

# SD202: Databases

## SQL language

---

Louis Jachiet based on Antoine Amarilli's slides.

Télécom Paris

- We saw generalities about databases:
  - it abstracts away all the messy things needed to deal with data
  - it provides a lot of features (multi-users, transactions, etc.)
- We saw a “toy” language based on the relational algebra

General information

Implementation and setup

Schema creation

Creating data

Reading data

Advanced SELECT clauses

Modifying and deleting data

## General information

---

- **SQL**: Structured Query Language
- Language to manage data in a relational database
- Several **sublanguages**:
  - Data **definition** language: create/modify table schema
  - Data **manipulation** language: create/edit/query data
  - Data **control** language: users and rights
  - **Procedural** extensions: PL/SQL, SQL/PSM, PL/pgSQL...
- Implemented by **common database engines**



EN

MENU

ICS > 35 > 35.060

## ISO/IEC 9075-1:2016

### Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework)

ABSTRACT

PREVIEW

ISO/IEC 9075-1:2016 describes the conceptual framework used in other parts of ISO/IEC 9075 to specify the grammar of SQL and the result of processing statements in that language by an SQL-implementation.

ISO/IEC 9075-1:2016 also defines terms and notation used in the other parts of ISO/IEC 9075.

#### GENERAL INFORMATION

Status : Published

Publication date : 2016-12

Edition : 5

Number of pages : 78

SQL is an **industry standard**:

- First version in **1974**
- Latest version in **2016**
- **78 pages** (not so long!)
- Price: **178 CHF** (!)
- **Theory**: easy **migration** from a database engine to another
- **Practice**: many **incompatibilities**
- **Practice**: database engines do not usually implement the **full standard**, and/or add **extensions**

# Declarativity

SQL is a **declarative** language:

- specify **what** you want
  - “find all female actors who played in Hollywood movies”
- not **how** to compute it

# Declarativity

SQL is a **declarative** language:

- specify **what** you want
  - “find all female actors who played in Hollywood movies”
- not **how** to compute it
  - this was presented as a “**relational language**” in the previous class (DROP, FILTER, etc.)
  - “take all films and keep the ones from Hollywood, then take the actors who played there and keep the ones who are female”
  - “take all actors and keep the ones who are female, then take the films where they played and keep the ones from Hollywood”

The database engine will translate SQL to a concrete **execution plan** (more later)



```
SELECT * FROM Movie WHERE title = 'Avatar';
```

- Keywords are **English words** typically used in **uppercase**
- Whitespaces are **ignored** (line breaks, etc.)
- **Statements** are finished by a **semicolon**
- **Comments**, with `--` or `/* ... */`

General information

Implementation and setup

Schema creation

Creating data

Reading data

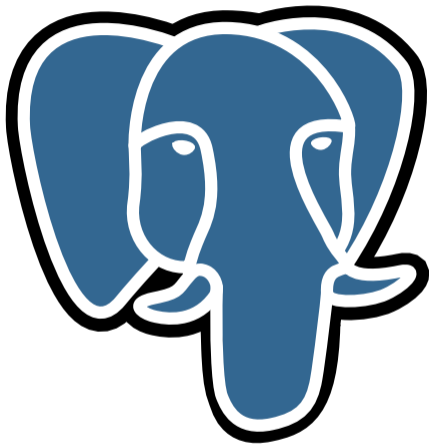
Advanced SELECT clauses

Modifying and deleting data

# Implementation and setup

---

## Step 1: installing an RDBMS



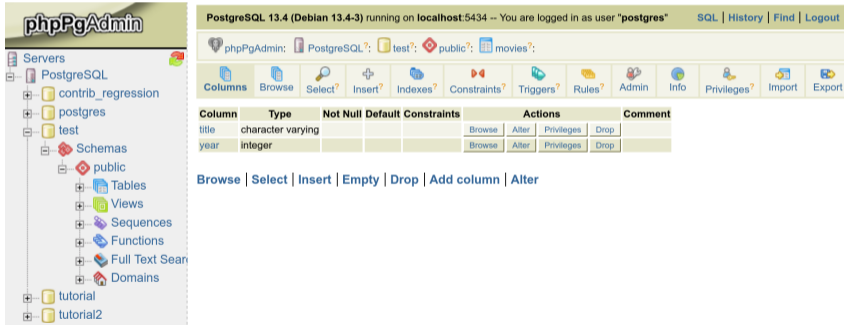
- Install a **Relational DataBase Management System**
- Let's choose **PostgreSQL**

