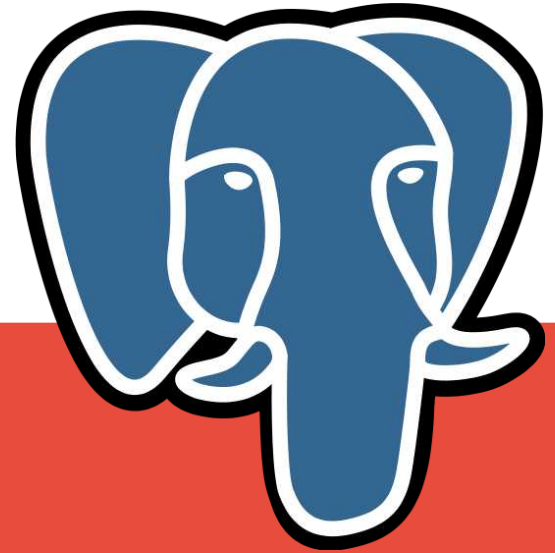


**MPS PostgreSQL**



# Architettura di Postgres

Postgres segue un modello client-server e la sua architettura è basata su processi differenti.

1. Processo Server (Postmaster), è il processo principale, (denominato postmaster o postgres a seconda della piattaforma e della versione) ed è anche il "padre" di tutti gli altri processi Postgres che coordina tutti gli altri processi e la memoria condivisa.

Rimane in attesa delle richieste di connessione dei client e, quando una richiesta arriva, avvia (fork) un nuovo processo backend per gestirla.

# Architettura di Postgres

## 2. Processi Backend o Server Process

Ogni connessione attiva di un client è gestita da un processo server dedicato, chiamato Processo Backend (o Processo Server).

Il processo esegue il parsing, l'ottimizzazione e l'esecuzione delle query SQL inviate dal client connesso.

Ogni backend è isolato dagli altri, garantendo che gli eventuali problemi di una sessione non influiscano sulle altre.

# Architettura di Postgres

3. Processi Ausiliari (o Background Processes), sono processi in background che eseguono attività di manutenzione e housekeeping (di governo) per il database, come:

- Checkpointer: che scrive periodicamente i dati sporchi (modificati in memoria, ma non ancora passati alla persistenza) su disco.
- WAL Writer: che scrive su disco i record del Write-Ahead Log (WAL).
- Autovacuum Daemon: che esegue automaticamente operazioni di pulizia per recuperare lo spazio occupato da righe obsolete (importante per il controllo della MVCC - Multi-Version Concurrency Control).
- Background Writer: che scrive i dati sporchi dai buffer su disco più frequentemente di quanto farebbe il checkpointer, riducendo i picchi di I/O.

4. Memoria Condivisa (Shared Memory), che è un'area di memoria utilizzata da tutti i processi per condividere informazioni e dati, il cui fine è migliorare le performance. Gli elementi principali includono:

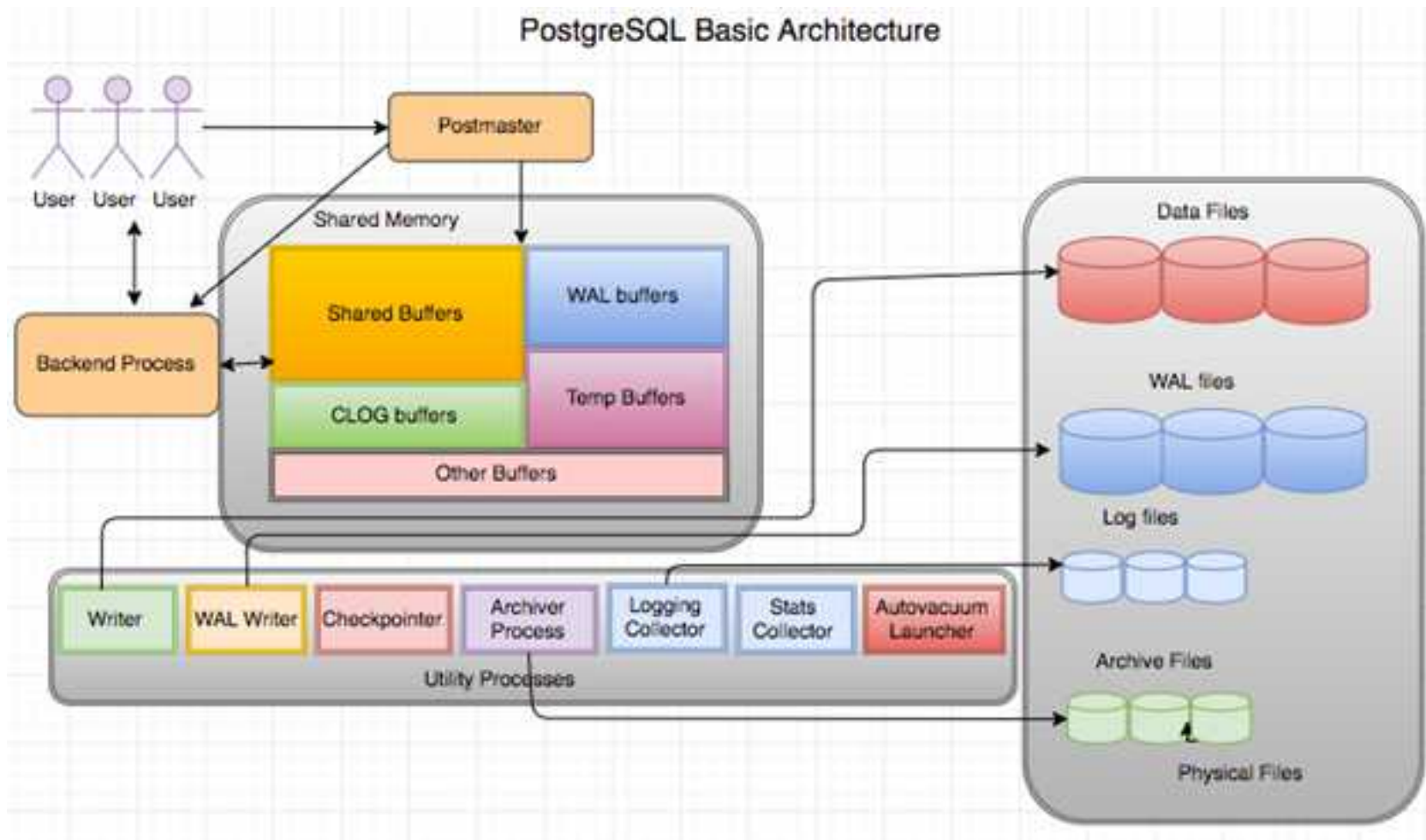
- **Shared Buffers:** ovvero l'area principale per la cache dei dati delle tabelle e degli indici. È il posto in cui i dati vengono letti dal disco e modificati prima di essere riscritti.
- **WAL Buffers:** Buffer per il Write-Ahead Log (WAL).
- **Catalog Cache:** Cache delle informazioni di sistema (metadati).

## Architettura di Postgres

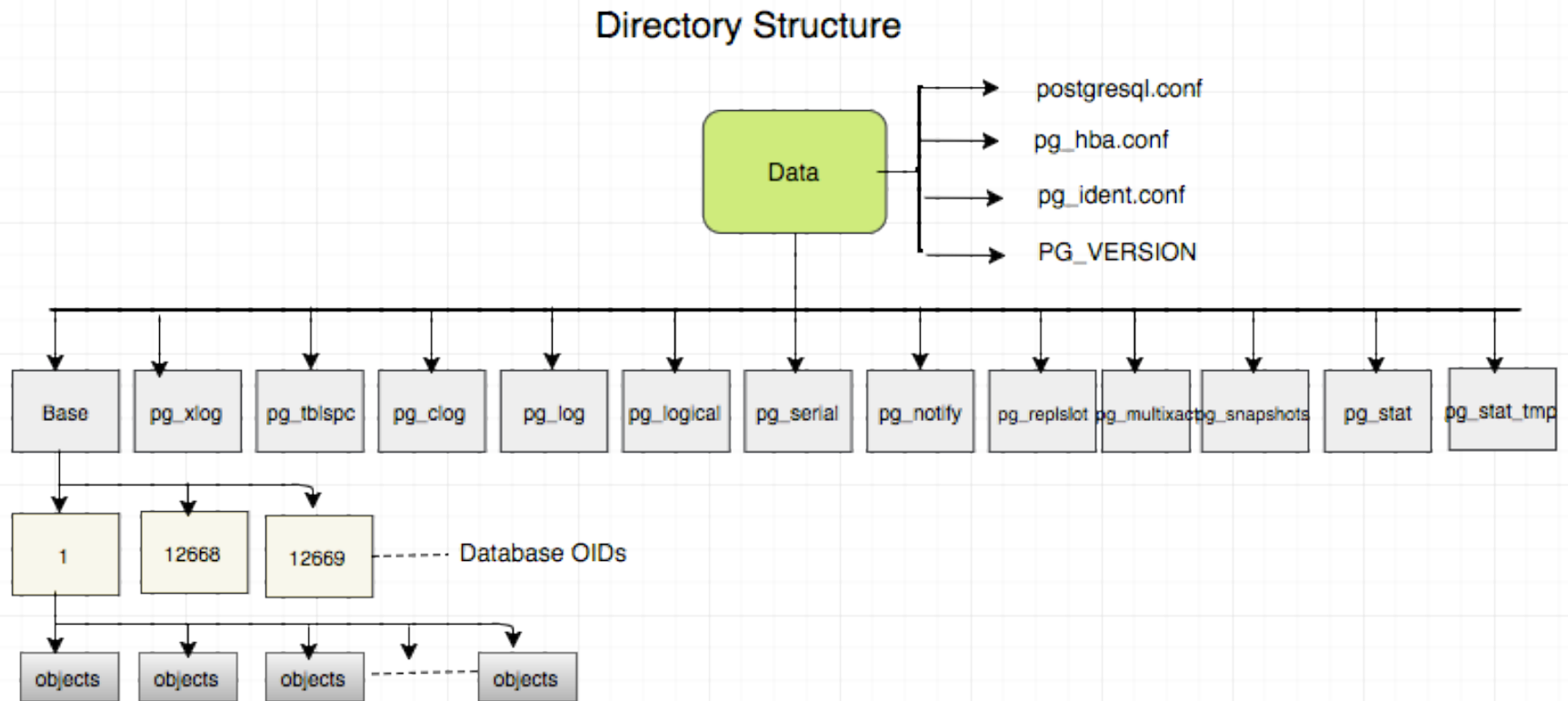
5. Struttura del Disco (Disk Structure), Postgres gestisce l'archiviazione dei dati su disco tramite diverse strutture:

