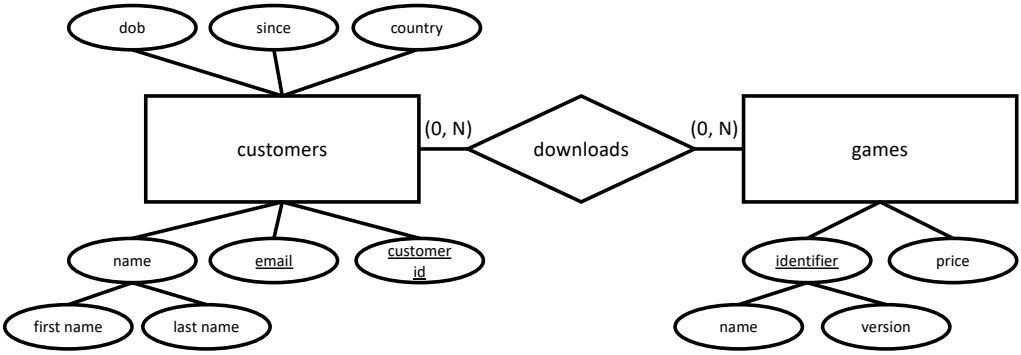
Creating and Populating Tables with Constraints

Stéphane Bressan



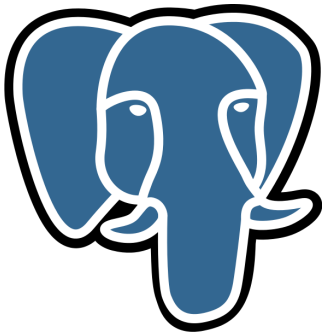


We have been commissioned to develop an application for managing the data of an online app store. We want to store several items of information about our customers such as their first name, last name, date of birth, e-mail, date and country of registration to our online sales service and the customer identifier that they have chosen. We also want to manage the list of our products, games, their name, their version, and their price. The price is fixed for each version of each game. Finally, our customers buy and download games. So we must remember which version of which game each customer has downloaded. It is not essential to keep the download date for this application.


dob
since
country
customers
(0, N)
downloads
(0, N)
games
name
email
customer id
first name
last name
identifier
price
name
version

In theory, the relational model proposes to organise data in relations. relations have a name. Relations have attributes. Attributes have a name. Mathematically, relations are subsets of the Cartesian product of the domains of their attributes. Domains extensions of types.

In practice, the relational database management systems organise data in tables. Tables have a name. Tables are multi-sets (not lists) of rows or records. Rows or records have fields corresponding to the columns of the table. Columns or fields have a name. Columns or fields also have an implicit position indicated by the order in the corresponding creation statement. Columns or fields have a domain which is a type to the extension of which is added the possibility of a null value.



We use PostgreSQL with psql or pgAdmin 4 (psql or Query Tool).

```sql
CREATE TABLE downloads(
    first_name VARCHAR(64),
    last_name VARCHAR(64),
    email VARCHAR(64),
    dob DATE,
    since DATE,
    customerid VARCHAR(16),
    country VARCHAR(16),
    name VARCHAR(32),
    version CHAR(3),
    price NUMERIC);
```

varchars are basically strings
VARCHAR = chars of variable length
VARCHAR(64) = chars of length up to 64

The choice of the number of columns and of their domains implicitly imposes structural constraints. For instance, if we use only one table, there can be no customer without a game and no game without a customer, unless we use null values. We can use the structural constraints as integrity constraints to control and maintain the integrity of data because data that does not fit in the table cannot be stored and only data that fit can be stored.

```sql
CREATE TABLE customers (
  first_name VARCHAR(64),
  last_name VARCHAR(64),
  email VARCHAR(64),
  dob DATE,
  since DATE,
  customerid VARCHAR(16),
  country VARCHAR(16));
```

```sql
CREATE TABLE games (
  name VARCHAR(32),
  version CHAR(3),
  price NUMERIC);
```

```sql
CREATE TABLE downloads(
  customerid VARCHAR(16),
  name VARCHAR(32),
  version CHAR(3));
```

We opt for a schema comprising the three tables above with the indicated columns and their respective domains (`VARCHAR(64)`, `DATE`, `CHAR(3)`, `NUMERIC`, etc.[a].)

