Aviation Database Management System

Cover

Presenters:

- Raphael Roshwalb (345304760)
- Shua Golubtchik (340880194)

System: Aviation Database Management System **Unit:** PostgreSQL Database Implementation

Table of Contents

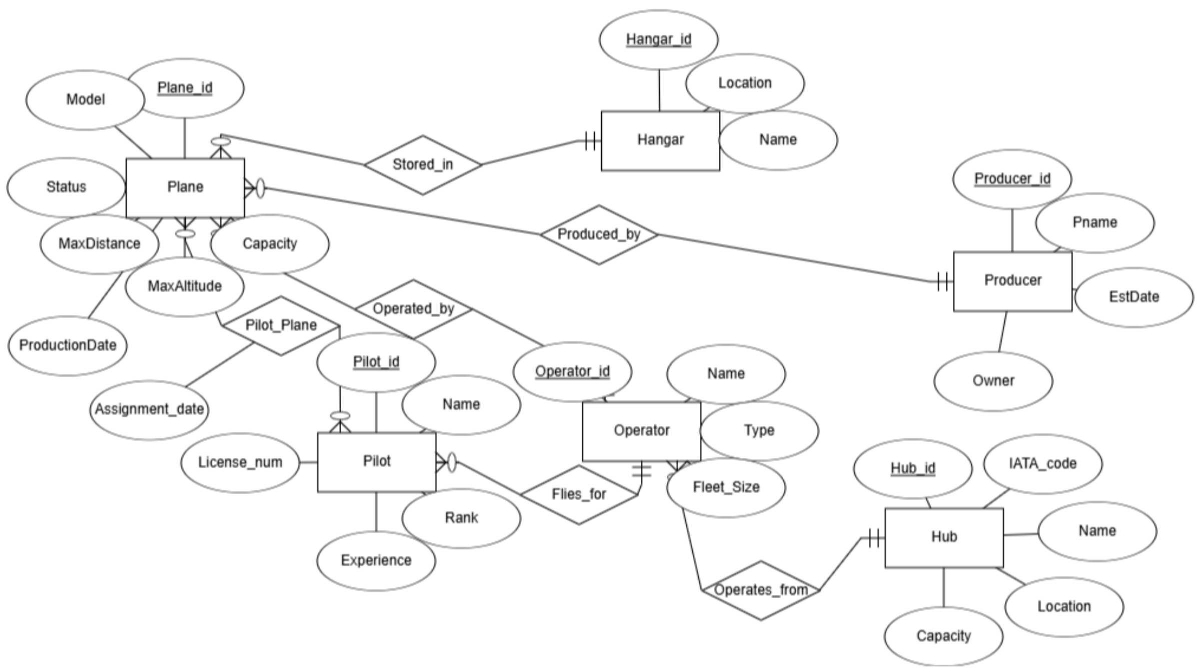
Introduction

Our Aviation Database Management System is designed to track and manage aviation resources across multiple airports and operators worldwide. The system stores comprehensive information about aviation infrastructure including hangars, planes, pilots, operators, producers, and hubs.