- **Data Files:** che contengono i dati effettivi delle tabelle e degli indici.
- **Write-Ahead Log (WAL):** tutte le modifiche al database vengono registrate prima nel WAL (su disco) e solo in seguito applicate ai data file. Questo garantisce che, in caso di crash, le transazioni committate possano essere recuperate (durability).
- **Configuration Files:** File come `postgresql.conf` e `pg_hba.conf` che controllano il comportamento del server e l'autenticazione.

# Architettura di Postgres

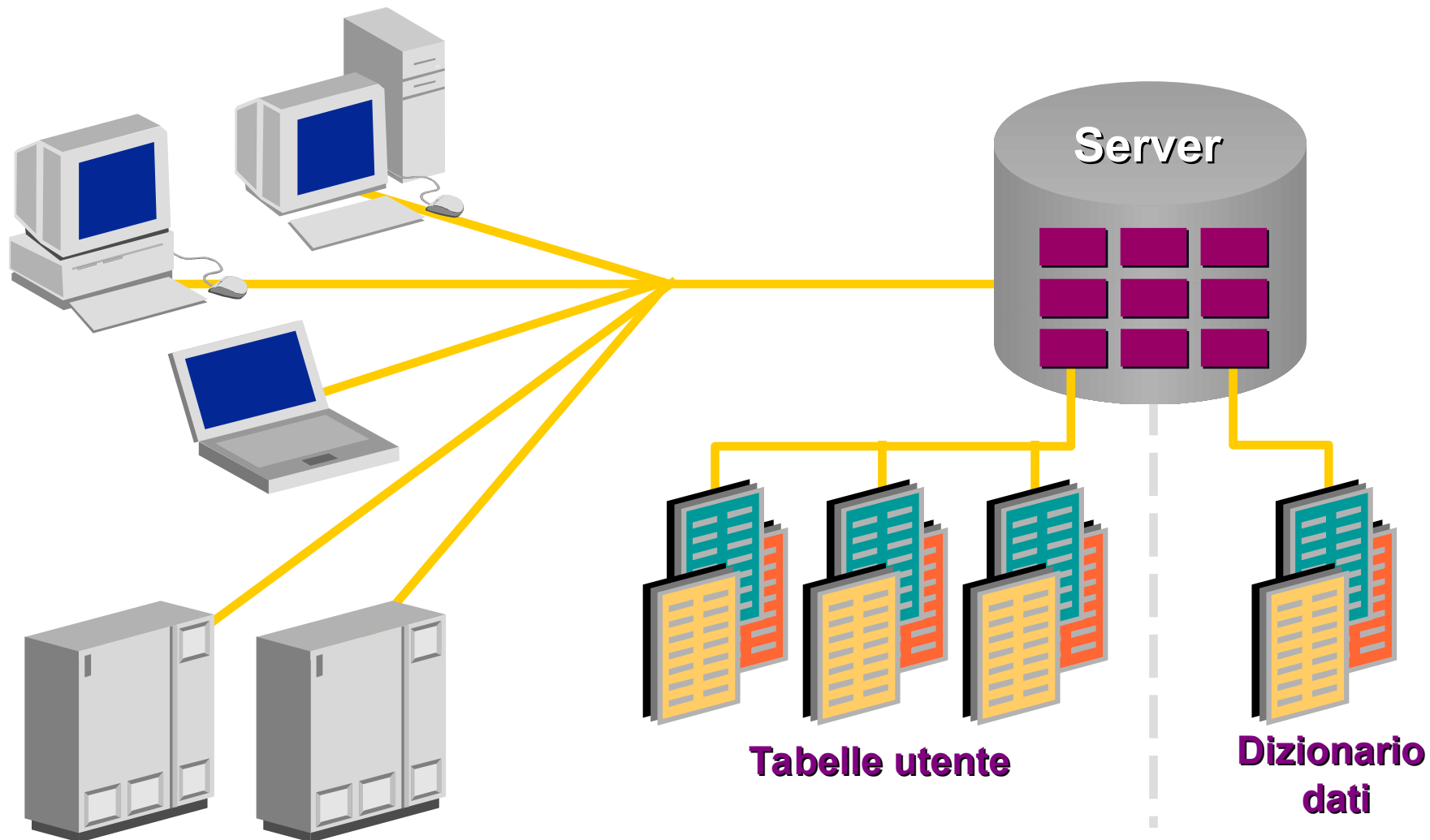


# Architettura di Postgres





# RDBMS



# Postgres - Connessione

Per connettersi al server è necessario fornire host, user e di seguito password

```
$ > psql -h localhost -U postgres
```

***host and user rappresentano:***

- Il server dove risiede Postgres;
- lo username di un utente che possiede un account sul server;

# Flusso di Esecuzione di una Query

- **Connessione:** Il client si connette al server (Processo Postmaster).
- **Forking:** Il Postmaster autentica il client e avvia un Processo Backend dedicato (Server Process).
- **Esecuzione:** Il client invia una query SQL al Processo Backend.
- **Elaborazione:** Il Backend esegue la query, interagendo con la Memoria Condivisa e i file su Disco (tramite il WAL).
- **Risultato:** Il Backend invia il risultato della query al client.
- **Disconnessione:** Al termine della sessione, il Processo Backend viene terminato.

L'architettura di Postgres si basa sulla separazione delle responsabilità tra un processo padre coordinatore (postmaster), processi figli esecutori (backend), e processi in background per la manutenzione.

## Architettura di Postgres

Il Data Dictionary in PostgreSQL (Dizionario Dati) è l'insieme delle tabelle e viste che contengono i metadati del database. In sostanza, è il database del database, dove sono memorizzate le informazioni sulla sua struttura, e non i dati effettivi degli utenti. PostgreSQL espone questi metadati principalmente attraverso due schemi di sistema: *pg\_catalog* e *information\_schema*.

# Architettura di Postgres

1. Il Catalogo di Sistema (pg\_catalog), è il cuore del Data Dictionary di PostgreSQL.

- Consiste in un insieme di tabelle di sistema sottostanti. Queste tabelle sono gestite internamente da PostgreSQL e contengono tutti i dettagli di implementazione.
- Contiene metadati molto dettagliati e specifici di PostgreSQL, come informazioni sui tipi di dati interni, sui processori, sulle statistiche e sui commenti (aggiunti tramite il comando COMMENT ON).
- Non è standard SQL. La struttura delle tabelle in pg\_catalog può cambiare tra le diverse versioni di PostgreSQL. Le query su questo schema sono quindi meno portabili su altri sistemi di database, ma permettono di accedere a tutti i dettagli unici di Postgres.
- Esempi di tabelle: pg\_class (tabelle, indici, viste), pg\_attribute (colonne), pg\_proc (funzioni e procedure).

# Architettura di Postgres

2. Lo Schema Informazioni (`information_schema`) fornisce una visione standardizzata dei metadati.

- Consiste in un insieme di viste che interrogano le tabelle sottostanti di `pg_catalog`.
- Offre una vista dei metadati più semplificata e limitata agli elementi definiti nello standard SQL. È focalizzato sulla struttura di base del database (tabelle, colonne, vincoli).
- È standard SQL (ANSI/ISO). Le query su queste viste sono più portabili su altri sistemi di gestione di database relazionali (RDBMS) che supportano lo standard `information_schema` (come MySQL, SQL Server, etc.).
- Esempi di viste: `information_schema.tables`, `information_schema.columns`, `information_schema.table_constraints`.

# Database Postgres

**1. Cluster di Database (Livello più Alto), il Cluster rappresenta l'intera installazione del server PostgreSQL. Contiene tutto ciò che riguarda una singola istanza in esecuzione:**

- **File di Configurazione:** file come postgresql.conf (parametri del server), pg\_hba.conf (autenticazione client) e pg\_ident.conf.
- **Database multipli:** La collezione di tutti i database gestiti dall'istanza.
- **WAL (Write-Ahead Log):** i log delle transazioni essenziali per la durabilità e il recupero.
- **Spazi Tabella (Tablespaces):** definizioni delle posizioni esterne sul file system dove possono essere archiviati i dati dei database.

