# Exploring BigQuery and Google Cloud as A Cloud Solution:

# Building a Data Pipeline with BigQuery and Google Cloud Storage

Joanne Senoren

Master of Science, Data Analytics

## Table of Contents

## A. Schema Objects Creation in BigQuery

This paper explores how Google Cloud and BigQuery can adequately create a database solution for Alliah's transaction records. Through this exploration, we seek to answer, "Can Alliah create a data pipeline utilizing the tools provided within the Google Cloud Platform?" The paper discusses the iterative steps of creating an automated data pipeline to an OLAP interface with querying features.

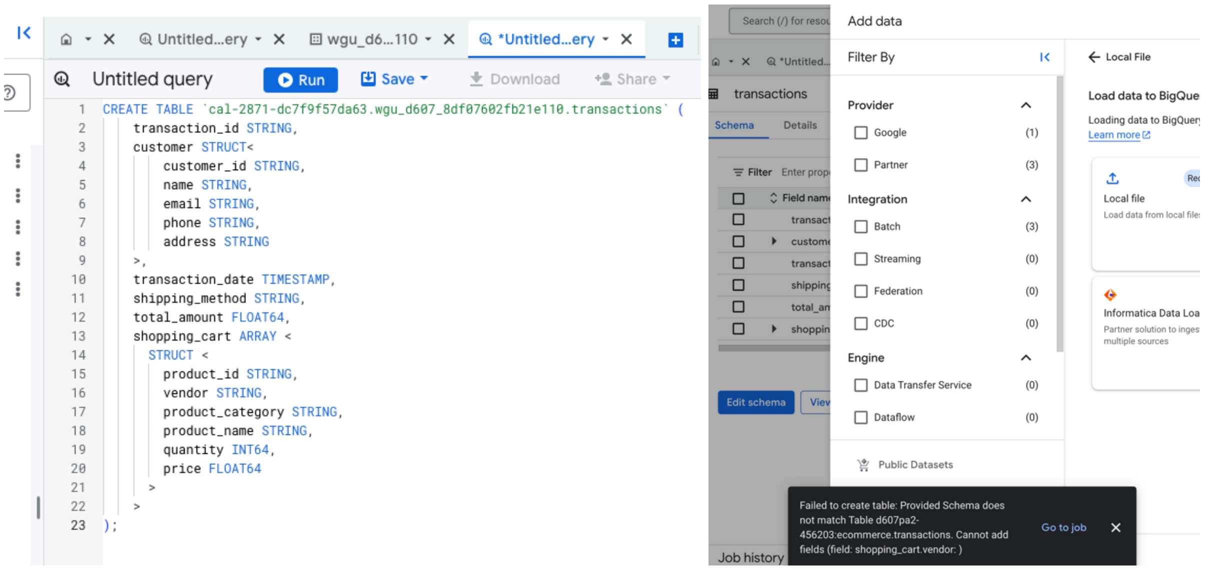
The creation of this data process and pipelines took two iterations, as explained in this paper. It is essential to discuss the iteration process because of limitations and issues that arise when establishing a cloud solution using sample data. Each iteration builds on the previous solution to address the challenges of creating a normalized, production-ready, automated data storage solution.

The first iteration involved data cleaning and transformation in a local Jupyter notebook before uploading the data directly into BigQuery for ingestion. BigQuery consists of a managed data warehouse and analytics service, so the most straightforward solution would be to upload the file directly into BigQuery. Figure 1 below shows the SQL code to create an empty table in BigQuery that follows the general JSON format of Alliah's transaction records. BigQuery's native support for JSON files has a caveat: it can automatically detect the schema for newline delimited JSON (JSONL) but has difficulty processing JSON payload formats—the JSON file needs to be reformatted into JSONL.

Additionally, there was an error when uploading the file to BigQuery after creating the schema (Figure 1) because the key 'vendor' had a trailing white space. The cleaning and transformation process involved fixing the 'vendor' key and reformatting the sample data file to JSONL with Python in a Jupyter Notebook environment (Figure 2).

Figure 1

*BigQuery Empty Table Generation and Upload Error*

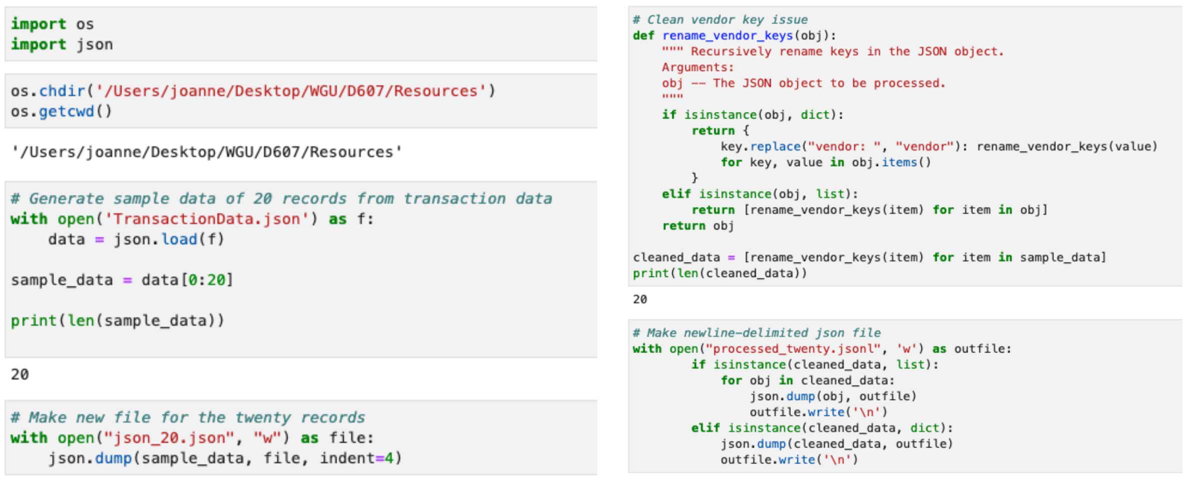


```sql
CREATE TABLE `cal-2871-dc7f9f57da63.wgu_d607_8df07602fb21e110.transactions` (
    transaction_id STRING,
    customer STRUCT<
        customer_id STRING,
        name STRING,
        email STRING,
        phone STRING,
        address STRING
    >,
    transaction_date TIMESTAMP,
    shipping_method STRING,
    total_amount FLOAT64,
    shopping_cart ARRAY <
      STRUCT <
        product_id STRING,
        vendor STRING,
        product_category STRING,
        product_name STRING,
        quantity INT64,
        price FLOAT64
      >
    >
);
```

```
Failed to create table: Provided Schema does
not match Table d607pa2-
456203:ecommerce.transactions. Cannot add
fields (field: shopping_cart.vendor: )
```

Figure 2

*Data Cleaning and Transformation – Jupyter Notebook*



```python
import os
import json
```

```python
os.chdir('/Users/joanne/Desktop/WGU/D607/Resources')
os.getcwd()
```

```
'/Users/joanne/Desktop/WGU/D607/Resources'
```

```python
# Generate sample data of 20 records from transaction data
with open('TransactionData.json') as f:
    data = json.load(f)

sample_data = data[0:20]

print(len(sample_data))
```

```
20
```

```python
# Make new file for the twenty records
with open("json_20.json", "w") as file:
    json.dump(sample_data, file, indent=4)
```

```python
# Clean vendor key issue
def rename_vendor_keys(obj):
    """ Recursively rename keys in the JSON object.
    Arguments:
    obj -- The JSON object to be processed.
    """
    if isinstance(obj, dict):
        return {
            key.replace("vendor: ", "vendor"): rename_vendor_keys(value)
            for key, value in obj.items()
        }
    elif isinstance(obj, list):
        return [rename_vendor_keys(item) for item in obj]
    return obj

cleaned_data = [rename_vendor_keys(item) for item in sample_data]
print(len(cleaned_data))
```

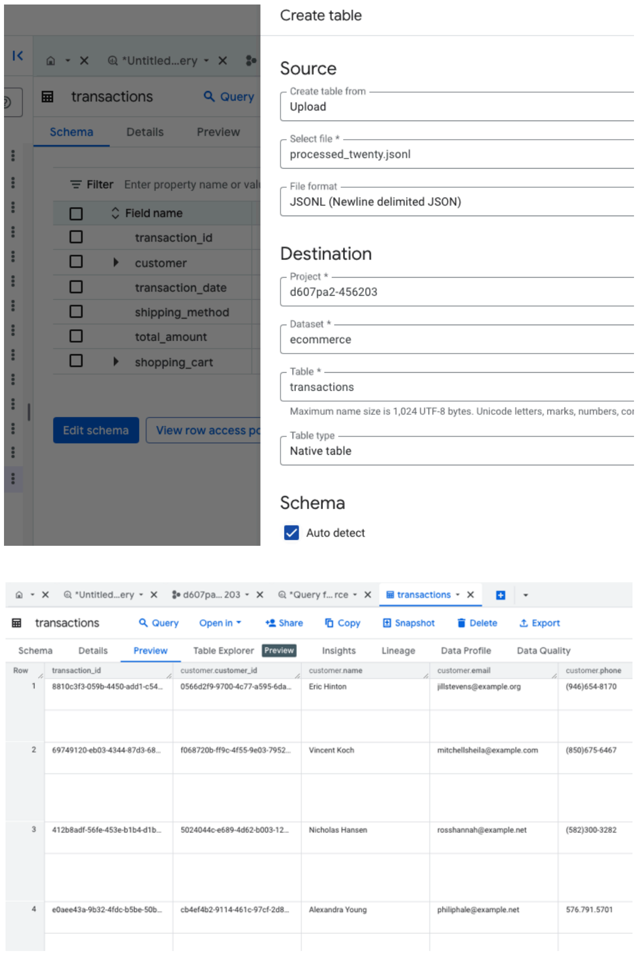
```
20
```

```python
# Make newline-delimited json file
with open("processed_twenty.jsonl", 'w') as outfile:
        if isinstance(cleaned_data, list):
            for obj in cleaned_data:
                json.dump(obj, outfile)
                outfile.write('\n')
        elif isinstance(cleaned_data, dict):
            json.dump(cleaned_data, outfile)
            outfile.write('\n')
```

The first screenshot in Figure 2 shows the code that opens the TransactionData.json file and extracts twenty records from the array, storing them in a variable called sample_data and then writing that list of sample data into its own JSON file. This will be the test data moving forward. The second screenshot shows how a helper function ran inside a for-loop to clean the 'vendor' key. A newline delimited JSON file with the cleaned data was written and saved. From here, the BigQuery console had features to manually upload the data file into the empty table (Figure 3).

