

SQL Databases Training

January 22 -25, 2019

Osman Aidel – Philippe Cheynet

- ▶ DB history (30 min)
- ▶ Relational Model (3h)
- ▶ SQL (3h)
- ▶ PL/SQL (4h)
- ▶ Advanced Objects (3h)
- ▶ Optimization (7h)



- ▶ DB history (30 min)
- ▶ Relational Model (3h)
- ▶ **SQL (3h)**
 - **Basic Information**
 - Advanced Information
- ▶ Advanced Objects (3h)
- ▶ PL/SQL (4h)
- ▶ Optimization (7h)



- ▶ SQL History
- ▶ Concepts
- ▶ Types
- ▶ Languages
 - DML
 - DDL
 - *DCL*
 - *TCL*
- ▶ *Advanced Usage*



Described in Edgar F. Codd's influential 1970 paper, "A Relational Model of Data for Large Shared Data Banks"

Originally based upon relational algebra and tuple relational calculus, SQL deviates in several ways from its theoretical foundation. In that model, a table is a set of tuples, while in SQL, tables and query results are lists of rows: the same row may occur multiple times, and the order of rows can be employed in queries

Despite not entirely adhering to the relational model as described by Codd, it became the most widely used database language.

SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987.[14] Since then, the standard has been revised to include a larger set of features. Despite the existence of such standards, most SQL code is not completely portable among different database systems without adjustments.

In the late 1970s, Relational Software, Inc. (now Oracle Corporation) saw the potential of the concepts described by Codd, Chamberlin, and Boyce, and developed their own SQL-based RDBMS. In June 1979, Relational Software, Inc. introduced the first commercially available implementation of SQL, Oracle V2 (Version2) for VAX computers.

What is and what is not SQL

Imperative programming languages. Every imperative language has a syntax, and a library of commands. You use these to tell the computer what to do. Like Prolog

SQL is a *declarative programming language*. You tell the database what data you want, and it will figure out the best way to get that data.

- ▶ Allows users to access data in the relational database management systems.
- ▶ Allows users to describe the data.
- ▶ Allows users to define the data in a database and manipulate that data.
- ▶ Allows to embed within other languages using SQL modules, libraries & pre-compilers.
- ▶ Allows users to create and drop databases and tables.
- ▶ Allows users to create views, stored procedures, functions in a database.
- ▶ Allows users to set permissions on tables, procedures and views.

However, extensions to Standard SQL add procedural programming language functionality, such as control-of-flow constructs.

SQL standard revisions:

Year	Name	Comments
1986	SQL-86	First formalized by ANSI.
1989	SQL-89	Minor revision that added integrity constraints, adopted as FIPS 127-1.
1992	SQL-92	Major revision (ISO 9075), Entry Level SQL-92 adopted as FIPS 127-2
1999	SQL:1999	Added regular expression matching, recursive queries (e.g. transitive closure), triggers, support for procedural and control-of-flow statements, non-scalar types (arrays), and some object-oriented features (e.g. structured types). Support for embedding SQL in Java (SQL/OLB) and vice versa (SQL/JRT).
2003	SQL:2003	Introduced XML-related features (SQL/XML), window functions, standardized sequences, and columns with auto-generated values (including identity-columns).
2006	SQL:2006	ISO/IEC 9075-14:2006 defines ways that SQL can be used with XML. It defines ways of importing and storing XML data in an SQL database, manipulating it within the database, and publishing both XML and conventional SQL-data in XML form. In addition, it lets applications integrate queries into their SQL code with XQuery, the XML Query Language published by the World Wide Web Consortium (W3C), to concurrently access ordinary SQL-data and XML documents.
2008	SQL:2008	Legalizes ORDER BY outside cursor definitions. Adds INSTEAD OF triggers, TRUNCATE statement, FETCH clause.
2011	SQL:2011	Adds temporal data (PERIOD FOR). Enhancements for window functions and FETCH clause.
2016	SQL:2016	Adds row pattern matching, polymorphic table functions, JSON.

Source : WIKIPEDIA « <https://en.wikipedia.org/wiki/SQL> »

Objects defined in SQL :

A relational database system (**RDBMS**) contains one or more related objects called **TABLES**. The data or information for the database are stored in these tables.

