**Basic version of Creating Fake data in Contact Management System:**

To generate 500,000 rows of fake data in the contacts table, you’ll need to update the init.sql script. Manually adding 500,000 rows is impractical, so we’ll use a loop within SQL to insert the data automatically. Here’s how:

1. **Modify** init.sql **to Generate 500,000 Rows Using a Loop:**

PostgreSQL allows you to use the generate\_series function in combination with a INSERT statement. Here’s how to set it up:

sql

Copy code

-- Create contacts table

CREATE TABLE contacts (

id SERIAL PRIMARY KEY,

name VARCHAR(100) NOT NULL,

phone VARCHAR(15),

email VARCHAR(100),

address TEXT

);

-- Insert 500,000 rows of fake data

INSERT INTO contacts (name, phone, email, address)

SELECT

'Name\_' || generate\_series(1, 500000), -- Generates unique names like 'Name\_1', 'Name\_2', etc.

'555-' || LPAD((RANDOM() \* 10000)::INT::TEXT, 4, '0'), -- Generates random phone numbers

'user\_' || generate\_series(1, 500000) || '@example.com', -- Generates unique emails like 'user\_1@example.com'

'Address ' || generate\_series(1, 500000) || ', Springfield' -- Generates unique addresses like 'Address 1, Springfield'

FROM generate\_series(1, 500000);

* + **Explanation of the Insert Statement:**
    - name: Creates names like Name\_1, Name\_2, etc., using generate\_series.
    - phone: Generates random phone numbers with the format 555-XXXX, where XXXX is a random 4-digit number.
    - email: Creates unique emails like user\_1@example.com, user\_2@example.com, etc.
    - address: Generates addresses like Address 1, Springfield.

1. **Run Docker Compose**

With this updated init.sql in place, simply run:

bash

Copy code

docker-compose up -d

PostgreSQL will execute the init.sql script on startup, creating the table and populating it with 500,000 rows of fake data.

This setup will efficiently generate and insert 500,000 records without manually writing each row. Let me know if you need further assistance!

### Step 1: Create Directory and Files

1. **Create a new directory** for the project, e.g., contact\_management.
2. Inside this directory, create two files:
   * docker-compose.yml
   * init.sql

### Step 2: Docker Compose Configuration

In your docker-compose.yml file, add the following content:

yaml

Copy code

version: '3.8'

services:

postgres:

image: postgres:latest

environment:

POSTGRES\_USER: admin

POSTGRES\_PASSWORD: password

POSTGRES\_DB: contact\_db

volumes:

- db\_data:/var/lib/postgresql/data

- ./init.sql:/docker-entrypoint-initdb.d/init.sql

ports:

- "5432:5432"

networks:

- contact\_network

volumes:

db\_data:

networks:

contact\_network:

### Step 3: SQL Initialization Script

In the init.sql file, add the following SQL script to create the table and populate it with 500,000 rows of fake data:

sql

Copy code

-- Create contacts table

CREATE TABLE contacts (

id SERIAL PRIMARY KEY,

name VARCHAR(100) NOT NULL,

phone VARCHAR(15),

email VARCHAR(100),

address TEXT

);

-- Insert 500,000 rows of fake data

INSERT INTO contacts (name, phone, email, address)

SELECT

'Name\_' || generate\_series(1, 500000), -- Generates unique names like 'Name\_1', 'Name\_2', etc.

'555-' || LPAD((RANDOM() \* 10000)::INT::TEXT, 4, '0'), -- Generates random phone numbers

'user\_' || generate\_series(1, 500000) || '@example.com', -- Generates unique emails like 'user\_1@example.com'

'Address ' || generate\_series(1, 500000) || ', Springfield' -- Generates unique addresses like 'Address 1, Springfield'

FROM generate\_series(1, 500000);

### Step 4: Running the Setup

1. Open a terminal, navigate to your project directory (e.g., contact\_management), and start Docker Compose:

bash

Copy code

docker-compose up -d

This command will:

* + Start a PostgreSQL container with the database contact\_db.
  + Run the init.sql script to create the contacts table and insert 500,000 rows of fake data.

