

A CASE STUDY

Enpal.

BY

FIDANSOY, ALTAR

Contents

Contents	i
List of Tables	ii
List of Figures	iii
List of Listings	iv
1 Data Generation	1
2 Data Infrastructure	4
3 Problem Solution	12

List of Tables

1	Sample Customer Data.	1
2	Team details.	2
3	Base details.	2

List of Figures

1	Creating the tables and pipeline.	10
2	Created tables under Enpal Schema	11
3	Single View for Teams and Bases	12
4	All routes for Customer 68	14
5	All possible routes for Customer 68	14
6	Shortest possible routes for Customer 68.	15
7	The best team match for Customer 68.	16
8	Complete Results	17
9	Team 5's daily performance.	18
10	Overall team performance.	18

List of Listings

1	PostgreSQL code of Teams table in Enpal schema	4
2	PostgreSQL code of Bases table in Enpal schema	4
3	PostgreSQL code of Customer Orders table in Enpal schema	4
4	Connection to AWS	5
5	Python table creation function.	6
6	The main function in create_table script.	6
7	Upload to AWS S3 function.	7
8	Truncate table function.	7
9	Data transfer function.	8
10	The main function in insert_data.py script.	9
11	PostgreSQL code of Capacity Assignment.	12
12	PostgreSQL code of all routes.	13
13	PostgreSQL code for all possible routes for Customer 68	14
14	PostgreSQL code for shortest possible routes for Customer 68.	15
15	PostgreSQL code for to find the best team match for Customer 68.	16
16	PostgreSQL code daily team performance.	17
17	PostgreSQL code for overall team performance.	19

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 any security leak. Therefore, these credentials are called via configparser.

```
1 import psycopg2
2 import os
3 import sys
4 import warnings
5 from configparser import ConfigParser
6 from psycopg2 import extras
7 import pandas
8 import boto3
9 import io
10 from io import StringIO
11
12 warnings.filterwarnings("ignore")
13 config = ConfigParser()
14 config.read('../config.ini')
15
16 user=config['altar']['user_name']
17 password=config['altar']['password']
18 db_name=config['altar']['db_name']
19 host=config['altar']['host']
20 port=config['altar']['port']
21 arn=config['altar']['arn']
22
23 params = {'host': host,
24           'database': db_name,
25           'user': user,
26           'password': password,
27           '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 and then it can execute it.

```
1 def create_table(conn, path):
2     f = open(path, 'r')
3     sql = f.read()
4     f.close()
5     commands = (sql)
6     try:
7         cur = conn.cursor()
8         cur.execute(commands)
9     except (Exception, psycopg2.DatabaseError) as error:
10        print(error)
11    finally:
12        cur.close()
13        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 to establish a connection to Redshift. Next, SQL file locations are listed. Afterwards, the create table function is called for each SQL script via for loop. Finally, connection is terminated.

```
1 if __name__ == '__main__':
2     try:
3         conn = psycopg2.connect(**params)
4         path=["./SQL/customer_orders.sql" , "./SQL/bases.sql", "./SQL/
5             teams.sql"]
6         for i in range(len(path)):
7             create_table(conn, path=path[i])
8             print("Created : " + path[i])
9         finally:
10            if conn is not None:
11                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, now we can focus on to create an ETL process so data can be stored 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.

```
1 def upload_to_s3(dataframe, bucket_name, file_name):
2     csv_buffer = StringIO()
3     dataframe.to_csv(csv_buffer, index=False)
4
5     s3_resource = boto3.resource('s3')
6     s3_resource.Object(bucket_name, file_name).put(Body=csv_buffer.
7         getvalue())
8
9     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 tables, we have to make sure that tables are 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.

```
1 def truncate(conn, tablename):
2     commands = ("""
3         TRUNCATE """ + tablename)
4     try:
5
6         cur = conn.cursor()
7         cur.execute(commands)
8
9     except (Exception, psycopg2.DatabaseError) as error:
10         print(error)
11     finally:
12         cur.close()
13         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 has the required SQL statement in it and all we need to do is to sent the required variables such as connection, table name, S3 path and column headers in the data set.