A single entry in a table is called a **TUPLE** or **RECORD**.

A tuple in a table represents a set of related data which can be broken down into several smaller parts of data known as **ATTRIBUTES**.

When an attribute is defined in a **RELATION** (TABLE), it is defined to hold only a certain type of values, which is known as **ATTRIBUTE DOMAIN**.

Tables are uniquely identified by their names and are comprised of columns and rows. Columns contain the column name, **data type**, and any other attributes for the column. Rows contain tuples/records for the relation.

Each column in a database table is required to have a name and a data type.

SQL General Data Types

- Boolean
- Character types such as char, varchar, and text.
- Numeric types such as integer and floating-point number.
- Temporal types such as date, time, timestamp, and interval
- Enumerated Types
- Binary Objects (BLOB)

Other Types

More details in the advanced part

Boolean

It can have three states: **TRUE**, **FALSE**, and **NULL**.

However, RDBMS are quite flexible when dealing with TRUE and FALSE values.

TRUE => 't', 'true', 'y', 'yes', '1'

FALSE => 'f', 'false', 'n', 'no', '0'

Uses with logical operator : =, ≠, not, and, or.

Three primary **character types**:

character(n) or **char(n)** : fixed-length, blank padded

character varying(n) or **varchar(n)** : variable-length with limit

text (varchar) : variable unlimited length

Both **char(n)** and **varchar(n)** can store up to **n** characters in length (**n** is a positive integer). If you try to store a longer string in the column that is either **char(n)** or **varchar(n)**, It will issue an error.

The text data type can store a string with unlimited length.

- **varchar** data type => **text** data type
- **character** or **char** => **character(1)** or **char(1)**

Different from other database systems, in PostgreSQL, there is no performance difference among three character types. In most situation, you should use **text** or **varchar**, and **varchar(n)** if you want PostgreSQL to check for the length limit.

Numeric Types

- Integers
- Arbitrary Precision Numbers
- Floating-Points Types
- Serial Types

NAME	STORAGE SIZE	DESCRIPTION	RANGE
Smallint	2 bytes	small-range integer	–32768 to +32767
Integer	4 bytes	typical choice for integer	–2147483648 to +2147483647
Bigint	8 bytes	large-range integer	–9223372036854775808 to +9223372036854775807
decimal	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
numeric	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
real	4 bytes	variable-precision, inexact	6 decimal digits precision
double precision	8 bytes	variable-precision, inexact	15 decimal digits precision
smallserial	2 bytes	small autoincrementing integer	1 to 32767
serial	4 bytes	autoincrementing integer	1 to 2147483647
bigserial	8 bytes	large autoincrementing integer	1 to 9223372036854775807

Integers

SQL only specifies the integer types integer (or int), smallint, and bigint. The type names int2, int4, and int8 are extensions, which are also used by some RDBMS.

Arbitrary Precision Numbers

The types decimal and numeric are equivalent. Both types are part of the SQL standard. Calculations with numeric values yield exact results whenever possible.

Floating-Point Types

These types are usually implementations of IEEE Standard 754 for Binary Floating-Point Arithmetic (single and double precision, respectively). The data types real and double precision are inexact.

Serial Types

The data types smallserial, serial and bigserial are not true types, but merely a notational convenience for creating unique identifier columns. They're similar to the AUTO_INCREMENT property supported by some other RDBMS.

Temporal types

Cover SQL date and time type full set

Dates are counted according to the Gregorian calendar, even in years before that calendar was introduced.

NAME	STORAGE SIZE	DESCRIPTION	LOW VALUE	HIGH VALUE	RESOLUTION
timestamp [(p)] [without time zone]	8 bytes	both date and time (no time zone)	4713 BC	294276 AD	1 microsecond
timestamp [(p)] with time zone	8 bytes	both date and time, with time zone	4713 BC	294276 AD	1 microsecond
date	4 bytes	date (no time of day)	4713 BC	5874897 AD	1 day
time [(p)] [without time zone]	8 bytes	time of day (no date)	00:00:00	24:00:00	1 microsecond
time [(p)] with time zone	12 bytes	time of day (no date), with time zone	00:00:00+1459	24:00:00-1459	1 microsecond
interval [fields] [(p)]	16 bytes	time interval	-178000000 years	178000000 years	1 microsecond

Value p specifies the number of fractional digits retained in the seconds field. Any date or time literal input needs to be enclosed in single quotes, like text strings.