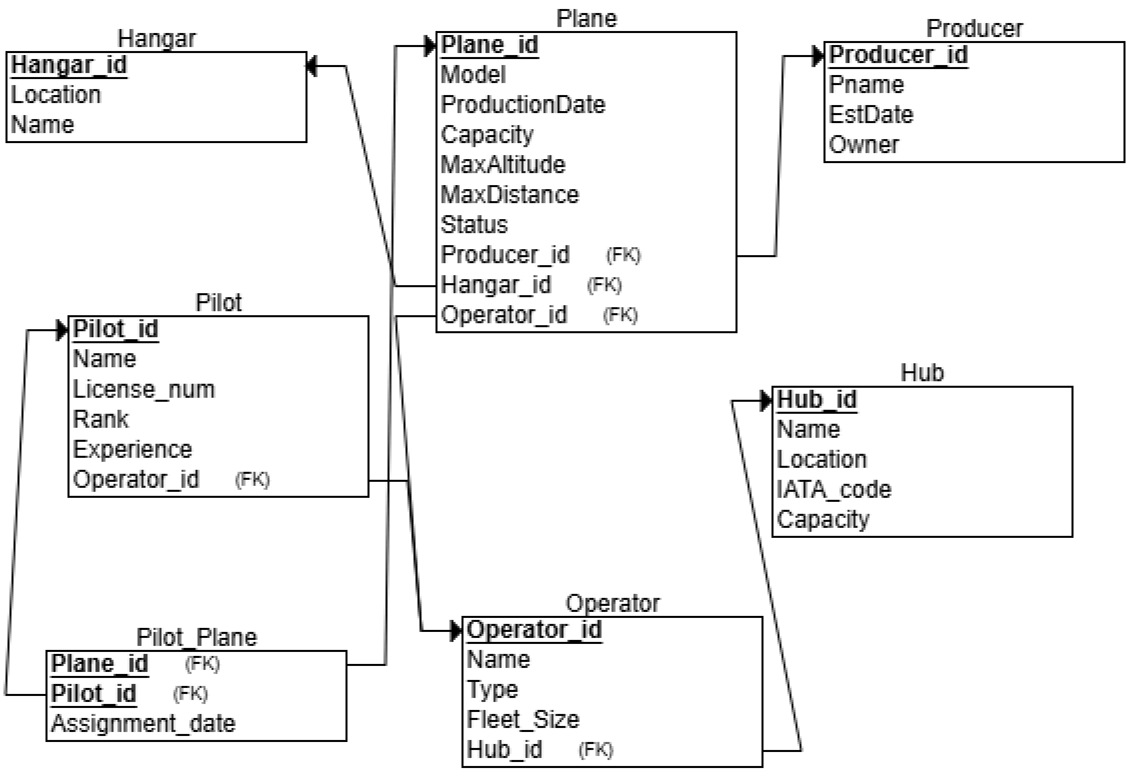
The database manages relationships between these entities, such as which pilots are assigned to which planes, which operators manage which fleets, and where aircraft are produced and stored. The main functionalities include tracking plane locations, monitoring pilot assignments, managing operational status, and facilitating fleet management for various operators.

This system would be utilized by aviation authorities, airlines, and airport management to efficiently allocate resources, schedule flight operations, and maintain regulatory compliance in the aviation sector.

Entity-Relationship Diagram



Database Schema Diagram



Design Decisions

Database Structure

We implemented a PostgreSQL relational database with seven tables to represent all entities and relationships in our aviation system. Key design decisions include:

1. **Many-to-Many Relationship Implementation**: We created a junction table (Pilot_Plane) to facilitate the many-to-many relationship between pilots and planes, allowing each pilot to be assigned to multiple planes and each plane to have multiple pilots.

2. **Normalization Approach**: The database follows the Third Normal Form (3NF) to eliminate data redundancy and improve data integrity. All entities have been properly separated with appropriate relationships.

3. **Primary and Foreign Key Strategy**: Each entity has a unique identifier (ID) as its primary key, and relationships are maintained using foreign keys. This ensures data

consistency and enables efficient queries across related tables.

4. **Data Types and Constraints**: We carefully selected appropriate data types for each attribute (VARCHAR for names and text fields, INT for numeric values, DATE for date fields) and added NOT NULL constraints to ensure data quality.

5. **Location Management**: We decided to store location information as text fields rather than geographic coordinates to simplify data entry, though this could be enhanced in future versions to support spatial queries.

6. **Identification Codes**: Special identification fields (like License_num for pilots and IATA_code for hubs) were included to align with industry standards and facilitate integration with external systems.

