A CASE STUDY

BY

FIDANSOY, ALTAR

Contents

List of Tables

List of Figures

List of Listings

1   Data Generation

The location data is gathered from Germany Cities Database in order to generate customer data with actual latitude and longitude information. The data set contains 513 German cities and each city is considered as a customer in this study. Therefore, a unique customer id is assigned to each city. Values in starting date and number of panels fields are randomly generated and randomization limits are set between '2022-01-01' - '2023-03-01' (two months) for starting date and '12' - '66' for number of panels fields. Additional columns such as city, state, are also added into table just for data visualization in Section 4. However, they are not used in any calculation. Table 1 shows the small portion of the created customer data but entire data set can be found in the attachment or in the git repository.

customer id	city	latitude	longitude	start date	# of panels
1	Aachen	50.7762	6.0838	2023-01-01	52
2	Aalen	48.8372	10.0936	2023-01-28	33
3	Achern	48.6314	8.0739	2023-02-22	66
4	Achim	53.013	9.033	2023-01-08	54
5	Ahaus	52.0794	7.0134	2023-02-22	54
6	Ahlen	51.7633	7.8911	2023-02-07	59
7	Ahrensburg	53.6747	10.2411	2023-02-17	18
8	Aichach	48.45	11.1333	2023-02-12	41
9	Albstadt	48.2119	9.0239	2023-01-18	13
10	Alsdorf	50.8744	6.1615	2023-02-13	40

Table 1: Sample Customer Data.

The second requested data set was for team details and each team must have availability range and skill level fields. Three different time periods are defined ('2023-01-01 / 2023-02-01', '2023-01-15 / 2023-02-15' and '2023-02-01 / 2023-03-01') and assigned to each team randomly. Additionally, predefined skill levels are added into the data set as Table 2 displays.

team id	availability from	availability till	skill level
1	2023-01-01	2023-02-01	1
2	2023-01-15	2023-02-15	1
3	2023-02-01	2023-03-01	1
4	2023-01-15	2023-02-15	2
5	2023-01-01	2023-02-01	2
6	2023-01-15	2023-02-15	2
7	2023-02-01	2023-03-01	3
8	2023-02-01	2023-03-01	3
9	2023-01-01	2023-02-01	3
10	2023-02-01	2023-03-01	1

Table 2: Team details.

The final data set is for Base details. I'd prefer to create this data set just for base information with base id field and call this data from base table into team table by using base id field but request was different. Therefore, I created the data set with three fields and these are team id, latitude and longitude, respectively. Lat and long fields define the location of each base and in this study Berlin, Cologne and Munich are selected. As Table 2 shows team 1, 4, 7 are in Berlin(52.5167 & 13.3833), team 2, 5, 8, 10 are in Cologne (50.9422 & 6.9578) and team 3, 6, 9 are located in Munich (48.1372 & 11.5755).

team id	latitude	longitude
1	52.5167	13.3833
2	50.9422	6.9578
3	48.1372	11.5755
4	52.5167	13.3833
5	50.9422	6.9578
6	48.1372	11.5755
7	52.5167	13.3833
8	50.9422	6.9578
9	48.1372	11.5755
10	50.9422	6.9578

Table 3: Base details.

These data sets are selected for this study because it is desired to check every single

possibility. As you notice, we have 513 customer orders within two months period but our capacity during this period, even in the best case scenario, is approximately 300 (10 teams * 30 days). Therefore, we will definitely miss some customer orders but the aim of this study is to minimize that number by following the requested conditions such as 1 day installation, closest team assignment, etc..

2   Data Infrastructure

In this study, an AWS Redshift cluster is used in my personal AWS account. I frequently
used AWS CLI, PostgreSQL and Python together because this is the most efficient way
to use the any of AWS services. First of all, a Redshift table needs to be created for each
data set which are generated in Section 1. Below queries are coded and saved under SQL
folder in Enpal git repository.

```
1  CREATE TABLE enpal.teams(
2      team_id integer PRIMARY KEY,
3      availability_from DATE,
4      availability_till DATE,
5      skill_level NUMERIC(20),
6      sysdatecreated timestamp NULL DEFAULT now(),
7      sysdatemodified timestamp DEFAULT CURRENT_TIMESTAMP NOT NULL)
```