Enumerated Types

Enumerated (enum) types are data types that comprise a static, ordered set of values. For example: ('sad', 'ok', 'happy')

BLOB

The SQL standard defines a different binary string type, called BLOB or BINARY LARGE OBJECT. MySQL and ORACLE give the same name. PostgreSQL has type bytea. The input format is different but the provided functions and operators are mostly the same.

Other types

- Monetary Type
- Geometric Types
- Network address
- UUID for storing Universally Unique Identifiers
- XML Type
- JSON Type
- Array for storing array strings, numbers, etc.
- Composite Types
- Range Types

<https://www.postgresql.org/docs/11/datatype.html>



NULL is not a value, it is a special marker to indicate that a data value does not exist in the database.

It is used to signify missing or unknown values.

The SQL keyword NULL is used to indicate these values.

NULL really isn't a specific value as much as it is an indicator.

Don't think of NULL as similar to zero or blank or false, it isn't the same.

Operators :

You must use the **IS NULL** or **IS NOT NULL** operators to check for a NULL value.

It is not possible to test for NULL values with comparison operators, such as =, <, or <>.

- Use comments to describe what SQL does
 - Put each major SQL statement on a new line
 - Put SQL keywords, built-in function and data types in the uppercase
- Use CamelCase capitalization and do not separate name parts with underscores
or use lower case and underscores to separate name parts
- Set standard abbreviations for frequently used objects, such as tbl for tables or idx for indexes
 - Use single quotation for characters, strings, binary and Unicode
 - Set a rule for naming aliases
 - Use indenting to align wrapped long lines
 - Use parentheses in complex mathematical expressions
 - Be consistent with indentation – use either tab or space
 - Don't avoid using line breaks to improve readability
 - Code grouping – keep the lines that execute a certain task in separate code blocks

SQL language is case insensitive (for the keywords).

Comments

-- line

/* ... */ block

DML Commands

Data Manipulation Language

Command	Description
SELECT	Retrieve certain records (get data) from a database
INSERT	Creates a record (insert data) into a table
UPDATE	Modifies records (update data) within a table
DELETE	Delete records (remove data) from a database table
MERGE (UPSERT)	Update a record if it already exists, otherwise, inserts a new row in a table.

The SELECT statement is one of the most complex statements in SQL. It has many clauses that you can combine to form a powerful query.

- **SELECT** – shows you how to query data from a single table.
- **ORDER BY**– guides you how to sort the result set returned from a query.
- **SELECT DISTINCT**– provides you a clause that removes duplicate rows in the result set.

```
SELECT column1, column2, ...  
FROM table1
```

```
SELECT DISTINCT column1 [,column2,..]  
FROM table1
```

```
SELECT column1, column2, ...  
FROM table1  
ORDER BY column1 [ASC/DESC]
```

It is not a good practice to use the asterisk (*) in the SELECT statement.

- **LIMIT** – gets a subset of rows generated by a query.

```
SELECT column1 FROM table LIMIT n OFFSET m;
```

- **FETCH**– limits the number of rows returned by a query.

```
OFFSET start { ROW | ROWS }  
FETCH { FIRST | NEXT } [ row_count ] { ROW | ROWS } ONLY
```

Conform with the SQL standard. FETCH clause introduced in SQL:2008

- **WHERE** – filters rows based on a specified condition.
where condition is any expression that evaluates to a result of type boolean.

Any row that does not satisfy this condition will be eliminated from the output → Evaluation returns TRUE

CONDITION

Conditions are built with:

- Logical Operators – AND, OR, NOT
- Comparison Operators – <, >, <=, >=, =, <> (or !=)
- Mathematical Functions and Operators – abs(), exp(), mod(), radians(),...
- [NOT] IN – selects data that matches any value in a list of values
- [NOT] BETWEEN [AND] – selects data that is a range of values
- [NOT] LIKE – filters data based on pattern matching
- IS [NOT] NULL – checks if a value is null or not.
- IS [NOT] DISTINCT FROM – not equal, treating NULL like an ordinary value


👉 7 = NULL or 7 <> NULL yields null.

