SQL

Data Definition Language

André Restivo

Index

Introduction

SQL

- Structured **Q**uery **L**anguage.
- A special purpose language to manage data stored in a **relational** database.
- Based on **relational algebra**.
- Pronounced *Sequel*

History

- Early 70's SEQUEL Developed at IBM
- 1986 SQL-86 and SQL-87 Ratified by ANSI and ISO.
- 1989 SQL-89
- 1992 SQL-92 Also know as SQL2.
- 1999 SQL:1999 Also known as SQL3 Includes regurlar expressions, recursive queries, triggers, non-scalar data types and some object-oriented expressions.
- 2003 SQL:2003 XML support and auto-generated values.
- 2006 SQL:2006 XQuery support.
- 2008 SQL:2008.
- 2011 SQL:2011.

Standard

- Although SQL is an ANSI/ISO standard, every database system implements it in a slightly different way.
- These slides will try to adhere to the standard as much as possible.
- Sometimes we'll deviate and talk specifically about **PostgreSQL**.

Table Basics

Creating Tables

The basic structure of a table creation statement in SQL:

```
CREATE TABLE <table_name> (
  <column_name> <data_type>,
  <column_name> <data_type>,
  ...
  <column_name> <data_type>
);
```

? Values between <> are to be replaced.

Deleting Tables

To delete a table we do:

```
DROP TABLE <table_name>;
```

If there are foreign keys referencing the table we must use the *cascade* keyword:

```
DROP TABLE <table_name> CASCADE;
```

Data Types

PostgreSQL

Numeric

- The types **SMALLINT**, **INTEGER**, and **BIGINT** store whole numbers.
- **NUMERIC(precision, scale)** stores numbers with a very large number of digits. The *scale* of a numeric data type is the count of decimal digits in the fractional part, to the right of the decimal point. The *precision* of a numeric is the total count of significant digits in the whole number.
- The data types **REAL** and **DOUBLE** precision are inexact, variable-precision numeric types.

Numeric Precision

Precision for numeric types:

SMALLINT	2 bytes	-32768 to +32767
INTEGER	4 bytes	-2147483648 to +2147483647
BIGINT	8 bytes	-9223372036854775808 to +9223372036854775807
NUMERIC	user-specified	numeric(3,2) -99.9 to 99.9
REAL	4 bytes	inexact 6 decimal digits precision
DOUBLE	8 bytes	inexact 15 decimal digits precision

Date / Time

- The type **DATE** stores date values in the ISO 8601 format.

  ? Example: 1980-12-25.

- The type **TIME** stores time values in the ISO 8601 format.

  ? Examples: 04:05 or 04:05:06.789

- The type **TIMESTAMP** store time and date values in the ISO 8601 format.

  ? Example: 1980-12-25 04:05:06.789.

- Date/time types can also store timezone information.

Text

- The type **CHARACTER VARYING(n)**, or **VARCHAR(n)**, stores variable-length text with a user defined limit.
- The type **CHARACTER(n)**, or **CHAR(n)**, stores fixed-length, blank-padded text.
- The type **TEXT** stores variable unlimited length text.

? VARCHAR, without a limit, is the same as TEXT

? CHAR, without a limit, is the same as CHAR(1)

Boolean

- The **BOOLEAN** type can only have two possible values: **true** or **false**.
- Possible values representing true: TRUE, 't', 'true', 'y', 'yes', 'on' and '1'
- Possible values representing false: FALSE, 'f', 'false', 'n', 'no', 'off' and '0'

Serial

- In PostgreSQL there is a *pseudo-type* that can be used to define **auto-generated** identifiers or **auto-counters**.
- To define a column as an auto-counter we use the type **SERIAL**.

Example

```
CREATE TABLE employee (
  id SERIAL,
  name VARCHAR(128),
  address VARCHAR(256),
  birthdate DATE,
  salary NUMERIC(6,2),
  taxes NUMERIC(6,2),
  card_number INTEGER,
  active BOOLEAN
);
```

Defaults

Default Values

For each column we can define its default value:

```
CREATE TABLE <table_name> (
  <column_name> <data_type> DEFAULT <default_value>,
  <column_name> <data_type>,
  ...
  <column_name> <data_type>
);
```

? The default default value is **NULL**

Example

```
CREATE TABLE employee (
  id SERIAL,
  name VARCHAR(128),
  address VARCHAR(256),
  birthdate DATE,
  salary NUMERIC(6,2),
  taxes NUMERIC(6,2),
  card_number INTEGER DEFAULT 0,
  active BOOLEAN DEFAULT TRUE
);
```

? Card number default value is 0.