[a] See `www.postgresql.org/docs/current/datatype.html`

Integrity Constraints

A consistent state of the database is a state which complies with the with the business rules as defined by the structural constraints and the integrity constraints in the schema.

SQL allows to express business rules, such as "*students who have not passed cs1010 cannot take cs2020*" as integrity constraints.

If an integrity constraint is violated by an operation or a transaction, the operation or the transaction is aborted and rolled back and its changes are undone, otherwise, it is committed and its changes are effective for all users.

In practice, there are different ways to specify the scope of transactions. One of them is to use blocks with keywords such as `BEGIN`, `END` or `COMMIT`, `ABORT` and `ROLLBACK`.

In most systems, unfortunately, integrity constraints are immediate, i.e. they are checked after each insertion, update and deletion. Whenever possible, it is preferable to set all integrity constraints to be differed, i.e. they are checked after the end of transactions. Unfortunately again, some systems, like PostgreSQL do only allow certain constraints to be deferred[a].

[a] See `www.postgresql.org/docs/current/sql-set-constraints.html`.

There are five main kinds of integrity constraints in SQL[a]:

- `NOT NULL`,
- `PRIMARY KEY`,
- `UNIQUE`,
- `FOREIGN KEY` and
- `CHECK`.

[a] PostgreSQL also implements `EXCLUDE` constraints.

Before recreating the tables with different definitions, let us delete the existing tables. The same applies for the subsequent examples with the same or other tables.

```
1 DELETE FROM customers;
```

Every SQL command ends with a ;

DELETE deletes content of the table but not its definition. In the example above, the content of the customers table is deleted. The table still exists and is empty. Instead, one should use DROP.

```
1 DROP TABLE customers;
```

DROP deletes the content of the table and its definition. the table does not exists any more.

We recommend `www.w3schools.com/sql` for details of the syntax and behaviour of standard SQL constructs. For proprietary syntax and behaviour, you may want to consult the PostgreSQL documentation at `www.postgresql.org/docs`.

A primary key is a set of columns that uniquely identifies a record in the table. Each table has at most one primary key.

The primary key can be one column or a combination of columns. An instance of the table cannot have two records with the same value or combination of values in the primary key columns and no primary key column cannot contain a null value[a].

[a] This is not implemented in all database management systems.

```
1 CREATE TABLE customers (
2 first_name VARCHAR(64),
3 last_name VARCHAR(64),
4 email VARCHAR(64),
5 dob DATE,
6 since DATE,
7 customerid VARCHAR(16) PRIMARY KEY,
8 country VARCHAR(16));
```

We can declare a primary key as a column constraint with the keyword `PRIMARY KEY`.

In the example above, the primary key of a customer is its customer identification number.

It is common practice to underline the primary key when informally writing the schema. $customers(\underline{first\_name}, last\_name, email, dob, since, customerid, country)$

We insert one customer.

```
1 INSERT INTO customers VALUES(
2   'Carole',
3   'Yoga',
4   'cyoga@glarge.org',
5   '1989-08-01',
6   '2016-09-15',
7   'Carole89',
8   'France');
```

We insert another customer with the same customer identification number.

```
1 INSERT INTO customers VALUES(
2   'Carole',
3   'Yoga',
4   'cyoga@glarge.org',
5   '1989-08-01',
6   '2016-09-15',
7   'Carole89',
8   'France');
```

```
1 ERROR:  duplicate key value violates unique constraint "customers_pkey"
2 DETAIL:  Key (customerid)=(Carole89) already exists.
3 SQL state: 23505
```

The primary key constraint prevents any operation or transaction that violates the constraint, for instance, inserting another customer with the same customer identification number. The operation or the transaction is aborted and rolled back.

```
1 CREATE TABLE games(
2 name VARCHAR(32),
3 version CHAR(3),
4 price NUMERIC,
5 PRIMARY KEY (name, version));
```

The primary key can be composite: it is the combination of several columns. The composite primary key constraint is declared as a table constraint with the keyword `PRIMARY KEY` after the row declarations.

In the example above, the primary key of a game is the combination of its name and version.