```
1 def s3_to_RS(conn, table_name, path, columns_list):
2     command = (
3         """
4         copy """ + table_name + """ (""" + columns_list + """)
5         from '""" + path + """'
6         IAM_ROLE """ + arn + """
7         delimiter ','
8         DATEFORMAT 'YYYY-MM-DD'
9         csv
10        IGNOREHEADER AS 1
11        """
12    )
13
14    try:
15        cur = conn.cursor()
16        cur.execute(command)
17
18    except (Exception, psycopg2.DatabaseError) as error:
19        print(error)
20
21    finally:
22        cur.close()
23        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 a python list over here so the order of the file names and table names must be synchronized but a python dictionary can be used as well to avoid synchronization. Afterwards, a for loop calls each file and applies 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 in the

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 a complete ETL process which can be schedule in AWS Lambda.

```
1 if __name__ == '__main__':
2     try:
3         conn = psycopg2.connect(**params)
4         s3 = boto3.client("s3")
5         filename=['customer_orders.csv', 'teams.csv', 'bases.csv']
6         table_name=["enpal.customer_orders", "enpal.teams", "enpal.
7         bases"]
8
9         bucket_name = "pipeline-redshift"
10
11         for i in range(len(filename)):
12             source_path=r'./data/' + filename[i]
13             destination_path = 's3://' + bucket_name + '/' +filename[i]
14
15             dataframe=pandas.read_csv(source_path)
16             columns_list=dataframe.columns
17             columns_list = ', '.join(columns_list)
18
19             if dataframe.empty:
20                 print("There is no data")
21             else:
22                 truncate(conn, table_name[i])
23                 upload_to_s3(dataframe, bucket_name, filename[i])
24                 s3_to_RS(conn, table_name[i], destination_path,
25                 columns_list)
26                 print("Upload to " + table_name[i] + " : Done")
27             except:
28                 print("Upload to " + table_name[i] + " : FAILED")
29             finally:
30                 if conn is not None:
31                     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, I 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).

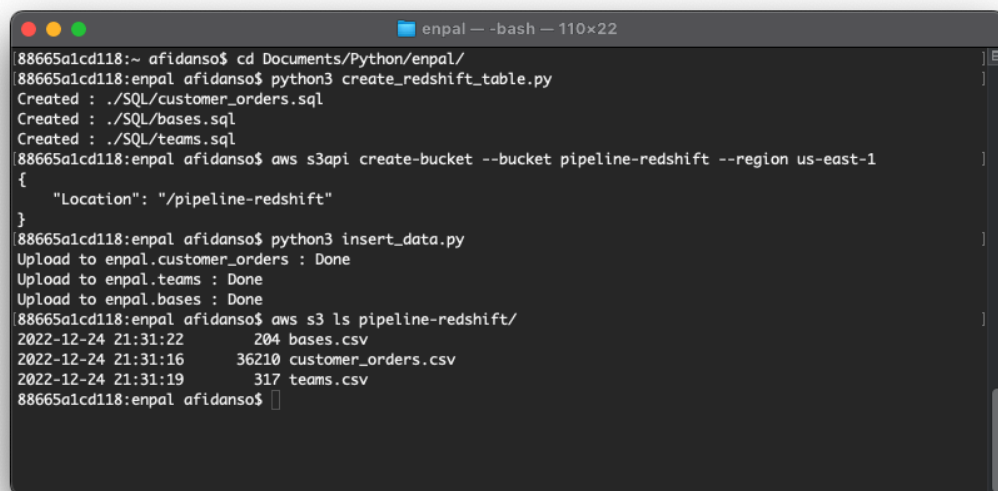
```
1 cd Documents/Python/enpal/  
2 python3 create_redshift_table.py
```

Then I run the below command to created a S3 bucket so insert_data.py script can dump the data into that 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 inserted the data into Redshift successfully (Figure 1 & 2)

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