## Step 2: interacting with the RDBMS

- Simplest way: use the **command line**
- Special commands (for PostgreSQL):
  - `\l` to list databases
  - `\c database` to change database
  - `\dt` to list tables
  - `\d table` to show details about a table
- You can issue **commands** (do not forget the **semicolon**)
- You can retrieve query results on **standard output**
- You can pipe a command from **standard input**

## Step 2a: using a graphical interface

You can install **phpPgAdmin**:



The screenshot displays the phpPgAdmin interface for PostgreSQL 13.4. The left sidebar shows a tree view of the database structure, including servers, databases (contrib\_regression, postgres, test), and schemas (public, tutorial, tutorial2). The main content area shows the 'test' database selected, with the 'public' schema expanded to show a table named 'movies'. The table structure is as follows:

Column	Type	Not Null	Default	Constraints	Actions	Comment
title	character varying				<a href="#">Browse</a> <a href="#">Alter</a> <a href="#">Privileges</a> <a href="#">Drop</a>	
year	integer				<a href="#">Browse</a> <a href="#">Alter</a> <a href="#">Privileges</a> <a href="#">Drop</a>	

Below the table, there are navigation links: [Browse](#) | [Select](#) | [Insert](#) | [Empty](#) | [Drop](#) | [Add column](#) | [Alter](#)

## Step 2b: using an API

First create a **user** to connect to the database:

```
CREATE USER testuser WITH ENCRYPTED PASSWORD 'PASS'  
GRANT ALL PRIVILEGES ON DATABASE test TO testuser;  
\\c test  
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO testuser;
```

Then write code (here, **psycopg2** with Python):

```
import psycopg2  
conn = psycopg2.connect(  
    "host=localhost dbname=test port=5432 user=testuser password=PASS")  
cur = conn.cursor()  
cur.execute("SELECT * FROM Movies")  
print (cur.fetchone())  
conn.close()
```

## Other approach: SQLite



- Install `sqlite`
- Run `sqlite3 file.sqlite`
- **That's it!**
- Graphical interface: `sqlitebrowser`



General information

Implementation and setup

Schema creation

Creating data

Reading data

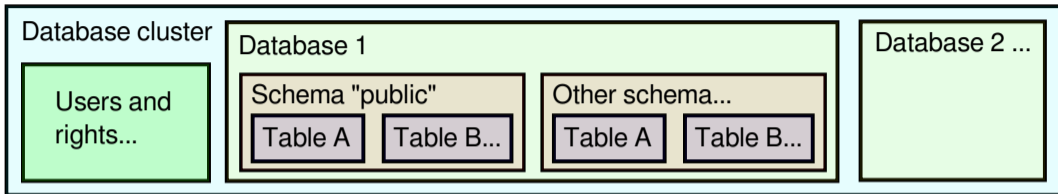
Advanced SELECT clauses

Modifying and deleting data

# Schema creation

---

# Overall structure



- A database cluster contains users/groups and **databases**
- A database contains several **schemas** (the default one is **public**)
- A schema contains **tables**
  - The same table name can occur in **multiple schemas**
  - Can be qualified with the schema name
  - Notion of **search path** to disambiguate unqualified names
- A table has a **structure** (also called its **schema**) and **data** (rows)

Recall that the relational model is composed of several **tables**:

<b>Movie</b>		
<b>id</b>	<b>name</b>	<b>year</b>
1	Avatar	2009
2	Avengers: Endgame	2019

Basic instruction to create a table:

```
CREATE TABLE Movie(id SERIAL, title VARCHAR, year INT);
```

## Naming tables and attributes

- Table names should be **singular**
- No **accents**, no special **characters**, **underscores** rather than spaces
  - You can use double quotes for this, but **discouraged**
- Table and attribute names are not **case-sensitive**
  - Except if using double quotes – still **discouraged**
- Probably have a column named **id** for the **primary key** (later)
- Several tables can have the **same attribute name**, but they will need to be disambiguated (e.g., `R.id` vs `S.id`)
- Avoid any **reserved names** (e.g., `end`)
- Most important: **consistency!**