## Database Postgres

**2. Database (Contenitore Logico), un ambiente isolato all'interno del Cluster a cui un utente si connette. Contiene:**

- **Schemi (Schemas):** L'organizzazione logica interna degli oggetti (vedi sotto).
- **Utenti e Privilegi:** Definizioni di quali ruoli (utenti) possono accedere al database e cosa possono fare.
- **Set di Caratteri e Locale:** Impostazioni regionali per l'ordinamento e la codifica dei dati.





# Database Postgres

3. Gli Schemi sono i contenitori logici che raggruppano gli oggetti all'interno di un Database, evitando collisioni di nomi e aiutando l'organizzazione. Contengono:

## Oggetti Utente:

Tabelle: Dove risiedono i dati.

Indici: Strutture per l'accesso veloce ai dati.

Viste (Views): Query salvate.

Funzioni e Procedure: Logica lato server (PL/pgSQL, ecc.).

Sequenze: Generatori di numeri interi univoci.

Tipi di Dati Utente: Tipi personalizzati.

## Schemi di Sistema (Data Dictionary):

`pg_catalog`: Le tabelle e viste interne che descrivono tutti gli oggetti e i metadati del sistema (ad esempio, `pg_class`, `pg_attribute`).

`information_schema`: Viste standardizzate (ANSI SQL) per l'interrogazione dei metadati.

## Database Postgres

4. A livello di file system (nella directory PGDATA), il contenuto è organizzato per garantire efficienza e recupero:

**Tabelle/Indici:** Ogni oggetto principale è un file separato (o più file chiamati "segmenti" se è grande). Il contenuto di questi file è diviso in pagine di 8 KB (l'unità di I/O).

**Dati MVCC:** Le righe non vengono sovrascritte, ma viene scritta una nuova versione. Le righe vecchie (non più visibili dalle transazioni) rimangono nei file finché non vengono rimosse dal processo di VACUUM.

**WAL Log:** I file pg\_wal contengono il registro sequenziale di tutte le modifiche, garantendo che le transazioni siano salvate in modo sicuro prima che i blocchi di dati vengano aggiornati in modo permanente.

**Sottodirectory base:** Contiene una sottodirectory per ogni database, identificata dal suo OID.

# Schema

In Postgres, uno schema è un Contenitore Logico che raggruppa logicamente oggetti correlati, come tabelle, viste, funzioni, indici e tipi di dati.

- **Nome Unico:** Gli oggetti devono avere un nome unico all'interno del loro schema. Questo significa che è possibile avere due tabelle con il medesimo nome all'interno dello stesso database, a patto che risiedano in schemi diversi (es. vendite.utenti e contabile.utenti).
- **Separazione degli Oggetti:** Gli schemi sono essenziali per evitare collisioni di nomi: Utile per le applicazioni che utilizzano librerie di terze parti o per sistemi complessi con molti moduli.
- **Gestione dei permessi:** È possibile definire diversi diritti di accesso e privilegi a livello di schema.
- **Organizzazione:** Permette di separare i dati, ad esempio quelli applicativi da quelli di configurazione o di reporting.

# Schema

Per accedere a un oggetto che risiede in uno schema, occorre qualificarlo usando il nome dello schema e il nome dell'oggetto, separati da un punto:

`nome_schema.nome_oggetto`

**Ogni database PostgreSQL viene creato con alcuni schemi predefiniti:**

- **public:** Lo schema predefinito dove vengono creati tutti gli oggetti utente se non si specifica uno schema diverso.
- **pg\_catalog:** Contiene tutte le tabelle di sistema (il Data Dictionary), con i metadati del database.
- **information\_schema:** Contiene le viste standard SQL sui metadati del database, utili per la portabilità.

# Schema

Il database usa la variabile di configurazione `search_path` per determinare in quali schemi cercare gli oggetti quando non è specificato un nome di schema.

Di default, il `search_path` include di solito `"$user", public`, dove `"$user"` è uno schema con lo stesso nome dell'utente connesso (se esiste).

Per impostare il `search_path` solo per la durata della connessione corrente (viene annullata al termine della sessione) del client `sql`:

```
SET search_path TO schema1, schema2, public;
```

**Per impostarlo a Livello di Utente (Permanente per Utente)**

```
ALTER ROLE nome_utente SET search_path TO schema_personale,  
public;
```

**A Livello di Database (Permanente per Database)**

```
ALTER DATABASE nome_database SET search_path TO vendite,  
report, public;
```

# Schema

Per creare un nuovo schema occorre avere il permesso CREATE nel database corrente.:

```
CREATE SCHEMA abc AUTHORIZATION MarioRossi;
```

Creare una tabella all'interno di uno schema:

```
CREATE TABLE abc.Dipendenti (  
    ID INT PRIMARY KEY,  
    Nome VARCHAR(100)  
);
```

Assegnare un permesso a un utente su uno schema:

```
GRANT SELECT ON ALL TABLES IN SCHEMA Vendite TO  
GiovanniBianchi;
```

```
ALTER DEFAULT PRIVILEGES IN SCHEMA Vendite GRANT  
SELECT ON TABLES TO GiovanniBianchi;
```

# Oggetti di database

## I tipi di oggetto più comuni includono:

Tabelle (Tables): Sono la base di un database, utilizzate per archiviare dati in formato righe e colonne.

Viste (Views): Tabelle “virtuali” basate su un set di risultati di una query. Le viste semplificano l'accesso a dati complessi e possono essere usate per la sicurezza, limitando l'accesso a determinate colonne o righe.

Stored Procedure: Blocchi di codice T-SQL precompilati e salvati nel database. Sono utili per eseguire operazioni complesse, migliorare le performance e aumentare la sicurezza.

Funzioni (Functions): Simili alle stored procedure, ma restituiscono un valore. Possono essere usate nelle query e non possono modificare i dati del database.

# Oggetti di database

Indici (Indexes): Strutture che migliorano le performance delle query, rendendo più veloce la ricerca dei dati nelle tabelle.

Trigger: Codice SQL che si attiva automaticamente in risposta a un evento (es. un'operazione di INSERT, UPDATE o DELETE).

Vincoli (Constraints): Regole applicate alle colonne di una tabella per limitare il tipo di dati che possono essere inseriti:

PRIMARY KEY: Garantisce l'univocità di ogni riga.

FOREIGN KEY: Assicura l'integrità referenziale tra le tabelle.

UNIQUE: Assicura che tutti i valori in una colonna siano diversi.

CHECK: Limita i valori che possono essere inseriti in una colonna.

NOT NULL: Forza una colonna a non accettare valori nulli.

Sinonimi (Synonyms): Alias per altri oggetti del database, utili per semplificare nomi lunghi o complessi.



# Authentication (Autenticazione)

L'autenticazione è il processo di verifica dell'identità di un utente che tenta di connettersi al database. È gestita principalmente dal file di configurazione `pg_hba.conf`.

Il file `pg_hba.conf` (Host-Based Authentication) è la regola d'oro per l'accesso e definisce chi può connettersi, da dove, a quale database e in che modo. Ogni riga del file è una regola che specifica:

Tipo, il tipo di connessione (locale, host TCP/IP).

Database, il nome del database a cui la regola si applica (es. `all`, `vendite`).

Utente, il nome del ruolo/utente a cui la regola si applica (es. `all`, `GiovanniBianchi`).

Indirizzo IP, l'indirizzo da cui proviene la connessione (es. `127.0.0.1/32`, `0.0.0.0/0`).

Metodo, il metodo di autenticazione da utilizzare.

# Authentication (Autenticazione)

PostgreSQL supporta diversi metodi di autenticazione:

**trust**: Consente la connessione senza alcuna password. Molto insicuro (usato solo per test locali o ambienti interni super protetti).

**password** / md5 / scram-sha-256: Richiede una password. scram-sha-256 è il metodo crittografato più sicuro e consigliato.

**peer** / ident: Utilizza l'identità dell'utente del sistema operativo (OS) locale per l'autenticazione, senza richiedere una password di database.

**ldap** / radius / cert: Metodi di autenticazione esterna per l'integrazione con servizi centralizzati o l'uso di certificati.

# Authorization (Autorizzazione)

L'autorizzazione è il processo che definisce ciò che un utente autenticato può fare all'interno del database. È gestita dal sistema di Ruoli e Privilegi (GRANT/REVOKE) di PostgreSQL.

Ruoli (Roles), un ruolo può essere considerato sia un utente (con la capacità di connettersi) sia un gruppo di utenti (con la capacità di ereditare i permessi).

Creazione: Si usa `CREATE ROLE nome_ruolo WITH LOGIN PASSWORD '...';`

Ereditarietà: Un ruolo può essere membro di un altro ruolo e ereditare tutti i suoi permessi (`GRANT gruppo TO utente`).

# Authorization (Autorizzazione)

I privilegi definiscono i diritti specifici su oggetti specifici del database (tabelle, schemi, funzioni, ecc.). A Livello di Oggetto (Oggetti Esistenti) si utilizza il comando GRANT per assegnare permessi su oggetti specifici:

Tabelle, SELECT, INSERT, UPDATE, DELETE, operazioni CRUD eseguibili sui dati.

Funzioni, EXECUTE diritto di eseguire una funzione.

Database, CONNECT, CREATE diritto di connettersi o creare schemi nel database.

Schema, USAGE, CREATE

...