Figure 3

JSON File Upload into Empty BigQuery Table and Data Preview



Field name
- transaction_id
- customer
- transaction_date
- shipping_method
- total_amount
- shopping_cart

Create table

Source

Create table from: Upload

Select file *: processed_twenty.jsonl

File format: JSONL (Newline delimited JSON)

Destination

Project *: d607pa2-456203

Dataset *: ecommerce

Table *: transactions

Maximum name size is 1,024 UTF-8 bytes. Unicode letters, marks, numbers, co

Table type: Native table

Schema

☑ Auto detect

| Row | transaction_id | customer.customer_id | customer.name | customer.email | customer.phone |
|---|---|---|---|---|---|
| 1 | 8810c3f3-059b-4450-add1-c54... | 0566d2f9-9700-4c77-a595-6da... | Eric Hinton | jillstevens@example.org | (946)654-8170 |
| 2 | 69749120-eb03-4344-87d3-68... | f068720b-ff9c-4f55-9e03-7952... | Vincent Koch | mitchellsheila@example.com | (850)675-6467 |
| 3 | 412b8adf-56fe-453e-b1b4-d1b... | 5024044c-e689-4d62-b003-12... | Nicholas Hansen | rosshannah@example.net | (582)300-3282 |
| 4 | e0aee43a-9b32-4fdc-b5be-50b... | cb4ef4b2-9114-461c-97cf-2d8... | Alexandra Young | philiphale@example.net | 576.791.5701 |

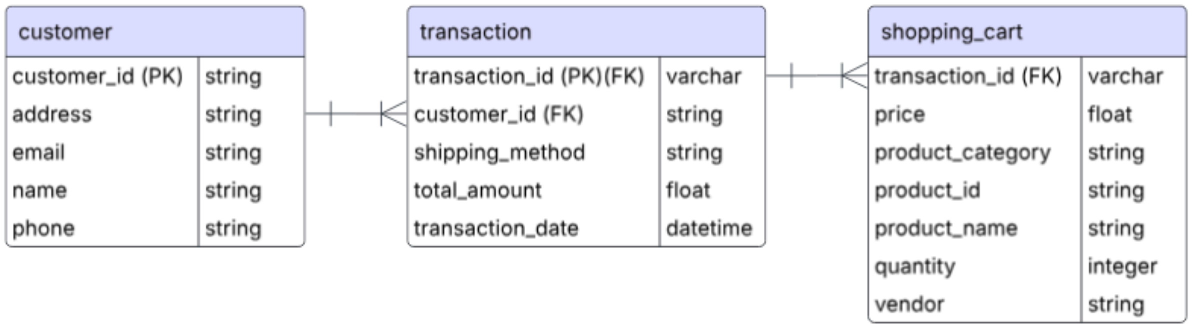
This process is highly inefficient for two main reasons. First, the BigQuery table does not reflect the schema developed in Task 1, which highlights normalized tables to minimize data redundancy. Second, the process involves a lot of manual modifications, which could result in multiple errors. A company such as Alliah will need to process their data into a normalized form for their OLAP needs so that teams only query the needed data. Axel Thevenot (2025) states, “the advantage of this kind of structure resides in the way that each table answers a specific entity.” Additionally, there is no data duplication or redundancy, which lets users update one specific data in one table (Thevenot, 2025).

**1. Create the Required Schema Objects**

The tables in BigQuery need to reflect the proposed schema from Task 1. Thus, the CREATE TABLE code was refactored to reflect the normalized schema in Figure 4. Figure 5 shows the updated code that reflects the creation of three tables to normalize the data.

Figure 4

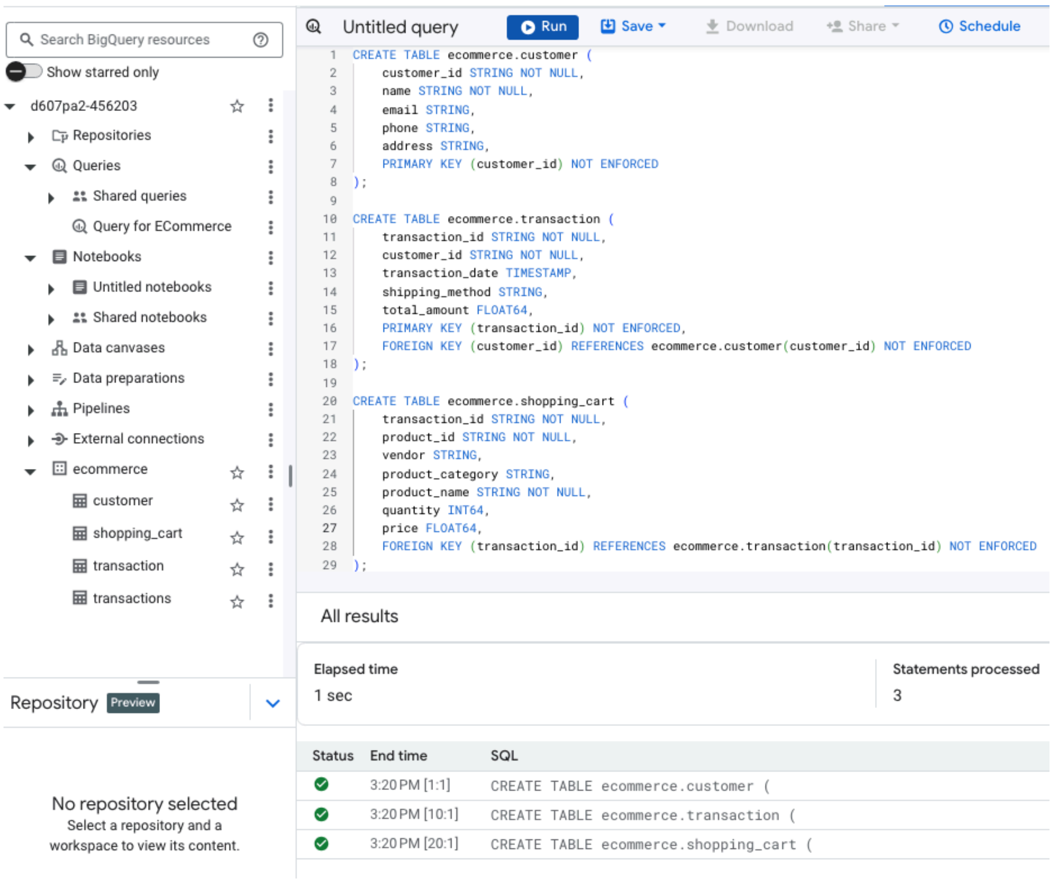
*Proposed ERD*



**1. Documentation of Database Code and Explanation of Components**

Figure 5

*Refactored CREATE TABLE Code*



```
CREATE TABLE ecommerce.customer (
    customer_id STRING NOT NULL,
    name STRING NOT NULL,
    email STRING,
    phone STRING,
    address STRING,
    PRIMARY KEY (customer_id) NOT ENFORCED
);

CREATE TABLE ecommerce.transaction (
    transaction_id STRING NOT NULL,
    customer_id STRING NOT NULL,
    transaction_date TIMESTAMP,
    shipping_method STRING,
    total_amount FLOAT64,
    PRIMARY KEY (transaction_id) NOT ENFORCED,
    FOREIGN KEY (customer_id) REFERENCES ecommerce.customer(customer_id) NOT ENFORCED
);

CREATE TABLE ecommerce.shopping_cart (
    transaction_id STRING NOT NULL,
    product_id STRING NOT NULL,
    vendor STRING,
    product_category STRING,
    product_name STRING NOT NULL,
    quantity INT64,
    price FLOAT64,
    FOREIGN KEY (transaction_id) REFERENCES ecommerce.transaction(transaction_id) NOT ENFORCED
);
```

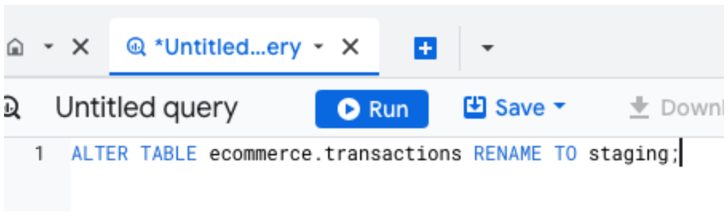
| Elapsed time | Statements processed |
|---|---|
| 1 sec | 3 |

| Status | End time | SQL |
|---|---|---|
| ✓ | 3:20 PM [1:1] | CREATE TABLE ecommerce.customer ( |
| ✓ | 3:20 PM [10:1] | CREATE TABLE ecommerce.transaction ( |
| ✓ | 3:20 PM [20:1] | CREATE TABLE ecommerce.shopping_cart ( |

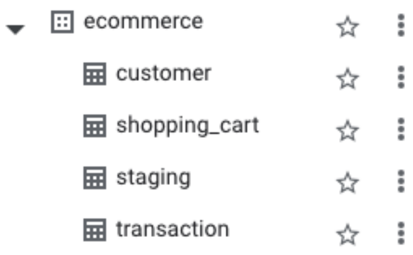
BigQuery does not support enforced primary and foreign key constraints to maintain pricing efficiency since forced key constraints can increase computational costs (*BigQuery Overview*, n.d.). However, using the NOT ENFORCED constraint on primary keys helps to optimize joins and provides relationship documentation (Alamoudi & Zhang, 2023). Consequently, the original transactions table was renamed “staging” to load data from the storage bucket to newly created BigQuery tables (Figure 6).