List of Listings 1: PostgreSQL code of Teams table in Enpal schema .

```
1  CREATE TABLE enpal.bases(
2      team_id NUMERIC(20),
3      latitude     DECIMAL(20,5),
4      longitude    DECIMAL(20,5),
5      sysdatecreated timestamp NULL DEFAULT now(),
6      sysdatemodified timestamp DEFAULT CURRENT_TIMESTAMP NOT NULL)
```

List of Listings 2: PostgreSQL code of Bases table in Enpal schema .

```
1  CREATE TABLE enpal.customer_orders(
2      customer_id integer PRIMARY KEY,
3      city     TEXT,
4      country  TEXT,
5      country_code     TEXT,
6      state    TEXT,
7      latitude     DECIMAL(20,5),
8      longitude    DECIMAL(20,5),
9      start_date DATE,
10     number_of_panels NUMERIC(20),
11     sysdatecreated timestamp NULL DEFAULT now(),
12     sysdatemodified timestamp DEFAULT CURRENT_TIMESTAMP NOT NULL)
```

List of Listings 3: PostgreSQL code of Customer Orders table in Enpal schema .

After creating the SQL codes, I wrote a python script which reads those SQL codes and runs them in AWS and create 3 Redshift tables. Listings 4 shows which libraries and parameters are used to connect AWS. This section is common all the other python scripts and that's why I will not return to this section for the other scripts. First 6 libraries such as psycopg2, configparser, sys etc. are common libraries which are used in all scripts while pandas, boto3, io are only used in the pipeline script (insert_data.py). All required credentials such as arn, server name, password etc., are stored in external location in order to avoid security leak and they are called via configparser when they are needed.

```python
import psycopg2
import os
import sys
import warnings
from configparser import ConfigParser
from psycopg2 import extras
import pandas
import boto3
import io
from io import StringIO

warnings.filterwarnings("ignore")
config = ConfigParser()
config.read('../config.ini')

user=config['altar']['user_name']
password=config['altar']['password']
db_name=config['altar']['db_name']
host=config['altar']['host']
port=config['altar']['port']
arn=config['altar']['arn']

params = {'host': host,
          'database': db_name,
          'user': user,
          'password': password,
          'port': port}
```

List of Listings 4: Connection to AWS

Afterwards, table creation function is coded (Listings 5). This function requires a connection and the path of the SQL code so it can open and read the code code fist and then execute it.

```python
def create_table(conn, path):
    f = open(path, 'r')
    sql = f.read()
    f.close()
    commands = (sql)
    try:
        cur = conn.cursor()
        cur.execute(commands)
    except (Exception, psycopg2.DatabaseError) as error:
        print(error)
    finally:
        cur.close()
        conn.commit()
```

List of Listings 5: Python table creation function.

Therefore, in order to call the create_table function, a connection must be build and path of each SQL script must be defined. These are done in the main function (see Listings 6). Required parameters are sent to psycopg2.connect function which establish the connection and SQL locations are listed. Afterwards, the create table function is called for each SQL script via for loop. Finally, connection is terminated.

```python
if __name__ == '__main__':
    try:
        conn = psycopg2.connect(**params)
        path=["./SQL/customer_orders.sql" , "./SQL/bases.sql", "./SQL/
    teams.sql"]
        for i in range(len(path)):
            create_table(conn, path=path[i])
            print("Created : " + path[i])
    finally:
        if conn is not None:
            conn.close()
```

List of Listings 6: The main function in create_table script.

Since we have the Python + SQL script which creates tables in AWS Redshift and now, the next step is to create an ETL process so we can store the data in those tables. In order to do that, first we need to store the data in a AWS S3 bucket. Upload to S3 function (see Listing 7), reads the data frame, destination bucket name and final file name and then uploads that data frame into the designated S3 bucket as csv file with a proper name.

```python
def upload_to_s3(dataframe, bucket_name, file_name):
    csv_buffer = StringIO()
    dataframe.to_csv(csv_buffer,index=False)

    s3_resource = boto3.resource('s3')
    s3_resource.Object(bucket_name, file_name).put(Body=csv_buffer.
    getvalue())

    return True
```

List of Listings 7: Upload to AWS S3 function.