For non-null inputs, IS DISTINCT FROM is the same as the '<>' operator and IS NOT DISTINCT FROM is identical to '='.

These predicates effectively act as though NULL were a normal data value, rather than “unknown”.

SQL – The complete postgresSQL SELECT clause

```
[ WITH [ RECURSIVE ] with_query [, ...] ]  
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]  
    [ * | expression [ [ AS ] output_name ] [, ...] ]  
    [ FROM from_item [, ...] ]  
    [ WHERE condition ]  
    [ GROUP BY grouping_element [, ...] ]  
    [ HAVING condition [, ...] ]  
    [ WINDOW window_name AS ( window_definition ) [, ...] ]  
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]  
    [ ORDER BY  
        expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]  
    [ LIMIT { count | ALL } ]  
    [ OFFSET start [ ROW | ROWS ] ]  
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]  
    [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE }  
        [ OF table_name [, ...] ] [ NOWAIT | SKIP LOCKED ] [...]
```



<https://www.postgresql.org/docs/11/sql-select.html>

When you create a new table, it does not have any data. The first thing you often do is to insert new rows into the table.

SQL provides the **INSERT** statement that allows you to insert one or more rows into a table at a time.

SQL INSERT syntax:

```
INSERT INTO table(column1, column2, ...)
VALUES (value1, value2, ...);
```

To add multiple rows into a table at a time:

```
INSERT INTO table (column1, column2, ...)
VALUES
(value1, value2, ...),
(value1, value2, ...),
...;
```


When you insert

- Character data, you must enclose it in single quotes (') for example 'SQL Tutorial'.
- Numeric data type, you don't need to do so, just use plain numbers such as 1, 2, 3.

If you omit any column that accepts the NULL value in the INSERT statement, the column will take its default value. In case the default value is not set for the column, the column will take the NULL value.

SQL provides a value for the serial column automatically so you do not and should not insert a value into the serial column.

When you insert

- Character data, you must enclose it in single quotes (') for example 'SQL Tutorial'. If a string contains a single quote character, you have to use a single quote (') escape character
J'arrive -> J"arrive
- Numeric data type, you don't need to do so, just use plain numbers such as 1, 2, 3.

If you omit any column that accepts the NULL value in the INSERT statement, the column will take its default value. In case the default value is not set for the column, the column will take the NULL value.

You can also use the DEFAULT keyword to set the default value for any column that has a default value.

SQL provides a value for the serial column automatically so you do not and should not insert a value into the serial column.

Very Usefull

To insert data that comes from another table, you can use the INSERT INTO SELECT statement

```
INSERT INTO table(column1,column2,...)
SELECT column1,column2,...
FROM another_table
WHERE condition;
```

The WHERE clause is used to filter rows that allow you to insert partial data

To change the values of the columns in a table, you use the **UPDATE** statement.

```
UPDATE table1  
SET column1 = value1 [, column2 = value2 ,...]  
WHERE condition;
```

If you omit the WHERE clause, all the rows in the table are updated.

To change the NULL values of a column, you can use the following statement:

```
UPDATE table1  
SET column1 = DEFAULT  
WHERE column1 IS NULL;
```

You can also update data of a column from another column within the same table:

```
UPDATE table1  
SET column1 = column2
```

If you want to update data of a column from column data in another table:

```
UPDATE table1
SET
    column2 = table2.column3
FROM
    table2
WHERE
    table1.column1 = table2.column1;
```

This kind of UPDATE statement is sometimes referred to as **UPDATE JOIN** or **UPDATE INNER JOIN** because two or more tables are involved in the UPDATE statement.

The FROM clause in the UPDATE statement specifies the second table
The join condition is specified in the WHERE clause.

JOINS will be detailed in the next session

To delete data from a table, you use **DELETE** statement

Syntax:

```
DELETE FROM table  
[ WHERE condition] ;
```

The WHERE clause is optional. However, if you omit it, the DELETE statement will delete all rows in the table.

