Solution

This document outlines a pragmatic, cloud-based solution for processing, storing and updating transaction data. The solution makes use of various **Amazon Web Services (AWS)** such as **AWS Lambda**, **Amazon Simple Queue Service (SQS)**, **Amazon RDS**, and **Amazon Redshift** to create an efficient and scalable data pipeline.

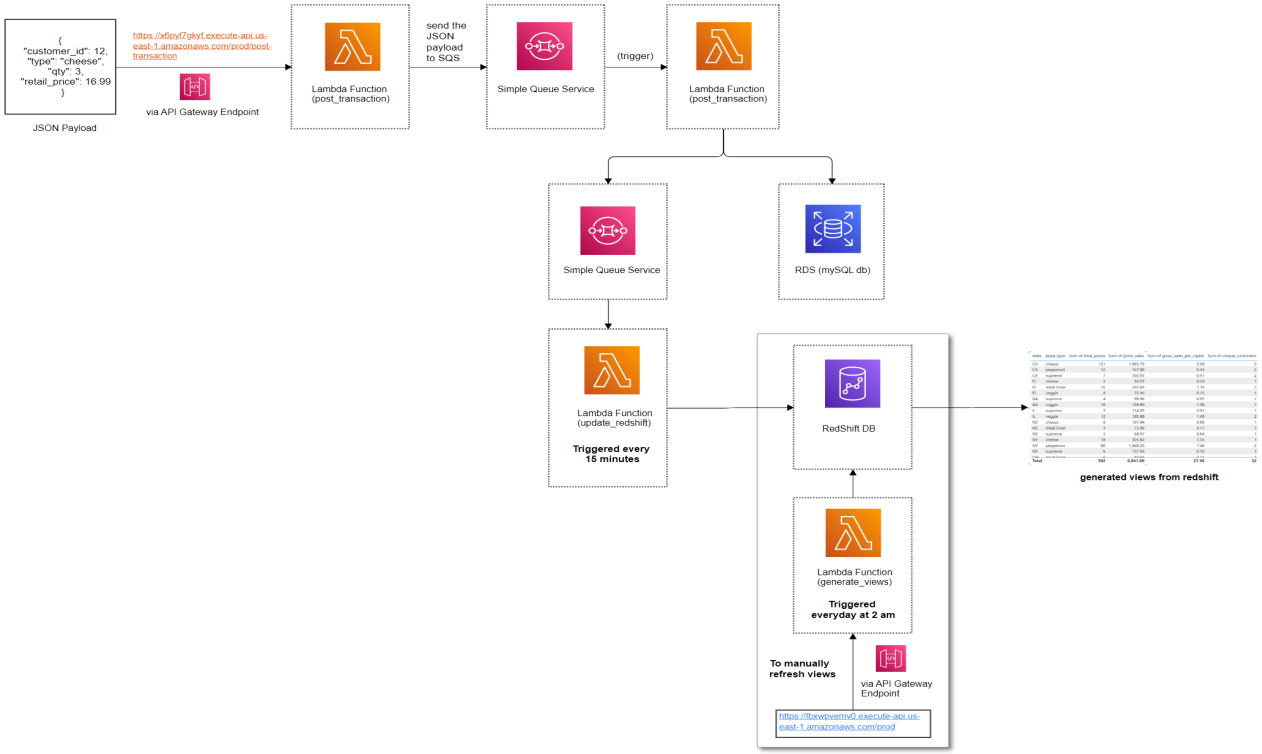
The solution begins with an **API Gateway** that receives a JSON payload of transaction data. This payload is then passed to a Lambda function, which processes the data and pushes it to an SQS queue called "transaction_queue". Another Lambda function, triggered by the SQS queue, then inserts the transaction data into an RDS database and sends the order_id of the newly inserted records to another SQS queue called "redshift_queue".

A **CloudWatch rule** is used to trigger a Lambda function called "update_redshift" **every 15 minutes**, which retrieves all the messages present in the "redshift_queue" and performs an insert operation in the Redshift table. Additionally, materialized views are generated from the Redshift table, and are updated **every 24 hours (at 2:00 AM)** for reporting and analytics purposes. Finally, the details of the order are deleted from the SQS queue.

The whole **data pipeline is designed to meet the low latency requirement of the dashboard** and ensure that adjusting the state filter requires 5 seconds or less for the new data to populate over a stable, high-bandwidth internet connection.

This document provides a detailed explanation of the solution, including the architecture, code snippets and necessary configurations for each step of the data pipeline.