Now we have the data in AWS S3 but before we insert that data into the AWS Redshift table, we have to make sure that table is clean. Therefore, in this study, I preferred to truncate the table before inserting the data (see Listing 8). By doing so, I avoid the duplication issue which is caused by multiple insert command.

```python
def truncate(conn, tablename):
    commands = ("""
        TRUNCATE """ + tablename)
    try:

        cur = conn.cursor()
        cur.execute(commands)

    except (Exception, psycopg2.DatabaseError) as error:
        print(error)
    finally:
        cur.close()
        conn.commit()
```

List of Listings 8: Truncate table function.

Now we have the data in S3 plus we have a clean table in Redshift so we can transfer the data from S3 to Redshift. To do that, I coded the s3_to_RS function (see Listing 9) which already has the required SQL statement in it and all we need to do is to sent the variables such as connection, table name, S3 path and column headers in the data set.

```python
def s3_to_RS(conn, table_name, path, columns_list):
    command = (
    """
        copy """ + table_name + """ (""" + columns_list + """)
        from '""" + path + """'
        IAM_ROLE """ + arn + """
        delimiter ','
        DATEFORMAT 'YYYY-MM-DD'
        csv
        IGNOREHEADER AS 1
    """
    )

    try:
        cur = conn.cursor()
        cur.execute(command)

    except (Exception, psycopg2.DatabaseError) as error:
        print(error)

    finally:
        cur.close()
        conn.commit()
```

List of Listings 9: Data transfer function.

The final step is to call all these functions that we create earlier. In the main function (see Listing 10), first we established the connection to Redshift and define the variables such as file names, Redshift table names and S3 bucket name. I used python lists over here so the order of the file name and table name must be synchronized but a python dictionary can be used as well. Afterwards, a for loop calls each file and does the following steps. First reads the data from local source and creates a pandas data frame. Then it gets the column headers from that data frame and stores as column_list. Afterwards, the script

calls truncate, upload_to_S3 and s3_to_RS functions, respectively. Finally, it terminates the any open connections. Now, we have an ETL process which can be schedule in AWS Lambda.

```python
if __name__ == '__main__':
    try:
        conn = psycopg2.connect(**params)
        s3 = boto3.client("s3")
        filename=['customer_orders.csv', 'teams.csv', 'bases.csv']
        table_name=["enpal.customer_orders", "enpal.teams", "enpal.bases"]
        bucket_name = "pipeline-redshift"

        for i in range(len(filename)):
            source_path=r'./data/' + filename[i]
            destination_path = 's3://' + bucket_name + '/' +filename[i]

            dataframe=pandas.read_csv(source_path)
            columns_list=dataframe.columns
            columns_list = ', '.join(columns_list)

            if dataframe.empty:
                print("There is no data")
            else:
                truncate(conn, table_name[i])
                upload_to_s3(dataframe, bucket_name, filename[i])
                s3_to_RS(conn, table_name[i], destination_path, columns_list)
                print("Upload to " + table_name[i] + " : Done")
    except:
        print("Upload to " + table_name[i] + " : FAILED")
    finally:
        if conn is not None:
            conn.close()
```

List of Listings 10: The main function in insert_data.py script.

Since we have a complete process, now we can run these scripts and store the data in Redshift. First, opened the terminal and move to project dir. Afterwards, I run the

"python3 create_redshift_table.py" command and I got the success message (Figure 1).

```
1  cd Documents/Python/enpal/
2  python3 create_redshift_table.py
```

Then I run the below command to created a S3 bucket so insert_data.py script can dump the data into that S3 bucket.

```
1  aws s3api create-bucket --bucket pipeline-redshift --region us-east-1
```

Now we have the S3 Bucket and the Redshift tables so I run the below command and insert the data into Redshift successfully (Figure 1 & 2)

```
1  python3 insert_data.py
2  aws s3 ls pipeline-redshift/
```


Figure 1: Creating the tables and pipeline.