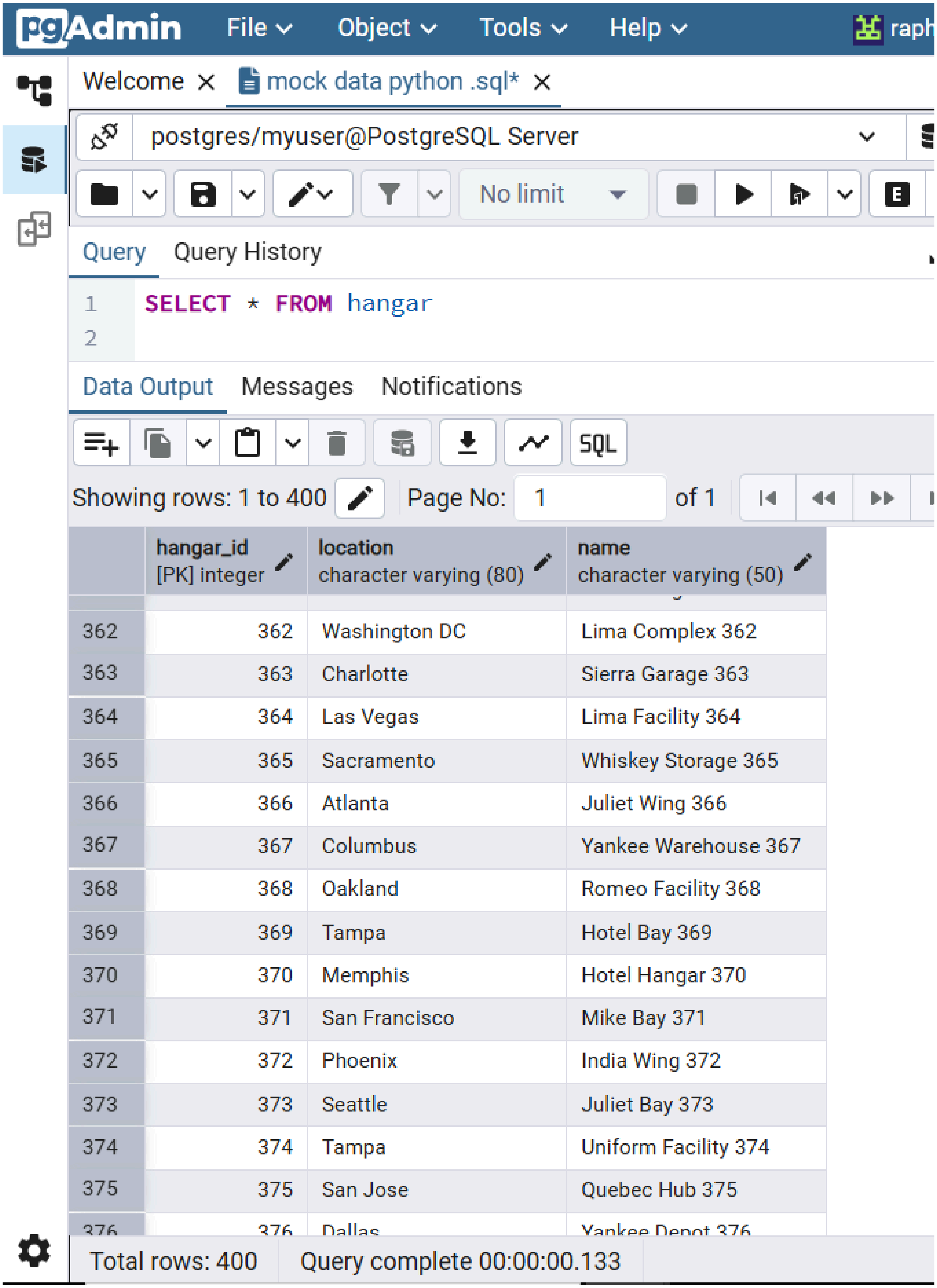
7. **Data Generation Strategy**: To populate the database with realistic test data, we developed a Python script (gen_data.py) that generates varied and representative mock data while maintaining referential integrity.

Alternative Approaches Considered

We initially considered a simpler model with fewer entities but decided on the current structure to better represent the complex relationships in aviation systems. We also evaluated NoSQL options but chose PostgreSQL for its robust transaction support, which is critical for aviation data.