High-level architecture diagram of the solution and explanation:



The architecture can be divided into three main components:

1. **Data Collection**: The solution uses an API Gateway endpoint to accept incoming JSON payloads containing transaction data. This payload is then passed to a Lambda function called "post_transaction" which processes the data and pushes it to an SQS queue called "transaction_queue".
2. **Data Processing**: The "transaction_queue" triggers another Lambda function called "process_transactions" which inserts all the records into an RDS database and pushes the order_id of the newly inserted records to another queue called "redshift_queue".
3. **Data Analysis**: Using CloudWatch rules, a Lambda function called "update_redshift" is triggered every 15 minutes which updates the new records in the Redshift database. All the newly inserted orders are pushed to "redshift_queue" because every 15 minutes, all the newly inserted records are to be updated in the Redshift database.

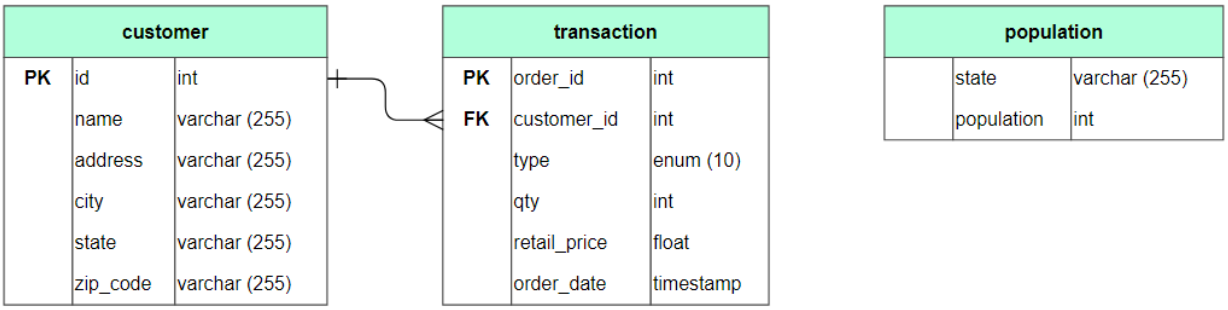
How it works:

- An API Gateway that handles the incoming JSON payload containing transaction details and triggers the first Lambda function, "post_transaction".
- The "post_transaction" Lambda function processes the transaction and pushes it to an SQS queue called "transaction_queue".
- The "transaction_queue" triggers another Lambda function called "process_transactions" which inserts all the records to an RDS database and pushes the order_id of the newly inserted records to another queue called "redshift_queue".
- The "redshift_queue" is monitored by a CloudWatch event that is set to trigger a Lambda function called "update_redshift" every 15 minutes. This function updates the new records to the Redshift table by fetching the details of the order from the RDS database using the order_id present in the SQS queue and inserting it into the Redshift table using the Redshift Data API. After the successful update, the messages in the SQS queue are deleted.
- The RDS database is used to store the transaction details.
- The Redshift table is used to store the transaction details for reporting and analytics purposes.

Note: All the Lambda functions are attached as code snippets below

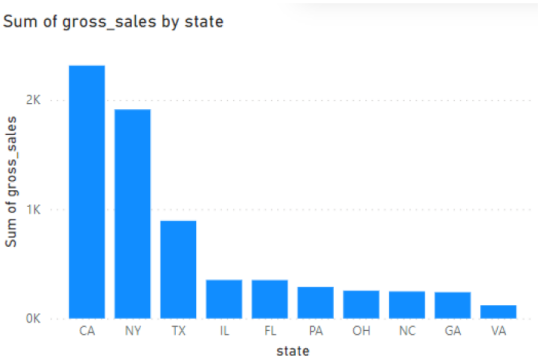
Data Sources:

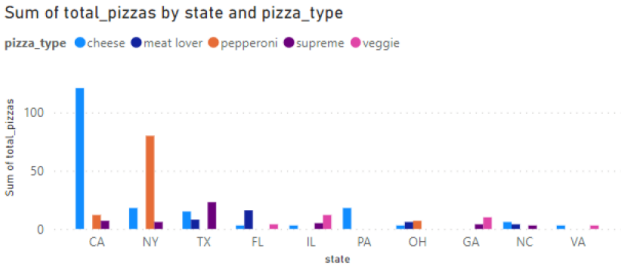
- Population data was sourced from United States Census Bureau.

