# Design Document: AI-Optimized Data Replication from NoSQL to SQL Using Predictive Caching and Load Balancing

## Project Overview

This project focuses on replicating NoSQL data to SQL databases using optimized caching and load balancing mechanisms. The script accomplishes this by:

* Downloading NoSQL data from a Google Cloud Platform (GCP) storage bucket.
* Caching data in Redis for efficiency.
* Converting JSON data into a structured DataFrame format
* Storing the DataFrames in a SQL database.

## 1. System Architecture

1. Data Source (NoSQL on GCP):

* The primary data source is a Google Cloud Storage bucket containing JSON documents that mimic NoSQL data structure.
* Each document is assumed to represent a NoSQL entity, such as a user profile, event log, or configuration entry.

2. Redis Caching Layer:

* A caching layer in Redis reduces data transfer costs by storing previously retrieved NoSQL data.
* Redis also stores DataFrame structures to minimize repeated transformations.

3. SQL Database (Target):

* The SQL database (e.g., MySQL, PostgreSQL) is the destination for structured, normalized data derived from NoSQL JSON documents.
* SQLAlchemy facilitates database communication, allowing for flexible SQL table creation and data insertion.

4. Execution Pipeline:

* The pipeline is controlled by a `nosql\_to\_sql\_pipeline` function that orchestrates data retrieval, caching, transformation, and insertion into SQL.

## 2. Core Components

The code consists of the following major components:

**def** initialize\_storage\_client(service\_account\_key\_path):  
**return** storage.Client.from\_service\_account\_json(service\_account\_key\_path)

1. **Initialize GCP Storage Client:**

This function authenticates and connects to GCP using a provided service account key.

1. **Redis Cache Management:**

redis\_client = redis.StrictRedis(host='localhost', port=6379, db=0, decode\_responses=**True**)

**def** fetch\_from\_cache(cache\_key):  
 """Fetch data from Redis cache"""  
 cached\_data = redis\_client.get(cache\_key)  
 **if** cached\_data:   
 # Return data if found in cache  
 **return** json.loads(cached\_data)  
 **else**:  
 **return** None

fetch\_from\_cache: Retrieves data from Redis using a generated cache key.

**def** save\_to\_cache(cache\_key, data):  
 """Save data to Redis cache"""  
 redis\_client.set(cache\_key, json.dumps(data), ex=3600)   
 # Expires in 1 hour (3600 seconds)

save\_to\_cache: Caches JSON data in Redis with a one-hour expiration.

1. **Download Data from GCP:**

**def** download\_nosql\_files(bucket\_name, prefix, service\_account\_key\_path):  
 cache\_key = f"nosql\_data\_{bucket\_name}\_{prefix}"  
 cached\_data = fetch\_from\_cache(cache\_key)  
 **if** cached\_data:  
 **print**("Data fetched from Redis cache")  
 **return** cached\_data # Return cached data if found

# If not cached, download the data as usual  
 client = initialize\_storage\_client(service\_account\_key\_path)  
 bucket = client.get\_bucket(bucket\_name)  
 blobs = bucket.list\_blobs(prefix=prefix)

nosql\_data = []

`download\_nosql\_files`: Downloads JSON files from the GCP bucket. If data is cached, it skips the download.

**for** blob **in** blobs:  
 file\_data = blob.download\_as\_text()  
 json\_data = json.loads(file\_data)  
 # Append each JSON document to a list  
 nosql\_data.append(json\_data)

# Cache the downloaded data  
 save\_to\_cache(cache\_key, nosql\_data)   
 **return** nosql\_data # Return the freshly fetched data

1. **Data Transformation:**

**def** convert\_json\_to\_dataframes(json\_data):  
 # Create a cache key based on the data  
 cache\_key = f"dataframe\_{hashlib.md5(  
 **str**(json\_data).encode()).hexdigest()  
 }"

cached\_dataframe = fetch\_from\_cache(cache\_key)

if cached\_dataframe:

print("DataFrame fetched from Redis cache")

return pd.DataFrame(cached\_dataframe)

# Normalize JSON data to DataFrame

df = pd.json\_normalize(json\_data)

save\_to\_cache(cache\_key, df.to\_dict(orient='records')) # Cache the dataframe as a list of records

return df # Return the new dataframe

`convert\_json\_to\_dataframes`: Converts JSON data into a Pandas DataFrame, caches the DataFrame in Redis, and ensures that transformed data is available for SQL insertion.