As Figure 2 shows, three tables, which are 'enpal.bases', 'enpal.teams' and 'enpal.customer_orders', are created and filled with correct data. Now we have the data in the cloud environment so in the next section we can continue on problem solution.
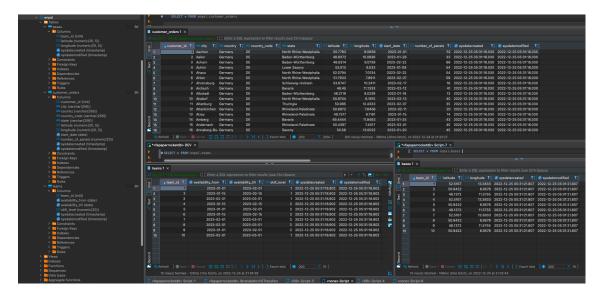
Figure 2: Created tables under Enpal Schema

3   Problem Solution

In this documentation a single order (Customer ID : 68) is randomly selected as an example and each steps are explained with this order.The main reason of that is we have more than 500 customer orders and it will use too much space to display in this documentation. However, complete results are given in the csv format and it can be found in attachment.

In the first step, I created a single temporary table join the teams table with bases table. Then I assign a daily capacity to each team based on their skill levels (Listings 11). According to this, teams with skill level 1 can assemble 100 units per day which is more than the maximum customer order, teams with skill level 2 can assemble maximum 22 units per day and teams with skill level 3 can assemble maximum 21 units per day. When I run the query I got the results in the Figure 3.

```
1  CREATE TEMP Table capacity AS (
2     SELECT
3        a.team_id,
4        b.latitude AS base_latitude,
5        b.longitude AS base_longitude,
6        a.availability_from,
7        a.availability_till,
8        a.skill_level,
9        CASE  WHEN a.skill_level = 1 THEN 100
10             WHEN a.skill_level = 2 THEN 22
11             ELSE 21 END AS capacity
12    FROM enpal.teams a
13    LEFT JOIN enpal.bases b ON a.team_id=b.team_id);
```

List of Listings 11: PostgreSQL code of Capacity Assignment.

team_id	base_latitude	base_longitude	availability_from	availability_till	skill_level	capacity
1	52.5167	13.3833	2023-01-01	2023-02-01	1	100
2	50.9422	6.9578	2023-01-15	2023-02-15	1	100
3	48.1372	11.5755	2023-02-01	2023-03-01	1	100
4	52.5167	13.3833	2023-01-15	2023-02-15	2	22
5	50.9422	6.9578	2023-01-01	2023-02-01	2	22
6	48.1372	11.5755	2023-01-15	2023-02-15	2	22
7	52.5167	13.3833	2023-02-01	2023-03-01	3	21
8	50.9422	6.9578	2023-02-01	2023-03-01	3	21
9	48.1372	11.5755	2023-01-01	2023-02-01	3	21
10	50.9422	6.9578	2023-02-01	2023-03-01	1	100

Figure 3: Single View for Teams and Bases

Afterwards, I need to find out all routes from each team to each customer. Since we do not have any foreign key in between customer order table and temporary table that I created earlier, I cross joined both tables and I got all possible routes between each customer and each team. Then I calculated the distance between them by using the mathematical formula in Listing 12, Line 9. I also need to know if these teams are available on the start date or not so I checked if start date is within availability date range (Listing 12, Line 13). Finally, for each team I calculated the number of required days to finish that specific customer order Listing 12, Line 17). At the end I ordered them based on customer id, availability, distance and capacity. The reason of that order is I want to utilize low skill level teams for small customer orders so I can create availability to better skill teams to cover bigger orders that other teams cannot cover it within a single day.

```
1  CREATE TEMP Table all_routes AS (
2    SELECT
3      b.customer_id,
4      a.team_id,
5      b.latitude  AS destination_latitude,
6      b.longitude AS destination_longitude,
7      a.base_latitude,
8      a.base_longitude,
9      SQRT(POW(111 * (base_latitude::float - destination_latitude::float)
      , 2) + POW(111 * (base_longitude::float - destination_longitude::
      float) * COS(base_latitude::float), 2)) AS distance,
10     b.start_date,
11     a.availability_from,
12     a.availability_till,
13     CASE WHEN b.start_date BETWEEN a.availability_from AND a.
      availability_till THEN TRUE ELSE FALSE END availabile,
14     b.number_of_panels,
15     a.capacity,
16     a.skill_level,
17     CEIL(b.number_of_panels/a.capacity) AS required_day
18   FROM capacity a
19   CROSS JOIN enpal.customer_orders b
20   ORDER BY 1,11,7,13  );
```

List of Listings 12: PostgreSQL code of all routes.