# Basic PostgreSQL types

- BOOLEAN for **Boolean values**
- INT for **integers** (4-byte)
  - SERIAL for an **auto-incrementing identifier** (4-byte), or AUTO\_INCREMENT with MySQL
- REAL for **floating-point numbers** (4-byte)
- NUMERIC for **high-precision numbers** (1000 digits)
- TEXT or VARCHAR: **text**
  - VARCHAR(42): text of length **at most 42**
- BYTEA or BLOB for **binary strings**
- TIMESTAMP for **date and time** (can be WITH TIME ZONE), DATE, etc.
- **Other**: money, enumerated types (enums), geometric types, JSON and XML, network addresses, UUIDs, arrays...

## Modifying and deleting a table

```
ALTER TABLE Movie ADD COLUMN test BOOLEAN;  
ALTER TABLE Movie ALTER COLUMN test TYPE int USING test::integer;  
ALTER TABLE Movie RENAME COLUMN test TO test2;  
ALTER TABLE Movie DROP COLUMN test;  
ALTER TABLE Movie RENAME TO Movie2;  
DROP TABLE Movie2;
```

We can enforce some **constraints** on the tuples that we create:

- **Check** constraints
- **Keys** and **uniqueness constraints** (related to schema design)
- **Foreign key** constraints



## Check constraints

```
CREATE TABLE Filming(id SERIAL PRIMARY KEY, title VARCHAR,  
  tstart DATE CHECK (tstart > '1895-01-01'),  
  tend DATE,  
  CHECK (tstart < tend));
```

- Constraints can check values in the **current tuple**
- You can give **names** to constraints to refer to them
- Special case: **NOT NULL** to disallow the default value (NULL)

# Primary keys and uniqueness constraints



- **PRIMARY KEY:** value is unique, non-NULL, and is the “**main way**” to refer to a tuple of the table
  - In practice, you often use a column `id` **just for that purpose**
  - Can be an **existing identifier** (e.g., ISBN) if you **trust** it
  - Can be **multiple columns** for an n:n-relation (more later)
- **UNIQUE:** value (or tuple of values) is **unique**

These constraints automatically create an **index** (see later)

# Foreign keys

```
ALTER TABLE Movie ADD COLUMN filming INT REFERENCES Filming(id)
```

The value of the **filming** attribute must be the **id** of a tuple in **Filming** relation

- You can have a foreign key on a **tuple** of columns, e.g.,  
`FOREIGN KEY (a, b) REFERENCES Table(c, d)`
- The target attribute(s) must have a **uniqueness constraint**
  - Usually, it is the **primary key**, and you can omit the attribute name
- NULL is **allowed** (unless you imposed NOT NULL)
- This constraint can be **broken** when changing the referenced table!

## Repairing foreign key violations

- Default: **prohibit deletion** (except in a transaction, see later)
- ON DELETE RESTRICT: **completely prohibit deletion**
- ON DELETE CASCADE: also delete referencing tuples (**dangerous**)
- ON DELETE SET NULL: replace the reference by a **NULL**
- ON DELETE SET DEFAULT: replace the reference by the **default value** (which must also obey the foreign key constraint)
- Same questions when **updating** tuples

General information

Implementation and setup

Schema creation

**Creating data**

Reading data

Advanced SELECT clauses

Modifying and deleting data

## Creating data

---

## INSERT query

```
INSERT INTO Movie (title, year) VALUES ('Titanic', '1997');
```

- Can specify **multiple tuples** to insert
- Can omit the **field names**, they are then filled in order
- Instead of specifying **VALUES**, we can put a **SELECT query** to execute on existing data (see later)

## Default values for an insert

For fields that are **not specified**:

- For a **SERIAL**, automatically use a **“next” value**
- If a **DEFAULT** value was supplied, **use it**
- Otherwise, use **NULL**
- (Failure if **NOT NULL** was specified)



## Performance of bulk INSERTs

Doing multiple INSERTs can be **slow**... to improve performance:

- Do the INSERTs within a **single transaction** rather than committing after each INSERT
- Run a single INSERT command with multiple values
- Temporarily **remove** keys and indexes (and rebuild them at the end)
- Use the **COPY** command to load a file directly

