E-Commerce Order Processing

And Aggregation

- Payal Yadav

Contents

[Problem Statement: 2](#_Toc187628559)

[Proposed Design: 2](#_Toc187628560)

[Detailed Description: 2](#_Toc187628561)

[Design Reasoning: 4](#_Toc187628562)

# Problem Statement:

* Poll messages from the SQS queue using Localstack’s endpoint.
* For each valid order:
  + Update Redis with user-level aggregates: increment the user’s total order count
  + and add the order\_value to that user’s cumulative spend.
* Update global aggregates: total number of orders processed and total revenue across all users.
* If an order is invalid or missing data, decide on a handling strategy (e.g., log and skip).

# Proposed Design:

All the orders are received in the SQS queue. A SQS consumer continuously polls the SQS queue for newer messages. As soon as a new message/order is received in the queue, it performs the necessary validation. If the validation is successful, it pushes the data into the redis DB after making some calculations as per the user and global.

A diagram of a process flow

Description automatically generated

# Detailed Description:

* **Data In Redis:**

A diagram of a software flowchart

Description automatically generated

* + First poll the messages from the sqs queue and the validate the orders.
  + Skip and delete the order if below are the validations have failed:
    - If user\_id, order\_id is null
    - If order\_value is not in int or float format
    - if order\_value is not equal to the quantity \* amount.
  + If all the validations are passed successfully then process the order for 2 different keys: user\_stats, global stats
  + For user\_stats key for each users:
    - If key doesn’t exists then set 1 for order\_count and order\_amount as total\_spend.
    - If it exists, then increment the order\_count with 1 and add the order\_amount to total\_spend for that user.
  + For global stats key :
    - If key doesn’t exists then set 1 for order\_count and order\_amount as total\_spend.
    - If it exists, then increment the order\_count with 1 and add the order\_amount to total\_spend for that user
  + Now push both the aggregated values for user and global\_stats to redis.
* **API Endpoints:**
  + For user\_stats API, get the order\_count, total\_spend from redis based on user\_id and return the result to the calling application.
  + For global\_stats API, get the total\_order, total\_revenue till now for the store from redis and return the result to the calling application.

## Design Reasoning:

* + Since the problem statement was to find out the total spends by a particular user and global sales of the store,
  + it was necessary to store the data on the basis of per customer as well as global for the store.
  + At the time of storing the data into Redis DB, we aggregated the data in redis for both the user and global to keep the data up-to-date.
  + When the rest api is called to fetch the data about any particular user or global data, we just need to get the data having the key as user\_id or global based on the query.
  + This makes the response to rest api call within milliseconds as we have stored the data pre-calculated and using the same key as in the query.