Hey database enthusiasts,

This note is dedicated to all the cool cats diving into the wild world of database design. Big high-fives to those nerding out over entities, relationships, and normalization – you're the real MVPs.

**CREATING A DATABASE**

To create a new database named "Test," use the following SQL command:

CREATE DATABASE Test;

To connect to the newly created database, use:

\c Test

**CAUTION: DROPPING A DATABASE**

Be extremely cautious with the `DROP DATABASE` command, as it permanently deletes a database and its contents. For instance, to drop the "Test" database:

DROP DATABASE Test;

**CLEARING THE CONSOLE**

You can clear the console screen using either `Control + L` or the command:

\! cls

**CREATING TABLES WITHOUT CONSTRAINTS**

To create a table named "Person" without constraints:

CREATE TABLE Person (

id INT,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

gender VARCHAR(7),

date\_of\_birth DATE

);

**VIEWING TABLES**

To view all tables in the current database:

\dt

To describe a specific table, like "Person":

\d person

**DROPPING A TABLE**

To drop a table, for example, the "Person" table:

DROP TABLE Person;

**CREATING TABLES WITH CONSTRAINTS**

To create a table "Person" with constraints:

CREATE TABLE Person (

id BIGSERIAL NOT NULL PRIMARY KEY,

first\_name VARCHAR(50) NOT NULL,

last\_name VARCHAR(50) NOT NULL,

gender VARCHAR(7) NOT NULL,

date\_of\_birth DATE NOT NULL,

email VARCHAR(150)

);

**INSERTING RECORDS INTO TABLES**

To insert a record into the "Person" table:

INSERT INTO person(

first\_name,

last\_name,

gender,

date\_of\_birth,

email)

VALUES ('Jake', 'Jones', 'MALE', DATE '1990-01-10', 'jake@example.com');

**USING MOCKAROO DATA: QUERIES FOR PRACTICE**

Use Mockaroo to generate 100 rows of mock data for the 'person' table

**SELECTING DATA**

Select all columns from the 'person' table

SELECT \* FROM person;

Select only the 'first\_name' column from the 'person' table

SELECT first\_name FROM person;

Select 'first\_name' and 'last\_name' columns from the 'person' table

SELECT first\_name, last\_name FROM person;

**ORDERING DATA**

Select all columns from 'person' table, ordered by 'country\_of\_birth' in ascending order

SELECT \* FROM person ORDER BY country\_of\_birth ASC;

Select all columns from 'person' table, ordered by 'country\_of\_birth' in descending order

SELECT \* FROM person ORDER BY country\_of\_birth DESC;

**DISTINCT VALUES**

Select distinct values of 'country\_of\_birth' from 'person' table, ordered alphabetically

SELECT DISTINCT country\_of\_birth FROM person ORDER BY country\_of\_birth;

**FILTERING DATA WITH WHERE CLAUSE & AND**

Select all columns from 'person' table where gender is 'Female'

SELECT \* FROM person WHERE gender = 'Female';

Select all columns from 'person' table where gender is 'Male'

SELECT \* FROM person WHERE gender = 'Male';

Select all columns from 'person' table where gender is 'Male' and country\_of\_birth is 'Poland'

SELECT \* FROM person WHERE gender = 'Male' AND country\_of\_birth = 'Poland';

Select all columns from 'person' table where gender is 'Male' and country\_of\_birth is either 'Poland' or 'China'

SELECT \* FROM person WHERE gender = 'Male' AND (country\_of\_birth = 'Poland' OR country\_of\_birth = 'China');

**LIMIT, OFFSET & FETCH**

Select all columns from 'person' table, limit to 10 rows

SELECT \* FROM person LIMIT 10;

Select all columns from 'person' table, skip the first 5 rows, and limit to the next 5 rows

SELECT \* FROM person OFFSET 5 LIMIT 5;

Select all columns from 'person' table, skip the first 5 rows, and fetch the next 5 rows only

SELECT \* FROM person OFFSET 5 FETCH FIRST 5 ROW ONLY;

**IN CLAUSE**

Select all columns from 'person' table where country\_of\_birth is in ('China', 'France', 'Brazil')