Figure 6

*Renaming “transactions” Table to “staging”*



```
ALTER TABLE ecommerce.transactions RENAME TO staging;
```



Keeping raw and cleaned data in storage buckets in the cloud is essential to centralizing the data. Cloud functions provide a fully serviced ETL environment and can be automated with trigger events. For example, a trigger event could be someone uploading a raw file into a specific storage bucket. Leveraging the free trial of Google Cloud helped demonstrate the ETL procedure using a cloud function that precludes the database from being populated.

Creating a simple ETL cloud function to clean the 'vendor' field and write a JSONL file helps to maintain data accuracy and consistency that is automatic and efficient (*Cloud Run Functions Overview*, n.d.). The ETL cloud function code for cleaning the 'vendor' field and creating the JSONL file is demonstrated in Figure 7. At this point, a cloud function was created to listen for an event: uploading a raw JSON file into the 'd607pa2-raw' storage bucket. This event triggers the cloud function in Figure 7 to run.

Figure 7

*Clean Vendor Field, Create JSONL file, and Upload to 'd607pa2-export' Bucket*

```python
import functions_framework
from google.cloud import storage
import json

# Helper function to rename keys in the JSON object
def rename_vendor_keys(obj):
    """ Recursively rename keys in the JSON object.
    Arguments:
    obj -- The JSON object to be processed.
    """
    if isinstance(obj, dict):
        return {
            key.replace("vendor: ", "vendor"): rename_vendor_keys(value)
            for key, value in obj.items()
        }
    elif isinstance(obj, list):
        return [rename_vendor_keys(item) for item in obj]
    return obj
```

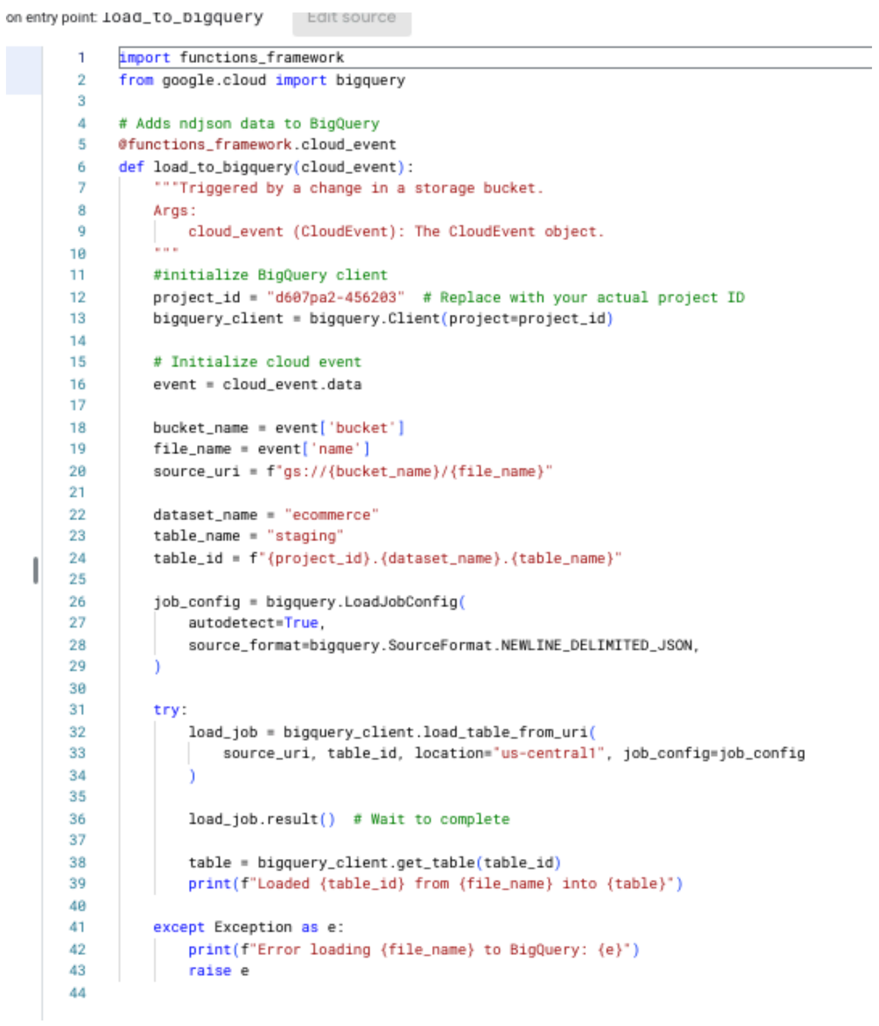
```python
@functions_framework.cloud_event
def rename_vendor_and_upload(cloud_event):
    """Triggered by a change in a storage bucket.
    Args:
        cloud_event (CloudEvent): The CloudEvent object.
    """
    event = cloud_event.data

    bucket_name = event["bucket"]
    file_name = event["name"]

    # change from json to .jsonl
    output_file_name = f"processed_{file_name.replace('.json', '.jsonl')}"

    storage_client = storage.Client()

    print(f"Input file name: {file_name}")
    print(f"Output file name: {output_file_name}")

    input_bucket = storage_client.bucket(bucket_name)
    input_blob = input_bucket.blob(file_name)

    output_bucket_name = "d607pa2-export"
    output_bucket = storage_client.bucket(output_bucket_name)
    output_blob = output_bucket.blob(output_file_name)

    try:
        json_data = json.loads(input_blob.download_as_text())

        cleaned_data = [rename_vendor_keys(item) for item in json_data]
        ndjson_output = "\n".join(json.dumps(item) for item in cleaned_data)

        output_blob.upload_from_string(ndjson_output, content_type="application/jsonl")
        print(f"Uploaded cleaned file to: gs://{output_bucket_name}/{output_file_name}")

    except Exception as e:
        print(f"Error processing {file_name}: {e}")
        raise e
```

The functions framework is a Python library that helps write cloud functions by accessing cloud events, such as changes in cloud storage (*Functions Framework*, n.d.). The google-cloud-bigquery is a Python library with several classes, methods, and attributes that give users tools to communicate and interact with BigQuery (*Python Client Library | Google Cloud*, n.d.). The 'rename_vendor_key' is a helper function that cleans up the 'vendor' field across the JSON payload, which removes white space. This function is called inside the 'rename_vendor_and_upload' function. The 'rename_vendor_and_upload' accesses the event and listens for a change in the specific bucket (d607pa-raw), ingests the raw JSON file, cleans up the 'vendor' field, and then transforms it to a newline delimited JSON file. This JSONL file is then uploaded to the 'd607pa2-export' bucket. This change in 'd607pa2-export' triggers the cloud function to listen for an event in that bucket to load the processed data to BigQuery.

The second cloud function in Figure 8 shows the code for loading the JSONL file to the BigQuery staging table.

Figure 8

*Load File from 'd607pa2-export' Bucket to BigQuery Staging Table*

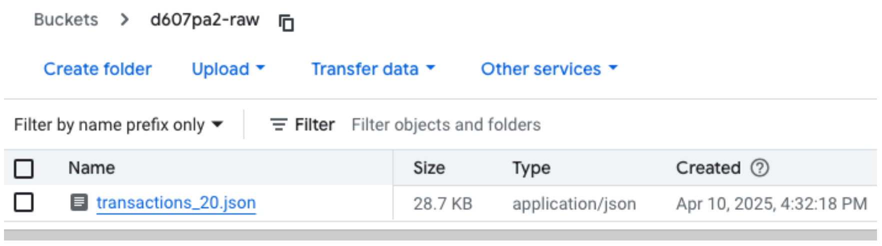


```python
import functions_framework
from google.cloud import bigquery

# Adds ndjson data to BigQuery
@functions_framework.cloud_event
def load_to_bigquery(cloud_event):
    """Triggered by a change in a storage bucket.
    Args:
        cloud_event (CloudEvent): The CloudEvent object.
    """
    #initialize BigQuery client
    project_id = "d607pa2-456203"  # Replace with your actual project ID
    bigquery_client = bigquery.Client(project=project_id)

    # Initialize cloud event
    event = cloud_event.data

    bucket_name = event['bucket']
    file_name = event['name']
    source_uri = f"gs://{bucket_name}/{file_name}"

    dataset_name = "ecommerce"
    table_name = "staging"
    table_id = f"{project_id}.{dataset_name}.{table_name}"

    job_config = bigquery.LoadJobConfig(
        autodetect=True,
        source_format=bigquery.SourceFormat.NEWLINE_DELIMITED_JSON,
    )

    try:
        load_job = bigquery_client.load_table_from_uri(
            source_uri, table_id, location="us-central1", job_config=job_config
        )

        load_job.result()  # Wait to complete

        table = bigquery_client.get_table(table_id)
        print(f"Loaded {table_id} from {file_name} into {table}")

    except Exception as e:
        print(f"Error loading {file_name} to BigQuery: {e}")
        raise e

```

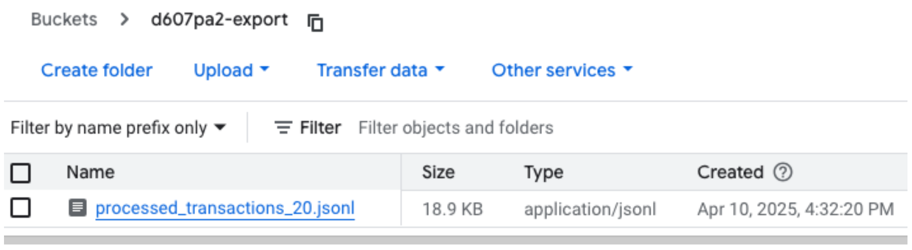
Keeping the ETL procedures within the Google Cloud Functions environment lets us monitor for errors and debug problems in one location. Figure 9 shows the Google Cloud Storage buckets with the raw JSON file (transactions_20) manually uploaded and the generated JSONL file (processed_transactions_20) loaded into BigQuery by the second cloud function.