It is common practice to underline primary key columns ('prime columns' or 'prime attributes') when informally writing the schema. $games(\underline{name}, \underline{version}, price)$

There cannot be two games with the same name and version.

```
1 INSERT INTO games VALUES ('Aerified', '1.0', 5), ('Aerified', '1.0', 6);
```

```
1 ERROR:  duplicate key value violates unique constraint "games_pkey"
2 DETAIL:  Key (name, version)=(Aerified, 1.0) already exists.
3 SQL state: 23505
```

However, there can be several games with the same name and several games with the same version.

```
1 INSERT INTO games VALUES ('Aerified', '1.0', 5), ('Aerified', '2.0', 6), ('Verified', '1.0', 7);
```

```
1 CREATE TABLE games(
2 name VARCHAR(32),
3 version CHAR(3),
4 price NUMERIC NOT NULL);
```

A not null constraint guarantees that no value of the corresponding column in any record of the table can be set to null. A not null constraint is always declared as a row constraint. When it is explicit, it is declared with the keyword `NOT NULL`. In the example above, the price of a game can never be null.

```
1 INSERT INTO games VALUES ('Aerified', '3.0', null);
```

or

no values given = null

```
1 INSERT INTO games (name, version) VALUES ('Aerified', '3.0');
```

```
1 ERROR:  null value in column "price" of relation "games" violates not-null constraint
2 DETAIL:  Failing row contains (Aerified, 1.0, null).
3 SQL state: 23502
```

Both operations above attempt to insert a null value for the price. An error is raised. The operation or the transaction violating the constraints is aborted and rolled back.

Alternatively, we could set a default value at table creation time with the following declaration when creating the table games.

```
1 price NUMERIC DEFAULT 1.00,
```

A unique constraint on a column or a combination of columns guarantees the table cannot contain two records with the same value in the corresponding column or combination of columns.

```
1 CREATE TABLE customers (
2   first_name VARCHAR(64),
3   last_name VARCHAR(64),
4   email VARCHAR(64) UNIQUE,
5   dob DATE,
6   since DATE,
7   customerid VARCHAR(16),
8   country VARCHAR(16),
9   UNIQUE (first_name, last_name));   combination of first_name & last_name unique
```

This is declared as a row or a table constraint with the `UNIQUE` keyword. In the example above, the email of a customer must be unique and the combination of her first name and last name must be unique. This is the same as a primary key constraint except that it does allow null values.

According to the SQL standard and in most systems, primary key columns must not contain null values. This is the case of PostgreSQL, but this is not the case of SQLite and IBM DB2, for which `NOT NULL` constraint must be added explicitly if one wants to prevent values of the prime columns from being null.

SQLite Apology: "According to the SQL standard, PRIMARY KEY should always imply NOT NULL. Unfortunately, due to a bug in some early versions, this is not the case in SQLite. Unless the column is an INTEGER PRIMARY KEY or the table is a WITHOUT ROWID table or the column is declared NOT NULL, SQLite allows NULL values in a PRIMARY KEY column. SQLite could be fixed to conform to the standard, but doing so might break legacy applications. Hence, it has been decided to merely document the fact that SQLite allowing NULLs in most PRIMARY KEY columns."
`http://www.sqlite.org/lang_createtable.html`

In many systems that consider that primary key columns should not contain null values, declaring a `NOT NULL` constraint on a prime columns raises an error. This is particularly annoying as it hinders the portability of the otherwise standard SQL DDL code.This is not the case of PostgreSQL, which allows it, precisely for portability reasons.

According to the QL standard, a `PRIMARY KEY` constraint is equivalent, from the point of view of integrity, to a combination of `UNIQUE` and `NOT NULL` constraints, but its declaration has side effects not fully discussed here. It creates and maintains indexes for fast access. In PostgreSQL, the unique indexes interfere with the deferrability of constraints. It can be referenced by foreign keys.

A foreign key constraint enforces referential integrity. The values in the columns for which the constraint is declared must exists in the corresponding columns of the referenced table.

