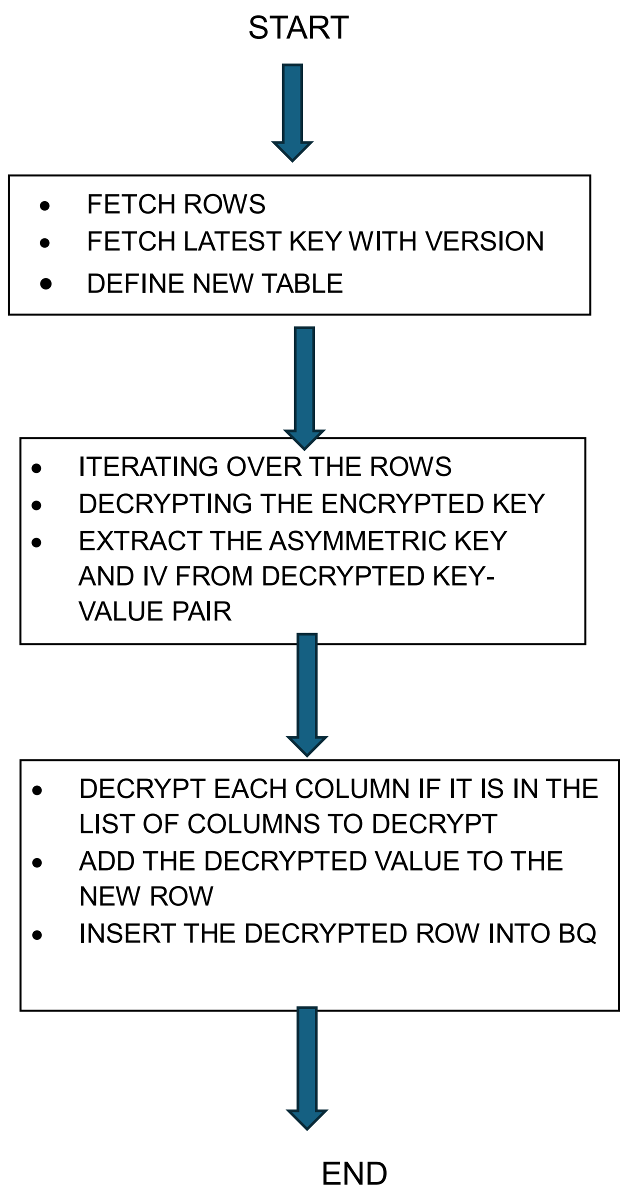
Decrypting Data in Transit/ A Plugin to Decrypt Data with The Payload, Encryption Key [KMS] and Key Version

METHOD 1)→ CREATING DIFFERENT FUNCTIONS USING PYTHON

DECRYPTION WORKFLOW:


START
FETCH ROWS
FETCH LATEST KEY WITH VERSION
DEFINE NEW TABLE
ITERATING OVER THE ROWS
DECRYPTING THE ENCRYPTED KEY
EXTRACT THE ASYMMETRIC KEY AND IV FROM DECRYPTED KEY-VALUE PAIR
DECRYPT EACH COLUMN IF IT IS IN THE LIST OF COLUMNS TO DECRYPT
ADD THE DECRYPTED VALUE TO THE NEW ROW
INSERT THE DECRYPTED ROW INTO BQ
END

Code and Process Explanation for Asynchronous Decryption

This code demonstrates an asynchronous approach to decrypting data stored in BigQuery . It utilizes Cloud KMS for key management and BigQuery for data retrieval and storage.

Key Components:

- **Libraries:**

- `google.cloud.bigquery`: Interacts with BigQuery for data access.
- `google.cloud.kms`: Interacts with Cloud KMS for key management.
- `Crypto.Cipher` and `Crypto.Util.Padding`: Used for low-level cryptographic operations
- `base64`: Encodes/decodes data between bytes and text representations.
- `json`: Parses JSON-formatted data.
- `asyncio`: Enables asynchronous programming for efficient handling of multiple tasks.
- **Functions:**

- `symmetric_decrypt`: Decrypts data using a symmetric key and initialization vector (IV).
- `decrypt_asymmetric` (asynchronous): Decrypts data using an asymmetric key retrieved from Cloud KMS.
- `fetch_latest_key_version` (asynchronous): Retrieves the latest version of a key from Cloud KMS.
- `fetch_rows` (asynchronous): Fetches rows from a BigQuery table.
- `insert_data_into_table` (asynchronous): Inserts data into a BigQuery table.
- `main` (asynchronous): Main function that orchestrates the decryption process.

Process Flow:

1. **Configuration:** Set up project IDs, dataset IDs, table IDs, key details (project, location, ring, and ID), and columns to be decrypted.
2. **Fetch Rows:** Asynchronously retrieve all rows from the BigQuery table containing encrypted data.
3. **Fetch Latest Key:** Asynchronously fetch the latest version of the asymmetric key used for encrypting the symmetric keys from Cloud KMS.
4. **Iterate Through Rows:** Loop through each row retrieved from BigQuery.

- **Decrypt Encrypted Key:** Use the `decrypt_asymmetric` function to decrypt the `encrypted_key` field in the current row using the retrieved asymmetric key. This provides the symmetric key and IV for further decryption.
- **Prepare Decryption:** Parse the decrypted key information (symmetric key and IV) from the JSON format.