Figure 9

*Populating Staging Table*



| Name | Size | Type | Created |
|---|---|---|---|
| transactions_20.json | 28.7 KB | application/json | Apr 10, 2025, 4:32:18 PM |



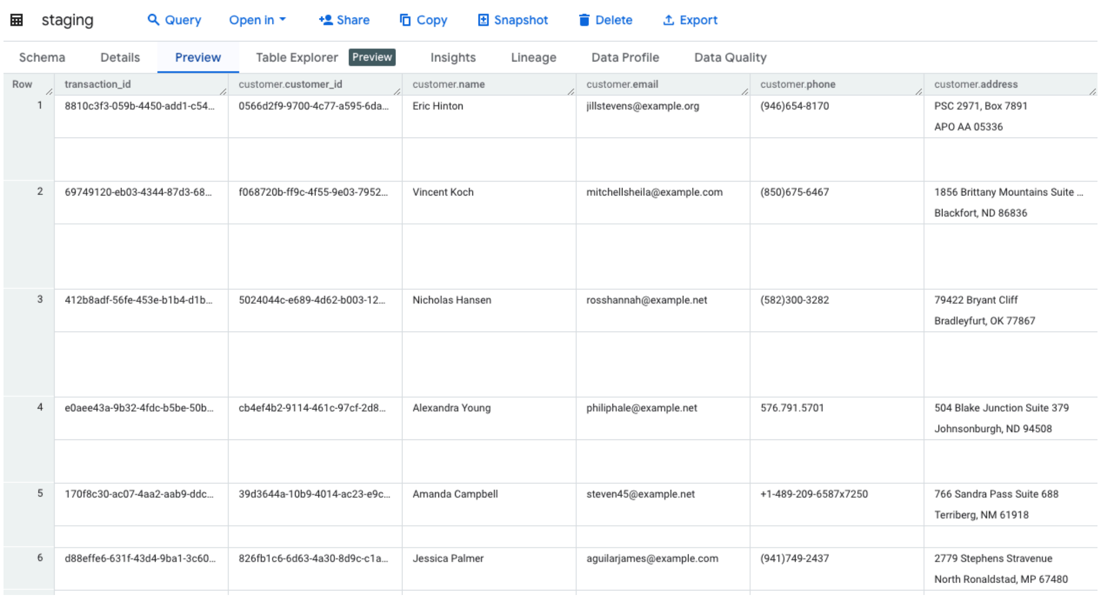
| Name | Size | Type | Created |
|---|---|---|---|
| processed_transactions_20.jsonl | 18.9 KB | application/jsonl | Apr 10, 2025, 4:32:20 PM |

## B. Populating the Database

Figure 10 shows the staging table with the loaded data from the ‘d607pa2-export’ bucket.

Figure 10

*Populated Staging Table*



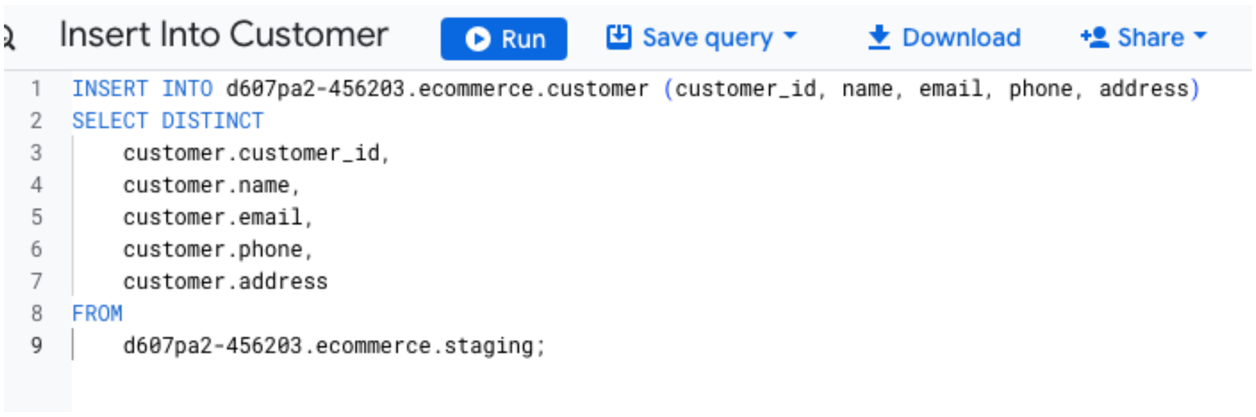
| Row | transaction_id | customer.customer_id | customer.name | customer.email | customer.phone | customer.address |
|---|---|---|---|---|---|---|
| 1 | 8810c3f3-059b-4450-add1-c54... | 0566d2f9-9700-4c77-a595-6da... | Eric Hinton | jillstevens@example.org | (946)654-8170 | PSC 2971, Box 7891<br>APO AA 05336 |
| 2 | 69749120-eb03-4344-87d3-68... | f068720b-ff9c-4f55-9e03-7952... | Vincent Koch | mitchellsheila@example.com | (850)675-6467 | 1856 Brittany Mountains Suite ...<br>Blackfort, ND 86836 |
| 3 | 412b8adf-56fe-453e-b1b4-d1b... | 5024044c-e689-4d62-b003-12... | Nicholas Hansen | rosshannah@example.net | (582)300-3282 | 79422 Bryant Cliff<br>Bradleyfurt, OK 77867 |
| 4 | e0aee43a-9b32-4fdc-b5be-50b... | cb4ef4b2-9114-461c-97cf-2d8... | Alexandra Young | philiphale@example.net | 576.791.5701 | 504 Blake Junction Suite 379<br>Johnsonburgh, ND 94508 |
| 5 | 170f8c30-ac07-4aa2-aab9-ddc... | 39d3644a-10b9-4014-ac23-e9c... | Amanda Campbell | steven45@example.net | +1-489-209-6587x7250 | 766 Sandra Pass Suite 688<br>Terriberg, NM 61918 |
| 6 | d88effe6-631f-43d4-9ba1-3c60... | 826fb1c6-6d63-4a30-8d9c-c1a... | Jessica Palmer | aguilarjames@example.com | (941)749-2437 | 2779 Stephens Stravenue<br>North Ronaldstad, MP 67480 |

## 1. Inserting Transaction Records

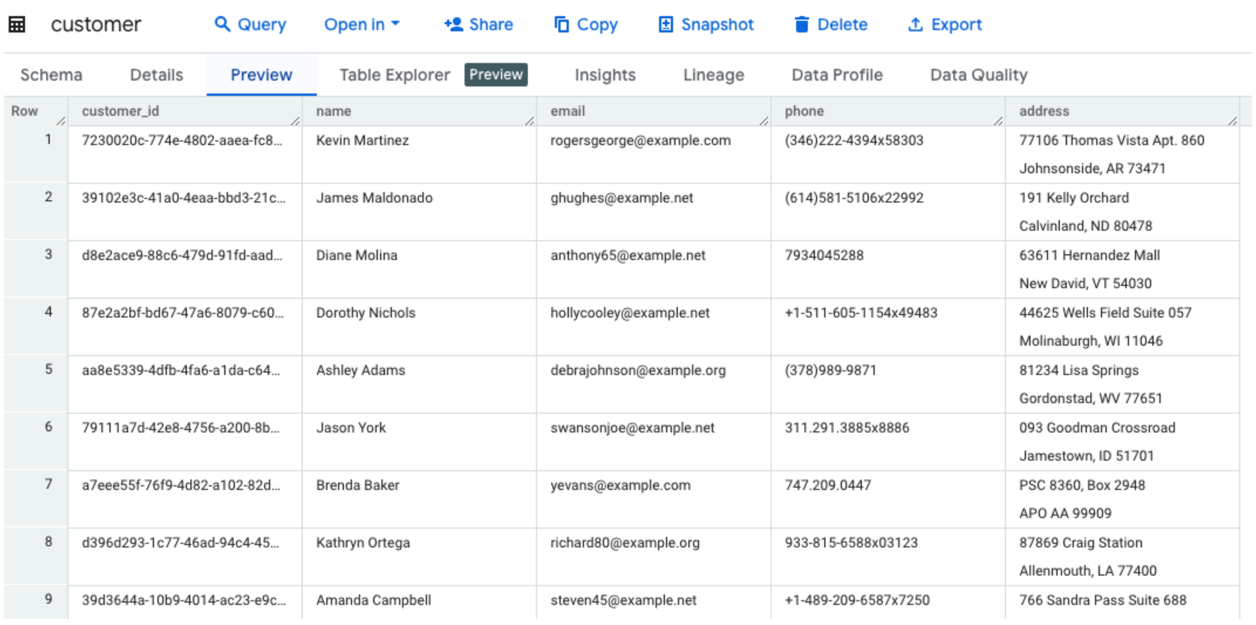
Figures 11 to 13 show the queries used to insert the data into the normalized tables. These queries can be scheduled to add incremental data to the tables as users upload additional JSON files to the storage bucket. The code must be refactored to include a NOT IN clause to avoid duplication since the JSONL loads BigQuery append data to the staging table. Figure 14 shows an example of the NOT IN clauses added to the INSERT INTO queries.