The referenced columns are usually required to be the primary key of the referenced table. Some systems relax this requirement. PostgreSQL allows to reference primary keys and unique columns and combinations of columns. It may be good practice to nevertheless enforce the requirement strictly for the sake of portability.

```
1  CREATE TABLE customers (
2   first_name VARCHAR(64),
3   last_name VARCHAR(64),
4   email VARCHAR(64),
5   dob DATE,
6   since DATE,
7   customerid VARCHAR(16) PRIMARY KEY,
8   country VARCHAR(16));
9
10 CREATE TABLE games(
11  name VARCHAR(32),
12  version CHAR(3),
13  price NUMERIC,
14  PRIMARY KEY (name, version));
```

```
1 CREATE TABLE downloads(
2 customerid VARCHAR(16) REFERENCES customers (customerid),
3 name VARCHAR(32),
4 version CHAR(3),
5 FOREIGN KEY (name, version) REFERENCES games(name, version));
```

A foreign key is declared using the keyword `REFERENCES` as a row constraint and the keywords `FOREIGN KEY` and `REFERENCES` as a table constraint. In the example above, the customer identification number and the combination of name and version of the game she downloaded as recorded in the `downloads` table must correspond to an existing customer and an existing game, they must be the corresponding primary keys, `customerid` in the `customers` table and the combination of `name` and `version` in the `games` table, respectively.

Note that, surprisingly, a null value does not violate referential integrity in SQL.

```
1 CREATE TABLE downloads(
2 customerid VARCHAR(16) REFERENCES customers (customerid),
3 name VARCHAR(32),
4 version CHAR(3),
5 FOREIGN KEY (name, version REFERENCES games(name, version));
```

When the referenced primary key is composite, the `FOREIGN KEY` constraint is declared as a table constraint using the additional keyword `FOREIGN KEY`. All foreign key constraints can be declared as table constraints.

```
1 INSERT INTO downloads VALUES ('Adam1983', 'Biodex', '1.0');
```

```
1 ERROR: insert or update on table "downloads" violates foreign key constraint "downloads_customerid_fkey"
2 DETAIL:  Key (customerid)=(Adam1983) is not present in table "customers".
3 SQL state: 23503
```

The column `customerid` in the table `downloads` references the primary key `customerid` of the table `customers` The column `customerid` in the table `downloads` can only take values that appears in the column `customerid` of the table `customers`. The customer `Adam1983` does not exist. The same thing happens for the game `Biodex 1.0`.

```
1 INSERT INTO customers VALUES ('Deborah', 'Ruiz', 'druiz0@drupal.org', '1984-08-01', '2016-10-17', '
      Deborah84', 'Singapore');
2 INSERT INTO games VALUES ('Aerified', '1.0', 12);
3 INSERT INTO downloads VALUES ('Deborah84', 'Aerified', '1.0');
```

```
1 DELETE FROM customers WHERE country='Singapore';
```

```
1 ERROR: update or delete on table "customers" violates foreign key constraint "downloads_customerid_fkey"
      on table "downloads"
2 DETAIL: Key (customerid)=(Deborah84) is still referenced from table "downloads".
3 SQL state: 23503
```

We cannot delete the Singapore customers because some of them have downloaded some games.

```
1 CREATE TABLE games (
2   name VARCHAR(32),
3   version CHAR(3),
4   price NUMERIC NOT NULL CHECK (price > 0));
```

A check constraint enforces any other condition that can be expressed in SQL. A check constraint is declared as a row or a table constraint with the `CHECK` keyword followed by the SQL condition in parenthesis. If the condition involves more than one row, it has to be a table constraint. In the example above, the price of game can not be zero or negative.

The SQL standard caters for very expressive CHECK constraints. It even allows that CHECK constraints can be implemented as assertions outside tables (using the construct `CREATE ASSERTION`) so that they can involve several tables and use aggregate functions, nested queries etc. In practice, the vendors only offer check constraints that are limited to very simple row and table checks. This is a great pity since the technology exists, is very well understood and its non availability jeopardises the integrity of enterprise data. The vendors usually advocate the use of triggers and stored functions, which is notoriously difficult and, therefore, error-prone. It largely defeats the objective of integrity management.

```
1 INSERT INTO games VALUES ('Aerified', '1.0', 12);
2 INSERT INTO games VALUES ('Aerified', '1.1', 3.99);
```

```
1 UPDATE games SET price = price - 10;
```

```
1 ERROR: new row for relation "games" violates check constraint "games_price_check"
2 DETAIL:  Failing row contains (Aerified, 1.1, -6.01).
3 SQL state: 23514
```