SELECT \* FROM person WHERE country\_of\_birth IN ('China', 'France', 'Brazil');

**BETWEEN CLAUSE**

Select all columns from 'person' table where date\_of\_birth is between '2000-01-01' and '2015-01-01'

SELECT \* FROM person WHERE date\_of\_birth BETWEEN DATE '2000-01-01' AND '2015-01-01';

**LIKE AND ILIKE**

Select all columns from 'person' table where email ends with '.com'

SELECT \* FROM person WHERE email LIKE '%.com';

Select all columns from 'person' table where email contains '@bloomberg.com'

SELECT \* FROM person WHERE email LIKE '%@bloomberg.com';

Select all columns from 'person' table where email contains '%google.%'

SELECT \* FROM person WHERE email LIKE '%google.%';

Select all columns from 'person' table where email has six characters before '@%'

SELECT \* FROM person WHERE email LIKE '\_\_\_\_\_\_@%';

Select all columns from 'person' table where country\_of\_birth starts with 'P'

SELECT \* FROM person WHERE country\_of\_birth LIKE 'P%';

Select all columns from 'person' table where country\_of\_birth starts with 'p' (case-insensitive)

SELECT \* FROM person WHERE country\_of\_birth ILIKE 'p%';

**GROUP BY CLAUSE**

Count the number of people born in each country and display the results

SELECT country\_of\_birth, COUNT(\*) FROM person GROUP BY country\_of\_birth ORDER BY country\_of\_birth;

**GROUP BY HAVING CLAUSE**

Count the number of people born in each country, but only display those with a count greater than 5

SELECT country\_of\_birth, COUNT(\*) FROM person GROUP BY country\_of\_birth HAVING COUNT(\*) > 5 ORDER BY country\_of\_birth;

**OTHER AGGREGATION FUNCTIONS**

Find the maximum price among all cars

SELECT MAX(price) FROM car;

Find the minimum price among all cars

SELECT MIN(price) FROM car;

Find the average price of all cars

SELECT AVG(price) FROM car;

Find the maximum price for each make and model of cars

SELECT make, model, MAX(price) FROM car GROUP BY make, model;

Find the total sum of all car prices

SELECT SUM(price) FROM car;

Find the total sum of prices for each make of cars

SELECT make, SUM(price) FROM car GROUP BY make;

**BASICS OF ARITHMETIC OPERATORS**

Select id, make, model, price, 10% of the price, and 90% of the price for each car

SELECT id, make, model, price, ROUND(price \* 0.10, 2) AS ten\_percent\_price, ROUND(price - (price \* 0.10), 2) AS discounted\_price FROM car;

**ALIAS IN SQL:**

In SQL, an alias is a temporary name assigned to a table or column for the duration of a query. It is used to make the SQL query more readable and concise.

Selecting specific columns and applying aliases

SELECT

id,

make,

model,

price AS original\_price, -- Renaming the 'price' column to 'original\_price'

-- Calculating 10% of the original price and rounding to 2 decimal places

ROUND(price \* 0.10, 2) AS ten\_percent,

-- Calculating the discounted price after a 10% reduction and rounding to 2 decimal places

ROUND(price - (price \* 0.10), 2) AS discount\_after\_ten\_percent

-- Specifying the source table 'car'

FROM car;

**COALESCE FUNCTION**

In SQL, the `COALESCE` function is used to handle NULL values. The primary purpose of `COALESCE` is to provide a default value when a NULL value is encountered in the result set.

-- Using COALESCE to handle NULL values in the 'email' column

SELECT COALESCE(email, 'Email Not Provided') FROM person;

**TIMESTAMPS AND DATES IN POSTGRESQL:**

In PostgreSQL, the `NOW()` function is used to obtain the current date and time.

