

How Do I Make Sure My Database Stays Intact?

Data Types As Constraints

Columns of a PostgreSQL database table must have a data type, which constrains the type of information that can be entered into that column. This is important in order to ensure data integrity and consistency over time. Some common PostgreSQL types are `integer`, `decimal`, `varchar`, and `boolean`. Data types are defined in a `CREATE TABLE` statement by indicating the data type after each column name.

```
CREATE TABLE tablename (  
    myNum integer,  
    myString varchar(50)  
);
```

Check Constraints

When using PostgreSQL, it can be important to enforce check constraints on columns of a database table in order to ensure data integrity and consistency over time. Check constraints can be enforced on a single column, multiple columns, or on all columns. They are implemented within a `CREATE TABLE` statement using the `CHECK` keyword.

```
CREATE TABLE table_name (  
    column_1 integer,  
    column_2 text,  
    column_3 numeric CHECK (column_3 > 0),  
    column_4 numeric CHECK (column_4 > 0),  
    CHECK (column_3 > column_4)  
);
```

Multiple Constraints

Columns in a database table can have multiple constraints. Multiple constraints can be implemented by listing them in a row following the relevant column name and data type in a `CREATE TABLE` statement. The order of the constraints does not matter.

```
CREATE TABLE table_name (  
    column_1 integer NOT NULL CHECK  
(column_3 > 0),  
    column_2 text UNIQUE NOT NULL,  
    column_3 numeric  
);
```

NOT NULL

In PostgreSQL, `NOT NULL` constraints can be used to ensure that particular columns of a database table do not contain missing data. This is important for ensuring database integrity and consistency over time. `NOT`

```
CREATE TABLE table_name (  
    column_1 integer NOT NULL,  
    column_2 text NOT NULL,
```

NULL constraints can be enforced within a `CREATE TABLE` statement using `NOT NULL`.

UNIQUE

In PostgreSQL, `UNIQUE` constraints can be used to ensure that elements of a particular column (or group of columns) are unique (i.e., no two rows have the same value or combination of values). This is important for ensuring database integrity and consistency over time.

`UNIQUE` constraints can be enforced within a `CREATE TABLE` statement using the `UNIQUE` keyword.

Primary Key Constraint

In PostgreSQL, a primary key constraint indicates that a particular column (or group of columns) in a database table can be used to identify a unique row in that table. In terms of restrictions, this is equivalent to a `UNIQUE NOT NULL` constraint; however, a table may only have one primary key, whereas multiple columns can be constrained as `UNIQUE NOT NULL`. A primary key constraint can be enforced within a `CREATE TABLE` statement using `PRIMARY KEY`.

Foreign Key Constraint

In PostgreSQL, a foreign key constraint ensures that the values in a particular column of a database table exactly match values in another database table. This is important to ensure the “referential integrity” of the database. A foreign key constraint can be enforced within a `CREATE TABLE` statement either by adding `REFERENCES other_table_name (other_table_primary_key)` after the relevant column name and type or using `FOREIGN KEY (column_1, column_2) REFERENCES other_table_name (other_key1, other_key2)` to indicate a link between groups of columns.

```
column_3 numeric
);
```

codecademy

```
CREATE TABLE table_name (
    column_1 integer UNIQUE,
    column_2 text UNIQUE,
    column_3 numeric,
    column_4 text,
    UNIQUE(column_3, column_4)
);
```

```
-- A primary key on one column
```

```
CREATE TABLE table_name (
    column_1 integer PRIMARY KEY,
    column_2 text,
);
```

```
-- A composite primary key
```

```
CREATE TABLE table_name (
    column_1 integer,
    column_2 text,
    column_2 integer,
    PRIMARY KEY (column_1, column_2),
);
```

```
CREATE TABLE table_1 (
    column_1 integer PRIMARY KEY,
    column_2 text,
    column_3 numeric
);
```

```
-- Option 1
```

```
CREATE TABLE table_2 (
    column_a integer PRIMARY KEY,
    column_b integer REFERENCES table_1
(column_2),
```

```
        column_c integer
    );

-- Option 2 - Creating columns b and c are
a composite foreign key referencing
columns c1 and c2 from other_table
CREATE TABLE t1 (
    a integer PRIMARY KEY,
    b integer,
    c integer,
    FOREIGN KEY (b, c) REFERENCES
other_table (c1, c2)
);
```

Cascade and Restrict

In PostgreSQL, when implementing foreign key constraints in a database table, it is possible to preemptively specify database behavior when values in a referenced column are updated or deleted. Specifying `ON DELETE RESTRICT` or `ON UPDATE RESTRICT` after a foreign key constraint ensures that referenced values/rows cannot be deleted/updated. Specifying `ON DELETE CASCADE` or `ON UPDATE CASCADE` ensures that updated/deleted values/rows in the referenced table are automatically updated/deleted in the referencing table.