If you want to remove all rows from the 'table1' that have values of the 'column1' in 'table2'.

```
DELETE FROM table1  
USING table2  
WHERE  
    table1.column1 = table2.column1;
```

In RDBMS, the term **UPSERT** is referred to as a **MERGE**.

The idea is that when you insert a new row into the table, the row will be updated if it already exists, otherwise, it's inserted.

That is why The action is call upsert (update or insert).

Concept introduced in the SQL:2006 standard.

RDBMS handle it from different ways:

`MERGE ... INTO ... USING ... ON ...`

Oracle

`INSERT ... ON DUPLICATE KEY UPDATE ...
REPLACE INTO ...`

MySQL

`INSERT INTO ... ON CONFLIT ...`

PostgreSQL

PosgreSQL UPSERT Syntax:

```
INSERT INTO table_name(column_list) VALUES(value_list)  
ON CONFLICT {DO NOTHING | DO UPDATE SET (value_list) [WHERE condition]}
```

Examples:

```
INSERT INTO customers (col2, col3)  
VALUES  
    ('val2', 'val3')  
ON CONFLICT (col2)  
DO NOTHING;
```

Column name

```
INSERT INTO customers (col2, col3)  
VALUES  
    ('val2', 'val3')  
ON CONFLICT ON CONSTRAINT col1  
DO NOTHING;
```

P.K.

```
INSERT INTO customers (col2, col3)  
VALUES  
    ('val2', 'val3')  
ON CONFLICT (col2)  
DO UPDATE  
    SET col3 = val3;
```


DDL Commands

Data Definition Language

Command	Description
CREATE	Create the database or its objects (like table, index, function, views, store procedure and triggers)
DROP	Deletes objects (like an entire table, a view of a table or other objects)
ALTER	Modifies an existing database object, such as a table
RENAME	Rename an object existing in the database
TRUNCATE	Remove all records from a table. All corresponded spaces allocated for them are released
COMMENT	Define or change the comment of an object

Create

The **CREATE** statement in SQL is used to create database objects.
You can use the CREATE statement in SQL to create databases, tables, indexes...

```
CREATE DATABASE database_name [parameters]  
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] table_name [parameters]  
CREATE [OR REPLACE] [TEMP | TEMPORARY] [RECURSIVE] view_name [parameters]  
CREATE [ CONCURRENTLY ] [ [ IF NOT EXISTS ] INDEX index_name [parameters]
```

But also all other objects in a database. For PostgreSQL, **42 commands**:

```
CREATE CAST — define a new cast  
CREATE EVENT TRIGGER — define a new event trigger  
CREATE EXTENSION — install an extension  
CREATE FUNCTION — define a new function  
CREATE GROUP — define a new database role  
CREATE SEQUENCE — define a new sequence generator  
CREATE STATISTICS — define extended statistics  
CREATE TRIGGER — define a new trigger  
CREATE TYPE — define a new data type  
CREATE USER — define a new database role  
CREATE VIEW — define a new view  
... ..
```

The most used **CREATE** statement in SQL is for tables:

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] table_name [parameters]
```

Syntax:

```
CREATE TABLE table_name (  
    column_name TYPE column_constraint,  
    table_constraint table_constraint,  
);
```

- Name of the new table after the **CREATE TABLE** clause.
- List the column names, their data types and constraints. A table may have multiple columns separated by a comma (,).
- Table constraints, that defines rules for the data in the table.

Note: Keyword **TEMPORARY** is for creating a temporary table.
Temporary tables are automatically dropped at the end of a session.

Create table

Column constraints

The following are the commonly used column constraints in SQL:

- **NOT NULL** – the value of the column cannot be NULL.
- **UNIQUE** – the value of the column must be unique across the whole table. SQL standard only allows one NULL value in the column that has the UNIQUE constraint (PostgreSQL treats each NULL value to be unique so such column can have many NULL values)
- **PRIMARY KEY** – this constraint is the combination of NOT NULL and UNIQUE constraints. You can define one column as PRIMARY KEY by using column-level constraint. In case the primary key contains multiple columns, you must use the table-level constraint.
- **CHECK** – enables to check a condition when you insert or update data. For example, the values in the price column of the product table must be positive values.
- **REFERENCES** – constrains the value of the column that exists in a column in another table. You use REFERENCES to define the foreign key constraint.

Create table

Table constraints

The table constraints are similar to column constraints except that they are applied to the entire table rather than to an individual column.

The following are the table constraints:

- **UNIQUE (column_list)** – to force the value stored in the columns listed inside the parentheses to be unique.
- **PRIMARY KEY(column_list)** – to define the primary key that consists of multiple columns.
- **CHECK (condition)** – to check a condition when inserting or updating data.
- **REFERENCES** – to constrain the value stored in the column that must exist in a column in another table..

Create table

Example:

create a new table named account that has the following columns with the corresponding constraints:

- user_id – primary key
- username – unique and not null
- password – not null
- email – unique and not null
- created_on – not null
- last_login – null