# Creazione database e utente

Connettendosi come utente postgres (o superuser), eseguire il comando:

```
CREATE DATABASE corsodb;
```

Creare il ruolo (utente) con la possibilità di accedere al database (LOGIN) ed impostare una password.

```
CREATE ROLE mpsuser WITH  
    LOGIN  
    PASSWORD 'LaTuaPasswordSicura!'  
    NOSUPERUSER  
    NOCREATEDB;
```

# Creazione utente

Garantire Tutti i Privilegi a mpsuser su corsodb, richiede l'assegnazione di permessi su tre livelli: Database, Schema e Oggetti futuri.

Permessi sul Database: concedere all'utente il permesso di connettersi (CONNECT) e di creare schemi/oggetti all'interno del database (CREATE).

```
GRANT CONNECT, CREATE ON DATABASE corsodb TO  
mpsuser;
```

Permessi sullo Schema Esistente (public), per impostazione predefinita, tutte le tabelle vengono create nello schema public dove è necessario dare a mpsuser tutti i permessi:

```
GRANT ALL ON SCHEMA public TO mpsuser;
```

# Creazione utente

Permessi sugli Oggetti Attuali e Futuri (Tabelle, Sequenze, Funzioni), per assegnare tutti i privilegi di amministrazione sul database (creare, cancellare, modificare tabelle, e fare tutte le operazioni CRUD sui dati) agire sui permessi predefiniti (DEFAULT PRIVILEGES).

Per gli oggetti Esistenti (Tabelle, Sequenze, Funzioni, ...):

```
GRANT ALL ON ALL TABLES IN SCHEMA public TO  
mpuser;
```

```
GRANT ALL ON ALL SEQUENCES IN SCHEMA public TO  
mpuser;
```

```
GRANT ALL ON ALL FUNCTIONS IN SCHEMA public TO  
mpuser;
```

# Creazione utente

Per gli oggetti futuri è necessario impostare i permessi in modo che qualsiasi tabella, sequenza o funzione che mpsuser (o altri utenti) venga creata in futuro nello schema public sia completamente accessibile a mpsuser.

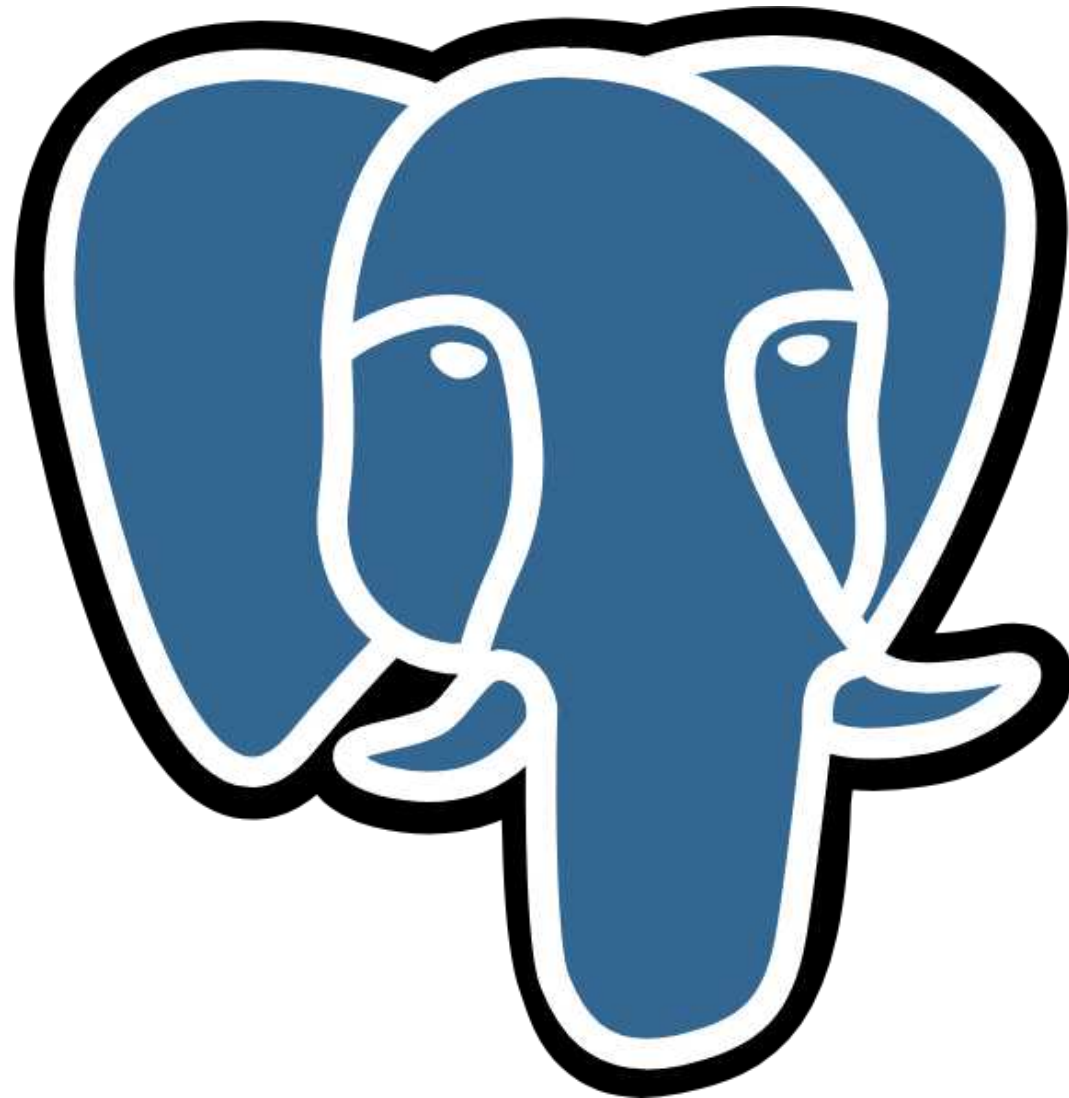
```
ALTER DEFAULT PRIVILEGES FOR ROLE postgres IN  
SCHEMA public GRANT ALL ON TABLES TO mpsuser;
```

```
ALTER DEFAULT PRIVILEGES FOR ROLE postgres IN  
SCHEMA public GRANT ALL ON SEQUENCES TO mpsuser;
```

```
ALTER DEFAULT PRIVILEGES FOR ROLE postgres IN  
SCHEMA public GRANT ALL ON FUNCTIONS TO mpsuser;
```

per vedere la procedura completa: [001-CreateUser.sql](#)





# Modello Relazionale (in breve)

Il modello relazionale, ideato da Edgar F. Codd nel 1970, è un modello logico per l'organizzazione e la gestione dei dati in un database.

La forza del modello relazionale sta nel combinare un concetto matematico formalizzato (la relazione) con un'idea intuitiva (la tabella).

# Modello Relazionale (in breve)

**Relazione (Tabella):** È la struttura fondamentale del modello relazionale. Una relazione è un insieme non ordinato di tuple (righe), e ogni tupla è un insieme non ordinato di attributi (colonne).

**Attributo (Colonna):** Rappresenta un tipo di dato specifico (es. Nome, Età, Codice Fiscale). Ogni attributo ha un dominio, che definisce l'insieme di valori che può assumere (es. il dominio dell'attributo "Età" è l'insieme dei numeri interi positivi).

**Tupla (Riga, Npla, Record, Row):** Rappresenta un'istanza o un record. Ogni tupla è unica all'interno della relazione.

**Chiavi:** Sono essenziali per stabilire le relazioni tra le tabelle e garantire l'integrità dei dati.

Chiave Primaria: Un attributo o un insieme di attributi che identifica in modo univoco ogni tupla in una relazione. Il suo valore non può essere nullo.

Chiave Esterna: Un attributo (o un insieme di attributi) in una tabella che fa riferimento alla chiave primaria di un'altra tabella. Serve a collegare le relazioni tra loro, creando un riferimento logico.

# Modello Relazionale (in breve)

**Indipendenza dei Dati:** il modello relazionale separa la visione logica dei dati (come l'utente li vede e li manipola) dalla loro implementazione fisica (come sono memorizzati). Ciò significa che è possibile modificare la struttura di memorizzazione senza alterare le applicazioni che accedono ai dati.

**Ridondanza Ridotta:** grazie alla possibilità di collegare le tabelle tramite chiavi, si evitano dati duplicati, migliorando l'integrità e riducendo lo spazio di archiviazione.

**Flessibilità e Semplicità:** i dati vengono gestiti con operatori basati sull'algebra relazionale (come la selezione, la proiezione e il join), che sono implementati in linguaggi come l'SQL (Structured Query Language).

# Modello Relazionale (in breve)

**Per garantire la coerenza dei dati, il modello relazionale si basa su regole precise:**

I valori degli attributi devono essere atomici: ovvero non possono essere ulteriormente scomponibili.

Ogni riga deve essere unica: non ci possono essere due righe identiche in una tabella.

L'ordine delle righe e delle colonne è irrilevante: non influisce sulla validità dei dati.

I nomi degli attributi devono essere univoci all'interno della stessa relazione.

Il valore della chiave primaria non può essere nullo.

Il valore di chiave esterna deve corrispondere a un valore di chiave primaria esistente nella tabella a cui fa riferimento, oppure essere nullo.

# Modello Relazionale (in breve)

**Le forme normali sono un insieme di regole che servono a ridurre la ridondanza dei dati e a migliorare l'integrità in un database relazionale.**

## **Prima Forma Normale (1FN)**

**La 1FN è il requisito di base e ogni database relazionale deve rispettarla. Una tabella è in 1FN se:**

Ogni colonna contiene valori atomici: Non ci sono valori multipli o liste all'interno di una singola cella.