Figure 11

*Insert Into 'customer' Table*



```
INSERT INTO d607pa2-456203.ecommerce.customer (customer_id, name, email, phone, address)
SELECT DISTINCT
    customer.customer_id,
    customer.name,
    customer.email,
    customer.phone,
    customer.address
FROM
    d607pa2-456203.ecommerce.staging;
```

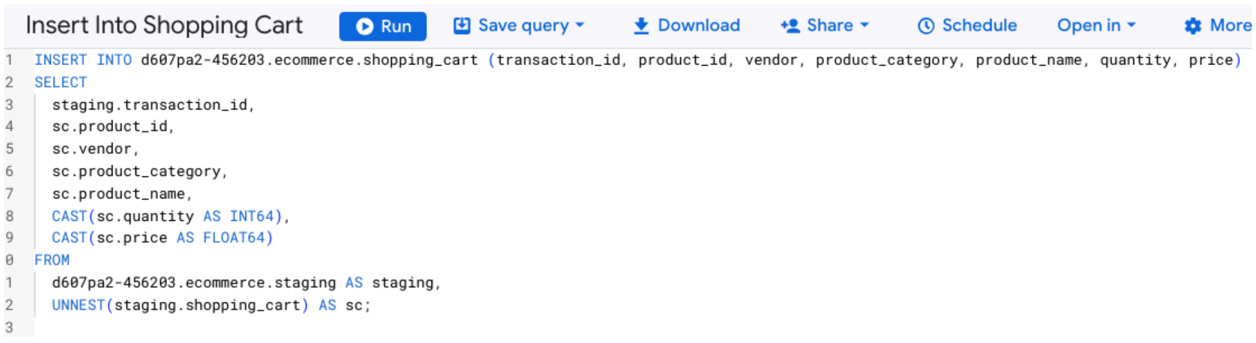


customer

| Row | customer_id | name | email | phone | address |
|---|---|---|---|---|---|
| 1 | 7230020c-774e-4802-aaea-fc8... | Kevin Martinez | rogersgeorge@example.com | (346)222-4394x58303 | 77106 Thomas Vista Apt. 860 Johnsonside, AR 73471 |
| 2 | 39102e3c-41a0-4eaa-bbd3-21c... | James Maldonado | ghughes@example.net | (614)581-5106x22992 | 191 Kelly Orchard Calvinland, ND 80478 |
| 3 | d8e2ace9-88c6-479d-91fd-aad... | Diane Molina | anthony65@example.net | 7934045288 | 63611 Hernandez Mall New David, VT 54030 |
| 4 | 87e2a2bf-bd67-47a6-8079-c60... | Dorothy Nichols | hollycooley@example.net | +1-511-605-1154x49483 | 44625 Wells Field Suite 057 Molinaburgh, WI 11046 |
| 5 | aa8e5339-4dfb-4fa6-a1da-c64... | Ashley Adams | debrajohnson@example.org | (378)989-9871 | 81234 Lisa Springs Gordonstad, WV 77651 |
| 6 | 79111a7d-42e8-4756-a200-8b... | Jason York | swansonjoe@example.net | 311.291.3885x8886 | 093 Goodman Crossroad Jamestown, ID 51701 |
| 7 | a7eee55f-76f9-4d82-a102-82d... | Brenda Baker | yevans@example.com | 747.209.0447 | PSC 8360, Box 2948 APO AA 99909 |
| 8 | d396d293-1c77-46ad-94c4-45... | Kathryn Ortega | richard80@example.org | 933-815-6588x03123 | 87869 Craig Station Allenmouth, LA 77400 |
| 9 | 39d3644a-10b9-4014-ac23-e9c... | Amanda Campbell | steven45@example.net | +1-489-209-6587x7250 | 766 Sandra Pass Suite 688 |

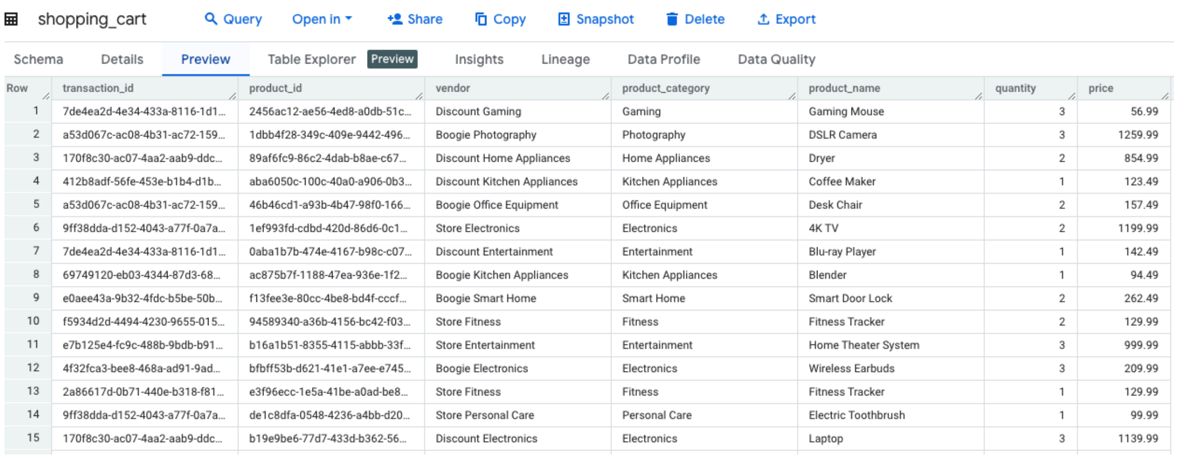
Figure 12

*Insert Into ‘shopping_cart’ Table*



Insert Into Shopping Cart

```
INSERT INTO d607pa2-456203.ecommerce.shopping_cart (transaction_id, product_id, vendor, product_category, product_name, quantity, price)
SELECT
  staging.transaction_id,
  sc.product_id,
  sc.vendor,
  sc.product_category,
  sc.product_name,
  CAST(sc.quantity AS INT64),
  CAST(sc.price AS FLOAT64)
FROM
  d607pa2-456203.ecommerce.staging AS staging,
  UNNEST(staging.shopping_cart) AS sc;
```

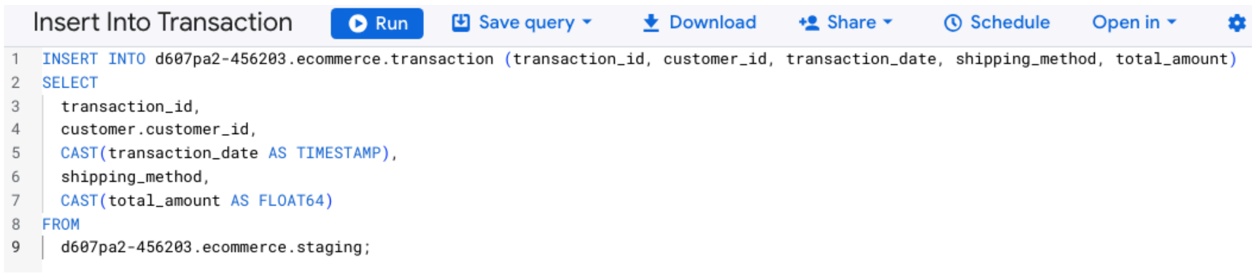


shopping_cart

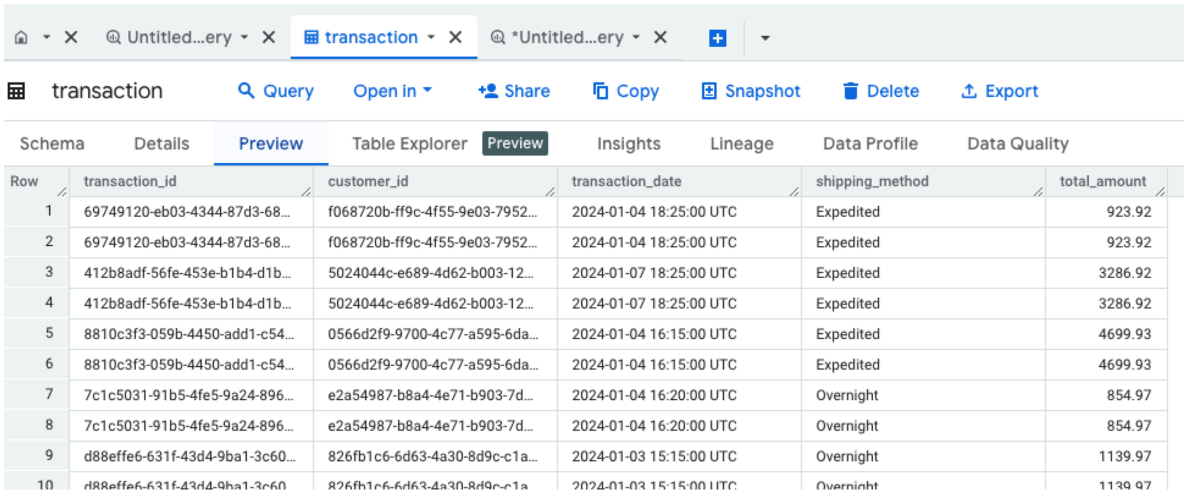
| Row | transaction_id | product_id | vendor | product_category | product_name | quantity | price |
|---|---|---|---|---|---|---|---|
| 1 | 7de4ea2d-4e34-433a-8116-1d1... | 2456ac12-ae56-4ed8-a0db-51c... | Discount Gaming | Gaming | Gaming Mouse | 3 | 56.99 |
| 2 | a53d067c-ac08-4b31-ac72-159... | 1dbb4f28-349c-409e-9442-496... | Boogie Photography | Photography | DSLR Camera | 3 | 1259.99 |
| 3 | 170f8c30-ac07-4aa2-aab9-ddc... | 89af6fc9-86c2-4dab-b8ae-c67... | Discount Home Appliances | Home Appliances | Dryer | 2 | 854.99 |
| 4 | 412b8adf-56fe-453e-b1b4-d1b... | aba6050c-100c-40a0-a906-0b3... | Discount Kitchen Appliances | Kitchen Appliances | Coffee Maker | 1 | 123.49 |
| 5 | a53d067c-ac08-4b31-ac72-159... | 46b46cd1-a93b-4b47-98f0-166... | Boogie Office Equipment | Office Equipment | Desk Chair | 2 | 157.49 |
| 6 | 9ff38dda-d152-4043-a77f-0a7a... | 1ef993fd-cdbd-420d-86d6-0c1... | Store Electronics | Electronics | 4K TV | 2 | 1199.99 |
| 7 | 7de4ea2d-4e34-433a-8116-1d1... | 0aba1b7b-474e-4167-b98c-c07... | Discount Entertainment | Entertainment | Blu-ray Player | 1 | 142.49 |
| 8 | 69749120-eb03-4344-87d3-68... | ac875b7f-1188-47ea-936e-1f2... | Boogie Kitchen Appliances | Kitchen Appliances | Blender | 1 | 94.49 |
| 9 | e0aee43a-9b32-4fdc-b5be-50b... | f13fee3e-80cc-4be8-bd4f-cccf... | Boogie Smart Home | Smart Home | Smart Door Lock | 2 | 262.49 |
| 10 | f5934d2d-4494-4230-9655-015... | 94589340-a36b-4156-bc42-f03... | Store Fitness | Fitness | Fitness Tracker | 2 | 129.99 |
| 11 | e7b125e4-fc9c-488b-9bdb-b91... | b16a1b51-8355-4115-abbb-33f... | Store Entertainment | Entertainment | Home Theater System | 3 | 999.99 |
| 12 | 4f32fca3-bee8-468a-ad91-9ad... | bfbff53b-d621-41e1-a7ee-e745... | Boogie Electronics | Electronics | Wireless Earbuds | 3 | 209.99 |
| 13 | 2a86617d-0b71-440e-b318-f81... | e3f96ecc-1e5a-41be-a0ad-be8... | Store Fitness | Fitness | Fitness Tracker | 1 | 129.99 |
| 14 | 9ff38dda-d152-4043-a77f-0a7a... | de1c8dfa-0548-4236-a4bb-d20... | Store Personal Care | Personal Care | Electric Toothbrush | 1 | 99.99 |
| 15 | 170f8c30-ac07-4aa2-aab9-ddc... | b19e9be6-77d7-433d-b362-56... | Discount Electronics | Electronics | Laptop | 3 | 1139.99 |