```
CREATE TABLE account (  
  user_id serial PRIMARY KEY,  
  username VARCHAR (50) UNIQUE NOT NULL,  
  password VARCHAR (50) NOT NULL,  
  email VARCHAR (355) UNIQUE NOT NULL,  
  created_on TIMESTAMP NOT NULL,  
  last_login TIMESTAMP  
);
```

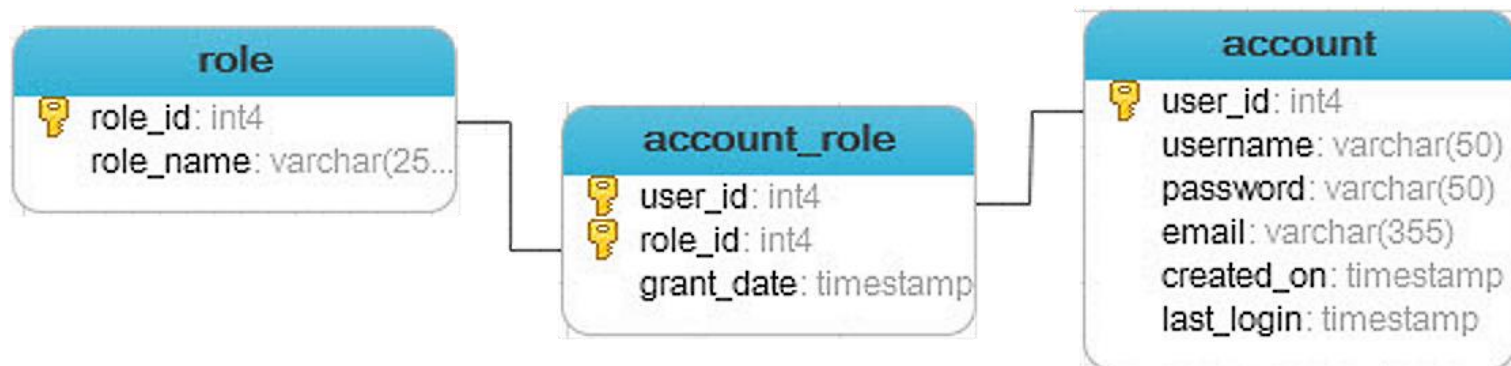
create a new table named role that has the following columns with the corresponding constraints:

- role_id – primary key
- Role_name – unique and not null

```
CREATE TABLE role(  
  role_id serial PRIMARY KEY,  
  role_name VARCHAR (255)  
              UNIQUE NOT NULL  
);
```

Create table

```
CREATE TABLE account_role (  
  user_id integer NOT NULL,  
  role_id integer NOT NULL,  
  grant_date timestamp without time zone,  
  PRIMARY KEY (user_id, role_id),  
  CONSTRAINT account_role_role_id_fkey FOREIGN KEY (role_id)  
    REFERENCES role (role_id) MATCH SIMPLE  
    ON UPDATE NO ACTION ON DELETE NO ACTION,  
  CONSTRAINT account_role_user_id_fkey FOREIGN KEY (user_id)  
    REFERENCES account (user_id) MATCH SIMPLE  
    ON UPDATE NO ACTION ON DELETE NO ACTION  
);
```



Create table

Create Table Using Another Table