1. **SQL Writing Function:**

**def** write\_to\_sql(dataframe, table\_name, engine):  
 dataframe.to\_sql(table\_name, con=engine, if\_exists='replace', index=**False**)

`write\_to\_sql`: Writes DataFrames into an SQL database, creating new tables or overwriting existing ones.

1. **Orchestration Pipeline:**

**def** nosql\_to\_sql\_pipeline(bucket\_name, prefix, service\_account\_key\_path, database\_uri):  
 # Download NoSQL documents from GCP bucket  
 nosql\_data = download\_nosql\_files(bucket\_name, prefix, service\_account\_key\_path)

# Convert JSON data to DataFrames  
 dataframes = [convert\_json\_to\_dataframes(doc) **for** doc **in** nosql\_data]

# Initialize database engine  
 engine = create\_engine(database\_uri)

# Write each dataframe to a separate SQL table  
 **for** idx, df **in** enumerate(dataframes):  
 table\_name = f'table\_{idx}'  
 write\_to\_sql(df, table\_name, engine)

`nosql\_to\_sql\_pipeline`: Orchestrates the full data retrieval, caching, transformation, and storage pipeline from NoSQL to SQL.

1. **Usage**

# Configure SQLAlchemy engine (to be replaced with our database URI)  
username = input('Enter username: ')  
password = getpass.getpass('Enter password: ')  
host = "127.0.0.1"  
port = "3306"  
database = "SDEProject"  
db\_uri = f'mysql+pymysql://{username}:{password}@{host}/{database}'  
engine = create\_engine(database\_uri)  
service\_key = 'service-account-file.json'  
bucket\_name = 'no\_sql\_files'  
prefix = ''

# Execute the pipeline  
nosql\_to\_sql\_pipeline(bucket\_name, prefix, service\_key, db\_uri)

After specifying database credentials and Redis configuration, the script triggers nosql\_to\_sql\_pipeline.

**JSON files:**

1. **data.json**:
   * Contains astronomical object data with attributes like designation, discovery\_date, h\_mag, moid\_au, and orbit\_class.
   * Each record represents a celestial object and its associated parameters, suitable for analytical and scientific SQL storage.
2. **employee.json**:
   * Contains employee details structured with nested fields for name, age, department, position, salary, and an address dictionary.
   * Each record represents an employee profile that can be stored in a SQL table for HR or management analysis.
3. **student.json**:
   * Contains student information with fields like name, class, subjects, age, and a nested address.
   * Each record represents a student’s academic details, structured for educational databases and suitable for storage in SQL tables.

## 3. Data Flow Diagram

1. Data Request: GCP storage client requests JSON documents from GCP.

2. Cache Check: Redis checks if data exists in cache.

* If cached, fetch data from Redis.
* If not cached, download data from GCP, cache it, and proceed.

3. Data Transformation: JSON data is transformed into a DataFrame.

4. SQL Insertion: DataFrames are saved to SQL tables.

## 4. Caching and Load Balancing Strategy

1. **Redis Caching:**   
   Cached data reduces repetitive downloads and transformation time, optimizing both performance and cost.
2. **Selective Replication:**By only retrieving non-cached or frequently accessed data, the script minimizes redundant operations, which is a foundational element of load balancing.

## 5. Security Considerations

1. **Credentials:**Service account key and database credentials are managed through environment variables or input prompts to prevent hardcoded sensitive information.
2. **Data Caching:**Redis caching is set with an expiration time to avoid storing stale data indefinitely.

## 6. Future Extensions

1. **AI-Enhanced Caching Logic:**Integrate predictive models to improve caching and replication frequency based on anticipated demand.
2. Event-Driven Data Refresh:  
   Implement event triggers on NoSQL updates to refresh cached data selectively.