# Design, Challenges & Approach

Masum Rahman

November 2020

## 1 Design Overview

### 1.1 Technology used

| | |
|---|---|
| **Back-end Server:** | Django |
| **Rest Framework Swagger Documentation:** | drf-yasg |
| **Containerization:** | Docker |

### 1.2 Simplification and Assumptions:

The problem can be designed by having following three tables.

- Driver Info (PK: Supply ID)
- Order Info (PK: Order ID)
- Supply Order Table ( FK:Supply ID, Order ID)

However, as our query is only related to last table, only last table is integrated in our model and the rest of the two tables are assumed to exist.
If any driver id is not present in our table (i.e. driver with 0 orders) we assume it to be an unavailable driver and throw exception accordingly.

### 1.3 Schema

| *Supply_Order* | |
|---|---|
| *puid* | *String* |
| *timestamp* | *DateTime* |
| *supply_id* | *long int* |
| *order_id* | *string* |
| *status* | *ChoiceField* |

### 1.4 Dummy Test Data Info

For unit-testing and seeding purposes, we use the following dataset. This dataset tries to cover all possible cases. We insert all completed orders first and then insert cancelled orders with ascending order of timestamp.
So, if we have 85 completed orders and 25 cancelled orders, the last 100 orders will contain 75 completed orders and 25 cancelled orders.

| Supply_ID | Num_Completed | Num_Cancelled | Total | Completion Rate |
|---|---|---|---|---|
| 1000 | 85 | 25 | 110 | 0.75 |
| 1001 | 70 | 40 | 110 | 0.6 |
| 1002 | 50 | 50 | 100 | 0.5 |
| 1003 | 50 | 30 | 80 | 0.85 |
| 1004 | 50 | 0 | 50 | 0.85 |

Table 1: Dummy Dataset Distribution

**For Rest API endpoints, Please refer to the homepage (http://127.0.0.1:8000/)**

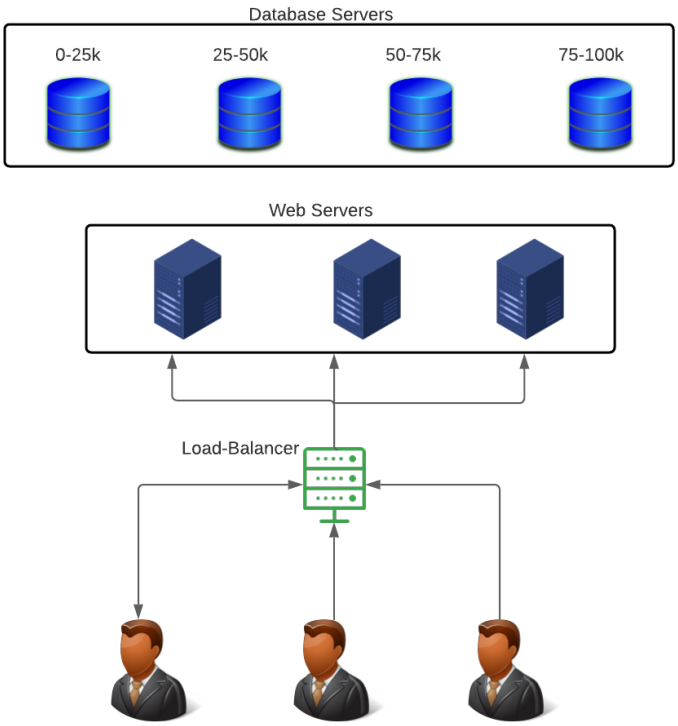
## 2 Bonus Point:

### 2.1 Scaling

This part asks us to scale the previous version so that concurrent 2-3k drivers can be notified. Database will contain more than 100k drivers.

A high level system design is provided below for this problem.

Let's first see what we can do from a server-architecture point of view. To address concurrency, few things can be done. Instead of one single machine, we can deploy our server to multiple machines. And in each machine multiple parallel processes will be spawned to serve requests.
Instead of one single database server, we can have multiple database server. Each server will contain entry within a range. For example DB server 1 contains the user info with id 0-25k. Then all query related to that id will be redirected to that db server.
Following picture depicts the whole idea.




Now let's see what optimization we can do at database design.
We can cache the calculated result ( completion rate, message) so that for next notification we can serve from cache and no need to make a query again. However, if any order insertion happens in between notification issuing, we need to invalidate cache.

If we don't mind having a bit older completion rate, we can create a table which contains a pre-calculated completion rate. And we can update the table by scheduling a crone service at certain interval. Then for notification we don't have to calculate completion every time, instead one single query to this table will give us the result.

## 2.2 Dockerize

Implemented as per document.

# 3 Challenges & Solutions

## 3.1 Indexing

As there will be lots of queries on the basis of supply_id, we have a secondary index on it.

## 3.2 Cascaded filtering

**Problems:** There might be cascaded filtering like, first filter by driver-id and then get the last 100 or n rows. Will there be multiple hits?

**Solutions:** Django itself takes care of it by lazy query evaluation. So cascading filters don't end up in multiple hits rather query gets evaluated only when queryset items are accessed.

## 3.3 Optimizing Database Traffic

**Problems:** There are 4-5 columns which can be even more in real production. Do I need to fetch all columns?

**Solutions:** For this particular problem we only need the status column filtered by driver-id. And that has been done by **.values_list('status')** function. It only fetches status column.

## 3.4 Invalid/Unavailable Driver Id

**Problems:** There might be invalid driver id like 'a23b' or legit driver id but not present in database. How to handle them?

**Solutions:** For non integer id, input validation is done at the client side. For valid integer id, we might want to use **.exists()** to see if present in database and if not then return an exception. And if present we would continue filtering by driver-id with another database query.

Although django optimizes **exists()** for presence checking, because of two database hits, it will eventually turn out to be costly. That's why .exists() has been avoided. Instead queryset length has been calculated by iterating once and returns an exception if length is 0.