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
);

CREATE TABLE order_items (
    product_no integer REFERENCES products
ON DELETE RESTRICT,
    order_id integer REFERENCES orders ON
DELETE CASCADE,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

Updating A Table With Constraints

In PostgreSQL, when implementing a constraint on an existing table, the table must already be consistent with the constraint or PostgreSQL will reject the new constraint. A DB user may backfill the table using `UPDATE` or `ALTER TABLE` statements to make

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
```

the table consistent with the constraint.

```
price numeric,
sale_price numeric
);
```



```
--Assume some values in `sale_price` are
missing, and we'd like to apply a `NOT
NULL` constraint on `sale_price`. We must
`UPDATE` the table before applying our
constraint.
```

```
UPDATE TABLE products
SET sale_price = 0 WHERE sale_price IS
NULL
```

PostgreSQL's default Superuser

PostgreSQL database clusters are created with a default role named `Postgres`. This is a special role that has `SUPERUSER` privileges and can modify all access restrictions within the database.

```
-- On a new database cluster you can check
the existing role with either:
```

```
-- From a Query, using `pg_roles` table
SELECT rolname, rolsuper
FROM pg_roles
WHERE rolsuper = true;
```

```
/*
+-----+-----+
|      rolname      | rolsuper |
+-----+-----+
| postgres          | t        |
+-----+-----+
*/
```

```
-- In PSQL
-- => \du
```

```
/*
List of roles
+-----+-----+
-----+-----
```

```

----+
|  Role name      |
Attributes          |
Member of |
+-----+-----+
-----+-----+
----+
| postgres        | Superuser, Create role,
Create DB, Replication, Bypass RLS | {}
|
+-----+-----+
-----+-----+
----+
*/

```

Principle of Least Privilege

The principle of least privilege says that applications and users should have the minimum permissions required for their function. In PostgreSQL, this means superusers should not be performing routine database tasks and specialized roles should be created for each user or application.

Checking User Permissions

In PostgreSQL, a user may check the system table, `pg_catalog.pg_roles` to understand what permissions users in the DB have. They may also check the `information_schema.table_privileges` table for more granular description of permissions at a table level.

```
-- Checking User Permissions (Results
orientation to horizontal...), true/false
for values like `rolsuper`, `rolcanlogin`,
`rolcreatorole`, etc.
```

```
SELECT *
FROM pg_catalog.pg_roles
LIMIT 1
```

```
/*
```

```

+--[ RECORD 1 ]-----+
| rolname          | pg_signal_backend |
| rolsuper         | f                  |
| rolinherit       | t                  |
| rolcreatorole    | f                  |
| rolcreatedb      | f                  |

```

```

| rolcanlogin      | f          |
| rolreplication  | f          |
| rolbypassrls     | f          |
+-----+-----+
*/

-- Checking Table Permissions

select
    grantor,
    grantee,
    table_schema,
    table_name,
    privilege_type
FROM information_schema.table_privileges
WHERE table_name = 'sample';

/*
+-----+-----+-----+
---+-----+-----+
| grantor | grantee |
table_schema | table_name | privilege_type
|
+-----+-----+-----+
---+-----+-----+
| postgres | user_1 | public
| sample   | INSERT |
| postgres | user_1 | public
| sample   | SELECT |
| postgres | user_1 | public
| sample   | UPDATE |
| postgres | user_1 | public
| sample   | DELETE |
| postgres | user_1 | public
| sample   | TRUNCATE |
| postgres | user_1 | public
| sample   | REFERENCES |
| postgres | user_1 | public
| sample   | TRIGGER |
+-----+-----+-----+
---+-----+-----+

```

*/

PostgreSQL Roles

In PostgreSQL, roles are a collection of privileges that can be granted to one or more users.

SET into other roles in PostgreSQL

In PostgreSQL, a superuser can use `SET ROLE` to modify the current session to test the permissions of another user.

```
-- Step 1.
SELECT current_user;

/*
+-----+
| current_user |
+-----+
| postgres     |
+-----+
*/

-- Step 2. Using SET, change the current
permissions to mimic another user's (if
user has proper permissions)
SET ROLE user_1;

-- Step 3. Confirm change
SELECT current_user;


/*
+-----+
| current_user |
+-----+
| user_1       |
+-----+
*/
```

Creating Roles in PostgreSQL

In PostgreSQL, `CREATE ROLE` statements allow a user to create a new role. Keywords passed to `CREATE ROLE` like `VALID UNTIL`, `LOGIN`, `IN GROUP`, `CREATEUSER`, `CREATEDB` and `PASSWORD` allow for customization of the new user's privileges.

GRANTING and REVOKING Database Privileges

In PostgreSQL, `GRANT` and `REVOKE` statements can be used to modify what privileges users have on an existing database object (e.g. schemas, databases, sequences, functions, etc.)