```
enpal -- bash -- 110x22  
88665a1cd118:~ afidanso$ cd Documents/Python/enpal/  
88665a1cd118:enpal afidanso$ python3 create_redshift_table.py  
Created : ./SQL/customer_orders.sql  
Created : ./SQL/bases.sql  
Created : ./SQL/teams.sql  
88665a1cd118:enpal afidanso$ aws s3api create-bucket --bucket pipeline-redshift --region us-east-1  
{  
  "Location": "/pipeline-redshift"  
}  
88665a1cd118:enpal afidanso$ python3 insert_data.py  
Upload to enpal.customer_orders : Done  
Upload to enpal.teams : Done  
Upload to enpal.bases : Done  
88665a1cd118:enpal afidanso$ aws s3 ls pipeline-redshift/  
2022-12-24 21:31:22      204 bases.csv  
2022-12-24 21:31:16    36210 customer_orders.csv  
2022-12-24 21:31:19     317 teams.csv  
88665a1cd118:enpal afidanso$
```

Figure 1: Creating the tables and pipeline.

As Figure 2 shows, three tables ('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 focus on the solution of the problem .

customer_orders 1 x

customer_id	city	country	country_code	state	latitude	longitude	start_date	number_of_panels	sysdatecreated	sysdatemodified
1	Aachen	Germany	DE	North Rhine-Westphalia	50.7762	6.0838	2023-01-01	53	2022-12-25 05:31:16.030	2022-12-25 05:31:16.030
2	Aalen	Germany	DE	Baden-Wuerttemberg	48.8372	10.0006	2023-01-26	29	2022-12-25 05:31:16.030	2022-12-25 05:31:16.030
3	Achern	Germany	DE	Baden-Wuerttemberg	48.8314	8.0739	2023-02-22	60	2022-12-25 05:31:16.030	2022-12-25 05:31:16.030
4	Achim	Germany	DE	Lower Saxony	53.013	9.033	2023-01-08	54	2022-12-25 05:31:16.030	2022-12-25 05:31:16.030
5	Ahaus	Germany	DE	North Rhine-Westphalia	52.2704	7.0234	2023-02-22	54	2022-12-25 05:31:16.030	2022-12-25 05:31:16.030
6	Ahaus	Germany	DE	North Rhine-Westphalia	51.7633	7.8911	2023-02-07	59	2022-12-25 05:31:16.030	2022-12-25 05:31:16.030
7	Ahrensburg	Germany	DE	Schleswig-Holstein	53.8747	10.2411	2023-02-17	18	2022-12-25 05:31:16.030	2022-12-25 05:31:16.030
8	Aichach	Germany	DE	Bavaria	48.45	11.1222	2023-02-12	41	2022-12-25 05:31:16.030	2022-12-25 05:31:16.030
9	Altach	Germany	DE	Baden-Wuerttemberg	48.2119	9.0239	2023-01-18	13	2022-12-25 05:31:16.030	2022-12-25 05:31:16.030
10	Alsdorf	Germany	DE	North Rhine-Westphalia	50.8244	6.1676	2023-02-13	40	2022-12-25 05:31:16.030	2022-12-25 05:31:16.030
11	Altburg	Germany	DE	Thuringia	50.965	12.4333	2023-02-07	16	2022-12-25 05:31:16.030	2022-12-25 05:31:16.030
12	Altankirchen	Germany	DE	Rhineland-Palatinate	50.8872	7.6456	2023-02-11	20	2022-12-25 05:31:16.030	2022-12-25 05:31:16.030
13	Alzey	Germany	DE	Rhineland-Palatinate	49.7517	8.1181	2023-01-15	14	2022-12-25 05:31:16.030	2022-12-25 05:31:16.030
14	Amberg	Germany	DE	Bavaria	49.4444	11.5463	2023-01-26	43	2022-12-25 05:31:16.030	2022-12-25 05:31:16.030
15	Andernach	Germany	DE	Rhineland-Palatinate	50.4397	7.4017	2023-02-01	46	2022-12-25 05:31:16.030	2022-12-25 05:31:16.030
16	Annaberg-Buchholz	Germany	DE	Saxony	50.58	13.0022	2023-01-30	48	2022-12-25 05:31:16.030	2022-12-25 05:31:16.030

teams 1 x

team_id	availability_from	availability_till	skill_level	sysdatecreated	sysdatemodified
1	2023-01-01	2023-02-01	1	2022-12-25 05:31:16.902	2022-12-25 05:31:16.902
2	2023-01-16	2023-02-16	1	2022-12-25 05:31:16.902	2022-12-25 05:31:16.902
3	2023-02-01	2023-03-01	1	2022-12-25 05:31:16.902	2022-12-25 05:31:16.902
4	2023-01-16	2023-02-16	2	2022-12-25 05:31:16.902	2022-12-25 05:31:16.902
5	2023-01-01	2023-02-01	2	2022-12-25 05:31:16.902	2022-12-25 05:31:16.902
6	2023-01-16	2023-02-16	2	2022-12-25 05:31:16.902	2022-12-25 05:31:16.902
7	2023-02-01	2023-03-01	3	2022-12-25 05:31:16.902	2022-12-25 05:31:16.902
8	2023-02-01	2023-03-01	3	2022-12-25 05:31:16.902	2022-12-25 05:31:16.902
9	2023-01-01	2023-02-01	3	2022-12-25 05:31:16.902	2022-12-25 05:31:16.902
10	2023-02-01	2023-03-01	1	2022-12-25 05:31:16.902	2022-12-25 05:31:16.902

bases 1 x

team_id	latitude	longitude	sysdatecreated	sysdatemodified
1	52.9167	13.9333	2022-12-25 05:31:21.807	2022-12-25 05:31:21.807