Non ci sono gruppi di colonne ripetitive: Le colonne non devono ripetersi (es. Telefono1, Telefono2, Telefono3). Se un'entità ha più attributi simili, questi dovrebbero essere spostati in una tabella separata.

# Modello Relazionale (in breve)

## Seconda Forma Normale (2FN)

Una tabella è in 2FN se è già in 1FN e ogni attributo non chiave dipende completamente dalla chiave primaria intera. Questo si applica solo a tabelle con chiavi primarie composte (formate da più colonne).

L'anomalia da evitare è la dipendenza parziale. Se un attributo non chiave dipende solo da una parte della chiave primaria composta, la tabella non è in 2FN.

## Esempio di tabella non in 2FN:

Ordine (ID\_Ordine, ID\_Prodotto, Quantità, Nome\_Prodotto)

Dove Nome\_Prodotto dipende solo da ID\_Prodotto, non dalla chiave composta (ID\_Ordine, ID\_Prodotto).

## Correzione: split della tabella

Ordine\_Dettagli (ID\_Ordine, ID\_Prodotto, Quantità)

Prodotti (ID\_Prodotto, Nome\_Prodotto).

# Modello Relazionale (in breve)

## Terza Forma Normale (3FN)

Una tabella è in 3FN se è in 2FN e non ha dipendenze transitive. Si ha una dipendenza transitiva se un attributo non chiave dipende da un altro attributo non chiave.

In altre parole l'anomalia da evitare è: se A determina B e B determina C, allora C ha una dipendenza transitiva da A.

## Esempio di tabella non in 3FN:

Dipendente (ID\_Dipendente, Nome, ID\_Reparto, Nome\_Reparto)

Dove Nome\_Reparto dipende da ID\_Reparto, che a sua volta dipende da ID\_Dipendente.  
Nome\_Reparto ha una dipendenza transitiva.

## Correzione split della tabella:

Dipendenti (ID\_Dipendente, Nome, ID\_Reparto)

Reparti (ID\_Reparto, Nome\_Reparto).



# Modello Relazionale (in breve)

## Forma Normale di Boyce-Codd (BCNF)

La BCNF è una versione più rigorosa della 3FN. Una tabella è in BCNF se, per ogni dipendenza funzionale  $X \rightarrow Y$  (dove  $X$  determina  $Y$ ),  $X$  è una superchiave. Una superchiave è un insieme di attributi che identifica in modo univoco una tupla.

La BCNF elimina alcune anomalie che la 3FN potrebbe non coprire, specialmente in casi complessi con più chiavi candidate. Spesso 3FN è sufficiente, ma BCNF è la norma per database ben progettati.

L'importanza delle forme normali risiede nel fatto che garantiscono un design del database logico, coerente e senza ridondanze inutili, rendendo più efficiente la gestione dei dati nel tempo.

# Modello Relazionale (in breve)

## Esempio: Tabella "Istruttori-Corsi"

Immaginiamo una tabella che gestisce l'assegnazione di istruttori a corsi specifici in una scuola.

ID\_Studente , ID\_Istruttore, Nome\_Istruttore, Nome\_Corso

La chiave primaria è (ID\_Studente, ID\_Istruttore). Questo significa che ogni riga è identificata in modo univoco dalla combinazione di un determinato studente e un istruttore.

### Dipendenze Funzionali:

{ID\_Studente, ID\_Istruttore} -> {Nome\_Corso}: La combinazione di uno studente e un istruttore determina il corso che stanno seguendo insieme.

{ID\_Studente, ID\_Istruttore} -> {Nome\_Istruttore}: La combinazione di uno studente e un istruttore determina il nome dell'istruttore.

{Nome\_Corso} -> {ID\_Istruttore}: Un corso è tenuto da un solo istruttore.

{ID\_Istruttore} -> {Nome\_Istruttore}: Ogni istruttore ha un nome.

# Modello Relazionale (in breve)

È in 3FN: perchè non ci sono dipendenze transitive ovvero nessun attributo non-chiave dipende da un altro attributo non-chiave, ma non è in BCNF:

La dipendenza Nome\_Corso  $\rightarrow$  ID\_Istruttore non rispetta la regola BCNF.

La regola dice che per ogni dipendenza  $X \rightarrow Y$ , X deve essere una superchiave.

In questo caso, Nome\_Corso non è una superchiave per la tabella, ma determina un'altra colonna (ID\_Istruttore).

Questo crea una potenziale ridondanza e anomalia. Ad esempio, se due studenti diversi seguono lo stesso corso, il Nome\_Istruttore e ID\_Istruttore verranno ripetuti.

# Modello Relazionale (in breve)

Per portare la tabella in BCNF, è necessario eliminare la dipendenza che causa il problema, creando una nuova tabella per l'informazione ridondante.

**Tabella 1: Istruttori\_Corsi** che gestisce la relazione tra corsi e istruttori.

ID\_Corso (Chiave Primaria), ID\_Istruttore (Chiave Esterna a Istruttori)

**Tabella 2: Iscrizioni** che gestisce le iscrizioni degli studenti ai corsi.

ID\_Studente, ID\_Corso

La relazione tra istruttori e corsi è ora gestita in una tabella separata e ogni dipendenza funzionale è rispettata.

# SQL

SQL, acronimo di Structured Query Language, è un linguaggio ""standard"" per la gestione e l'interazione con i database relazionali. Non è un linguaggio di programmazione generico, ma un linguaggio dichiarativo, il che significa che l'utente specifica "cosa" vuole fare, e il database si occupa di determinare "come" farlo.

È il linguaggio universale utilizzato per creare, interrogare, modificare e controllare la struttura e i dati all'interno di un database. Viene utilizzato per definire schemi di database (CREATE, ALTER), manipolare i dati (INSERT, UPDATE, DELETE), recuperare informazioni (SELECT) e gestire i permessi degli utenti (GRANT, REVOKE).

La sua forza risiede nella sua capacità di operare su insiemi di dati, rendendolo estremamente efficiente per l'elaborazione di grandi volumi di informazioni.

È alla base di tutti i principali sistemi di gestione di database relazionali (RDBMS) come SQL Server, MySQL, PostgreSQL e Oracle...

# PL/pgSQL

Il PL/pgSQL è l'estensione proprietaria del linguaggio SQL di Postgres SQL.

A differenza dell'SQL standard, che si concentra sulle query dichiarative, il PL/pgSQL aggiunge funzionalità di Linguaggio procedurale costruito sopra SQL, possiede variabili, condizioni, loop, eccezioni.

Viene usato per stored procedures, functions, triggers ed è simile a PL/SQL di Oracle.

**L'SQL (Structured Query Language) può essere suddiviso in diversi sotto-linguaggi, ognuno con uno scopo specifico.**

**1. Data Definition Language (DDL) che viene usato per definire la struttura del database e dei suoi oggetti, non manipola i dati, ma gestisce lo schema.**

CREATE: Crea oggetti nel database (es. CREATE TABLE, CREATE VIEW, CREATE INDEX).

ALTER: Modifica la struttura di un oggetto esistente (es. ALTER TABLE per aggiungere o rimuovere una colonna).

DROP: Cancella un oggetto dal database (es. DROP TABLE, DROP VIEW).

## **2. Data Manipulation Language (DML) che è usato per manipolare i dati all'interno degli oggetti del database, come tabelle e viste.**

INSERT: Inserisce nuove righe in una tabella.

UPDATE: Modifica i valori delle righe esistenti.

DELETE: Rimuove righe da una tabella.

SELECT: Recupera i dati dal database in base a criteri specificati.



## 3. Data Control Language che gestisce i permessi e il controllo di accesso agli oggetti del database.

GRANT: Concede permessi agli utenti (es. GRANT SELECT su una tabella).

REVOKE: Revoca i permessi precedentemente concessi.

DENY: Nega esplicitamente i permessi, impedendo a un utente di ereditarli da un ruolo.

**4. Transaction Control Language che controlla le transazioni, ovvero sequenze di istruzioni DML eseguite come una singola unità logica. Questo assicura che tutte le modifiche avvengano con successo o che nessuna venga applicata.**

COMMIT: Salva in modo permanente tutte le modifiche fatte in una transazione.

ROLLBACK: Annulla tutte le modifiche fatte nella transazione, ripristinando lo stato precedente.

SAVEPOINT: Imposta un punto di salvataggio all'interno di una transazione, permettendo di fare un rollback parziale.

# Istruzioni SQL

<b>SELECT</b>	<b>Estrazione Dati</b>
---------------	------------------------

<b>INSERT</b> <b>UPDATE</b> <b>DELETE</b>	<b>Data manipulation language (DML)</b>
---	---

<b>CREATE ALTER</b> <b>DROP RENAME</b> <b>TRUNCATE</b>	<b>Data definition language (DDL)</b>
--	---------------------------------------

<b>COMMIT</b> <b>ROLLBACK</b> <b>SAVEPOINT</b>	<b>controllo Transazioni</b>
--	------------------------------

<b>GRANT</b> <b>REVOKE</b>	<b>Data control language (DCL)</b>
-------------------------------	------------------------------------

# Creazione di un database

Si creano le tabelle:

```
CREATE TABLE IF NOT EXISTS province (  
    id int NOT NULL,  
    provincia varchar(128) NOT NULL,  
    sigla varchar(5) NOT NULL,  
    regione varchar(128) DEFAULT NULL,  
    PRIMARY KEY (id)  
);
```

# Keys

**KEY è sinonimo di INDEX. Gli attributi PRIMARY KEY possono essere specificati con la parola chiave KEY quando vengono definiti.**

**Una PRIMARY KEY è una KEY univoca e le colonne sono definite NOT NULL. Una tabella può avere solo una PRIMARY KEY.**

# Primary Key

```
drop table if exists student;
```

```
CREATE TABLE student (  
    name varchar(100) NOT NULL,  
    sex char(1) NOT NULL,  
    student_id integer,  
    PRIMARY KEY (student_id)  
);
```


Una PRIMARY KEY è una KEY univoca e le colonne sono definite NOT NULL. Una tabella può avere solo una PRIMARY KEY.