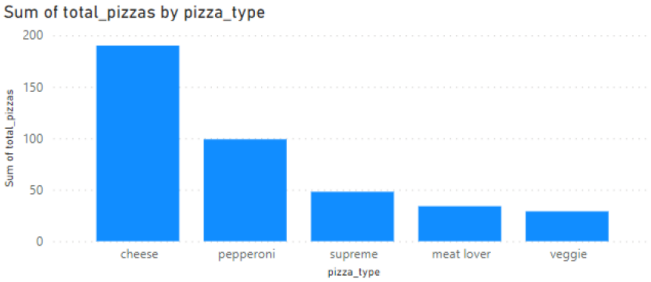


Materialized Views - Power BI (Dashboarding Tool):

state	pizza_type	Sum of total_pizzas	Sum of gross_sales	Sum of gross_sales_per_capita	Sum of unique_customers
CA	cheese	121	1,985.79	5.09	3
CA	pepperoni	12	167.88	0.43	2
CA	supreme	7	160.93	0.41	2
FL	cheese	3	50.97	0.23	1
FL	meat lover	16	243.84	1.10	2
FL	veggie	4	55.96	0.25	1
GA	supreme	4	99.96	0.92	1
GA	veggie	10	139.90	1.28	1
IL	cheese	3	50.97	0.41	1
IL	supreme	5	114.95	0.91	1
IL	veggie	12	185.88	1.48	2
NC	cheese	6	101.94	0.95	1
NC	meat lover	4	75.96	0.71	1
NC	supreme	3	68.97	0.64	1
NY	cheese	18	305.82	1.55	1
NY	pepperoni	80	1,468.20	7.46	2
NY	supreme	6	137.94	0.70	1
Total		**400**	**6,971.00**	**28.59**	**34**







Code Snippets:

post_transaction:

```
import json
import boto3

def lambda_handler(event, context):
    # create an SQS client
    sqs = boto3.client('sqs')

    # To process single JSON payload
    if(type(event) == type({})):
        try:
            sqs.send_message(
                    QueueUrl='https://sqs.us-east-
1.amazonaws.com/351366174178/transaction_queue',
                    MessageBody=json.dumps(event)
                )
            print(f"Successfully sent order to SQS.")
            # return a success response
            return {
                'statusCode': 200,
                'body': json.dumps('Success')
            }
        except Exception as e:
                print(f"Error sending order to SQS: {e}")
```

```
    else:
        #iterate over the array of orders
        for order in event:
            try:
                # send the message to the SQS queue
                sqs.send_message(
                    QueueUrl='https://sqs.us-east-
1.amazonaws.com/351366174178/transaction_queue',
                    MessageBody=json.dumps(order)
                )
                print(f"Successfully sent order to SQS.")
            except Exception as e:
                print(f"Error sending order to SQS: {e}")
        # return a success response
        return {
                    'statusCode': 200,
                    'body': json.dumps('Success')
                }
```

process_transactions:

```
import json
import pymysql
import boto3

def lambda_handler(event, context):
    # Connect to the RDS instance
    connection = pymysql.connect(
        host="database-1.crcjvvpqohqv.us-east-1.rds.amazonaws.com",
        user="admin",
        passwd="Welcome123",
        database="mysql_test"
    )

    # Create a cursor
    cursor = connection.cursor()

    # Create an SQS client
    sqs = boto3.client('sqs')

    # Loop through the "Records" list in the SQS message
    for sqs_message in event['Records']:
        message_body = json.loads(sqs_message['body'])

        try:
            # SQL statement to insert data into the table
            sql_statement = f"INSERT INTO transaction (customer_id, type, qty,
retail_price) VALUES({message_body['customer_id']}, '{message_body['type']}',
{message_body['qty']}, {message_body['retail_price']});"

            # Execute the SQL statement
```

```
            cursor.execute(sql_statement)

            # Commit the changes
            connection.commit()

            # Get the ID of the newly inserted row
            new_order_id = cursor.lastrowid

            # Send the new order ID to another SQS queue
            sqs.send_message(QueueUrl='https://sqs.us-east-
1.amazonaws.com/351366174178/redshift_queue', MessageBody=str(new_order_id))
            print(f"New order ID: {new_order_id}")

        except Exception as e:
            print(f"Error inserting data into RDS or sending to SQS: {e}")
            # handle the exception

    # Close the RDS connection
    connection.close()

    return {
        "statusCode": 200,
        "body": json.dumps("Records inserted successfully!")
    }
```