- **Decrypt Columns:** For each column specified in `columns_to_decrypt`, use the `symmetric_decrypt` function (**Security Note:** Consider using higher-level libraries) to decrypt the data using the retrieved symmetric key and IV.
- **Create Decrypted Row:** Build a new row containing only the decrypted values for the specified columns.

5. **Insert Decrypted Data:** Asynchronously insert the newly created row with decrypted data into a new BigQuery table.

   Asynchronous Approach:

   This code utilizes asynchronous functions and the `asyncio` library for efficient handling of independent tasks like fetching data and keys. This allows for better performance compared to a synchronous approach, especially when dealing with large datasets or slow network connections.

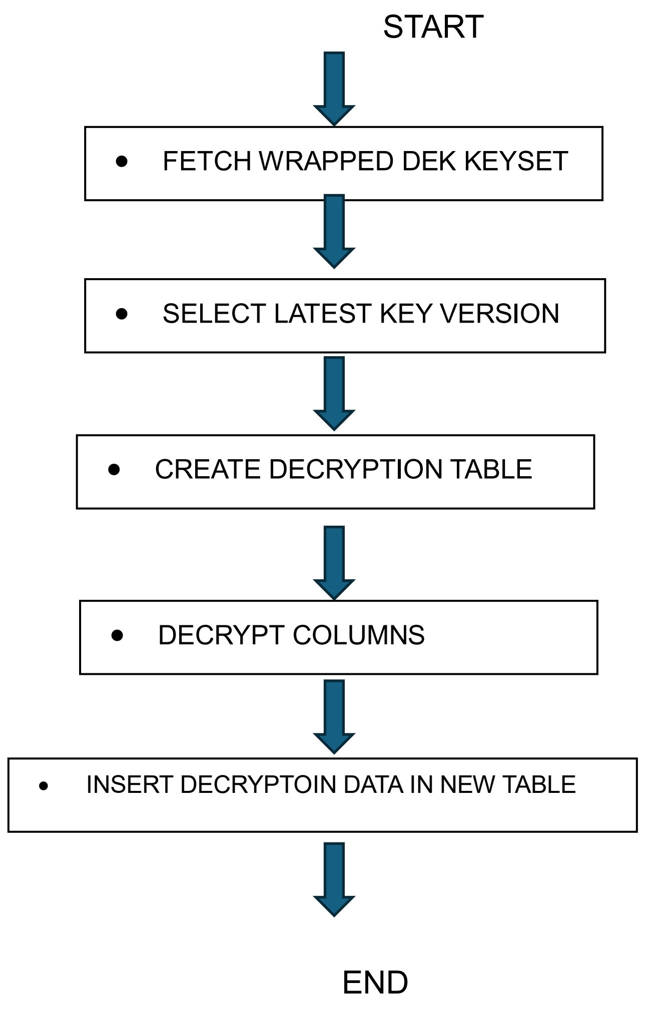
   Security Note:

   While the code demonstrates the decryption process, it's important to be cautious about using the `Crypto` library for production environments. Consider using higher-level, well-maintained cryptographic libraries provided by Google Cloud or other trusted sources to ensure secure implementation.

Method 2→

USING THE INBUILT COLUMN LEVEL ENCRYPTION METHOD OF BIGQUERY:

Workflow:



Decrypting Data with Column-Level Encryption in BigQuery

Here's a breakdown of the decryption process and how it complements the encryption process:

1. Selecting a Key for Decryption

Similar to encryption, decryption also relies on Key Encryption Keys (KEKs) and Data Encryption Keys (DEKs). This code snippet retrieves the wrapped DEK keyset for the specific table you want to decrypt.

- The `selected_keyset_query` builds a SQL query that:
- Declares a variable `selected_keyset` to hold the retrieved keyset in bytes.
- Uses a `SELECT` statement to fetch the `keyset` from the `my_keysets` table in BigQuery.
- Filters the results based on the `id` which matches the `encrypted_table_id`.
- This query is executed using `client.query(selected_keyset_query).result()`.

2. Decrypting the Data

The retrieved `selected_keyset` holds the encrypted DEK used during the original encryption process. This step utilizes the DEK to decrypt the chosen columns in the encrypted table.

- The `decrypt_query` builds another SQL query that:
- Uses `CREATE OR REPLACE TABLE` to define a new table, `decrypted_table_id`, to store the decrypted data.
- The `SELECT` clause specifies the columns to be decrypted.
- For each column in `columns_to_decrypt`, the `DETERMINISTIC_DECRYPT` function is used. This function takes three arguments:
- The key chain formed using `KEYS.KEYSET_CHAIN` with the KEK location and the retrieved `selected_keyset`.
- The column name to be decrypted.
- An empty string (`''`) as padding for the decrypted data.
- The decrypted columns are aliased with their original names using `AS`.
- `FROM` clause specifies the source table, `encrypted_table_id`.
- Finally, the query is executed with `client.query(decrypt_query).result()`.

Key Points:

- This code assumes you have already set up Cloud KMS and created the necessary KEKs and DEKs for encryption.
- Ensure the `project_id`, `dataset_id`, `encrypted_table_id`, `decrypted_table_id`, and `columns_to_decrypt` variables are replaced with your specific values.

In Conclusion:

This approach effectively demonstrates how to decrypt data encrypted with column-level encryption in BigQuery. By utilizing the KEK to access the DEK and then decrypting the chosen columns, you can gain access to the original sensitive data while maintaining security through the encryption process.