? Employee is active by default.

Constraints

Constraint Types

Several types of constraints can be defined using SQL:

- Check
- Not Null
- Unique Keys
- Primary Keys
- Foreign Key

Constraints can be **column** based or **table** based.

- **Column** constraints appear **in front** of the column they are referring to.
- **Table** constraints appear as a **separate** clause.

Check

Check Constraint

Check contraints allows to define a constraint on the column values using an expression that the values must follow:

```
CREATE TABLE <table_name> (
  <column_name> <data_type> CHECK <check_expression>,
  <column_name> <data_type>,
  ...
  <column_name> <data_type>
);
```

Example

```
CREATE TABLE employee (
  id SERIAL,
  name VARCHAR(128),
  address VARCHAR(256),
  birthdate DATE,
  salary NUMERIC(6,2) CHECK (salary > 500),
  taxes NUMERIC(6,2),
  card_number INTEGER DEFAULT 0 CHECK (card_number >= 0),
  active BOOLEAN DEFAULT TRUE
);
```

? Salary must be larger than 500.

? Card number must be larger or equal to 0.

Constraint Naming

Giving names to constraints allows us to better identify them when errors occur or when we want to refer to them.

```
CREATE TABLE <table_name> (
  <column_name> <data_type> CONSTRAINT <constraint_name> CHECK <check_expression>,
  <column_name> <data_type>,
  ...
  <column_name> <data_type>
);
```

? Naming constraints is optional but is a good practice.

Example

```
CREATE TABLE employee (
  id SERIAL,
  name VARCHAR(128),
  address VARCHAR(256),
  birthdate DATE,
  salary NUMERIC(6,2)
    CONSTRAINT minimum_wage CHECK (salary > 500),
  taxes NUMERIC(6,2),
  card_number INTEGER DEFAULT 0 CHECK (card_number >= 0),
  active BOOLEAN DEFAULT TRUE
);
```

Multiple Column Check

If the check constraint refers to more than one column, we must use a table based constraint:

```
CREATE TABLE employee (
  id SERIAL,
  name VARCHAR(128),
  address VARCHAR(256),
  birthdate DATE,
  salary NUMERIC(6,2)
    CONSTRAINT minimum_wage CHECK (salary > 500),
  taxes NUMERIC(6,2),
  card_number INTEGER DEFAULT 0 CHECK (card_number >= 0),
  active BOOLEAN DEFAULT TRUE,
  CONSTRAINT taxes_lower_salary CHECK (taxes < salary)
);
```

? Taxes have to be lower than salary.

Not Null

Not Null Constraint

We can define that a certain column does not allow NULL values:

```
CREATE TABLE <table_name> (
  <column_name> <data_type> NOT NULL,
  <column_name> <data_type>,
  ...
  <column_name> <data_type>
);
```

Example

```
CREATE TABLE employee (
  id SERIAL,
  name VARCHAR(128) NOT NULL,
  address VARCHAR(256),
  birthdate DATE,
  salary NUMERIC(6,2)
    CONSTRAINT minimum_wage CHECK (salary > 500),
  taxes NUMERIC(6,2),
  card_number INTEGER DEFAULT 0 CHECK (card_number >= 0),
  active BOOLEAN DEFAULT TRUE,
  CONSTRAINT taxes_lower_salary CHECK (taxes < salary)
);
```

? Name cannot be null.

Primary Keys

Primary Key Constraints

We can define one, and **only one**, primary key for our table:

```
CREATE TABLE <table_name> (
  <column_name> <data_type> PRIMARY KEY,
  <column_name> <data_type>,
  ...
  <column_name> <data_type>
);
```

Example

```
CREATE TABLE employee (
  id SERIAL PRIMARY KEY,
  name VARCHAR(128) NOT NULL,
  address VARCHAR(256),
  birthdate DATE,
  salary NUMERIC(6,2)
    CONSTRAINT minimum_wage CHECK (salary > 500),
  taxes NUMERIC(6,2),
  card_number INTEGER DEFAULT 0 CHECK (card_number >= 0),
  active BOOLEAN DEFAULT TRUE,
  CONSTRAINT taxes_lower_salary CHECK (taxes < salary)
);
```

? Id cannot be null and cannot have repeated values.