Discounting all the prices by 10 dollars creates negative prices. This operation is aborted and rolled back.

fail check

Consider the following schema.

```
1  CREATE TABLE IF NOT EXISTS customers (
2   first_name VARCHAR(64) NOT NULL,
3   last_name VARCHAR(64) NOT NULL,
4   email VARCHAR(64) UNIQUE NOT NULL,
5   dob DATE NOT NULL,
6   since DATE NOT NULL,
7   customerid VARCHAR(16) PRIMARY KEY,
8   country VARCHAR(16) NOT NULL);
9
10 CREATE TABLE downloads(
11  customerid VARCHAR(16) REFERENCES customers(customerid)
12  name VARCHAR(32),
13  version CHAR(3),
14  PRIMARY KEY (customerid, name, version),
```

downloads

name	version	customerid
Skype	1.0	tom1999
Comfort	1.1	john88
Skype	2.0	tom1999
...		

customers

customerid	email	...
tom1999	tlee@gmail.com	...
john88	al@hotmail.com	...
walnuts	dcs@nus.edu.sg	...
...		

What happens if the customer identification number `john88` is changed to `john1988` in the table `downloads`? error

What happens if the customer identification number `tom1999` is changed to `thom1999` in the table `customers`?

What happens if the record for the customer with identification number `tom1999` is deleted from the table `customer`?

```
1  CREATE TABLE downloads(
2   customerid VARCHAR(16) REFERENCES customers(customerid)
3      ON UPDATE CASCADE
4      ON DELETE CASCADE,
5   name VARCHAR(32),
6   version CHAR(3),
7   PRIMARY KEY (customerid, name, version),
8   FOREIGN KEY (name, version) REFERENCES games(name, version)
9      ON UPDATE CASCADE
10     ON DELETE CASCADE);
```

if update/delete from customers, do the same on downloads

The annotations `ON UPDATE/DELETE` with the option `CASCADE` propagate the update or deletion. They can also be used with the self-describing options: `NO ACTION`, `SET DEFAULT` and `SET NULL`.

Update and deletion cascades are powerful mechanisms that can result in chain reactions when they are chains of foreign key dependencies.

There is an even more powerful but difficult to program generalisation of such propagation mechanisms in the form of triggers using stored functions. Triggers with stored procedures are very expressive but leave the responsibility of the control to the programmer.

The exact syntax and behaviour of these constructs may vary from one DBMS to another and from one version to the next.

This is the complete schema for our example

```
1  CREATE TABLE IF NOT EXISTS customers (
2   first_name VARCHAR(64) NOT NULL,
3   last_name VARCHAR(64) NOT NULL,
4   email VARCHAR(64) UNIQUE NOT NULL,
5   dob DATE NOT NULL,
6   since DATE NOT NULL,
7   customerid VARCHAR(16) PRIMARY KEY,
8   country VARCHAR(16) NOT NULL);
9
10 CREATE TABLE IF NOT EXISTS games(
11  name VARCHAR(32),
12  version CHAR(3),
13  price NUMERIC NOT NULL,
14  PRIMARY KEY (name, version));
15
16  CREATE TABLE downloads(
17  customerid VARCHAR(16) REFERENCES customers(customerid)
18     ON UPDATE CASCADE ON DELETE CASCADE
19     DEFERRABLE INITIALLY DEFERRED,
20  name VARCHAR(32),
21  version CHAR(3),
22  PRIMARY KEY (customerid, name, version),
23  FOREIGN KEY (name, version) REFERENCES games(name, version)
24     ON UPDATE CASCADE ON DELETE CASCADE
25     DEFERRABLE INITIALLY DEFERRED); check constraints at the end
```

It is generally a good idea to constraint all columns not to be null unless there is a good design or tuning reason for not doing so.

Think carefully about which foreign keys should be subject to cascade.

It is generally a good idea to defer all the constraints that can be deferred. A deferred constraint is checked at the end of a transaction and not immediately after each operation.

Given the complete schema for the three tables, try and find out the scenarii in which an operation (insertion, deletion, update) on a table or a transaction containing a set of operation on one or more tables violate which constraint on which table.