# Primary Key e auto increment



## Metodo Raccomandato (ISO)

```
CREATE TABLE clienti (  
    id INTEGER GENERATED [BY DEFAULT | ALWAYS] AS  
    IDENTITY PRIMARY KEY,  
    ...  
);
```

 \*\*Metodo Tradizionale (SERIAL)\*\*

```
CREATE TABLE clienti (  
    id SERIAL PRIMARY KEY,  
    ...  
);
```

# Null

Il valore NULL per un campo assume il significato di: *mancante* oppure *sconosciuto* ed è trattato diversamente dagli altri valori.

Per controllare il valore NULL non possono essere usati gli operatori di confronto quali =, <, o <> occorre utilizzare l'operatore *is* null.

Quando si utilizza ORDER BY, i valori NULL sono inseriti all'inizio con ASC ed alla fine con ORDER BY ... DESC.



# INSERT

## Standard

```
INSERT INTO TABLE_NAME  
(column1, column2, column3, .....columnN)  
VALUES (value1, value2, value3, .... valueN);
```

## Simplified

```
INSERT INTO TABLE_NAME  
VALUES (value1, value2, value3, .... valueN);
```

## From Select

```
INSERT INTO table_name(column1, column2,... )  
SELECT expression1, expression2, ...
```

# UPDATE







```
UPDATE table_name  
SET column1 = value1,  
column2 = value2....,  
columnN = valueN  
WHERE  
condition;
```

# DELETE

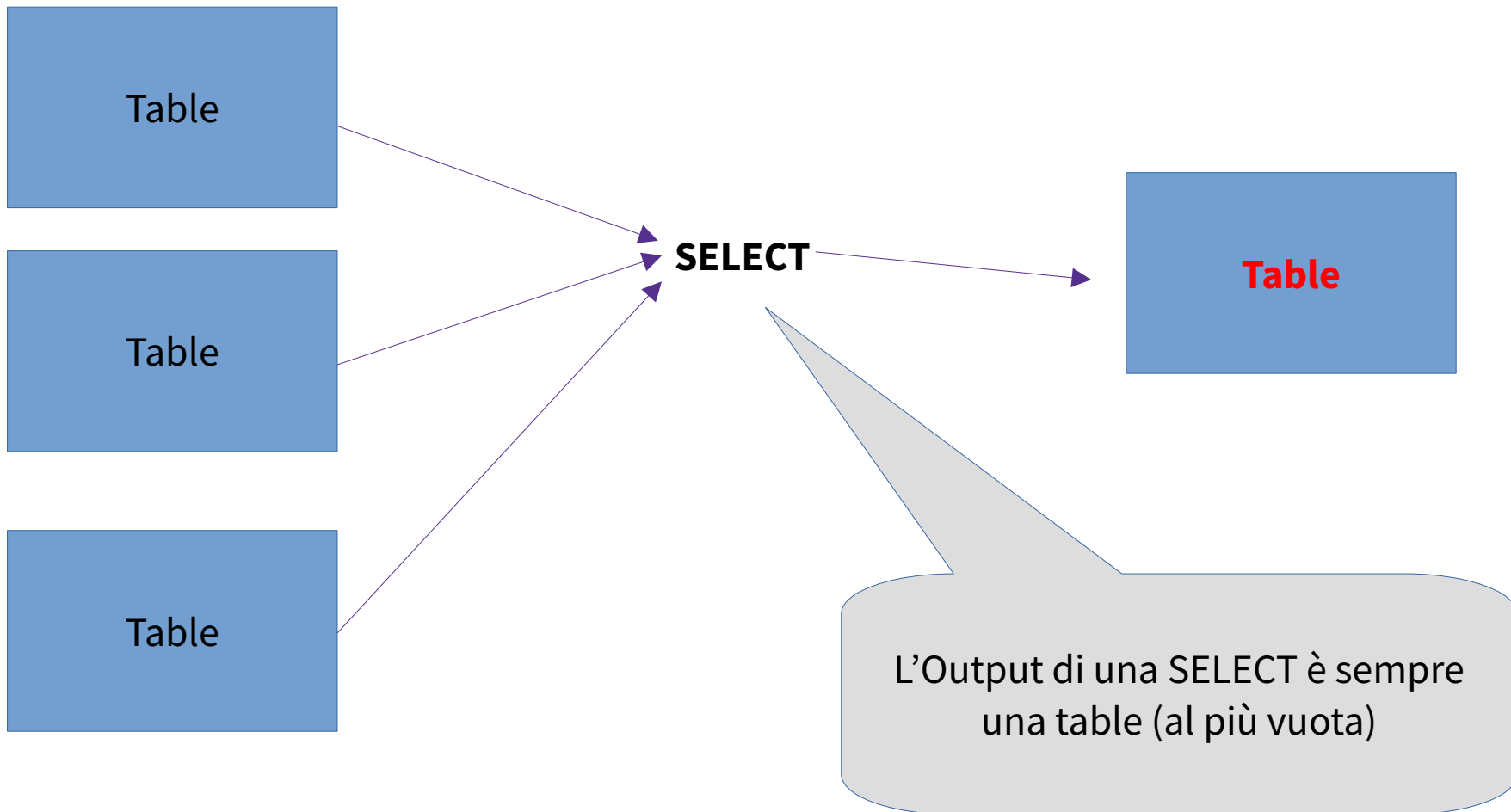
```
DELETE FROM table_name  
WHERE [condition];
```

# SELECT

```
select id, nome, sigla_automobilistica as  
"Targa" from province;
```

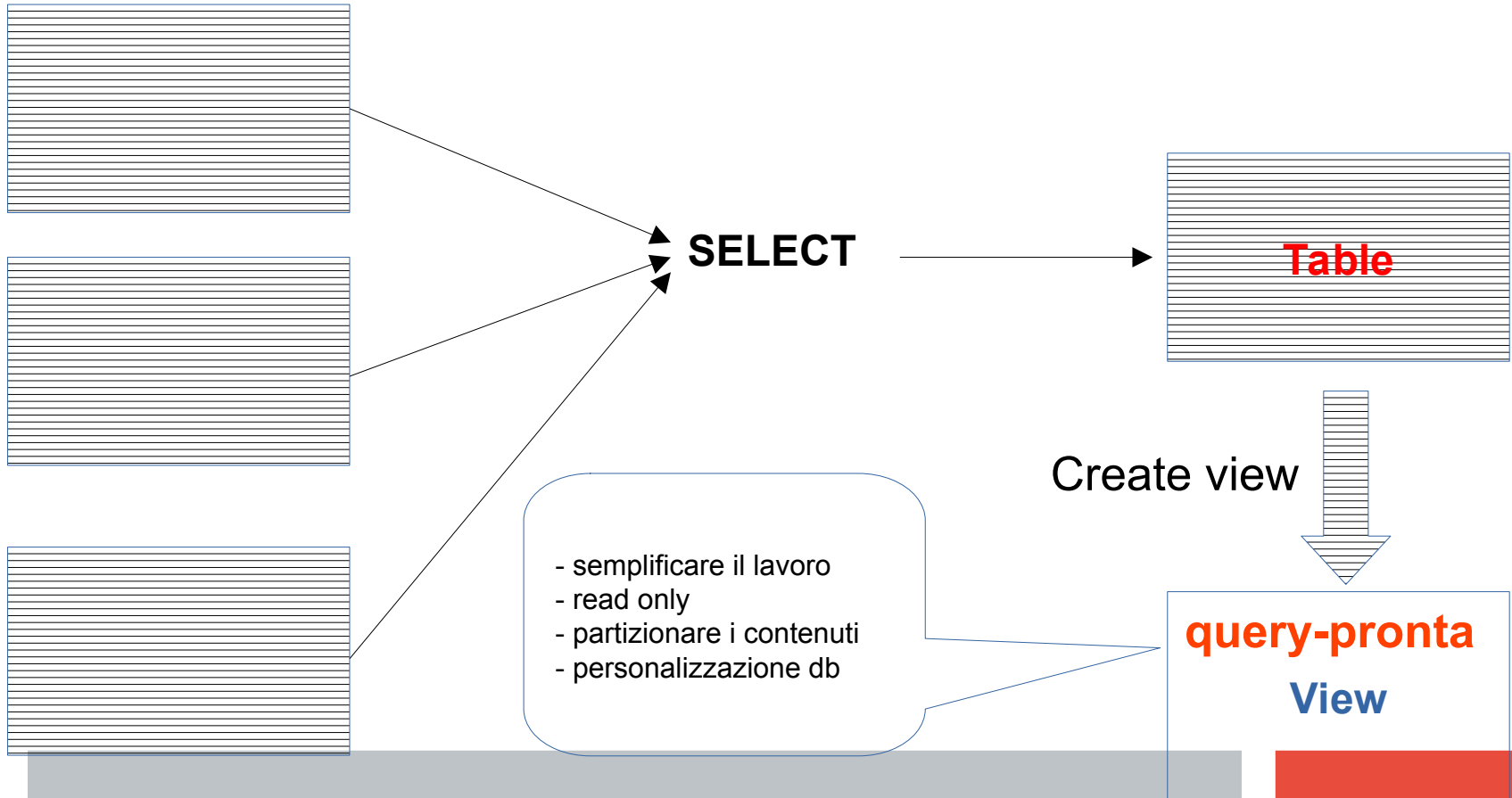
	 id 	 nome 	 Targa 
1	1	Torino	TO
2	2	Vercelli	VC
3	3	Novara	NO
4	4	Cuneo	CN
5	5	Asti	AT
6	6	Alessandria	AL
7	7	Valle d'Aosta/Vallée d'Aoste	AO
8	8	Imperia	IM