Multiple Column Primary Key

If a primary key is composed by **more than one** column, we must use a **table based** constraint:

```
CREATE TABLE <table_name> (
  <column_name> <data_type>,
  <column_name> <data_type>,
  ...
  <column_name> <data_type>,
  PRIMARY KEY (<column_name>, <column_name>)
);
```

Example

```
CREATE TABLE telephone (
  employee INTEGER,
  phone VARCHAR,
  PRIMARY KEY (employee, phone)
);
```

? An employee cannot have the same phone number twice.

Unique Keys

Unique Key Contraints

Unique keys are identical to primary keys but:

- they **allow NULL** values;
- and there can be **multiple unique keys** in one table.

They are created using the same type of syntax:

Example

```
CREATE TABLE employee (
  id SERIAL PRIMARY KEY,
  name VARCHAR(128) NOT NULL,
  address VARCHAR(256),
  birthdate DATE,
  salary NUMERIC(6,2)
    CONSTRAINT minimum_wage CHECK (salary > 500),
  taxes NUMERIC(6,2),
  card_number INTEGER
    UNIQUE
    DEFAULT 0 CHECK (card_number >= 0),
  active BOOLEAN DEFAULT TRUE,
  CONSTRAINT taxes_lower_salary CHECK (taxes < salary)
);
```

? Card keys cannot have repeated values.

Multiple Column Unique Keys

If a unique key is composed by **more than one** column, we must use a **table based** constraint:

```
CREATE TABLE <table_name> (
  <column_name> <data_type>,
  <column_name> <data_type>,
  ...
  <column_name> <data_type>,
  UNIQUE (<column_name>, <column_name>)
);
```

Foreign Keys

Foreign Key Constraints

- We can also declare foreign keys.
- A foreign key must always **reference a key** (primary or unique) from another (or the same) table.
- Databases don't allow columns with a foreign key containing values that do not exist in the referenced column.

```
CREATE TABLE <table_A> (
  <column_A> <data_type> PRIMARY KEY,
  <column_B> <data_type>,
  ...
  <column_C> <data_type>
);

CREATE TABLE <table_B> (
  <column_X> <data_type> PRIMARY KEY,
  <column_Y> <data_type>,
  ...
  <column_Z> <data_type> REFERENCES <table_A>(<column_A>)
);
```

Example

```
CREATE TABLE employee (
  id SERIAL PRIMARY KEY,
  name VARCHAR(128) NOT NULL,
  address VARCHAR(256),
  birthdate DATE,
  salary NUMERIC(6,2)
    CONSTRAINT minimum_wage CHECK (salary > 500),
  taxes NUMERIC(6,2),
  card_number INTEGER
    UNIQUE
    DEFAULT 0 CHECK (card_number >= 0),
  active BOOLEAN DEFAULT TRUE,
  department_id integer REFERENCES department(id),
  CONSTRAINT taxes_lower_salary CHECK (taxes < salary)
);
```

? The id of the department references the id column in the department table.

? Employees cannot have a department number that doesn't exist in the department table.

Foreign Key to Primary Key

If the referenced column is the primary key of the other table, we can omit the name of the column:

```
CREATE TABLE employee (
  id SERIAL PRIMARY KEY,
  name VARCHAR(128) NOT NULL,
  address VARCHAR(256),
  birthdate DATE,
  salary NUMERIC(6,2)
    CONSTRAINT minimum_wage CHECK (salary > 500),
  taxes NUMERIC(6,2),
  card_number INTEGER
    UNIQUE
    DEFAULT 0 CHECK (card_number >= 0),
  active BOOLEAN DEFAULT TRUE,
  department_id integer REFERENCES department,
  CONSTRAINT taxes_lower_salary CHECK (taxes < salary)
);
```

? The id of the department references the primary key in the department table.