When I run the query and filter the data for customer id 68, I got the results in Figure 4. As the figure shows, there are only 6 available teams (team 5, 2 ,9 ,6, 4 ,1), only 5 of them can do the job within a day and only 2 of them (team 5 and 2) are very close to the customer. Since the order quantity can be cover with the lower skilled team, my goal is to assign team 5 to customer 68.

Figure 4: All routes for Customer 68

In order to do that, first, I need to remove unavailable teams and team that cannot completed the job within the same day.

```
1  CREATE TEMP Table availabile_routes AS (
2    SELECT
3      *
4    FROM all_routes
5    WHERE availabile=TRUE
6    AND required_day=1
7    ORDER BY 1,7,13
8  );
```

List of Listings 13: PostgreSQL code for all possible routes for Customer 68 .

By running the query in Listing 13, I obtained the results in Figure 5. Now I have 5 teams that can do the job within a single day and all of them are available in the order start date. The next step is to get the closest teams only.
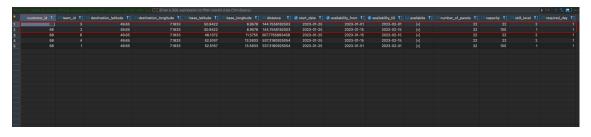

Figure 5: All possible routes for Customer 68

The query in the Listing 14, first finds the closest distance for each customer and then joins with the temporary table that we create in previous step, based on customer id and the distance. By doing so, query returns only the closest teams, which are team 5 and 2 (Figure 6).

```sql
CREATE TEMP Table shortest_availabile_routes AS (
  SELECT
    a.customer_id,
    a.destination_latitude,
    a.destination_longitude,
    a.team_id,
    a.base_latitude,
    a.base_longitude,
    a.distance,
    a.start_date,
    a.availability_from,
    a.availability_till,
    a.availabile,
    a.number_of_panels,
    a.capacity,
    a.skill_level,
    a.required_day
  FROM availabile_routes a join
      (
       SELECT
         customer_id,
         min(distance) AS distance
       FROM availabile_routes
       WHERE availabile=TRUE
       GROUP BY 1
      ) b on a.customer_id = b.customer_id AND a.distance=b.distance
  ORDER BY 1,7,13 );
```

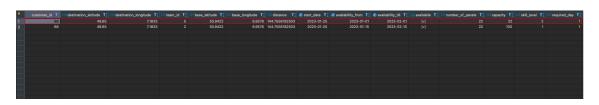List of Listings 14: PostgreSQL code for shortest possible routes for Customer 68.

Figure 6: Shortest possible routes for Customer 68.

Now we have 2 teams that can cover all requirements but we need to assign one to one customer. As I mention in the beginning of this chapter, I'd like to cover small orders with low skilled teams so I can utilize high skilled teams in bigger orders. Therefore, I order them based on their daily installment capacities and chose the first team for each customer by using windows function (Listing 15, line 18-26). Finally, I got the result in Figure 7.

```sql
1  CREATE TEMP Table chosen_routes AS (
2    SELECT
3      a.customer_id,
4      a.destination_latitude,
5      a.destination_longitude,
6      a.team_id,
7      a.base_latitude,
8      a.base_longitude,
9      a.distance,
10     a.start_date,
11     a.availability_from,
12     a.availability_till,
13     a.availabile,
14     a.number_of_panels,
15     a.capacity,
16     a.skill_level,
17     a.required_day
18   FROM (
19         SELECT
20         ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY
     customer_id) AS row_number,
21         b.*
22         FROM ( SELECT * FROM shortest_availabile_routes ) b
23       ) a
24   WHERE a.row_number = 1
25   ORDER BY 1);
```

List of Listings 15: PostgreSQL code for to find the best team match for Customer 68.

Figure 7: The best team match for Customer 68.