2	50.9422	6.9678	2022-12-25 05:31:21.807	2022-12-25 05:31:21.807
3	48.1372	11.0760	2022-12-25 05:31:21.807	2022-12-25 05:31:21.807
4	52.9167	13.9333	2022-12-25 05:31:21.807	2022-12-25 05:31:21.807
5	50.9422	6.9678	2022-12-25 05:31:21.807	2022-12-25 05:31:21.807
6	48.1372	11.0760	2022-12-25 05:31:21.807	2022-12-25 05:31:21.807
7	52.9167	13.9333	2022-12-25 05:31:21.807	2022-12-25 05:31:21.807
8	50.9422	6.9678	2022-12-25 05:31:21.807	2022-12-25 05:31:21.807
9	48.1372	11.0760	2022-12-25 05:31:21.807	2022-12-25 05:31:21.807
10	50.9422	6.9678	2022-12-25 05:31:21.807	2022-12-25 05:31:21.807

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 everything in a single page. However, complete results can be found in the attachment.

In the first step, I created a single temporary table and joined the teams table with bases table. Then I assigned a daily capacity to each team based on their skill levels (Listings 11). According to this, teams with skill level 1 can install 100 units per day which is more than the maximum customer order, teams with skill level 2 can install maximum 22 units per day and teams with skill level 3 can install 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	1	52.5167	13.3833	2023-01-01	2023-02-01	1	100
2	2	50.9422	6.9578	2023-01-15	2023-02-15	1	100
3	3	48.1372	11.5755	2023-02-01	2023-03-01	1	100
4	4	52.5167	13.3833	2023-01-15	2023-02-15	2	22
5	5	50.9422	6.9578	2023-01-01	2023-02-01	2	22
6	6	48.1372	11.5755	2023-01-15	2023-02-15	2	22
7	7	52.5167	13.3833	2023-02-01	2023-03-01	3	21
8	8	50.9422	6.9578	2023-02-01	2023-03-01	3	21
9	9	48.1372	11.5755	2023-01-01	2023-02-01	3	21
10	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 availability of each team. Therefore, I checked whether start date is within team's 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 by customer id, availability, distance and capacity, respectively. The reason of that order is I want to utilize low skilled teams for small customer orders so I can create availability to high skilled 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)
10    , 2) + POW(111 * (base_longitude::float - destination_longitude::
11    float) * COS(base_latitude::float), 2)) AS distance,
12    b.start_date,
13    a.availability_from,
14    a.availability_till,
15    CASE WHEN b.start_date BETWEEN a.availability_from AND a.
16    availability_till THEN TRUE ELSE FALSE END available,
17    b.number_of_panels,
18    a.capacity,
19    a.skill_level,
20    CEIL(b.number_of_panels/a.capacity) AS required_day
21  FROM capacity a
22  CROSS JOIN enpal.customer_orders b
23  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.

customer_id	team_id	destination_latitude	destination_longitude	base_latitude	base_longitude	distance	start_date	availability_from	availability_to	available	number_of_panels	capacity	skill_level	required_day
68	8	49.65	7.1833	50.9422	6.9578	144.7656182503	2023-01-25	2023-02-01	2023-03-01	[]	22	21	3	2
68	10	49.65	7.1833	50.9422	6.9578	144.7656182503	2023-01-25	2023-02-01	2023-03-01	[]	22	100	1	1
68	3	49.65	7.1833	48.1372	11.5765	307.776893456	2023-01-25	2023-02-01	2023-03-01	[]	22	100	1	1
68	7	49.65	7.1833	52.5167	13.3833	537.3160925054	2023-01-25	2023-02-01	2023-03-01	[]	22	21	3	2
68	5	49.65	7.1833	50.9422	6.9578	144.7656182503	2023-01-25	2023-01-01	2023-02-01	[x]	22	22	2	1
68	2	49.65	7.1833	50.9422	6.9578	144.7656182503	2023-01-25	2023-01-15	2023-02-15	[x]	22	100	1	1
68	9	49.65	7.1833	48.1372	11.5765	307.776893456	2023-01-25	2023-01-01	2023-02-01	[x]	22	21	3	2
68	6	49.65	7.1833	48.1372	11.5765	307.776893456	2023-01-25	2023-01-15	2023-02-15	[x]	22	22	2	1
68	4	49.65	7.1833	52.5167	13.3833	537.3160925054	2023-01-25	2023-01-15	2023-02-15	[x]	22	22	2	1
68	1	49.65	7.1833	52.5167	13.3833	537.3160925054	2023-01-25	2023-01-01	2023-02-01	[x]	22	100	1	1

Figure 4: All routes for Customer 68

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