Data Entry Methods

Method 1: Direct SQL Insertion




```sql
SELECT * FROM hangar
```

	hangar_id [PK] integer	location character varying (80)	name character varying (50)
362	362	Washington DC	Lima Complex 362
363	363	Charlotte	Sierra Garage 363
364	364	Las Vegas	Lima Facility 364
365	365	Sacramento	Whiskey Storage 365
366	366	Atlanta	Juliet Wing 366
367	367	Columbus	Yankee Warehouse 367
368	368	Oakland	Romeo Facility 368
369	369	Tampa	Hotel Bay 369
370	370	Memphis	Hotel Hangar 370
371	371	San Francisco	Mike Bay 371
372	372	Phoenix	India Wing 372
373	373	Seattle	Juliet Bay 373
374	374	Tampa	Uniform Facility 374
375	375	San Jose	Quebec Hub 375
376	376	Dallas	Yankee Depot 376

Direct SQL insertion was implemented using prepared INSERT statements. This method provides precise control over data entry and is suitable for batch processing. Our insertTables.sql script demonstrates this approach with structured INSERT statements for all tables.

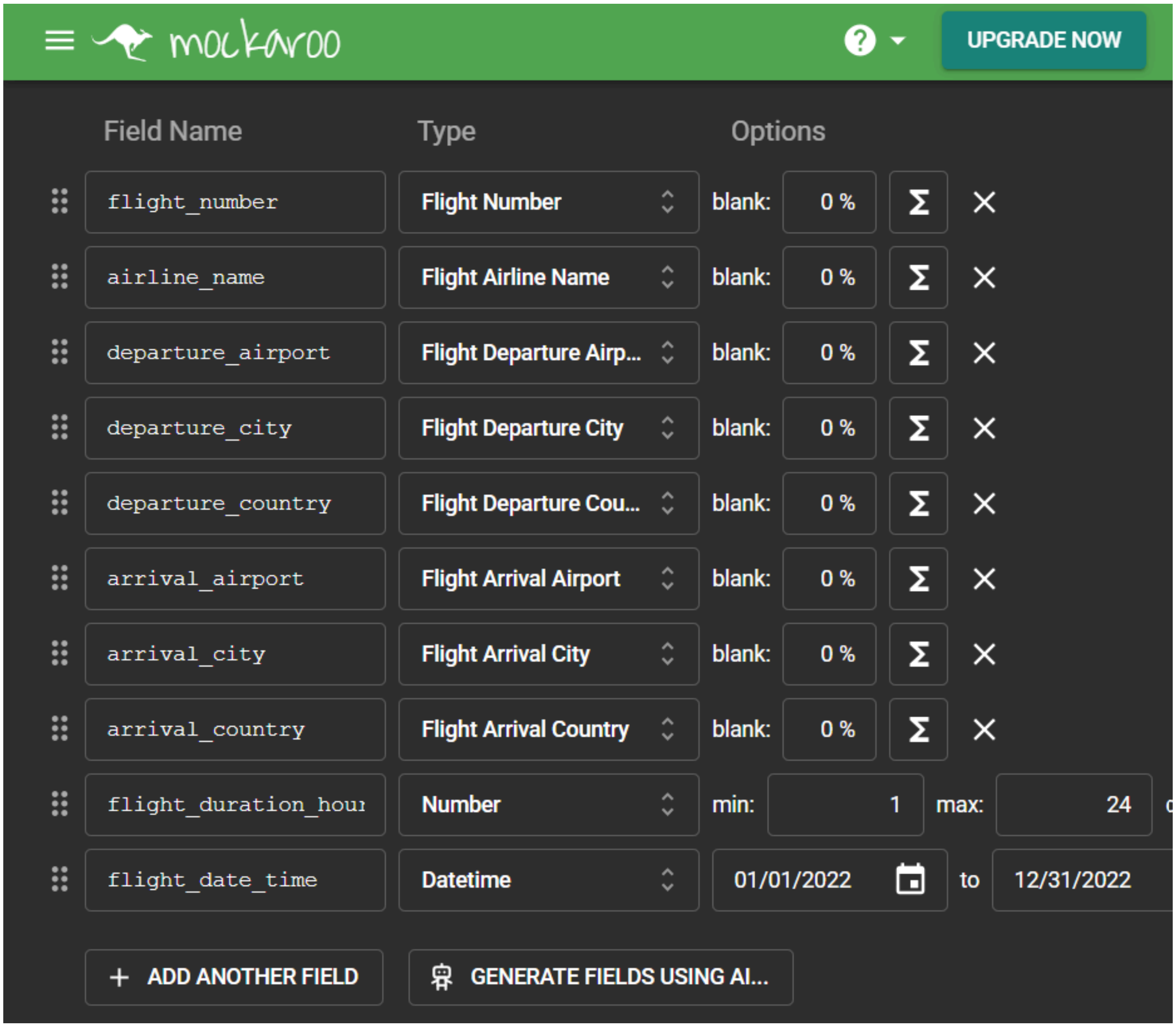
Method 2: Python Data Generation Script

```python
def main():

    # Open file for writing SQL statements
    with open('aviation_mock_data.sql', 'w') as f:
        f.write("-- Generated Mock Data for Aviation Database (PostgreSQL)\n\n")

        # 1. Hangar Table (400 rows)
        f.write("-- Hangar Table Data\n")
        for i in range(1, 401):
            location = random.choice(hangar_locations)
            name = f"{random.choice(hangar_name_prefixes)} {random.choice(hangar_name_suffixes)} {i}"
            f.write(f"INSERT INTO Hangar (Hangar_id, Location, Name) VALUES ({i}, '{location}', '{name}');\n")
        f.write("\n")

        # 2. Producer Table (400 rows)
        f.write("-- Producer Table Data\n")
        for i in range(1, 401):
            name_parts = []
            if random.random() < 0.7:  # 70% chance to use a predefined name
                name_parts.append(random.choice(producer_names))
            else:
                # Generate a random producer name
                name_parts.append(random.choice(["Aero", "Sky", "Air", "Jet", "Flight", "Wing", "Cloud"]) +
                                  random.choice(["Tech", "Systems", "Dynamics", "Motors", "Craft", "Works"]))

            if random.random() < 0.5:  # 50% chance to add a suffix
                name_parts.append(random.choice(producer_owner_types))

            pname = " ".join(name_parts)
            est_date = random_date(producer_start, producer_end)
            owner = f"{random.choice(['Dr.', 'Mr.', 'Ms.', 'Prof.'])} {random.choice(['John', 'Jane', 'Robert',
```