Figure 13

*Insert Into 'transaction' Table*



```sql
INSERT INTO d607pa2-456203.ecommerce.transaction (transaction_id, customer_id, transaction_date, shipping_method, total_amount)
SELECT
  transaction_id,
  customer.customer_id,
  CAST(transaction_date AS TIMESTAMP),
  shipping_method,
  CAST(total_amount AS FLOAT64)
FROM
  d607pa2-456203.ecommerce.staging;
```



| Row | transaction_id | customer_id | transaction_date | shipping_method | total_amount |
|---|---|---|---|---|---|
| 1 | 69749120-eb03-4344-87d3-68... | f068720b-ff9c-4f55-9e03-7952... | 2024-01-04 18:25:00 UTC | Expedited | 923.92 |
| 2 | 69749120-eb03-4344-87d3-68... | f068720b-ff9c-4f55-9e03-7952... | 2024-01-04 18:25:00 UTC | Expedited | 923.92 |
| 3 | 412b8adf-56fe-453e-b1b4-d1b... | 5024044c-e689-4d62-b003-12... | 2024-01-07 18:25:00 UTC | Expedited | 3286.92 |
| 4 | 412b8adf-56fe-453e-b1b4-d1b... | 5024044c-e689-4d62-b003-12... | 2024-01-07 18:25:00 UTC | Expedited | 3286.92 |
| 5 | 8810c3f3-059b-4450-add1-c54... | 0566d2f9-9700-4c77-a595-6da... | 2024-01-04 16:15:00 UTC | Expedited | 4699.93 |
| 6 | 8810c3f3-059b-4450-add1-c54... | 0566d2f9-9700-4c77-a595-6da... | 2024-01-04 16:15:00 UTC | Expedited | 4699.93 |
| 7 | 7c1c5031-91b5-4fe5-9a24-896... | e2a54987-b8a4-4e71-b903-7d... | 2024-01-04 16:20:00 UTC | Overnight | 854.97 |
| 8 | 7c1c5031-91b5-4fe5-9a24-896... | e2a54987-b8a4-4e71-b903-7d... | 2024-01-04 16:20:00 UTC | Overnight | 854.97 |
| 9 | d88effe6-631f-43d4-9ba1-3c60... | 826fb1c6-6d63-4a30-8d9c-c1a... | 2024-01-03 15:15:00 UTC | Overnight | 1139.97 |
| 10 | d88effe6-631f-43d4-9ba1-3c60... | 826fb1c6-6d63-4a30-8d9c-c1a... | 2024-01-03 15:15:00 UTC | Overnight | 1139.97 |

## 2. Explanation of Insertion Code and Functions

The INSERT INTO queries programmatically adds selected data from the staging table into the respective tables. All customer data is inserted into the customer table. Since a customer is considered a record within the staging table, the dot notation (e.g., customer.name) specifies that each value within each customer record needs to be exported into the table.

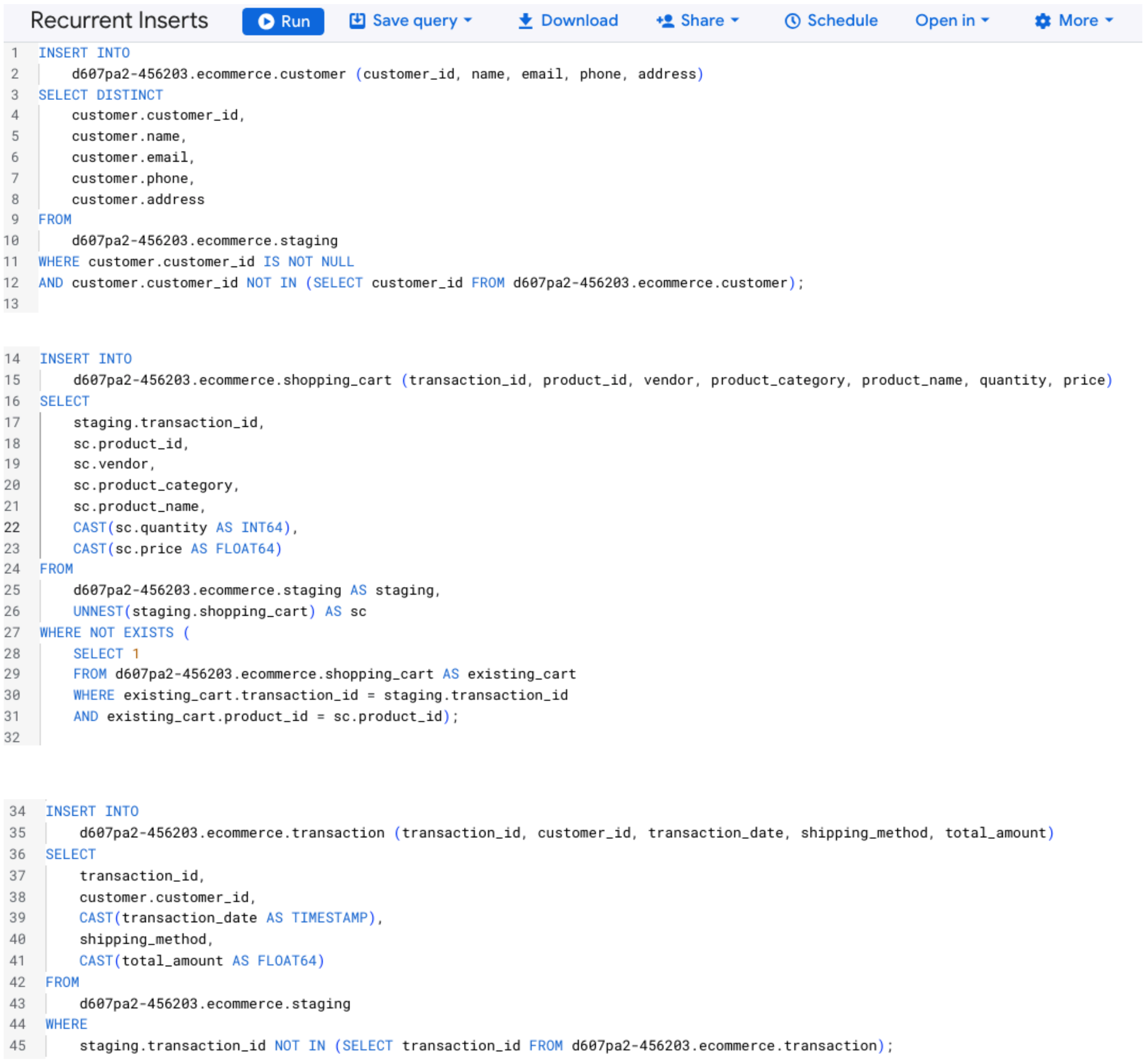
The query to load data into the shopping_cart table involves the UNNEST operator. Since the shopping_cart field in the JSON file is an array of products, we must create a temporary view of the unpacked rows per shopping_cart field. These temporary rows are inserted as rows into the shopping_cart table. In each array object, several attributes, such as product_id, vendor, product_category, and product_name, are selected and inserted into the table as columns. The transaction_id is also inserted as a column for reference in table joining. Price and quantity are cast as FLOAT64 and INT64, respectively, as a data type double-check.

The query for the transaction table acts as the fact table from the proposed ERD. It consists of customer_id and transaction_id and other information that relates directly to each transaction, such as transaction_date, shipping_method, and total_amount. The transaction_date is cast as a timestamp type because it is most likely stored as a string in the JSON files. Casting the data type of timestamp allows users to perform time-related operations.

The code in Figure 14 shows a different version of all table inserts for all subsequent inserts. This code is designed explicitly for recurring inserts. It leverages BigQuery's scheduled query feature to update the database as new data is added continuously. The query schedule should align with the raw data upload frequency to ensure a consistent update cadence.