General information

Implementation and setup

Schema creation

Creating data

**Reading data**

Advanced SELECT clauses

Modifying and deleting data

## Reading data

---

## Most basic SELECT query

```
SELECT * FROM Movie;
```

id	title	year
1	Frozen II	2019
2	Titanic	1997
3	Avengers: Endgame	2019
4	Avengers: Infinity War	2018
5	Star Wars: The Force Awakens	2015
6	Avatar	2009

## Projection: removing some attributes

```
SELECT year FROM Movie;
```

```
SELECT DISTINCT year FROM Movie;
```

```
SELECT DISTINCT ON (year) * FROM movie;
```

The DISTINCT keyword is **not** enabled by default (performance)

## Renaming output attributes

```
SELECT title AS t, year AS y FROM Movie;
```

id	t	y
1	Frozen II	2019
2	Titanic	1997
3	Avengers: Endgame	2019
4	Avengers: Infinity War	2018
5	Star Wars: The Force Awakens	2015
6	Avatar	2009

## Expressions over attributes

```
SELECT CONCAT(title, ' - ', year) AS long_name FROM Movie;
```

long\_name

---

Frozen II - 2019

Titanic - 1997

Avengers: Endgame - 2019

Avengers: Infinity War - 2018

Star Wars: The Force Awakens - 2015

Avatar - 2009

## Filtering: the WHERE clause

```
SELECT * FROM Movie WHERE year = '2019';
```

```
SELECT title FROM Movie WHERE year = '2019';
```

```
SELECT title FROM Movie WHERE year = '2019' AND title LIKE '%Frozen%';
```



## Possible conditions

- Compare attribute values **to constants**, or **among themselves**
- Test **equality**, **inequality**, **order**
- **Boolean conditions** : AND, OR, NOT
- **Value lists**: `year IN ('2019', '43')`
- **LIKE operator**: tests string equality to a pattern with '%' and '\_'
  - **ILIKE**: case-insensitive
- More **complex** expressions, e.g., `WHERE LENGTH(title) > 10`
- For **performance**, distinguish between:
  - Conditions that require a **full scan** of the table
  - Conditions implementable using **indexes**

## Selecting multiple tables: PRODUCT

```
SELECT * FROM Movie, Actor;
```

```
SELECT * FROM Movie, Actor, Actor_in_movie;
```

**Exercise:** how to select the titles of movies and the names of actors who played in that movie?

## Selecting multiple tables: PRODUCT

```
SELECT * FROM Movie, Actor;
```

```
SELECT * FROM Movie, Actor, Actor_in_movie;
```

**Exercise:** how to select the titles of movies and the names of actors who played in that movie?

We want to do a **join**:

```
SELECT title, year, name FROM Movie, Actor, Actor_in_movie
    WHERE Actor.id = Actor_in_movie.actor
    AND Movie.id = Actor_in_movie.movie;
```

Note the disambiguation of **ambiguous** attribute names

## Other kinds of join

```
SELECT title, actor FROM Movie, Actor_in_movie  
WHERE Movie.id = Actor_in_movie.movie;
```

```
SELECT title, actor FROM Movie INNER JOIN Actor_in_movie  
ON Movie.id = Actor_in_movie.movie;
```

```
SELECT title, actor FROM Movie LEFT OUTER JOIN Actor_in_movie  
ON Movie.id = Actor_in_movie.movie;
```

```
SELECT title, actor FROM Movie RIGHT OUTER JOIN Actor_in_movie  
ON Movie.id = Actor_in_movie.movie;
```

```
SELECT title, actor FROM Movie FULL OUTER JOIN Actor_in_movie  
ON Movie.id = Actor_in_movie.movie;
```

Can you guess the **difference**?