We download SQL file with the `CREATE TABLE` and `INSERT` statements.

```
1  CREATE TABLE games (
2    name VARCHAR(50),
3    version VARCHAR(50),
4    price DECIMAL(2,2)
5  );
6  INSERT INTO games (name, version, price) VALUES ('Voyatouch', '6.6', 6.36);
7  INSERT INTO games (name, version, price) VALUES ('Voltsillam', '7.4', 3.76);
8  INSERT INTO games (name, version, price) VALUES ('Stim', '8.2', 3.04);
9  INSERT INTO games (name, version, price) VALUES ('Y-Solowarm', '8.9', 9.99);
10 INSERT INTO games (name, version, price) VALUES ('Quo Lux', '9.9', 2.64);
11 INSERT INTO games (name, version, price) VALUES ('Biodex', '5.3', 7.3);
12 INSERT INTO games (name, version, price) VALUES ('Biodex', '8.7', 4.13);
```

You can now delete the tables you have created. `DROP` deletes both the content of the table and the table definition.

```
cs2102=\# DROP TABLE downloads
cs2102=\# DROP TABLE customers
cs2102=\# DROP TABLE games
```

From Canvas download the files[a] `AppStoreSchema.sql` `AppStoreCustomers.sql`, `AppStoreGames.sql` and `AppStreDownloads.sql`. Navigate psql to the directory where you downloaded them. These four SQL files (you should read them) are creating and populating the database or our application. Execute them. `\i` executes the SQL files in psql. The file `AppStoreClean.sql` cleans up the database if you need to start again.

[a] We use variants of the same data and that the results on your computer may differ slightly from the results given in the slides for illustration purposes.

```
cs2102=\# \cd 'C:/Users/.../Code'
cs2102=\# \i AppStoreSchema.sql
CREATE TABLE
CREATE TABLE
CREATE TABLE
cs2102=\# \i AppStoreCustomers.sql
INSERT 0 1
INSERT 0 1
...
cs2102=\# \i AppStoreGames.sql
INSERT 0 1
INSERT 0 1
...
cs2102=\# \i AppStoreDownloads.sql
INSERT 0 1
INSERT 0 1
...
```

Alternatively you can use pgAdmin Query Tool.

The following query prints the `customers` table.

```
1 SELECT *
2 FROM customers;
```

first_name	last_name	email	dob	since	customerid	country
Deborah	Ruiz	druiz0@drupal.org	1984-08-01	2016-10-17	Deborah84	Singapore
Rebecca	Garza	rgarza2@cornell.edu	1984-06-11"	2016-09-26	RebeccaG84	Malaysia"
Walter	Leong	wleong3@shop-pro.jp	1983-06-26	"2016-06-12"	Walter83	Singapore
...						

The following query prints the games table.

```
1 SELECT *
2 FROM games;
```

name	version	price
Aerified	1.0	12
Aerified	1.1	3.99
Aerified	1.2	1.99
...		

The following query prints the `downloads` table.

```
1 SELECT *
2 FROM downloads;
```

customerid	name	version
Adam1983	Biodex	1.0
Adam1983	Domainer	2.1
Adam1983	Subin	1.1
...		

You can give a name to a query. This is called a view. Once created, a view can be queried like any other table.

```
1 CREATE VIEW singapore_customers1 AS
2 SELECT c.first_name, c.last_name, c.email, c.dob, c.since, c.customerid
3 FROM customers c
4 WHERE country='Singapore';
```

```
1 SELECT *
2 FROM singapore_customers1;
```

Creating a view is generally a better option than creating and populating a table, temporary or not.

```
1  CREATE TABLE singapore_customers2 (
2    first_name VARCHAR(64) NOT NULL,
3    last_name VARCHAR(64) NOT NULL,
4    email VARCHAR(64) UNIQUE NOT NULL,
5    dob DATE NOT NULL,
6    since DATE NOT NULL,
7    customerid VARCHAR(16) PRIMARY KEY REFERENCES customers(customerid));
8
9  INSERT INTO singapore_customers2
10 SELECT c.first_name, c.last_name, c.email, c.dob, c.since, c.customerid
11 FROM customers c
12 WHERE country='Singapore';
```

There are further issues concerning views that we are not discussing here: view updates and materalised views, for instance.

```
1 DROP VIEW singapore_customers1;
2 DROP TABLE singapore_customers2;
```