```

1 CREATE TEMP Table available_routes AS (
2     SELECT
3         *
4     FROM all_routes
5     WHERE available=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.

customer_id	team_id	destination_latitude	destination_longitude	base_latitude	base_longitude	distance	start_date	availability_from	availability_to	available	number_of_panels	capacity	skill_level	required_day
68	5	49.65	7.1833	50.9422	6.9578	144.7656182503	2023-01-25	2023-01-01	2023-02-01	[x]	22	22	2	1
68	2	49.65	7.1833	50.9422	6.9578	144.7656182503	2023-01-25	2023-01-15	2023-02-15	[x]	22	100	1	1
68	9	49.65	7.1833	48.1372	11.5765	307.776893456	2023-01-25	2023-01-15	2023-02-15	[x]	22	22	2	1
68	6	49.65	7.1833	48.1372	11.5765	307.776893456	2023-01-25	2023-01-15	2023-02-15	[x]	22	22	2	1
68	4	49.65	7.1833	52.5167	13.3833	537.3160925054	2023-01-25	2023-01-15	2023-02-15	[x]	22	22	2	1

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).

```

1 CREATE TEMP Table shortest_avaiabile_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.avaiabile,
14        a.number_of_panels,
15        a.capacity,
16        a.skill_level,
17        a.required_day
18    FROM available_routes a join
19        (
20            SELECT
21                customer_id,
22                min(distance) AS distance
23            FROM available_routes
24            WHERE avaiabile=TRUE
25            GROUP BY 1
26        ) b on a.customer_id = b.customer_id AND a.distance=b.distance
27    ORDER BY 1,7,13 );

```

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

	customer_id T1	destination_latitude T1	destination_longitude T1	team_id T2	base_latitude T1	base_longitude T1	distance T1	start_date T1	availability_from T1	availability_till T1	avaiabile T1	number_of_panels T1	capacity T1	skill_level T1	required_day T1
1	68	49.65	7.1833	5	50.9422	6.9578	144.7558182503	2023-01-25	2023-01-01	2023-02-01	[v]	21	22	2	1
2	68	49.65	7.1833	2	50.9422	6.9578	144.7558182503	2023-01-25	2023-01-15	2023-02-15	[v]	22	100	1	1

Figure 6: Shortest possible routes for Customer 68.