# Difference between joins

These two are equivalent: they **drop** any rows that do not match:

```
SELECT title, actor FROM Movie, Actor_in_movie WHERE Movie.id = Actor_in_movie.movie;  
SELECT title, actor FROM Movie INNER JOIN Actor_in_movie ON Movie.id = Actor_in_movie.movie;
```

This one adds **one copy** of the left table rows that do not match (with NULLs):

```
SELECT title, actor FROM Movie LEFT OUTER JOIN Actor_in_movie ON Movie.id = Actor_in_movie.movie;
```

Likewise for the **right table** rows:

```
SELECT title, actor FROM Movie RIGHT OUTER JOIN Actor_in_movie ON Movie.id = Actor_in_movie.movie;
```

Likewise for **both tables**:

```
SELECT title, actor FROM Movie FULL OUTER JOIN Actor_in_movie ON Movie.id = Actor_in_movie.movie;
```

**Exercise:** In this example, some of these are **equivalent**. Why?

## Unioning: UNION

```
SELECT * FROM Teacher UNION SELECT * FROM Actor;
```

- The **number of columns** and **types** must be the same (but the **names** do not have to be)
- Removes **duplicates** unless you use `UNION ALL`

## Difference: EXCEPT

```
SELECT id FROM Teacher EXCEPT SELECT id FROM Actor;
```

- Same condition on **columns**; also **EXCEPT ALL**
- Also **INTERSECT** and **INTERSECT ALL** for intersection

## Difference: EXCEPT

```
SELECT id FROM Teacher EXCEPT SELECT id FROM Actor;
```

- Same condition on **columns**; also **EXCEPT ALL**
- Also **INTERSECT** and **INTERSECT ALL** for intersection

**Exercise:** how to express **INTERSECT** using a **WHERE** clause?



# Ordering data

By default, the data is **not sorted** and the order is **not consistent**:

- `ORDER BY date`
- `ORDER BY date DESC`
- `ORDER BY a + b`

Sometimes, we do not want the **full result** (e.g., pagination)

- LIMIT 1
- LIMIT 2
- OFFSET 1 LIMIT 1
- OFFSET 1 LIMIT 2

**Do not forget to use ORDER BY!**

General information

Implementation and setup

Schema creation

Creating data

Reading data

**Advanced SELECT clauses**

Modifying and deleting data

## Advanced SELECT clauses

---

# Groups and aggregation

```
SELECT genre, MAX(year) FROM Movie GROUP BY genre;
```

- Create one **group** per value of `genre` (could be multiple attributes)
- Attributes not in `GROUP BY` can only be **aggregated**
- Common **aggregate functions**: min, max, count, average, sum

## Advanced aggregation

- You can **filter out** groups with a **HAVING** clause (like **WHERE**, but evaluated after the aggregation)
- You can also use **window functions** to compute aggregates in a rolling fashion following some partitioning of the data

```
SELECT *, RANK() OVER (PARTITION BY genre ORDER BY id) AS rank
FROM Movie;
```

```
SELECT *, MIN(year) OVER (PARTITION BY genre ORDER BY year) AS minyear
FROM Movie;
```

## Subqueries in FROM

The **FROM** clause of a **SELECT** query can refer to a table evaluated with **another SELECT** query:

```
SELECT * FROM
  (SELECT * FROM Movie WHERE title LIKE 'Avengers%') AS M1
WHERE year = '2019';
```

The subquery **must** have an alias (here, M1), even if it is not used

## Subqueries in FROM

The **FROM** clause of a **SELECT** query can refer to a table evaluated with **another SELECT** query:

```
SELECT * FROM
  (SELECT * FROM Movie WHERE title LIKE 'Avengers%') AS M1
WHERE year = '2019';
```

The subquery **must** have an alias (here, M1), even if it is not used

**Exercise:** Can you **simplify** this query?



## Subqueries in WHERE (with EXISTS, etc.)

The WHERE clause of a SELECT query can **check existence** of a query result

```
SELECT id, title FROM Movie WHERE EXISTS  
    (SELECT 1 FROM Actor_in_movie WHERE movie = Movie.id);
```

Other possibility:

```
SELECT id, title FROM Movie WHERE id IN  
    (SELECT movie FROM Actor_in_movie);
```

## Subqueries in WHERE (with EXISTS, etc.)

The WHERE clause of a SELECT query can **check existence** of a query result

```
SELECT id, title FROM Movie WHERE EXISTS  
    (SELECT 1 FROM Actor_in_movie WHERE movie = Movie.id);
```

Other possibility:

```
SELECT id, title FROM Movie WHERE id IN  
    (SELECT movie FROM Actor_in_movie);
```

