Single Table

Dr Peadar Grant

October 1, 2024

Contents

1 Introduction

This week's class covers the setup and usage of a basic single-table database.

As a data analyst:

- You will often encounter situations where you will run queries on databases that already exist.
- Other times you will be tasked with setting up a database to conduct one-off or ongoing analyses.

We will therefore look at how data needs to be modelled for a **single table**, focusing on:

1. Column names
2. Data types
3. Constraints to improve data integrity.

1.1 Prerequisities

Before starting make sure that you:

- created your SSH public/private key-pair.
- can login over SSH to the shared server.

If not

Talk to me immediately!

2 Single-table databases

Recall that:

- A PostgreSQL *cluster* provides:
- One or more **databases** that each have
- (One or more **schemas** that each hold)
- One or more **tables**

where each table:

- has separate **columns** holding individual attribute values
- stores data each record in **rows**

2.1 Data definition language

Key concepts will be easily learned through repeated practice and reference to

CREATE table to define table name, columns (type, constraints)

ALTER table for incremental changes.

DROP table without warning!

Sometimes it's easiest to specify information up front in the `CREATE` statement.

Often it's better to start with a workable definition and amend using `ALTER`.

2.2 Sample problem

We will mainly do this by example.

Example

Mary is a local crafter who makes her own pottery. She runs a small shop where she sells her own wares.
At the moment she writes down all sales in a book:

- Date and sometimes time
- Description of what sold
- Quantity
- Price
- Total

Having done a database class, she realises that she could easily put together a table to replace the book.

2.3 Table names

Following a consistent and workable pattern with table names will help a lot:

1. names should be **descriptive**
    - decide on a sensible structure on pluralisation
2. try to keep names reasonablly **short**
    - You will be typing them a lot!
3. do not use spaces, use the underscore instead (_)
    - spaces in most text-based systems are a receipe for disaster!
4. table names are **case insensitive**

Table names *can* be changed afterwards.

Our table will be called **sales**.

2.4 Column names

Similar set of rules for columns as for tables:

Again following a consistent and workable pattern with column names will help a lot:

1. names should be **descriptive**
2. try to keep names reasonablly **short**
3. do not use spaces, use the underscore instead (_)

Column names *can* be changed afterwards.

Example

In our example, we'll choose the following column names:

1. `timestamp`
2. `description`
3. `quantity`
4. `total_price`

3 Datatypes and constraints

Choosing the correct datatypes is vitally important for making a usable database!

TEXT for text (only!)

- Other database systems use CHAR, VARCHAR etc. Avoid!

Numeric data largely falls into the categories of: integer, fixed precision and arbitary precision data.

INT for integers

- Also have `SMALLINT`, `BIGINT` etc for range

NUMERIC for numeric data incl. decimals and currency

- Easy to make mistakes choosing float instead!

3.1 Good practices

1. Do not store numbers as text.
2. Do not store boolean true / false (or any synonyms) as text.
3. Do not sure enumerated types as text. Either use an ENUM and/or a foreign-keyed table.
4. Anything where you need absolute precision after the decimal point must be NUMERIC, not FLOAT.
5. Do not be tempted to store a single logical date/time as separate date and time columns. Always use a single timestamp for this.

3.2 NOT NULL

The NOT NULL constraint prevents a particular column value being null. Writes will fail if set to NULL or if unspecified unless default column value specified.

```
-- During table creation:
CREATE TABLE tasks (
   description text not null,
   /* other columns */
);
```

3.2.1 Changing NOT NULL status

Whether NULL should be allowed depends on problem of interest. By default NULL is allowed. Must specify NOT NULL if not.

Can modify the column's NULLable status afterwards:

```
-- make a column NOT NULL
ALTER TABLE tasks ALTER COLUMN description SET NOT NULL

-- remove the NOT NULL constraint from a column
ALTER TABLE tasks ALTER COLUMN description DROP NOT NULL
```

Example

In our example we will set all columns to not null, as none are optional.

3.3 UNIQUE

The UNIQUE constraint prohibits two rows from having an equal set of attribute values for the columns specified:

- Use UNIQUE after column definition if column should be unique
- Use UNIQUE(col1, col2) if two or more columns should be unique after the column definitions

```
-- During table creation:
CREATE TABLE tasks (
   description text unique,
   /* other columns as necessary */
   project_code text,
   subproject_code text,
   unique(project,subproject)
);
```

Example

In our example we've no UNIQUE columns (yet!)

3.3.1 Modifying UNIQUE afterwards

Just as with NOT NULL we can modify UNIQUE later on:

```
-- Afterwards
ALTER TABLE tasks ADD UNIQUE (project, subproject);
```

3.4 Default values

Default values are automatically inserted when unspecified in INSERT statement. Often used in conjunction with NOT NULL. Can be static value or based on functions (see later).

```
-- when creating a table
CREATE TABLE sales (
    /* other columns  */
    quantity smallint not null default 1;
    timestamp timestamp not null default now();
);
```

3.4.1 Altering DEFAULT

```
-- afterwards
ALTER TABLE tasks ALTER COLUMN description SET default 'xyz'
ALTER TABLE tasks ALTER COLUMN description DROP default
```

4 Primary key

Primary key value is a **unique** AND **not null**.

Unambiguously identifies each row.

Every table should have 1 primary key. Cannot have more than 1.

Most DBMS do not enforce this, but you should consider it mandatory. Use PRIMARY KEY if column is the primary key.

Example

In our example, we *could* use the timestamp as the primary key.
But it would be much better to make an autoincrementing integer `id` column.

```
-- specifying primary key
CREATE TABLE accounts (
    account_number bigint primary key
    /* other columns as necessary */
);

-- primary key covering two columns
CREATE TABLE accounts (
    branch bigint, -- not null automatic
    account_number bigint,  -- not null automatic
    /* other columns */
    primary key (branch, account_number)
    /* other constraints etc. */
)
```

4.1 Auto-increment column

We can use the column types SERIAL or BIGSERIAL for auto-incrementing columns. Generally prefer SERIAL or BIGSERIAL.

Shorthand for default value based on *sequence generator* (later on).

```
CREATE TABLE sales (
    id BIGSERIAL PRIMARY KEY,
    /* other columns & constraints */
);
```

5 Transaction control

PostgreSQL transaction control can be used in simplest form to give basic “undo” capability:

```
-- start a new transaction
BEGIN;

/* execute SQL statements then either: */

COMMIT; /* save the changes */
-- or
ROLLBACK; /* undo the changes */
```

5.1 Error handling

If we don't BEGIN a transaction we implicitly BEGIN before the statement and COMMIT after it. Errors (e.g. syntax) will abort the transaction, requiring a ROLLBACK before any more statements will be accepted.

5.2 Savepoints

```
BEGIN;

/* sql statement block 1 */

-- define a savepoint
SAVEPOINT sp1;

/* sql statement block 2 */

-- rollback to the savepoint undoes block 2;
ROLLBACK TO sp1;

-- then either:
COMMIT; /* or ROLLBACK */
```

6 DML operations

The main **Data Manipulation Language** operations you'll use are grouped into:

INSERT rows populating specified columns with data given.

UPDATE table, optionally select rows with `WHERE`.

DELETE rows from a table, optionally select rows with `WHERE`.

We'll mainly cover these by example!