Now we have 2 teams that can cover all requirements but we need to assign only one team to one customer. As I mentioned 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 ordered them by 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.

```

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
21            customer_id) AS row_number, 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.

customer_id	destination_latitude	destination_longitude	team_id	base_latitude	base_longitude	distance	start_date	availability_from	availability_till	available	number_of_panels	capacity	skill_level	required_day
68	49.65	7.834	5	50.8422	6.5678	144.766162503	2022-01-25	2023-01-01	2023-02-01	t	22	22	2	1

Figure 7: The best team match for Customer 68.

When I run the above SQL codes for all customers, I see that we assigned a single team to each customer (see Figure 8) and validated the logic.

customer_id	destination_latitude	destination_longitude	team_id	base_latitude	base_longitude	distance	start_date	availability_from	availability_till	available	number_of_panels	capacity	skill_level	required_day
1	50.7762	6.0838	508422	50.7762	6.0838	13.3833	2023-01-01	2023-01-01	2023-02-01	[v]	52	100	1	1
2	48.8372	10.0395	508422	48.8372	10.0395	6.9578	2023-01-28	2023-01-15	2023-02-15	[v]	33	100	1	1
3	48.6214	8.0739	508422	48.6214	8.0739	11.5765	2023-02-22	2023-02-01	2023-03-01	[v]	66	100	1	1
4	53.013	9.013	508422	53.013	9.013	13.3833	2023-01-08	2023-01-01	2023-02-01	[v]	54	100	1	1
5	52.0784	7.0154	508422	52.0784	7.0154	6.9578	2023-02-22	2023-02-01	2023-03-01	[v]	54	100	1	1
6	51.7633	7.8911	508422	51.7633	7.8911	12.1779	2023-02-07	2023-01-15	2023-02-15	[v]	59	100	1	1
7	53.6747	10.2411	508422	53.6747	10.2411	13.3833	2023-02-17	2023-02-01	2023-03-01	[v]	18	21	3	1
8	48.445	11.1333	508422	48.445	11.1333	11.5765	2023-02-12	2023-02-01	2023-03-01	[v]	41	100	1	1
9	48.2119	9.0239	508422	48.2119	9.0239	11.5765	2023-01-18	2023-01-01	2023-02-01	[v]	13	21	3	1
10	50.8744	6.1615	508422	50.8744	6.1615	6.9578	2023-02-13	2023-01-15	2023-02-15	[v]	40	100	1	1
11	50.885	12.1333	508422	50.885	12.1333	11.5765	2023-02-07	2023-02-01	2023-03-01	[v]	16	100	1	1
12	50.6872	7.6458	508422	50.6872	7.6458	6.9578	2023-02-11	2023-02-01	2023-03-01	[v]	20	21	3	1
13	49.7517	8.1181	508422	49.7517	8.1181	6.9578	2023-01-15	2023-01-01	2023-02-01	[v]	14	22	2	1
14	49.8444	11.4433	508422	49.8444	11.4433	13.3833	2023-01-26	2023-01-01	2023-02-01	[v]	48	100	1	1
15	50.4397	7.4017	508422	50.4397	7.4017	6.9578	2023-01-15	2023-01-01	2023-02-01	[v]	46	100	1	1
16	50.548	13.0222	508422	50.548	13.0222	13.3833	2023-01-26	2023-01-01	2023-02-01	[v]	46	100	1	1
17	51.0247	11.1339	508422	51.0247	11.1339	13.3833	2023-01-06	2023-01-01	2023-02-01	[v]	32	100	1	1
18	51.3967	8.0644	508422	51.3967	8.0644	6.9578	2023-02-19	2023-02-01	2023-03-01	[v]	47	100	1	1
19	50.8442	10.1444	508422	50.8442	10.1444	11.5765	2023-02-24	2023-02-01	2023-03-01	[v]	46	100	1	1
20	49.9797	9.1478	508422	49.9797	9.1478	6.9578	2023-02-26	2023-02-01	2023-03-01	[v]	60	100	1	1
21	51.75	11.4667	508422	51.75	11.4667	13.3833	2023-01-28	2023-01-01	2023-02-01	[v]	27	100	1	1
22	48.3717	10.8983	508422	48.3717	10.8983	13.3833	2023-01-11	2023-01-01	2023-02-01	[v]	68	100	1	1
23	53.4714	7.6836	508422	53.4714	7.6836	6.9578	2023-02-24	2023-02-01	2023-03-01	[v]	59	100	1	1
24	48.9464	9.4306	508422	48.9464	9.4306	11.5765	2023-02-09	2023-02-01	2023-03-01	[v]	60	100	1	1
25	50.3881	7.7106	508422	50.3881	7.7106	6.9578	2023-02-24	2023-02-01	2023-03-01	[v]	46	100	1	1
26	50.3695	9.0957	508422	50.3695	9.0957	6.9578	2023-02-23	2023-02-01	2023-03-01	[v]	63	100	1	1
27	50.8883	9.7067	508422	50.8883	9.7067	6.9578	2023-01-27	2023-01-15	2023-02-15	[v]	30	100	1	1
28	50.2262	8.8165	508422	50.2262	8.8165	6.9578	2023-01-22	2023-01-15	2023-02-15	[v]	52	100	1	1
29	50.445	7.2259	508422	50.445	7.2259	6.9578	2023-02-24	2023-02-01	2023-03-01	[v]	52	100	1	1
30	50.2	10.0667	508422	50.2	10.0667	11.5765	2023-02-13	2023-02-01	2023-03-01	[v]	27	100	1	1
31	48.8469	7.8989	508422	48.8469	7.8989	6.9578	2023-02-16	2023-02-01	2023-03-01	[v]	68	100	1	1
32	50.3667	8.75	508422	50.3667	8.75	6.9578	2023-01-25	2023-01-15	2023-02-15	[v]	63	100	1	1
33	50.5447	7.1133	508422	50.5447	7.1133	6.9578	2023-02-04	2023-01-15	2023-02-15	[v]	49	100	1	1
34	50.3219	10.2761	508422	50.3219	10.2761	11.5765	2023-02-15	2023-02-01	2023-03-01	[v]	16	22	2	1
35	52.2	8.3	508422	52.2	8.3	6.9578	2023-02-21	2023-02-01	2023-03-01	[v]	32	100	1	1
36	53.8117	10.3742	508422	53.8117	10.3742	13.3833	2023-02-22	2023-02-01	2023-03-01	[v]	60	100	1	1
37	47.2247	12.8768	508422	47.2247	12.8768	13.3833	2023-01-20	2023-01-01	2023-02-01	[v]	23	100	1	1
38	50.7075	6.7075	508422	50.7075	6.7075	13.3833	2023-01-03	2023-01-01	2023-02-01	[v]	36	100	1	1
39	50.8117	10.2333	508422	50.8117	10.2333	6.9578	2023-02-12	2023-01-15	2023-02-15	[v]	59	100	1	1
40	50.8401	8.8904	508422	50.8401	8.8904	6.9578	2023-02-26	2023-02-01	2023-03-01	[v]	60	100	1	1
41	53.9356	10.1999	508422	53.9356	10.1999	6.9578	2023-02-12	2023-01-15	2023-02-15	[v]	62	100	1	1
42	50.7781	8.7261	508422	50.7781	8.7261	6.9578	2023-02-02	2023-01-15	2023-02-15	[v]	39	100	1	1
43	53.1336	8.0097	508422	53.1336	8.0097	6.9578	2023-01-21	2023-01-01	2023-02-01	[v]	30	100	1	1
44	48.7819	8.3408	508422	48.7819	8.3408	6.9578	2023-01-31	2023-01-15	2023-02-15	[v]	33	100	1	1
45	50.9	6.1833	508422	50.9	6.1833	6.9578	2023-01-28	2023-01-15	2023-02-15	[v]	40	100	1	1
46	48.2271	8.8406	508422	48.2271	8.8406	11.5765	2023-01-06	2023-01-01	2023-02-01	[v]	16	21	3	1
47	49.8917	10.8917	508422	49.8917	10.8917	13.3833	2023-01-03	2023-01-01	2023-02-01	[v]	66	100	1	1
48	52.3031	9.4608	508422	52.3031	9.4608	6.9578	2023-02-20	2023-02-01	2023-03-01	[v]	42	100	1	1

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.