-- Retrieving the current timestamp

SELECT NOW();

-- Extracting the date part from the current timestamp

SELECT NOW()::DATE;

-- Extracting the time part from the current timestamp

SELECT NOW()::TIME;

-- Calculating the timestamp one year ago

SELECT NOW() - INTERVAL '1 YEAR';

-- Calculating the timestamp ten years ago

SELECT NOW() - INTERVAL '10 YEARS';

-- Calculating the timestamp ten months ago

SELECT NOW() - INTERVAL '10 MONTHS';

-- Calculating the timestamp ten days ago

SELECT NOW() - INTERVAL '10 DAYS'

-- Calculating the timestamp ten days in the future

SELECT NOW() + INTERVAL '10 DAYS';

-- Calculating the date after adding ten months to the current timestamp

SELECT (NOW() + INTERVAL '10 MONTHS')::DATE;

**EXTRACTING FIELDS IN POSTGRESQL:**

In PostgreSQL, the `EXTRACT` function is used to retrieve date and time fields (such as year, month, day, etc.) from a timestamp or interval. It allows for more granular control when working with date and time values.

-- Retrieving the current timestamp

SELECT NOW();

-- Extracting the year from the current timestamp

SELECT EXTRACT(YEAR FROM NOW());

-- Extracting the month from the current timestamp

SELECT EXTRACT(MONTH FROM NOW());

-- Extracting the day from the current timestamp

SELECT EXTRACT(DAY FROM NOW());

-- Extracting the day of the week (0-6, where 0 is Sunday) from the current timestamp

SELECT EXTRACT(DOW FROM NOW());

-- Extracting the century from the current timestamp

SELECT EXTRACT(CENTURY FROM NOW());

**AGE FUNCTION IN POSTGRESQL**

The AGE function in PostgreSQL is a powerful tool for calculating the age of individuals by comparing their date of birth with the current date.

SELECT

first\_name,

gender,

date\_of\_birth,

AGE(NOW(), date\_of\_birth) AS age

FROM

person;

**PRIMARY KEYS IN POSTGRESQL**

In PostgreSQL, a primary key is a unique identifier for a record in a table. It ensures that each row in a table is uniquely identified, and it cannot contain NULL values. The primary key is a critical concept in database design, as it facilitates efficient data retrieval and maintains data integrity.

Creating a Primary Key:

-- Adding a primary key to the "employees" table on the "employee\_id" column

ALTER TABLE employees

ADD CONSTRAINT pk\_employees PRIMARY KEY (employee\_id);

Dropping a Primary Key:

-- Dropping the primary key constraint "pk\_employees" from the "employees" table

ALTER TABLE employees

DROP CONSTRAINT pk\_employees;

**FOREIGN KEYS IN POSTGRESQL**

A foreign key establishes a link between two tables, ensuring referential integrity between them. It is a field in one table that refers to the primary key in another table. Foreign keys play a crucial role in maintaining relationships and consistency in a relational database.

Creating a Foreign Key:

-- Adding a foreign key to the "orders" table referencing the "customer\_id" column in the "customers" table

ALTER TABLE orders

ADD CONSTRAINT fk\_orders\_customers

FOREIGN KEY (customer\_id)

REFERENCES customers (customer\_id);

Dropping a Foreign Key:

-- Dropping the foreign key constraint "fk\_orders\_customers" from the "orders" table

ALTER TABLE orders

DROP CONSTRAINT fk\_orders\_customers;

**UNIQUE CONSTRAINTS IN POSTGRESQL**

A unique constraint ensures that the values in a column or a set of columns are unique across all the rows in a table. This constraint is helpful when you want to enforce the uniqueness of data in a specific column or combination of columns.

Creating a Unique Constraint:

-- Adding a unique constraint to the "email" column in the "users" table

ALTER TABLE users

ADD CONSTRAINT uk\_users\_email UNIQUE (email);

Dropping a Unique Constraint:

-- Dropping the unique constraint "uk\_users\_email" from the "users" table

ALTER TABLE users

DROP CONSTRAINT uk\_users\_email;

Certainly! Here's an enhanced note on deleting and updating records using the `DELETE` and `UPDATE` statements in PostgreSQL:

**DELETING RECORDS IN POSTGRESQL**

To delete all records from a table, you can use the `DELETE` statement without a `WHERE` clause:

-- Deleting all records from the "person" table

DELETE FROM person;

This operation removes all rows from the specified table, effectively emptying it. Use this with caution, as it irreversibly removes all data.