Figure 14

*Recurring Inserts Code*



```sql
INSERT INTO
    d607pa2-456203.ecommerce.customer (customer_id, name, email, phone, address)
SELECT DISTINCT
    customer.customer_id,
    customer.name,
    customer.email,
    customer.phone,
    customer.address
FROM
    d607pa2-456203.ecommerce.staging
WHERE customer.customer_id IS NOT NULL
AND customer.customer_id NOT IN (SELECT customer_id FROM d607pa2-456203.ecommerce.customer);


INSERT INTO
    d607pa2-456203.ecommerce.shopping_cart (transaction_id, product_id, vendor, product_category, product_name, quantity, price)
SELECT
    staging.transaction_id,
    sc.product_id,
    sc.vendor,
    sc.product_category,
    sc.product_name,
    CAST(sc.quantity AS INT64),
    CAST(sc.price AS FLOAT64)
FROM
    d607pa2-456203.ecommerce.staging AS staging,
    UNNEST(staging.shopping_cart) AS sc
WHERE NOT EXISTS (
    SELECT 1
    FROM d607pa2-456203.ecommerce.shopping_cart AS existing_cart
    WHERE existing_cart.transaction_id = staging.transaction_id
    AND existing_cart.product_id = sc.product_id);


INSERT INTO
    d607pa2-456203.ecommerce.transaction (transaction_id, customer_id, transaction_date, shipping_method, total_amount)
SELECT
    transaction_id,
    customer.customer_id,
    CAST(transaction_date AS TIMESTAMP),
    shipping_method,
    CAST(total_amount AS FLOAT64)
FROM
    d607pa2-456203.ecommerce.staging
WHERE
    staging.transaction_id NOT IN (SELECT transaction_id FROM d607pa2-456203.ecommerce.transaction);
```

## C. Queries

The screenshots below demonstrate the queries required in the performance assessment rubric.

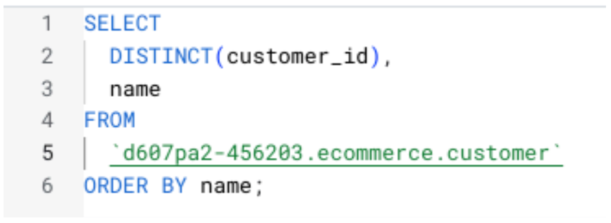
### 1. List All Unique Customers

The code below shows a selection of all unique customer names and customer IDs inside the customer table. This table is helpful when querying a need specifically for customer analysis.

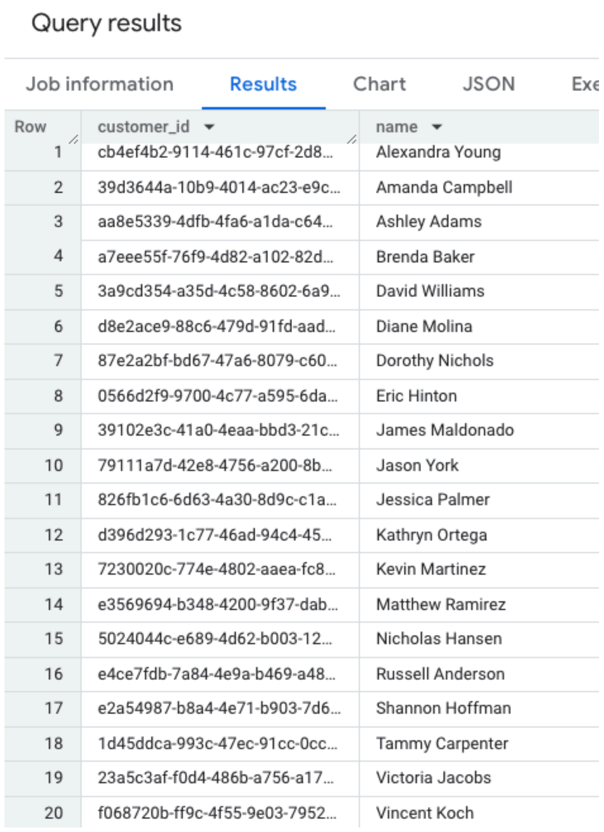
Any customer data updates can also occur inside the customer table without updating additional rows. Alliah should also anticipate that there will be customers who do not necessarily have any products in their shopping carts, and new customers may not have made transactions yet. Therefore, not all customers may exist in the transaction or shopping_cart table.

Figure 15

*List All Unique Customers Query*



```sql
SELECT
  DISTINCT(customer_id),
  name
FROM
  `d607pa2-456203.ecommerce.customer`
ORDER BY name;
```



Query results

| Row | customer_id | name |
|---|---|---|
| 1 | cb4ef4b2-9114-461c-97cf-2d8... | Alexandra Young |
| 2 | 39d3644a-10b9-4014-ac23-e9c... | Amanda Campbell |
| 3 | aa8e5339-4dfb-4fa6-a1da-c64... | Ashley Adams |
| 4 | a7eee55f-76f9-4d82-a102-82d... | Brenda Baker |
| 5 | 3a9cd354-a35d-4c58-8602-6a9... | David Williams |
| 6 | d8e2ace9-88c6-479d-91fd-aad... | Diane Molina |
| 7 | 87e2a2bf-bd67-47a6-8079-c60... | Dorothy Nichols |
| 8 | 0566d2f9-9700-4c77-a595-6da... | Eric Hinton |
| 9 | 39102e3c-41a0-4eaa-bbd3-21c... | James Maldonado |
| 10 | 79111a7d-42e8-4756-a200-8b... | Jason York |
| 11 | 826fb1c6-6d63-4a30-8d9c-c1a... | Jessica Palmer |
| 12 | d396d293-1c77-46ad-94c4-45... | Kathryn Ortega |
| 13 | 7230020c-774e-4802-aaea-fc8... | Kevin Martinez |
| 14 | e3569694-b348-4200-9f37-dab... | Matthew Ramirez |
| 15 | 5024044c-e689-4d62-b003-12... | Nicholas Hansen |
| 16 | e4ce7fdb-7a84-4e9a-b469-a48... | Russell Anderson |
| 17 | e2a54987-b8a4-4e71-b903-7d6... | Shannon Hoffman |
| 18 | 1d45ddca-993c-47ec-91cc-0cc... | Tammy Carpenter |
| 19 | 23a5c3af-f0d4-486b-a756-a17... | Victoria Jacobs |
| 20 | f068720b-ff9c-4f55-9e03-7952... | Vincent Koch |

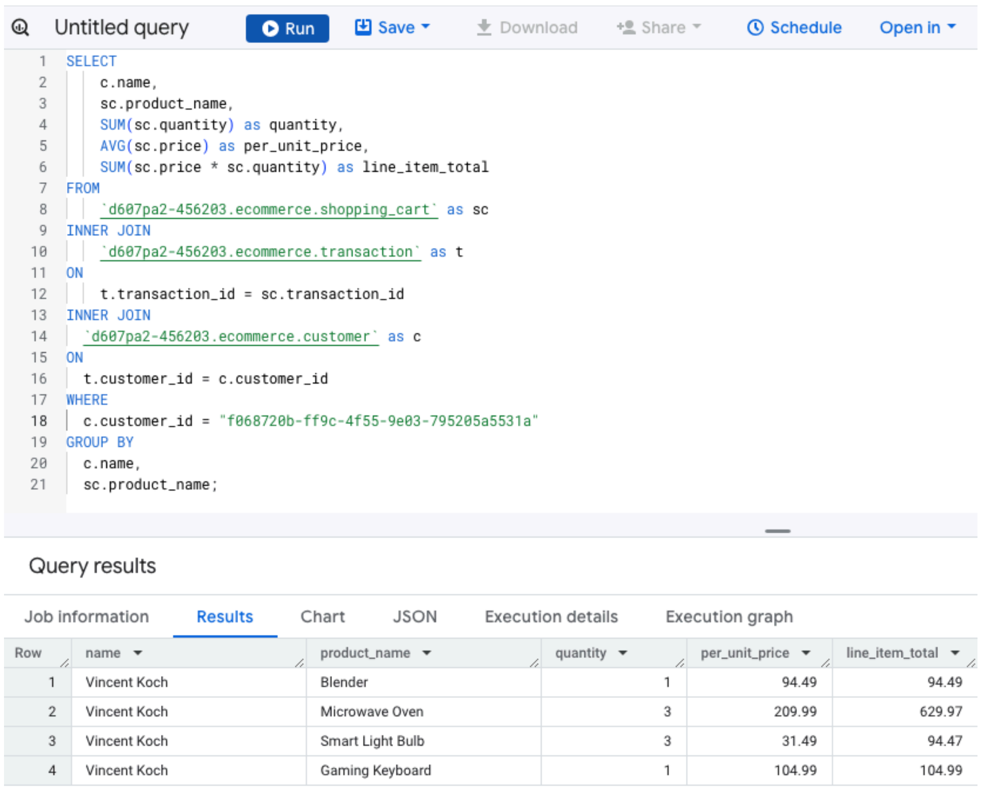
**2. List All Items of One Customer's Shopping Cart**

This query demonstrates how normalized tables must be joined to get the specified list of a customer's shopping cart items. The tables are joined by their respective primary and foreign keys. BigQuery can optimize its joins based on the primary and foreign key references. The JOIN operator combines the shopping_cart table with the transaction table on transaction_id. Then, the combined table undergoes another join with the customer table, joining on the customer id. The column name is derived from the customer table, while the product name comes from the shopping_cart table.

Additional information about a customer's line-item totals demonstrates how data in the table can be manipulated to provide additional information. This customer purchased three microwave ovens, spending over $600. This detail can better inform the marketing team with their data-driven strategy for marketing plans.