1. To verify the data, access the PostgreSQL container:

bash

Copy code

docker exec -it contact\_management\_postgres\_1 psql -U admin -d contact\_db

1. In the PostgreSQL command line, check the data by running:

sql

Copy code

SELECT COUNT(\*) FROM contacts;

This setup should create the database and populate it with the specified data volume. Let me know if you're ready to proceed to the next use case!

---------------------------------------------------------------------------------------------------------------------

INSERT INTO contacts (name, phone, email, address) SELECT 'Name\_' || generate\_series(1, 500000), -- Generates unique names like 'Name\_1', 'Name\_2', etc. '555-' || LPAD((RANDOM() \* 10000)::INT::TEXT, 4, '0'), -- Generates random phone numbers 'user\_' || generate\_series(1, 500000) || '@example.com', -- Generates unique emails like 'user\_1@example.com' 'Address ' || generate\_series(1, 500000) || ', Springfield' -- Generates unique addresses like 'Address 1, Springfield' FROM generate\_series(1, 500000); explain this part what is meant by '|'

###### ChatGPT said:

In PostgreSQL, the || symbol is used as the **concatenation operator** for strings. It joins or "concatenates" multiple string elements together into a single string. Here's a breakdown of how it's used in this part of the SQL statement:

sql

Copy code

INSERT INTO contacts (name, phone, email, address)

SELECT

'Name\_' || generate\_series(1, 500000), -- Concatenates 'Name\_' with numbers 1 to 500000

'555-' || LPAD((RANDOM() \* 10000)::INT::TEXT, 4, '0'), -- Concatenates '555-' with a random 4-digit number

'user\_' || generate\_series(1, 500000) || '@example.com', -- Concatenates 'user\_' with numbers and an email domain

'Address ' || generate\_series(1, 500000) || ', Springfield' -- Concatenates 'Address ' with numbers and a location

FROM generate\_series(1, 500000);

Here's what each part does:

1. 'Name\_' || generate\_series(1, 500000):
   * Concatenates the string 'Name\_' with numbers from 1 to 500,000 (e.g., 'Name\_1', 'Name\_2', …, 'Name\_500000').
2. '555-' || LPAD((RANDOM() \* 10000)::INT::TEXT, 4, '0'):
   * Creates random 4-digit phone numbers by:
     + Generating a random number (using RANDOM() \* 10000), converting it to an integer (::INT), then to text (::TEXT).
     + Padding this number to 4 digits with leading zeros using LPAD (if needed).
     + Concatenating the prefix '555-' with the generated 4-digit number, creating values like '555-0234'.
3. 'user\_' || generate\_series(1, 500000) || '@example.com':
   * Concatenates 'user\_' with numbers 1 to 500,000 and the domain '@example.com', generating email addresses like 'user\_1@example.com', 'user\_2@example.com', etc.
4. 'Address ' || generate\_series(1, 500000) || ', Springfield':
   * Concatenates 'Address ' with numbers 1 to 500,000 and the suffix ', Springfield', creating addresses like 'Address 1, Springfield', 'Address 2, Springfield', etc.

The || operator is essential here to build dynamic strings with unique values for each row.

INSERT INTO contacts (name, phone, email, address) SELECT 'Name\_' || generate\_series(1, 500000), -- Generates unique names like 'Name\_1', 'Name\_2', etc. '555-' || LPAD((RANDOM() \* 10000)::INT::TEXT, 4, '0'), -- Generates random phone numbers explain what is meant by LPAD, RANDOM(), and what is meant by ':'

###### ChatGPT said:

In your SQL query:

1. LPAD: This function stands for "left pad." It takes a string and adds a specified character (in this case, '0') to the left side of the string until it reaches a specified length. Here, LPAD((RANDOM() \* 10000)::INT::TEXT, 4, '0') means that the generated number is padded with zeros on the left if it has fewer than 4 digits. For example, if the random number is 123, it becomes 0123.
2. RANDOM(): This function generates a random decimal number between 0 and 1. Multiplying RANDOM() by 10000 gives a random number between 0 and 10000. By casting this to an integer, you get a random integer between 0 and 9999.
3. :: **(Double Colon)**: This is a casting operator in PostgreSQL. It converts a value from one data type to another. For example, (RANDOM() \* 10000)::INT casts the result to an integer, and then ::TEXT converts it to a text (string) format so that LPAD can work with it.