```
CREATE TABLE new_table_name AS
SELECT column1, column2,...
FROM existing_table_name
WHERE conditions;
```

For complete definition, see :

<https://www.postgresql.org/docs/current/sql-createtable.html>

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } [ IF NOT EXISTS ] table_name ( [
column_name data_type [ COLLATE collection ] [ column_constraint [ ... ] ]
...
]
[ LIKE source_table [ ( like_option ... ) ]
]
)
[ INHERITS ( parent_table [ , ... ] ) ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } [ IF NOT EXISTS ] table_name
of type_name [ ( ( column_name [ data_type [ COLLATE collection ] [ options ] [ , ... ] ) ]
...
)
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } [ IF NOT EXISTS ] table_name
PARTITIONED BY ( partition_by ) ( ( column_name [ data_type [ COLLATE collection ] [ options ] [ , ... ] ) ]
...
)
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } [ IF NOT EXISTS ] table_name
PARTITIONED BY ( partition_by ) ( ( column_name [ data_type [ COLLATE collection ] [ options ] [ , ... ] ) ]
...
)
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]

where column_constraint is:
CONSTRAINT constraint_name
[ NOT NULL ]
CHECK ( expression ) [ NO INHERIT ]
GENERATED ALWAYS AS IDENTITY ( ( sequence_options ) )
UNIQUE index_parameters
PRIMARY KEY index_parameters
FOREIGN KEY ( column_name [ , ... ] ) REFERENCES referable [ ( on_delete_action ) ]
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
and table_constraint is:
CONSTRAINT constraint_name
[ NOT NULL ]
CHECK ( expression ) [ NO INHERIT ]
UNIQUE ( column_name [ , ... ] ) index_parameters
PRIMARY KEY ( column_name [ , ... ] ) index_parameters
FOREIGN KEY ( column_name [ , ... ] ) REFERENCES referable [ ( on_delete_action ) ]
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
and like_option is:
{ INCLUDING | EXCLUDING } { COMMENTS | CONSTRAINTS | DEFAULTS | IDENTITIES | INDEXES | STATISTICS | STORAGE | ALL }
and partition_bound_spec is:
IN ( ( numeric_literal | string_literal | time | value | NULL ) [ , ... ] )
FROM ( ( numeric_literal | string_literal | time | value | NULL | MAXVALUE ) [ , ... ] )
WITH ( ( numeric_literal | string_literal | time | value | NULL | MAXVALUE ) [ , ... ] )
index_parameters is UNIQUE, PRIMARY KEY, and EXCLUDE constraints are:
EXCLUDE ( column_name [ , ... ] )
WITH ( ( numeric_literal | string_literal | time | value | NULL ) [ , ... ] )
USING index_tablespace tablespace_name
exclude_element is an EXCLUDE constraint is:
( column_name [ ( expression ) ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ] )
```


Drop

The **DROP** statement in SQL is used to delete created database objects. You can use the DROP statement in SQL to delete databases, tables, indexes...

```
DROP DATABASE [ IF EXISTS ] database_name
DROP TABLE [ IF EXISTS ] table_name [ CASCADE ]
DROP VIEW [ IF EXISTS ] table_name [ CASCADE ]
DROP INDEX [ IF EXISTS ] index_name [ CASCADE ]
```

But also all other objects in a database. For PostgreSQL, **43 commands**:

```
DROP CAST — remove a cast
DROP EXTENSION — remove an extension
DROP FOREIGN TABLE — remove a foreign table
DROP FUNCTION — remove a function
DROP LANGUAGE — remove a procedural language
DROP OPERATOR — remove an operator
DROP OWNED — remove database objects owned by a database role
DROP SCHEMA — remove a schema
DROP SEQUENCE — remove a sequence
DROP TRIGGER — remove a trigger
DROP TYPE — remove a data type
... ..
```

The most used **DROP** statement in SQL is for tables:

```
DROP TABLE [IF EXISTS] table_name [CASCADE | RESTRICT];
```



If you drop a table, all the rows in the table is deleted and the table structure is removed from the database.