# SELECT

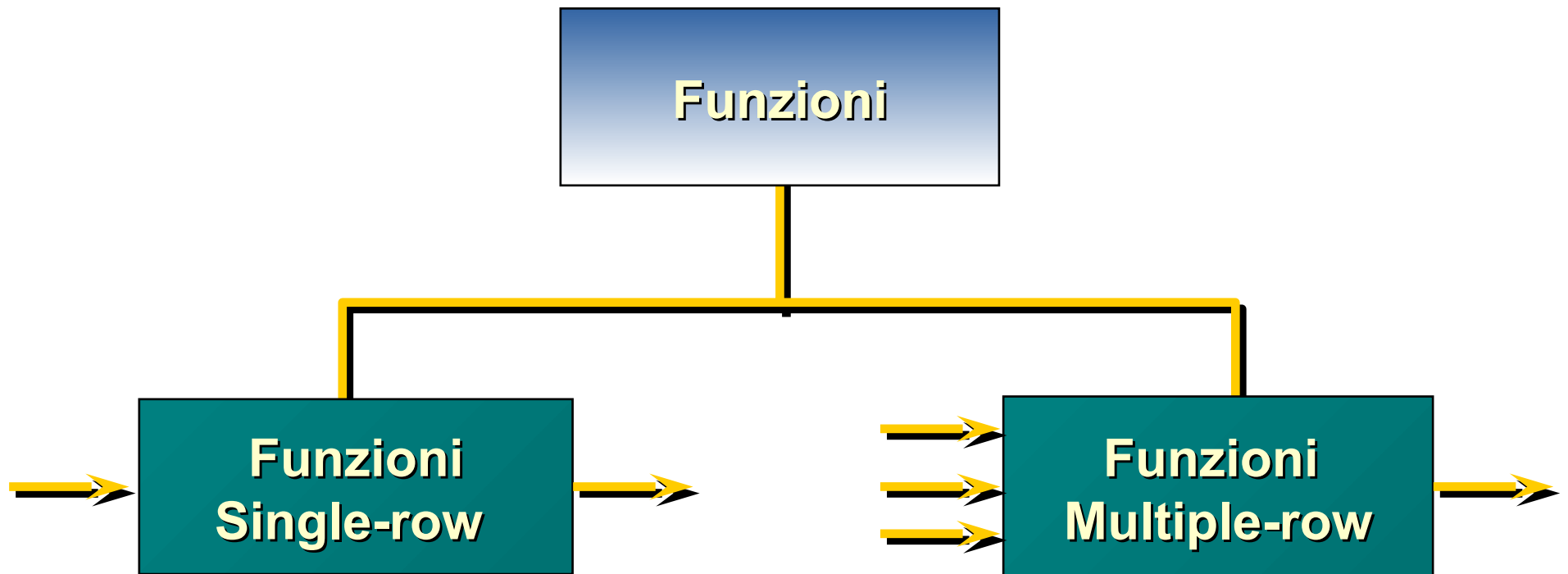


# Views

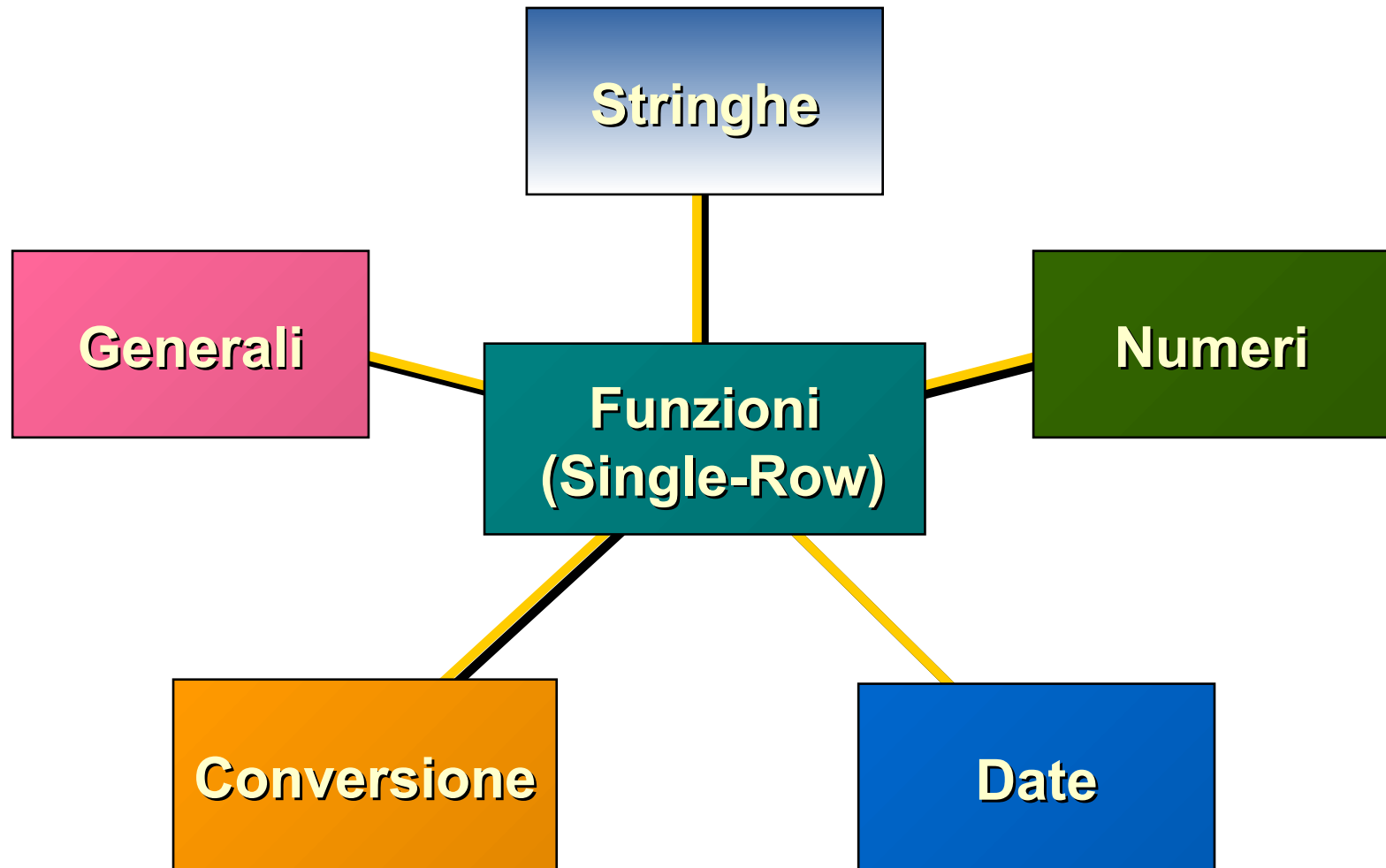
redazione(script) → inviare al DB → parsing → compilazione → ottimizzazione (query plan) (**query-pronta**) → esecuzione → tabella