```
-- Create a role named `miriam` 
`NOSUPERUSER` (is not a superuser) and
`LOGIN` (can log in to the DB)
CREATE ROLE miriam WITH NOSUPERUSER LOGIN;
```

```
-- To GRANT Permissions
GRANT SELECT, UPDATE
ON finance.revenue
TO analyst;
```

```
-- To REVOKE Permissions
REVOKE SELECT, UPDATE
ON finance.revenue
FROM analyst;
```

ALTER DEFAULT Permissions and Privilege

In PostgreSQL, `ALTER DEFAULT` statements can be used to set privileges on all new objects in a database or schema, this statement does not affect existing objects.

```
-- Alter the default permissions to a
schema, combine `ALTER DEFAULT` with a
`GRANT` statement
ALTER DEFAULT PRIVILEGES IN SCHEMA finance
GRANT SELECT ON TABLES TO analyst;
```

Creating a Group Role

In PostgreSQL, group roles are used to distribute permissions to sets of users. A `CREATE ROLE` statement that includes `WITH ROLE name name name....` can be used to create a group comprised of a set of existing roles.

```
-- Create a group role named `marketing`
with `alice` and `bob` as members.
CREATE ROLE marketing
WITH NOLOGIN ROLE alice, bob;
```

Groups and Roles in PostgreSQL

In PostgreSQL, because groups are roles themselves, you can create a role and `GRANT` access to that group. The privileges granted to that group will also apply to the group's members.

```
-- Group roles can be created with CREATE
ROLE
CREATE ROLE finance WITH NOLOGIN;
```



```
-- Then the permissions of that role
(`finance`) can be
-- `GRANT`ed to another role
GRANT finance TO charlie;
```

GRANT Permissions for subsets of columns

In PostgreSQL, GRANT statements can be applied to specific columns in a PostgreSQL table to limit the permissions a given user can access or modify.

```
-- GRANT SELECT on columns a, b, d. When
user_1 queries sample table for a, b, c, d
they'll receive an error.
GRANT SELECT (a, b, d)
ON sample_table TO user_1;

-- Set role into user_1...
SET ROLE user_1;

-- Attempted query on sample_table
SELECT * FROM sample_table;

-- ERROR:  permission denied for table
sample_table.
```

PostgreSQL Row Level Security

In PostgreSQL, row level security is a pattern that only grants access to certain rows of a relation to a given role/group. Row level security can be implemented with CREATE POLICY and ENABLE ROW LEVEL SECURITY statements.

```
-- Create a RLS Policy, only allow user's
to return rows where current_user is equal
to the row's column `salesperson`
CREATE POLICY emp_rls_policy ON accounts
FOR SELECT TO employees USING
(salesperson=current_user);

-- Alter table to enable the new RLS
policy
ALTER TABLE accounts ENABLE ROW LEVEL
SECURITY;
```

Database Transaction with ACID Properties

When working with a database, a single unit of work is defined as a transaction. While this unit of work can vary between databases in terms of its complexity, every transaction must maintain ACID properties. This allows for concurrent use of the database and helps with recovery in the case of a system failure.

Step Number	Actions
1	SELECT * FROM database.db
2	INSERT INTO database.db
3	SELECT * FROM database.db

// The above set of queries is an example of a possible database transaction. Note that in some databases a transaction could be a single query.

Atomicity in ACID

When working with a database transaction, the ACID property *Atomic* means that if any part of the transaction fails, then the entire transaction will fail as well. This means that the database would remain unchanged once the transaction has finished if any one part fails.

Step Number	Actions
1	SELECT * FROM database.db
2	INSERT INTO database.db
3	SELECT * FROM database.db

// In the above transaction, to demonstrate atomicity, should any one step fail then every other step in the transaction would fail. For instance, if steps 1 and 2 passed but step 3 failed, then the transaction would revert and fail steps 1 and 2.

Consistency in ACID

In most databases, certain constraints and triggers exist within it to ensure that all of the data is consistent. Transaction within the database must also follow these

Database Rules

rules to maintain Consistency in the ACID properties.

```

|-----codecademy
----|
| All employee salaries must be an integer
|
| All employee IDs must be unique
|
| No store can have more than fifty
employees |

// In the above set of rules for a
database, any transaction wanting to
maintain Consistency must follow each
rule. Should any rule be broken, the
transaction would no longer maintain ACID
properties.

```

Isolation in ACID

When using multiple database transactions at once, the ACID property of Isolation ensures that no two transactions interact with each other at the same time. Should two transactions end up interacting, they will be performed sequentially as to maintain isolation.

```

| Transaction 1          | Transaction 2
|
|-----|-----
----|
| SELECT * FROM table1 | SELECT * FROM
table2 |
| INSERT INTO table2   | INSERT INTO
table1   |
| INSERT INTO table 1  | INSERT INTO
table1   |
// Notice how in these two transactions
that both are set to interact with table1
and table2. Because of this, the
transactions would occur sequentially to
maintain the Isolation ACID property.

```

Durability in ACID

Once a database transaction has finished its operations, it must ensure that its operations were fully committed. This act is the Durability part of ACID properties, and makes sure the data can be recovered should anything go wrong.