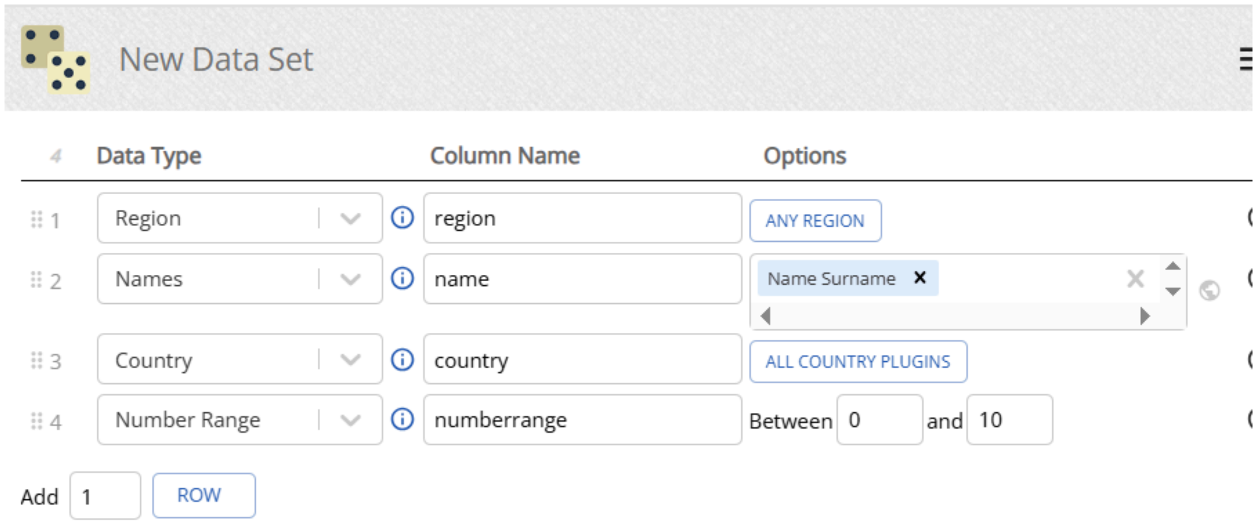
For bulk data entry, we developed a Python script (gen_data.py) that generates large amounts of realistic mock data. This approach is ideal for testing and development purposes. The script creates randomized but sensible data for all entities, maintaining proper relationships and constraints.