When I run the SQL code for all customers, I see that we assigned a single team to all 513 customers (see Figure 8) and make sure the logic is work for each customer.
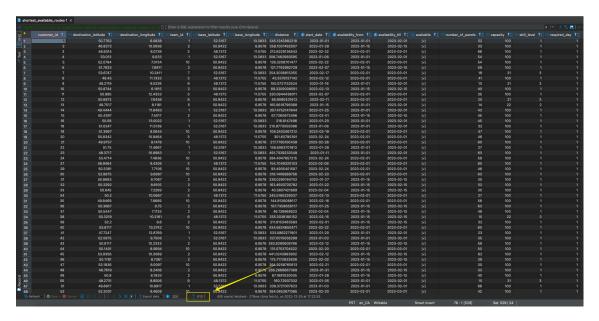

Figure 8: Complete Results

As I mentioned at the end of the Section 1, we will miss some orders no matter what due to high number orders and low capacity. Therefore, I wanted to check how we performed under given conditions and I wanted to look for optimization possibilities. After analyzing the data for team 5 by running the query in Listing 16, I obtained the results in Figure 9.

```sql
CREATE TEMP Table daily_performance AS (
    SELECT
    team_id,
    availability_till-availability_from AS total_capacity,
    start_date,
    count(customer_id) AS assigned_orders,
    1 AS fulfilled_orders,
    CASE WHEN count(customer_id) - 1 < 0 THEN 0
    ELSE count(customer_id) - 1 END missed_orders
    FROM chosen_routes
    GROUP BY 1,2,3
    ORDER BY 1,2) ;
```

List of Listings 16: PostgreSQL code daily team performance.

As Figure 9 shows, in some dates multiple customer orders are assigned to team 5 because in those days team 5 meets all the criteria. For instance on 2023-01-02, Team 5 is avail-

able for 4 customer orders, Team 5 can install panels within the same day for the same 4 customer orders and also Team 5 is closest team to those same 4 customers. However, since Team 5 cannot be in four different places in the same day. Therefore, 3 customer orders out of 4 will be missed.
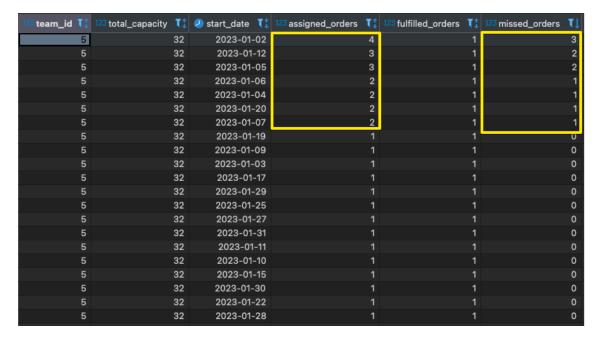

Figure 9: Team 5's daily performance.

In order to see the overall situation, I run the query in Listing 17 and got the results which are given in Figure 10. As you can see, with the given conditions we can only fulfill 171 customers orders out of 513. That means, 342 orders will be missed, and teams will be stand idle for 137 days. Therefore, this problem can be turn into an optimization problem since there are a lot to improve.
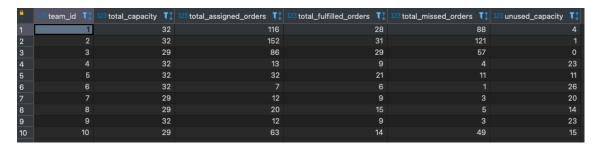

	team_id	total_capacity	total_assigned_orders	total_fulfilled_orders	total_missed_orders	unused_capacity
1	1	32	116	28	88	4
2	2	32	152	31	121	1
3	3	29	86	29	57	0
4	4	32	13	9	4	23
5	5	32	32	21	11	11
6	6	32	7	6	1	26
7	7	29	12	9	3	20
8	8	29	20	15	5	14
9	9	32	12	9	3	23
10	10	29	63	14	49	15

Figure 10: Overall team performance.

```sql
1  WITH summary AS (
2      SELECT
3      team_id,
4      total_capacity,
5      sum(assigned_orders) AS total_assigned_orders,
6      sum(fulfilled_orders) AS total_fulfilled_orders,
7      sum(missed_orders) AS total_missed_orders
8      FROM daily_performance
9      GROUP BY 1,2
10     ORDER BY 1
11 )
12 SELECT *,
13 total_capacity-total_fulfilled_orders AS unused_capacity
14     FROM summary
```

List of Listings 17: PostgreSQL code for overall team performance.