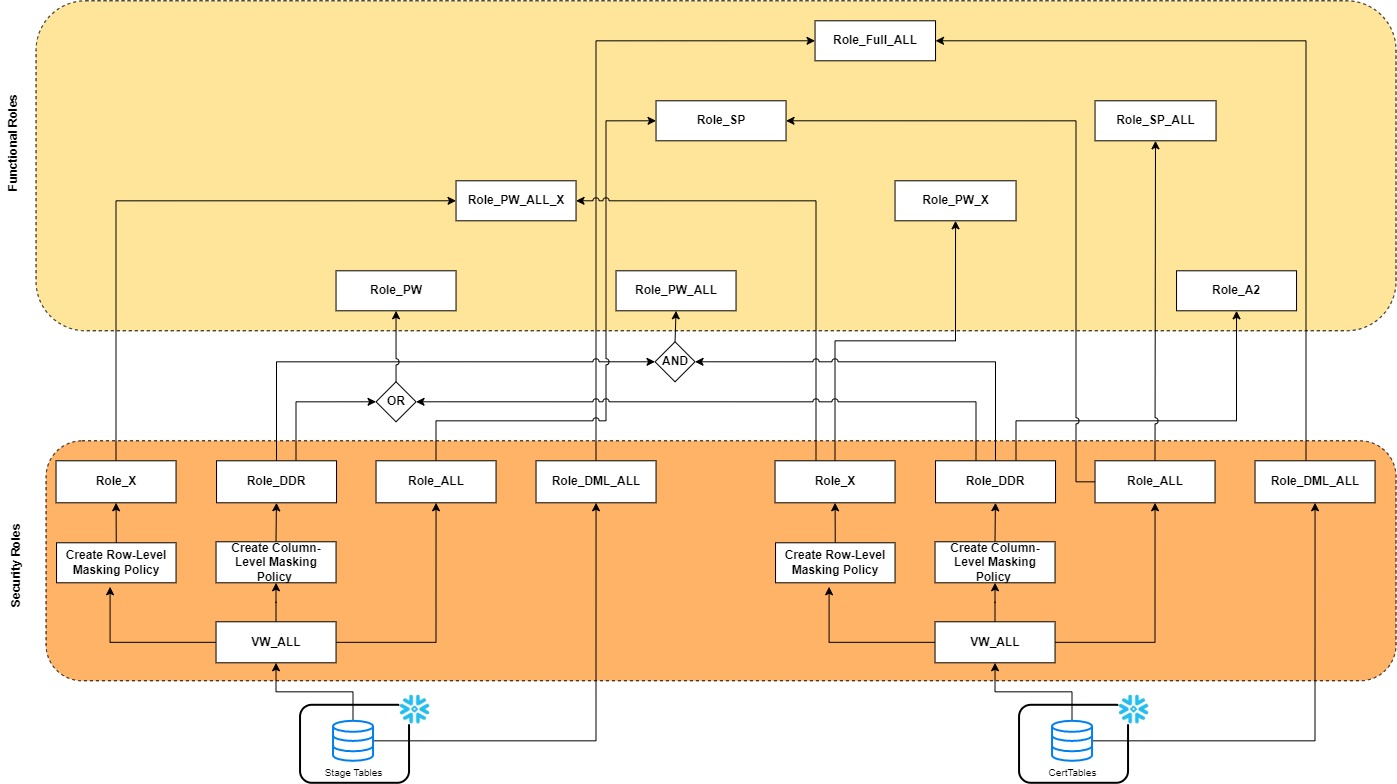
Each database which should sync between Netezza, and Snowflake will have a separate schema in Landing database. We should create Snowflake Landing table, Snowflake Stage and Snow Pipe objects for each table associated with the database in corresponding schema. The Landing tables are Truncate and Load for every run. Below are the suggested object naming conventions. These object names are managed in Control table and verified and deployed on runt time.



# Snowflake Environments

## Environment Setup

# RBAC Configurations



The diagram above illustrates the overall multi-layer RBAC architecture which consistent of the following layers.

1. Create the views on the Stage & Cert tables.
2. Create the Security Role Role\_X with Row-level masking policies to restrict the 2J/confidential data.
3. Create the Security Role\_DDR with Column-level masking policies to restrict the PHI & PII column data.
4. Create the Functional roles such as Role\_A2, Role\_PW, Role\_PW\_ALL\_X,Role\_SP etc., according to the security requirements.
5. Assign the functional roles to the Snowflake users.

## Role Based Column Level Security

Column Level Security in Snowflake allows to mask the columns which contain sensitive/confidential data and it can be achieved by creating a Data Masking policy at the column level.

Dynamic Data Masking uses the masking policies to mask the data in the selective columns in a table or view. This means the actual data in the table is not masked (no static masking) but while querying the table, based on user role/user group snowflake apply masking policies to show either the masked/unmasked actual data for authorized users.

Below is the process to create column-level masking policies:

1. Create the Dynamic masking policies to restrict the PII or PHI columns based on the user role grants
2. Apply the masking policies to the Stage & Cert views
3. Create the roles (Role\_DDR) and grant to the functional roles (Role\_PW & Role\_PW\_All)
4. Assign the functional roles to the Snowflake users.

## Role Based Row Level Security

Snowflake achieves Row Level Security using [Row Access Policies](https://docs.snowflake.com/en/user-guide/security-row-intro.html).  A row access policy is a small, centrally defined procedure that returns a Boolean value (TRUE or FALSE) depending whether the user is allowed to view the specific row.  The steps to defining row level security include:

1. Decide the view needs to be secured.
2. Create a mapping table which includes the role & column to be filtered
3. Create a **Row Access Policy** to implement the rule
4. Deploy the **Row Access Policy** against the Stage & Cert views

## Role Based Hierarchy

Functional Roles such as Role\_A2, Role\_PW, Role\_SP etc., will serve as a security role which will be inherited from security roles i.e., Role\_X, Role\_DD, Role\_All & Role\_Full\_All. At any point, only security roles will be amended/updated when a new view or table or columns or rows needs to be restricted which will be automatically applied to the functional roles which inturn will be applied to Snowflake users.

# General Design and Implementation

## Interfaces with Other Systems

## List of Impacted Inventory

| A) Program | | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| Sr. No. | Name | | | New/Old | Change Summary | Requirement Number |
|  |  | | |  |  |  |
|  |  | | |  |  |  |
|  |  | | |  |  |  |
|  |  | | |  |  |  |
|  |  | | |  |  |  |
|  | | | | | | |
| Sr. No. | | Name | New/New | | Change Summary | Requirement Number |
|  | |  |  | |  |  |
| H) Database | | | | | | |
| Sr. No. | | Name | New/New | | Change Summary | Requirement Number |
|  | |  |  | |  |  |

## Scheduling Impact

# Database Design

# Inputs and Outputs

## Screen Layouts

## Report Layouts

### Input Structure

### Output Structure

# *Non-Development Services* Considerations

# Constraints

# Assumptions and Issues

# Appendices

# Amendment History

| Version | Date | Additions / Modifications | Prepared / Revised By |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Approval of Final Document

| Name | Role / Position | Signature / Date |
| --- | --- | --- |
|  |  |  |