Figure 16

*List of Products in Customer's Shopping Cart (by customer ID)*



```sql
SELECT
    c.name,
    sc.product_name,
    SUM(sc.quantity) as quantity,
    AVG(sc.price) as per_unit_price,
    SUM(sc.price * sc.quantity) as line_item_total
FROM
    `d607pa2-456203.ecommerce.shopping_cart` as sc
INNER JOIN
    `d607pa2-456203.ecommerce.transaction` as t
ON
    t.transaction_id = sc.transaction_id
INNER JOIN
  `d607pa2-456203.ecommerce.customer` as c
ON
  t.customer_id = c.customer_id
WHERE
  c.customer_id = "f068720b-ff9c-4f55-9e03-795205a5531a"
GROUP BY
  c.name,
  sc.product_name;
```

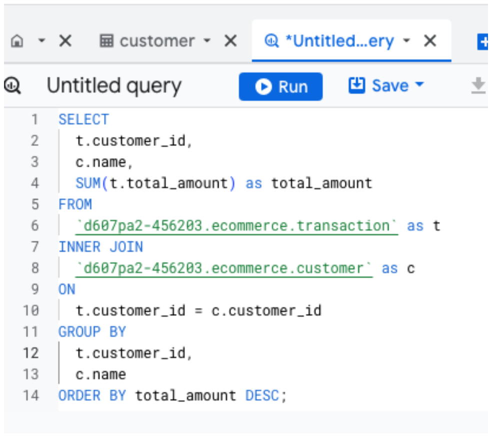
| Row | name | product_name | quantity | per_unit_price | line_item_total |
|---|---|---|---|---|---|
| 1 | Vincent Koch | Blender | 1 | 94.49 | 94.49 |
| 2 | Vincent Koch | Microwave Oven | 3 | 209.99 | 629.97 |
| 3 | Vincent Koch | Smart Light Bulb | 3 | 31.49 | 94.47 |
| 4 | Vincent Koch | Gaming Keyboard | 1 | 104.99 | 104.99 |

## 3. List the Total Purchase Amounts for all Customers in Descending Order

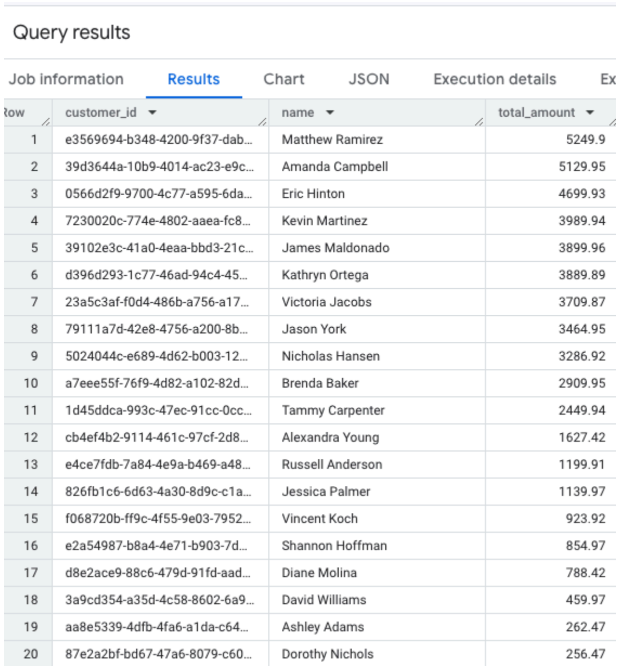
The query below selects the customer_id from the transaction table and the name from the customer table to identify the customer. Then, it calculates the sum of the total amount from the transaction table per customer. It joins the transaction table and the customer table based on customer_id to select only the customers with transactions. The results are grouped by customer ID and name. The total amount is shown in descending order. Matthew Ramirez has spent the most, with a total of $5,249.90.

Figure 17

*Total Amounts by Customer in Descending Order*



```sql
SELECT
  t.customer_id,
  c.name,
  SUM(t.total_amount) as total_amount
FROM
  `d607pa2-456203.ecommerce.transaction` as t
INNER JOIN
  `d607pa2-456203.ecommerce.customer` as c
ON
  t.customer_id = c.customer_id
GROUP BY
  t.customer_id,
  c.name
ORDER BY total_amount DESC;
```



| Row | customer_id | name | total_amount |
|---|---|---|---|
| 1 | e3569694-b348-4200-9f37-dab... | Matthew Ramirez | 5249.9 |
| 2 | 39d3644a-10b9-4014-ac23-e9c... | Amanda Campbell | 5129.95 |
| 3 | 0566d2f9-9700-4c77-a595-6da... | Eric Hinton | 4699.93 |
| 4 | 7230020c-774e-4802-aaea-fc8... | Kevin Martinez | 3989.94 |
| 5 | 39102e3c-41a0-4eaa-bbd3-21c... | James Maldonado | 3899.96 |
| 6 | d396d293-1c77-46ad-94c4-45... | Kathryn Ortega | 3889.89 |
| 7 | 23a5c3af-f0d4-486b-a756-a17... | Victoria Jacobs | 3709.87 |
| 8 | 79111a7d-42e8-4756-a200-8b... | Jason York | 3464.95 |
| 9 | 5024044c-e689-4d62-b003-12... | Nicholas Hansen | 3286.92 |
| 10 | a7eee55f-76f9-4d82-a102-82d... | Brenda Baker | 2909.95 |
| 11 | 1d45ddca-993c-47ec-91cc-0cc... | Tammy Carpenter | 2449.94 |
| 12 | cb4ef4b2-9114-461c-97cf-2d8... | Alexandra Young | 1627.42 |
| 13 | e4ce7fdb-7a84-4e9a-b469-a48... | Russell Anderson | 1199.91 |
| 14 | 826fb1c6-6d63-4a30-8d9c-c1a... | Jessica Palmer | 1139.97 |
| 15 | f068720b-ff9c-4f55-9e03-7952... | Vincent Koch | 923.92 |
| 16 | e2a54987-b8a4-4e71-b903-7d... | Shannon Hoffman | 854.97 |
| 17 | d8e2ace9-88c6-479d-91fd-aad... | Diane Molina | 788.42 |
| 18 | 3a9cd354-a35d-4c58-8602-6a9... | David Williams | 459.97 |
| 19 | aa8e5339-4dfb-4fa6-a1da-c64... | Ashley Adams | 262.47 |
| 20 | 87e2a2bf-bd67-47a6-8079-c60... | Dorothy Nichols | 256.47 |

In conclusion, this paper achieves the practical application and creation of a data pipeline within the Google Cloud Platform for Alliah's transaction records. Let us revisit the research question, “Can Alliah create a data pipeline utilizing the tools provided within the Google Cloud Platform?” By leveraging Google Cloud Storage buckets along with cloud functions for raw data ETL procedures and BigQuery for analytical processing, Alliah can definitively create a data pipeline using the Google Cloud Platform for their business needs.

The iterative process of development, starting with manual cleaning in a local environment to direct BigQuery uploads, which evolved to an automated solution using cloud functions for ETL, highlighted the challenges and necessary changes in establishing a production-ready cloud solution. Ultimately, the normalized schema implemented in BigQuery and its querying capabilities provides Alliah with a scalable and efficient data warehouse for OLAP purposes, enabling valuable insights into their business transactions.

## D1. Resources (Code Used and Tutorials)

1. *Class Blob (3.1.0)*. (n.d.). Google Cloud. https://cloud.google.com/python/docs/reference/storage/latest/google.cloud.storage.blob.Blob#google_cloud_storage_blob_Blob_download_as_text
2. GoogleCloudPlatform. (n.d.). *GitHub - GoogleCloudPlatform/functions-framework-python: FaaS (Function as a service) framework for writing portable Python functions*. GitHub. https://github.com/GoogleCloudPlatform/functions-framework-python
3. *json — JSON encoder and decoder*. (n.d.). Python Documentation. https://docs.python.org/3/library/json.html
4. *Loading JSON data from Cloud Storage*. (n.d.). Google Cloud. https://cloud.google.com/bigquery/docs/loading-data-cloud-storage-json
5. Nestoranaranjo. (2022, January 16). *Convert json to jsonl with python for BigQuery*. Kaggle. https://www.kaggle.com/code/nestoranaranjo/convert-json-to-jsonl-with-python-for-bigquery
6. *Python client library | Google Cloud*. (n.d.). Google Cloud. https://cloud.google.com/python/docs/reference/bigquery/latest
7. Qwiklabs. (n.d.). *Use cloud run functions to load BigQuery | Google Cloud Skills Boost*. https://www.cloudskillsboost.google/focuses/102965?parent=catalog

8. *Specify nested and repeated columns in table schemas*. (n.d.). Google Cloud. https://cloud.google.com/bigquery/docs/nested-repeated#sql
9. TechTrapture. (2025a, January 20). *Cloud Run Functions Explained: evolution from Cloud Function to Cloud Run Function* [Video]. YouTube. https://www.youtube.com/watch?v=V_fI2eAC5g4
10. TechTrapture. (2025b, January 20). *Deploying a Cloud Run Function with GCS Trigger | Step-by-Step Tutorial* [Video]. YouTube. https://www.youtube.com/watch?v=-_9fotoQZbc
11. *Upload objects from a file system*. (n.d.). Google Cloud. https://cloud.google.com/storage/docs/uploading-objects#storage-upload-object-code-sample
12. *Work with arrays*. (n.d.). Google Cloud. https://cloud.google.com/bigquery/docs/arrays
13. *Working with JSON data in GoogleSQL*. (n.d.). Google Cloud. https://cloud.google.com/bigquery/docs/json-data

## D2. References (In-Text Citations)

1. Alamoudi, A., & Zhang, Z. (2023, July 14). Join Optimizations with BigQuery Primary and Foreign Keys. *Google Cloud Blog*. https://cloud.google.com/blog/products/data-analytics/join-optimizations-with-bigquery-primary-and-foreign-keys
2. *BigQuery overview*. (n.d.). Google Cloud. https://cloud.google.com/bigquery/docs/introduction
3. *Cloud Run functions overview*. (n.d.). Google Cloud. https://cloud.google.com/functions/docs/concepts/overview
4. *Functions Framework*. (n.d.). Google Cloud. https://cloud.google.com/functions/docs/functions-framework

5. *Python client library | Google Cloud*. (n.d.). Google Cloud. https://cloud.google.com/python/docs/reference/bigquery/latest

6. Thevenot, A. (2025, March 12). Efficient BigQuery Data Modeling: a storage and compute comparison. *Medium*. https://medium.com/google-cloud/efficient-bigquery-data-modeling-a-storage-and-compute-comparison-ca7f3744e467#:~:text=Choosing%20Your%20BigQuery%20Schema%20Design,-Let's%20summarize%20some&text=Normalized%20schema%20offers%20the%20advantage,this%20data%20can%20be%20costly.