# Funzioni SQL

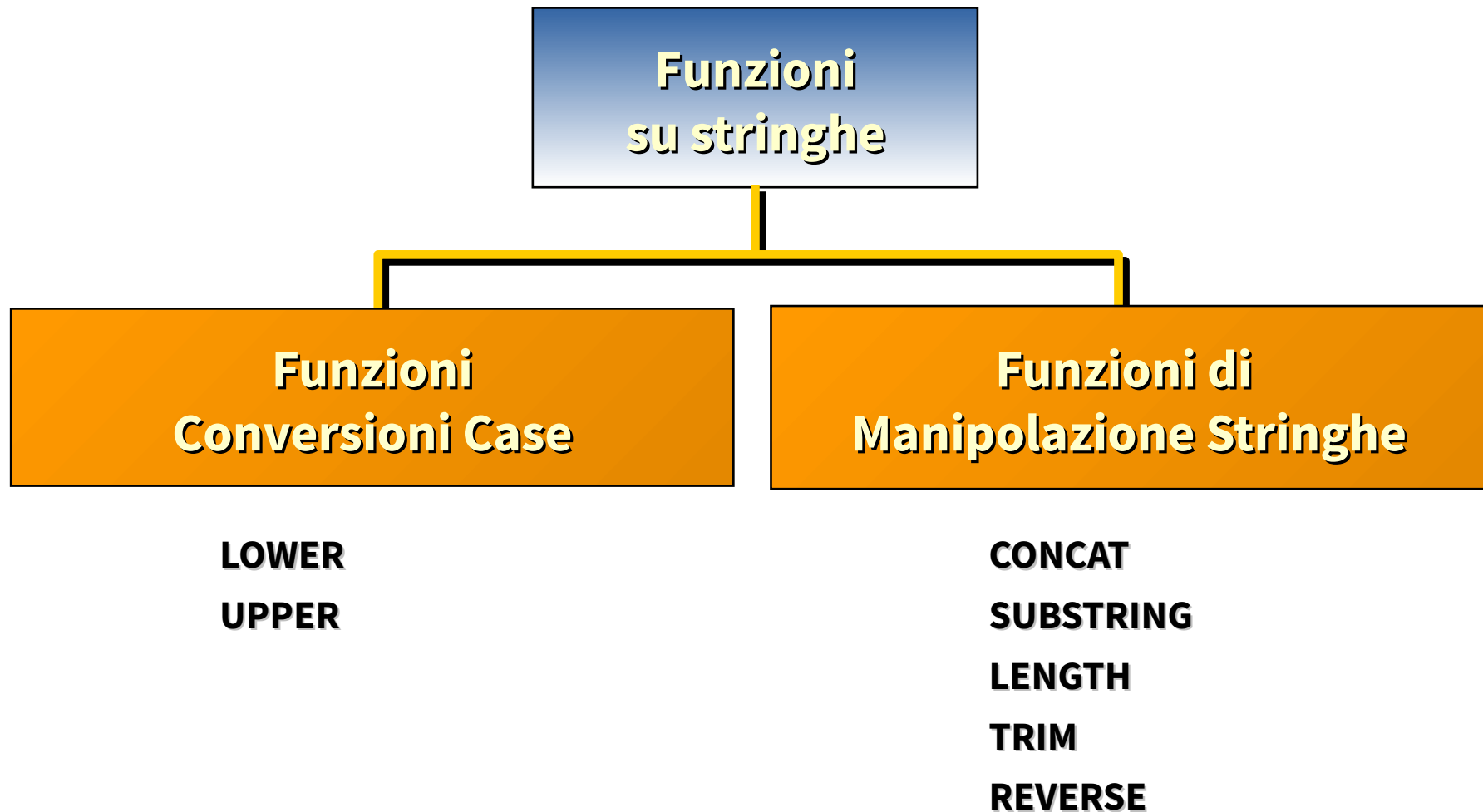


# Funzioni





# Funzioni su Stringhe



# Function

```
CREATE OR REPLACE FUNCTION generate_random_date( min_date DATE, max_date DATE )  
  
RETURNS DATE AS $$  
  
DECLARE  
  
    date_range INTEGER; -- Variabile per memorizzare l'intervallo totale di giorni  
  
    random_days INTEGER; -- Variabile per memorizzare i giorni casuali da aggiungere  
  
BEGIN  
  
    IF min_date > max_date THEN -- 1. Verifica che la data minima non sia maggiore della massima  
  
        RAISE EXCEPTION 'La data minima (%) maggiore della data massima (%)', min_date, max_date;  
  
    END IF;  
  
    date_range := max_date - min_date; -- 2. Calcola il numero totale di giorni  
  
    -- 3. Genera un numero casuale di giorni all'interno dell'intervallo (da 0 a date_range)  
  
    random_days := FLOOR(RANDOM() * (date_range + 1)); -- +1 per includere max_date  
  
    RETURN min_date + random_days; -- 4. Restituisce la data di inizio più i giorni casuali  
  
END;  
  
$$ LANGUAGE plpgsql IMMUTABLE; -- perché la funzione restituisce sempre lo stesso risultato per gli  
stessi argomenti
```

# Trigger

Un trigger SQL trigger è costituito da un gruppo di statement SQL memorizzati nel database.

Un trigger SQL è eseguito o fired quando si scatena un evento che è associato ad una tabella (insert, update, delete).

I trigger SQL vengono invocati e eseguiti senza che il client (l'applicazione) ne sia a conoscenza, quindi può essere difficoltoso capire cosa succede nel database layer e può essere complesso effettuarne il debugging.

# Trigger

I trigger vengono generalmente usati per gestire vincoli di integrità, calcolare dati derivati, gestire eccezioni, ecc.

Mediante i trigger è il sistema RDBMS che si fa carico di garantire la consistenza del DB, sollevando quindi le applicazioni da tale onere.

Un trigger può attivarsi prima (BEFORE) o dopo (AFTER) l'evento scatenante.

gli “before trigger” vengono usati per “condizionare” l'esito dell'operazione oppure per bloccarla segnalando errore

Gli “after trigger” servono a “reagire” alla modifica del DB mediante opportune azioni

# Trigger

**Nel caso di eventi che coinvolgano più tuple della tabella target, un trigger può essere attivato come:**

Row trigger: per ognuna di queste tuple (FOR EACH ROW)

Statement trigger: solo una volta per la data istruzione (FOR EACH STATEMENT).

# Trigger

In un trigger si ha frequentemente necessità di fare riferimento allo stato del DB prima e/o dopo l'evento.

A tale scopo si possono usare 2 variabili e 2 tabelle di transizione OLD e NEW.

**Non tutti i riferimenti hanno senso per tutti i tipi di eventi: OLD non ha senso in caso di INSERT, NEW non ha senso in caso di DELETE.**

**OLD:** valore della tupla prima della modifica, una tabella virtuale che contiene tutte le tuple con i valori prima della modifica

**NEW:** valore della tupla dopo la modifica, una tabella virtuale che contiene tutte le tuple modificate, con i valori dopo la modifica

# Trigger

**BEFORE INSERT:** attivato prima dell'inserimento dei dati nella table.

**AFTER INSERT:** attivato dopo l'inserimento dei dati nella table.

**BEFORE UPDATE:** attivato prima dell'aggiornamento

**AFTER UPDATE:** attivato dopo l'aggiornamento

**BEFORE DELETE:** attivato prima della cancellazione

**AFTER DELETE:** attivato dopo la cancellazione

# Trigger

## Data la tabella

```
CREATE TABLE prodotti (  
    id SERIAL PRIMARY KEY,  
    nome VARCHAR(100),  
    prezzo NUMERIC,  
    data_creazione TIMESTAMP DEFAULT NOW(),  
    data_ultima_modifica TIMESTAMP  
);
```

Il primo passo consiste nel creare la Funzione Trigger (Logica) dove la funzione trigger è una speciale funzione PostgreSQL che non accetta argomenti (anche se i dati della riga sono disponibili tramite variabili speciali) e deve restituire il tipo di dato TRIGGER.

La funzione contiene la logica che verrà eseguita quando il trigger si attiva.

Supponiamo di dover aggiornare automaticamente una colonna `data_ultima_modifica` al momento di un'UPDATE.



# Trigger

```
CREATE OR REPLACE FUNCTION aggiorna_data_modifica()  
RETURNS TRIGGER AS $$  
  
BEGIN  
    -- NEW è una variabile speciale che rappresenta la nuova riga solo per eventi INSERT o UPDATE  
    NEW.data_ultima_modifica = NOW();  
    RETURN NEW; -- Restituisce la riga modificata (NEW)  
  
END;  
  
$$ LANGUAGE plpgsql;
```

**Dove sono impiegabili le seguenti variabili speciali:**

**NEW, Contiene la nuova riga per le operazioni INSERT o UPDATE.**

**OLD, Contiene la vecchia riga per le operazioni UPDATE o DELETE.**

**TG\_OP, Contiene la stringa dell'operazione che ha attivato il trigger ('INSERT', 'UPDATE', 'DELETE').**

# Trigger

Dopo aver definito la funzione, occorre utilizzare il comando **CREATE TRIGGER** per associarla a una tabella e a un evento.

```
CREATE TRIGGER nome_trigger
{ BEFORE | AFTER } { evento [ OR evento ... ] }
ON nome_tabella
FOR EACH { ROW | STATEMENT }
EXECUTE FUNCTION nome_funzione_trigger();
```

**BEFORE**, Eseguire la funzione prima che l'operazione di **UPDATE** sia completata. (Utile per modificare i dati prima del salvataggio).

Evento **UPDATE**, attivare il trigger solo quando viene eseguito un comando **UPDATE**.

**ON** prodotti, la tabella su cui si applica la regola.

Livello **FOR EACH ROW**, eseguire la funzione una volta per ogni riga modificata dall'operazione. (L'alternativa **FOR EACH STATEMENT** esegue la funzione una volta per l'intera query).

Azione: **EXECUTE FUNCTION ...** Chiama la funzione trigger definita in precedenza

# Trigger

Dopo aver definito la funzione, occorre utilizzare il comando **CREATE TRIGGER** per associarla a una tabella e a un evento.

```
CREATE TRIGGER trigger_data_modifica_prodotto  
BEFORE UPDATE ON prodotti  
FOR EACH ROW  
EXECUTE FUNCTION aggiorna_data_modifica();
```

**BEFORE**, Eseguire la funzione prima che l'operazione di **UPDATE** sia completata. (Utile per modificare i dati prima del salvataggio).

Evento **UPDATE**, attivare il trigger solo quando viene eseguito un comando **UPDATE**.

**ON prodotti**, la tabella su cui si applica la regola.

Livello **FOR EACH ROW**, eseguire la funzione una volta per ogni riga modificata dall'operazione. (L'alternativa **FOR EACH STATEMENT** esegue la funzione una volta per l'intera query).

Azione: **EXECUTE FUNCTION ...** Chiama la funzione trigger definita in precedenza