```

1 CREATE TEMP Table daily_performance AS (
2     SELECT
3         team_id,
4         availability_till-availability_from AS total_capacity,
5         start_date,
6         count(customer_id) AS assigned_orders,
7         1 AS fulfilled_orders,
8         CASE WHEN count(customer_id) - 1 < 0 THEN 0
9         ELSE count(customer_id) - 1 END missed_orders
10        FROM chosen_routes
11        GROUP BY 1,2,3
12        ORDER BY 1,2) ;

```

List of Listings 16: PostgreSQL code daily team performance.

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

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

123 team_id	123 total_capacity	start_date	123 assigned_orders	123 fulfilled_orders	123 missed_orders
5	32	2023-01-02	4	1	3
5	32	2023-01-12	3	1	2
5	32	2023-01-05	3	1	2
5	32	2023-01-06	2	1	1
5	32	2023-01-04	2	1	1
5	32	2023-01-20	2	1	1
5	32	2023-01-07	2	1	1
5	32	2023-01-19	1	1	0
5	32	2023-01-09	1	1	0
5	32	2023-01-03	1	1	0
5	32	2023-01-17	1	1	0
5	32	2023-01-29	1	1	0
5	32	2023-01-25	1	1	0
5	32	2023-01-27	1	1	0
5	32	2023-01-31	1	1	0
5	32	2023-01-11	1	1	0
5	32	2023-01-10	1	1	0
5	32	2023-01-15	1	1	0
5	32	2023-01-30	1	1	0
5	32	2023-01-22	1	1	0
5	32	2023-01-28	1	1	0

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.

	123 team_id	123 total_capacity	123 total_assigned_orders	123 total_fulfilled_orders	123 total_missed_orders	123 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.

```
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.