Multiple Column Foreign Key

If the referenced table has a key with **multiple columns**, we must use a **table based** constraint to define our foreign key:

```
CREATE TABLE telephone (
  employee INTEGER,
  phone VARCHAR,
  PRIMARY KEY (employee, phone)
);

CREATE TABLE call (
  id INTEGER PRIMARY KEY,
  employee INTEGER,
  phone INTEGER,
  when DATE,
  caller INTEGER,
  FOREIGN KEY (employee, phone) REFERENCES telephone (employee, phone)
);
```

Example

We can also ommit the referenced columns if they are the primary keys:

```
CREATE TABLE telephone (
  employee INTEGER,
  phone VARCHAR,
  PRIMARY KEY (employee, phone)
);

CREATE TABLE call (
  id INTEGER PRIMARY KEY,
  employee INTEGER,
  phone INTEGER,
  when DATE,
  caller INTEGER,
  FOREIGN KEY (employee, phone) REFERENCES telephone
);
```

Deleting Referenced Values

- Declaring a foreign key means that values in one table must also appear in the referenced column.
- When a line having values referencing it is deleted, three different things can occur:
  - An **error** is thrown.
  - The referencing values becomes **NULL**.
  - All referencing **lines are deleted** (this might cause a cascade effect).

Updating Referenced Values

- The same problem happens when we update a line that is referenced by another column.
- When a line having values referencing it is updated, three different things can occur:
  - An **error** is thrown.
  - The referencing values becomes **NULL**.
  - All referencing **values are updated** to the new value (this might cause a cascade effect).

On Delete and On Update

To define the desired behavior, we should use the **ON DELETE** and **ON UPDATE** clauses with one of three possible values:

- **RESTRICT** (throws an error)
- **SET NULL** (values become null)
- **CASCADE** (lines are deletes or values updated)

? RESTRICT is the default value.

On Delete and On Update

```
CREATE TABLE <table_A> (
  <column_A> <data_type> PRIMARY KEY,
  <column_B> <data_type>,
  ...
  <column_C> <data_type>
);

CREATE TABLE <table_B> (
  <column_X> <data_type> PRIMARY KEY,
  <column_Y> <data_type>,
  ...
  <column_Z> <data_type> REFERENCES <table_A>(<column_A>)
    ON DELETE SET NULL ON UPDATE CASCADE
);
```

Example

```
CREATE TABLE employee (
  id SERIAL PRIMARY KEY,
  name VARCHAR(128) NOT NULL,
  address VARCHAR(256),
  birthdate DATE,
  salary NUMERIC(6,2)
    CONSTRAINT minimum_wage CHECK (salary > 500),
  taxes NUMERIC(6,2),
  card_number INTEGER
    UNIQUE
    DEFAULT 0 CHECK (card_number >= 0),
  active BOOLEAN DEFAULT TRUE,
  department_id integer REFERENCES department
    ON DELETE SET NULL ON UPDATE CASCADE,
  CONSTRAINT taxes_lower_salary CHECK (taxes < salary)
);
```

? If a department with id 1 is deleted, all employess with department id equal to 1 will start having a null department number.

? If a department with id 1 is updated to id 2, all employess with department id equal to 1 will start having a department number equal to 2.

Sequences

Sequences

- A sequence is a special kind of database object designed for generating unique numeric identifiers.
- They ensure that a different value is generated for every client.

Sequences

```
CREATE SEQUENCE <name>;
```

Get the next and the current value of the sequence.

```
SELECT nextval('<name>');
SELECT currval('<name>');
```

Getting the current value only works if called after calling nextval and in the same transaction.

The SERIAL type

The data type serial is not a true type, but merely a notational convenience for setting up unique identifier columns.

```
CREATE TABLE <tablename> (
    <colname> SERIAL
);
```

Is equivalent to:

```
CREATE SEQUENCE <tablename_colname_seq>;
CREATE TABLE <tablename> (
    <colname> integer DEFAULT nextval('<tablename_colname_seq>') NOT NULL
);
```

Example

```
CREATE TABLE category (
  id SERIAL PRIMARY KEY,
  name VARCHAR
)

CREATE TABLE product (
  id SERIAL PRIMARY KEY,
  name VARCHAR,
  cat_id INTEGER REFERENCES category
);

INSERT INTO category VALUES(DEFAULT, 'Fruits');
INSERT INTO products VALUES(DEFAULT, 'Lemon', currval('category_id_seq'));
```