Method 3: Using Mackaroo and GenerateData

MACKAROO:



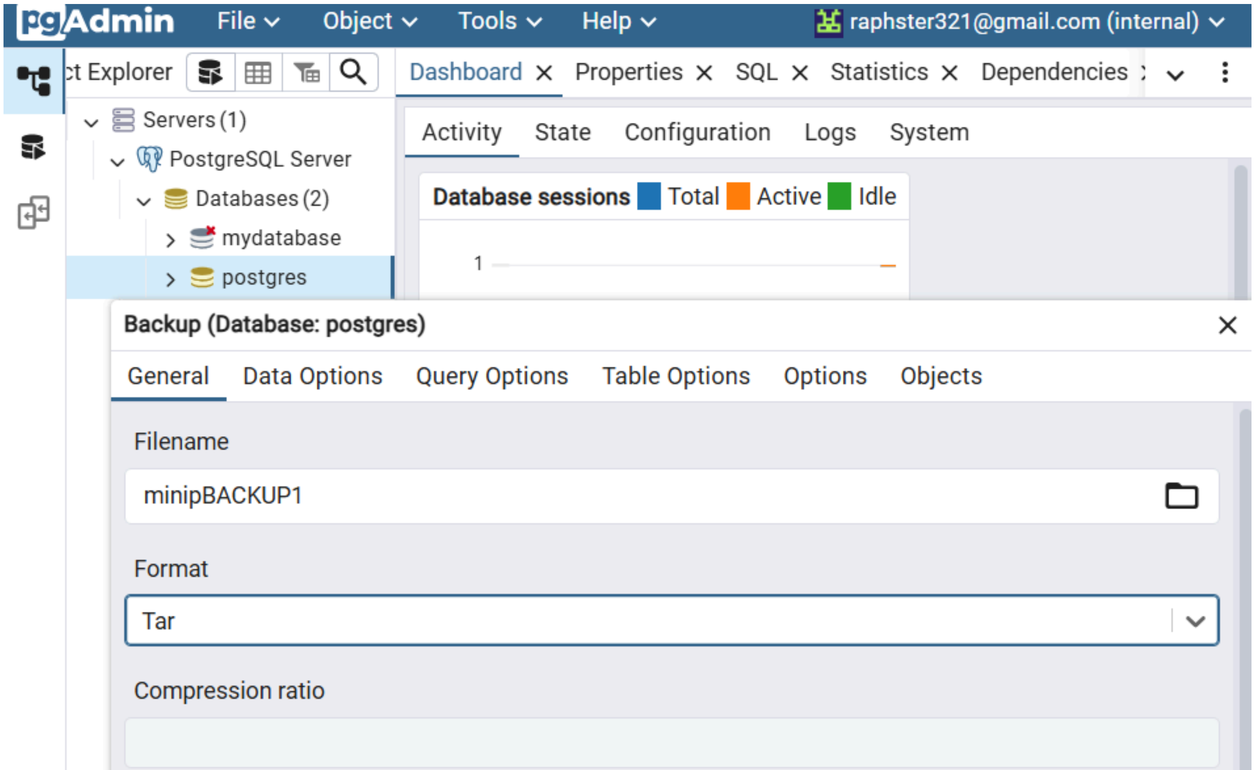
GENERATE DATA:



The database can also be maintained through PostgreSQL's client interfaces, which provide user-friendly forms for data entry. This method is appropriate for individual record management and ad-hoc updates by aviation staff.