To delete specific records based on a condition, you can use the `DELETE` statement with a `WHERE` clause. Here's an example:

-- Deleting records from the "person" table where the "id" is 1000

DELETE FROM person WHERE id = 1000;

This statement deletes rows from the "person" table where the value in the "id" column is equal to 1000. Adjust the condition based on your specific criteria.

**UPDATING RECORDS IN POSTGRESQL**

To update existing records in a table, you can use the `UPDATE` statement.

Here's an example of updating the "email" column in the "person" table for a specific record:

-- Updating the "email" column to 'omar@gmail.com' for the record where "id" is 200

UPDATE person SET email = 'omar@gmail.com' WHERE id = 200;

This statement modifies the "email" column value to 'omar@gmail.com' for the record in the "person" table where the "id" is 200. Adjust the column and condition based on your specific requirements.

**CONFLICT RESOLUTION IN POSTGRESQL**

The `ON CONFLICT DO NOTHING` clause allows you to handle conflicts during `INSERT` operations by taking no action when a conflict (violation of a unique constraint) occurs. This is useful when you want to avoid inserting a record if a conflicting record with the same unique key already exists.

-- Inserting a record into the "person" table with id 1000 and first\_name 'Russ'

-- If a conflict occurs (if a record with id 1000 already exists), do nothing

INSERT INTO person (id, first\_name) VALUES (1000, 'Russ') ON CONFLICT (id) DO NOTHING;

**UPSERT (ON CONFLICT DO UPDATE):**

The `ON CONFLICT DO UPDATE` clause, commonly known as "upsert," allows you to handle conflicts by updating the existing record when a conflict occurs.

-- Inserting a record into the "person" table with id 1000, first\_name 'Russ', and email 'russ@mail.com'

-- If a conflict occurs (if a record with id 1000 already exists), update the email column with the new value

INSERT INTO person (id, first\_name, email) VALUES (1000, 'Russ', 'russ@mail.com') ON CONFLICT (id) DO UPDATE SET email = EXCLUDED.email;

**JOINS IN POSTGRESQL**

**Inner Join**

An inner join retrieves records from two or more tables where there is a match based on the specified join condition.

-- Retrieving all columns from the "person" and "car" tables where the person's car\_id matches the car's id

SELECT \* FROM person JOIN car ON person.car\_id = car.id;

This statement retrieves all columns from both the "person" and "car" tables where there is a match between the "car\_id" column in the "person" table and the "id" column in the "car" table.

-- Retrieving specific columns from the "person" and "car" tables based on the join condition

SELECT person.first\_name, car.make, car.model, car.price FROM person JOIN car ON person.car\_id = car.id;

This statement retrieves specific columns from the "person" and "car" tables, including the person's first name, and the car's make, model, and price.

**Left Join**

A left join (or left outer join) retrieves all records from the left table (the table mentioned before `LEFT JOIN`) and the matching records from the right table.

-- Retrieving all columns from the "person" table and matching records from the "car" table based on the person's car\_id

SELECT \* FROM person LEFT JOIN car ON person.car\_id = car.id;

This statement retrieves all columns from the "person" table and includes matching records from the "car" table based on the condition that the "car\_id" in the "person" table matches the "id" in the "car" table. If there is no match, columns from the "car" table will contain NULL values.

**UUID**

A UUID, or Universally Unique Identifier, is a standardized 128-bit identifier that is unique across both space and time. The uniqueness of a UUID is primarily guaranteed by the combination of current timestamp, clock sequence, and a node identifier, which is typically the machine's network address.

PostgreSQL has built-in support for UUIDs. You can use the uuid data type to store UUID values in your tables. Additionally, PostgreSQL provides functions for generating UUIDs (uuid-ossp extension) and various operators for working with them.

-- Creating a table with a UUID column

CREATE TABLE my\_table (

id UUID PRIMARY KEY,

name VARCHAR(50)

);

-- Inserting a record with a UUID

INSERT INTO my\_table (id, name) VALUES ('550e8400-e29b-41d4-a716-446655440000', 'John Doe');

-- Querying records based on a UUID

SELECT \* FROM my\_table WHERE id = '550e8400-e29b-41d4-a716-446655440000';

Big Thanks!