You can also use the DROP statement in SQL in conjunction with the **ALTER TABLE** statement to remove fields from tables i.e. columns.

```
ALTER TABLE table1  
  DROP COLUMN [ IF EXISTS ] column1 [,  
  DROP COLUMN [ IF EXISTS ] column2 ]
```

CASCADE:

Automatically drop objects that depend on the table (such as views, constraints or any other objects)

According to the SQL standard, specifying either **RESTRICT** or **CASCADE** is required in a DROP command.

Dependency Tracking

When you create complex database structures involving many tables with foreign key constraints, views, triggers, functions, etc.

You implicitly create a net of dependencies between the objects.

```
DROP TABLE table1 [RESTRICT];
```

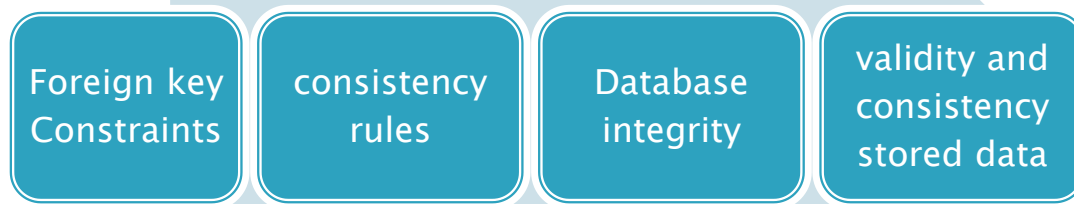
➡ ERROR: cannot drop table *table1* because other objects depend on it

The error message contains a useful hint:

...constraint *foreign_key_1* on table *table2* depends on table *table1*...

```
DROP TABLE table1 CASCADE;
```

and all the dependent objects will be removed, as will any objects that depend on them, recursively.



Alter

Modifies the definition of an existing database object, such as a table.

42 different **ALTER** commands in PostgreSQL

ALTER TABLE changes the definition of an existing table. There are several subforms. Note that the lock level required may differ for each subform.

Modify Table columns

```
ALTER TABLE table_name ADD COLUMN column_name data_type;
```

```
ALTER TABLE table_name DROP COLUMN [IF EXISTS] column_name [RESTRICT | CASCADE]
```

Alter

Modify Table Constraints

```
ALTER TABLE table_name ADD table_constraint [ NOT VALID ]
```

```
ALTER TABLE table_name VALIDATE CONSTRAINT constraint_name
```

This form validates a foreign key or check constraint that was previously created as **NOT VALID**, by scanning the table to ensure there are no rows for which the constraint is not satisfied.

```
ALTER TABLE table_name DROP CONSTRAINT [IF EXISTS] constraint_name [RESTRICT | CASCADE]
```

Note: ALTER TABLE that act on a single table can be combined into a list of multiple alterations to be applied together.
For example, it is possible to add several columns and/or alter the type of several columns in a single command.

Rename

Only 3 **RENAME** Commands in PostgreSQL

```
ALTER TABLE [ IF EXISTS ] [ ONLY ] table_name  
    RENAME [ COLUMN ] column_name TO new_column_name
```

```
ALTER TABLE [ IF EXISTS ] [ ONLY ] name  
    RENAME CONSTRAINT constraint_name TO new_constraint_name
```

```
ALTER TABLE [ IF EXISTS ] name  
    RENAME TO new_name
```

Note : When renaming a constraint that has an underlying index, the index is renamed as well

Truncate

You can use SQL **TRUNCATE** TABLE statement to remove all data from large tables quickly.

To remove all data from a table, you use the DELETE statement. However, for a large table, it is more efficient to use the TRUNCATE TABLE statement.

The TRUNCATE TABLE statement removes all rows from a table without scanning it.

```
TRUNCATE TABLE table_name1, table_name2, ...
```

- ✓ Reclaim Space immediately
- ✓ No impact on transaction logs

Comment

There is no **COMMENT** command in the SQL standard.

You can use a SQL COMMENT statement to store a comment about a database object.

Only one comment string is stored for each object, so to modify a comment, issue a new COMMENT command for the same object.

To remove a comment, write NULL in place of the text string.

```
COMMENT ON object_name IS 'text'
```


SELECT questions FROM audience;