Data Backup and Restoration

Database Backup Process

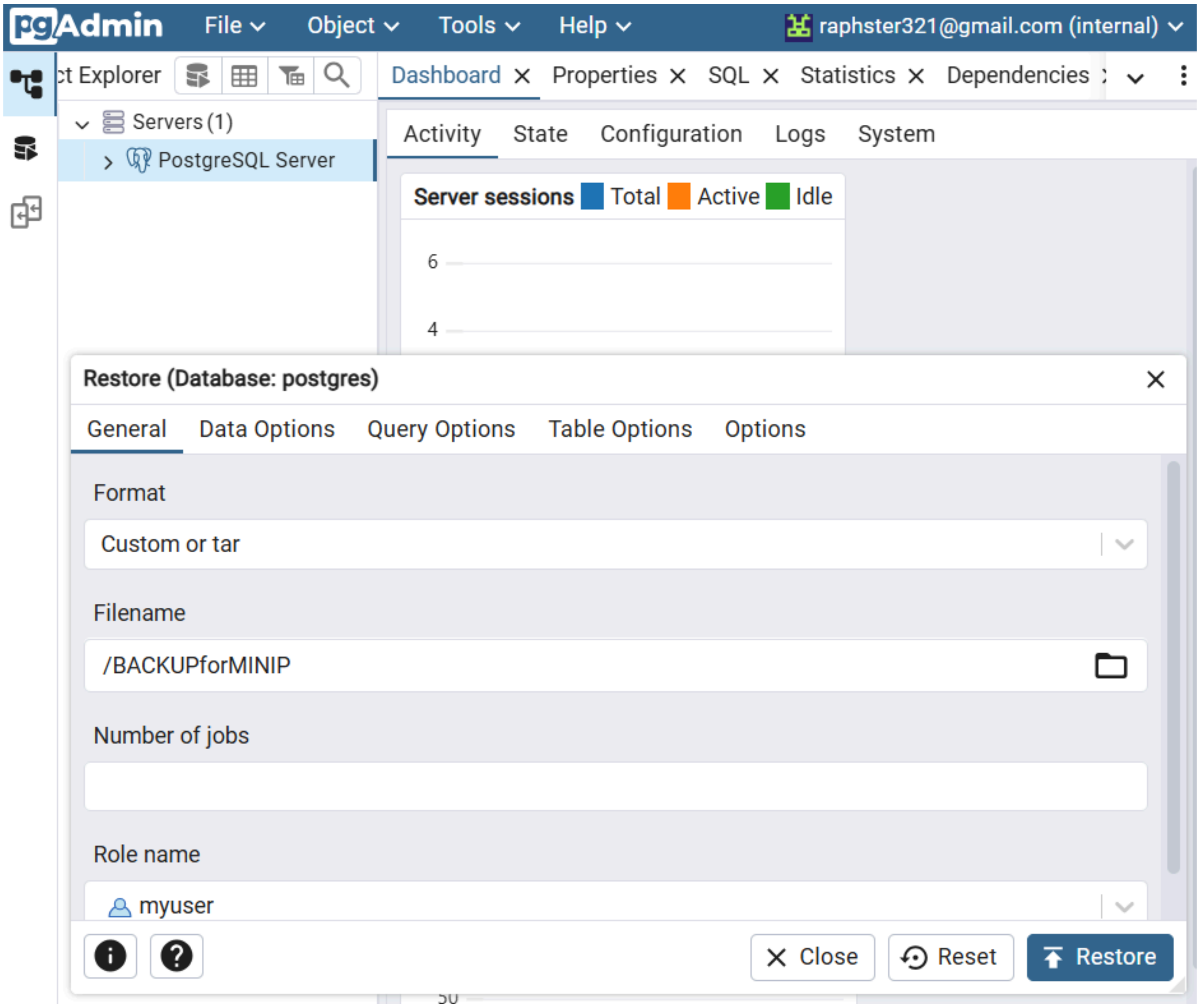


We implemented a systematic backup strategy using PostgreSQL's native pg_dump utility. This approach creates a complete snapshot of the database schema and data, which can be stored securely for disaster recovery purposes. The backup files are created in SQL format, allowing for easy inspection and modification if needed.

Command used:

pg_dump -U username -d aviation_db -f aviation_backup.sql

Database Restoration Process



The restoration process utilizes PostgreSQL's psql utility to reinstate the database from backup files. This method ensures that the entire database structure and content can be recovered in case of data loss or corruption.

Command used:

psql -U username -d aviation_db -f aviation_backup.sql

We tested the backup and restoration process thoroughly to ensure data integrity is maintained throughout the recovery procedure. The system successfully recovered all aviation data, including complex relationships between pilots, planes, and operators.