In summary:

* RANDOM() generates a random number.
* :: converts data types.
* LPAD pads the number to ensure it has four digits.

4o

##### You said:

I cant understand this line. I will say what it means if it is wrong correct me, The Random function will randomly give number between 0 and 1 then it will multiply by 10000 so the outcome will either be 0 or 10000 then why are you converting this to Integer 0 and 10000 is already an integer right then why are you converting it to a text and why you are padding it towards left.

###### ChatGPT said:

Memory updated

You're close in your understanding! Here’s a step-by-step explanation to clarify each part of that line:

1. RANDOM(): The RANDOM() function in PostgreSQL generates a decimal (floating-point) number between 0 and 1 (e.g., 0.1234, 0.8765), not strictly 0 or 1.
2. **Multiplying by** 10000: When you multiply the random decimal by 10000, you get a number between 0 and 9999.9999. This range gives you a wider variety of values, not just 0 or 10000.
3. **Casting to Integer** ::INT: After multiplying, the result might be a number with decimal points (e.g., 8765.4321). Casting it to INT removes the decimal part, leaving a whole number between 0 and 9999.
4. **Casting to Text** ::TEXT: After getting an integer (like 123), it’s converted to text so that the LPAD function can handle it. LPAD works on text values rather than numbers.
5. **Padding with** LPAD(..., 4, '0'): This ensures the final result is a four-digit string. If the integer result is 123, LPAD converts it to '0123'. Padding to four digits standardizes the phone number format.

So, the line ultimately generates a random 4-digit phone suffix by:

* Generating a decimal, multiplying, casting to integer, and converting it to a text string.
* Padding with zeros to ensure a consistent 4-digit format.

When multiplying the decimal value between 0 and 1 like ‘0.4476323295847222’ to 10000 it will move the four digit forward like this 4476.323295847222 then by converting into an integer you will get 4476., the four digit number we are expecting.

**Example 1**

* **Random Decimal**: 0.447632
* **Multiplied Value**: 0.447632 \* 10000 = 4476.323296
* **Integer Value**: 4476
* **Padded Result**: "4476" (no padding needed as it’s already 4 digits)
* **Example 2**
* **Random Decimal**: 0.108448
* **Multiplied Value**: 0.108448 \* 10000 = 1084.479132
* **Integer Value**: 1084
* **Padded Result**: "1084" (no padding needed as it’s already 4 digits)
* **Example 3**
* **Random Decimal**: 0.454212
* **Multiplied Value**: 0.454212 \* 10000 = 4542.115748
* **Integer Value**: 4542
* **Padded Result**: "4542" (no padding needed as it’s already 4 digits)

then obviously we are getting 4 numbers whatis the purpose of padding.

###### ChatGPT said:

While it's true that multiplying by 10000 followed by casting to an integer typically gives a 4-digit number, the padding with LPAD is a precautionary measure to ensure a uniform, consistent output. Random numbers in this range can occasionally result in fewer than four digits—for example, values such as 7, 45, or 302—because the integer casting doesn't inherently enforce a digit count; thus, without padding, you might end up with numbers that vary in length. By using LPAD, you guarantee that all results will appear as exactly four characters, with leading zeros added only when needed; this approach ensures that every output, regardless of initial length, adheres to the four-digit format, maintaining a standardized look across all entries. This technique is especially useful in scenarios like phone number generation, where uniformity is visually and functionally important.

****Copy****init.sql****into the Container****:

bash

Copy code

docker cp init.sql <correct\_container\_name>:/init.sql

****Enter the PostgreSQL Container****:

bash

Copy codedocker exec -it <correct\_container\_name> psql -U admin -d inventory\_db

****Run the Script****:

In the PostgreSQL prompt, execute:

sql

Copy code\i /init.sql