update_redshift:

```
import json
import boto3
import pymysql

# Create SQS and Redshift Data API clients
sqs = boto3.client('sqs')
redshift_data = boto3.client('redshift-data')

def lambda_handler(event, context):
    # Connect to the RDS instance
    connection = pymysql.connect(
        host="database-1.crcjvvpqohqv.us-east-1.rds.amazonaws.com",
        user="admin",
        passwd="Welcome123",
        database="mysql_test"
    )

    # Create a cursor
    cursor = connection.cursor()

    try:
        all_messages = []
        response = sqs.receive_message(QueueUrl='https://sqs.us-east-
1.amazonaws.com/351366174178/redshift_queue',
```

```
                                        MaxNumberOfMessages=10)
        while 'Messages' in response:
            all_messages.extend(response['Messages'])
            response = sqs.receive_message(QueueUrl='https://sqs.us-east-
1.amazonaws.com/351366174178/redshift_queue',
                                           MaxNumberOfMessages=10)

        # Loop through the messages
        for message in all_messages:
            # Get the order ID from the message body
            order_id = json.loads(message['Body'])
            print(order_id)
            # Fetch the details of the order from the RDS database
            cursor.execute(f"SELECT * FROM transaction WHERE order_id =
{order_id}")
            order_details = cursor.fetchone()
            print(order_details)
            print(order_details[1])
            # Prepare the SQL statement to insert the order details into Redshift
            sql_statement = f"INSERT INTO transaction (order_id, customer_id,
type, qty, retail_price, order_date) VALUES({order_details[0]},
{order_details[1]}, '{order_details[2]}', {order_details[3]}, {order_details[4]},
'{order_details[5]}')"
            # Use the Redshift Data API to execute the SQL statement
            response = redshift_data.execute_statement(
                ClusterIdentifier='redshift-cluster-1',
                Database='dev',
                SecretArn='arn:aws:secretsmanager:us-east-
1:351366174178:secret:redshift_secret-ZdHOu0',
                Sql=sql_statement
            )
            # Get the execution ID
            execution_id = response['Id']
            # Wait for the query to complete
            result_status = None
            while result_status != 'FINISHED':
                describe_response =
redshift_data.describe_statement(Id=execution_id)
                result_status = describe_response['Status']
                print(result_status)
            sqs.delete_message(QueueUrl='https://sqs.us-east-
1.amazonaws.com/351366174178/redshift_queue',
                               ReceiptHandle=message['ReceiptHandle'])
        return {'statusCode': 200, 'body': 'Order details fetched and inserted
into Redshift successfully'}
    except Exception as e:
        return {'statusCode': 500, 'body': f'Error fetching order details from RDS
or inserting into Redshift: {e}'}
    finally:
        # Close the RDS and Redshift connections
        cursor.close()
        connection.close()
```

generate_views:

```
import boto3

def lambda_handler(event, context):
    # Create a Redshift Data API client
    redshift_data = boto3.client('redshift-data')

    try:
        # Refresh the view
        response = redshift_data.execute_statement(
            ClusterIdentifier='redshift-cluster-1',
            Database='dev',
            SecretArn='arn:aws:secretsmanager:us-east-
1:351366174178:secret:redshift_secret-ZdHOu0',
            Sql='REFRESH MATERIALIZED VIEW sales_by_state;'
        )
        # Get the execution ID
        execution_id = response['Id']
        # Wait for the query to complete
        result_status = None
        while result_status != 'FINISHED':
            describe_response = redshift_data.describe_statement(Id=execution_id)
            result_status = describe_response['Status']
            print(result_status)

        return {'statusCode': 200, 'body': 'View refreshed successfully'}

    except Exception as e:
        return {'statusCode': 500, 'body': f'Error creating view: {e}'}
```

Conclusion:

In conclusion, the proposed solution for pizza delivery service includes a data pipeline that integrates streaming and static data from various sources to feed the data warehouse and power a reporting tool. The data pipeline starts with the transactional data streaming in real-time via an API, which is then joined with the customer data from a RDBMS. The final record is then outputted into a data warehousing tool, which stores all transactional records for historical analysis. The data warehouse table feeds into the data source used for the reporting tool, which contains the summary metrics as required by the analytics team.

The key features of the solution include:

- Real-time integration of streaming and static data from various sources
- Data warehousing for historical analysis
- Low latency for the reporting tool with a refresh of the data every day at 2:00 am
- Compliance with the SLA of 15 minutes for transactional data to be available in the data warehouse

This document provides a detailed explanation of the architecture, code snippets, and necessary configurations for each step of the data pipeline. With this solution, the analytics team can easily create a report based on orders and customer information, allowing them to select a single US state and display the top 3 best-selling pizza combinations, as well as other relevant data such as total number of pizzas sold, gross sales, and number of unique customers.