**Exercise:** Can you simplify this query?

## Subqueries in WHERE (with EXISTS, etc.)

The WHERE clause of a SELECT query can **check existence** of a query result

```
SELECT id, title FROM Movie WHERE EXISTS
  (SELECT 1 FROM Actor_in_movie WHERE movie = Movie.id);
```

Other possibility:

```
SELECT id, title FROM Movie WHERE id IN
  (SELECT movie FROM Actor_in_movie);
```

**Exercise:** Can you simplify this query?

```
SELECT DISTINCT Movie.id, title FROM Movie, Actor_in_movie
  WHERE Movie.id = Actor_in_movie.movie;
```

## Subqueries in WHERE (with EXISTS, etc.)

The WHERE clause of a SELECT query can **check existence** of a query result

```
SELECT id, title FROM Movie WHERE EXISTS
  (SELECT 1 FROM Actor_in_movie WHERE movie = Movie.id);
```

Other possibility:

```
SELECT id, title FROM Movie WHERE id IN
  (SELECT movie FROM Actor_in_movie);
```

**Exercise:** Can you simplify this query?

```
SELECT DISTINCT Movie.id, title FROM Movie, Actor_in_movie
  WHERE Movie.id = Actor_in_movie.movie;
```

**Other operators** : = ANY, = SOME, etc.

## Exercise

**Exercise:** How can you find the **latest films** in the database? (the ones whose year is the greatest)

## Exercise

**Exercise:** How can you find the **latest films** in the database? (the ones whose year is the greatest)

*-- first select the greatest year*

```
SELECT max(year) AS maxyear FROM Movie;
```

*-- now select where the year is greatest*

```
SELECT title, year
```

```
    FROM Movie, (SELECT max(year) AS maxyear FROM MOVIE) AS T
```

```
    WHERE Movie.year = T.maxyear;
```

## Exercise

**Exercise:** How can you find the **latest films** in the database? (the ones whose year is the greatest)

```
-- first select the greatest year
```

```
SELECT max(year) AS maxyear FROM Movie;
```

```
-- now select where the year is greatest
```

```
SELECT title, year
```

```
FROM Movie, (SELECT max(year) AS maxyear FROM MOVIE) AS T
```

```
WHERE Movie.year = T.maxyear;
```

How can you find the **latest films** for each genre?

## Exercise

**Exercise:** How can you find the **latest films** in the database? (the ones whose year is the greatest)

```
-- first select the greatest year
SELECT max(year) AS maxyear FROM Movie;
-- now select where the year is greatest
SELECT title, year
       FROM Movie, (SELECT max(year) AS maxyear FROM MOVIE) AS T
       WHERE Movie.year = T.maxyear;
```

How can you find the **latest films** for each genre?

```
SELECT title, year, Movie.genre FROM Movie,
       (SELECT genre, max(year) AS maxyear FROM MOVIE GROUP BY genre) AS T
       WHERE Movie.year = T.maxyear AND Movie.genre = T.genre;
```



## Recap: full SELECT syntax

```
SELECT [DISTINCT] [attrs]
FROM [tables, possibly with subexpressions]
WHERE [condition]
GROUP BY [grouping element]
HAVING [filter on groups]
UNION/INTERSECT/EXCEPT [ALL] [other queries...]
ORDER BY [criterion]
LIMIT [limit]
OFFSET [offset]
```

General information

Implementation and setup

Schema creation

Creating data

Reading data

Advanced SELECT clauses

**Modifying and deleting data**

## Modifying and deleting data

---

## DELETE and TRUNCATE commands

To **remove tuples**:

```
DELETE FROM Table WHERE [condition]
```

**Warning:**

- This can remove **more data than expected** if the condition is wrong
- There is **no confirmation**

To remove **all rows** (faster):

```
TRUNCATE TABLE Table
```

**Warning:** this will remove all data (without confirming)

## UPDATE command

```
UPDATE Actor SET name = 'Eliott Page' WHERE id = 42;  
UPDATE Movie SET year = year+1 WHERE title LIKE 'Avengers%';
```

### Warning:

- This can **mess up** more data than expected if the **condition** is wrong
- There is **no confirmation!**

# Avoiding data disasters

- Use a **transaction** and only COMMIT when you are sure of the result
- Have **backups**, e.g., use `pg_dump`
- **Run a SELECT** before UPDATE or DELETE, with the same WHERE clause