Appunti di Basi di dati

Giacomo Simonetto

Primo semestre 2025-26

Sommario

Appunti del corso di Basi di dati della facoltà di Ingegneria Informatica dell'Università di Padova.

Indice

1 Introduzione

Dato vs informazione

- il dato è la registrazione di un veneto effettuata attraverso dei simboli e salvata su un supporto, in un dato puro non c'è nessun contesto per interpretarlo
- l'informazione è il dato con il contesto associato, ovvero è un dato interpretabile

Dati strutturati vs non strutturati

- i dati strutturati sono dati che possono facilmente ed efficientemente essere organizzati in tabelle, sono ad esempio numeri, date, stringhe, ecc., sono facili da memorizzare ed analizzare
- i dati non strutturati sono dati che non possono essere facilmente organizzati in tabelle, sono ad esempio immagini, video, audio, documenti di testo, ecc., richiedono più spazio per poter essere memorizzati e serve un processo di strutturazione per poterli analizzare

Basi di dati

Le basi di dati o database sono collezioni organizzate e permanenti di dati strutturati coerenti che permettono di memorizzare, gestire e recuperare informazioni (dati e contesto) in modo efficiente. Vengono realizzati su misura per rispondere alle esigenze di specifiche applicazioni e utenti. Modellano un piccolo aspetto del mondo reale, ovvero rappresentano un miniworld. I database sono:

- centralizzati (i dati sono memorizzati su un unico server) o distribuiti (i dati sono memorizzati su più server collegati in rete)
- persistenti: i dati rimangono memorizzati anche quando il sistema viene spegnimento
- condivisi: più utenti e applicazioni possono accedere e manipolare i dati contemporaneamente

DataBase Managment System (DBMS)

Il DBMS sono software che permetton di gestire e interagire con i dati all'interno di un database. Garantiscono:

- integrità: mantengono i dati corretti e coerenti secondo le regole definite
- sicurezza: proteggono i dati da accessi non autorizzati
- efficienza: ottimizzano l'archiviazione e il recupero dei dati

Oltre ai dati veri e propri, i DBMS memorizzano anche metadati, ovvero informazioni testuali su come interpretare e organizzare i dati.

2 Modello ER

3 Modello relazionale

4 Algebra relazionale

5 Linguaggio SQL

5.1 Elementi fondamentali al linguaggio SQL

Tabella

Una tabella è una collezione di zero o più colonne ordinate e zero o più righe non ordinate. Non è possibile avere tabelle con lo stesso nome e non è possibile avere righe duplicate. Ogni colonna può memorizzare solo uno specifico tipo di dato. Le tabelle vengono utilizzate nei database relazionali per memorizzare e organizzare i dati in modo strutturato. Esistono due tipi di tabelle in SQL:

- **tabelle base**: tabelle che rispettano i vincoli e contengono i dati memorizzati nel database
- **tabelle derivate**: tabelle ottenute da una query e non necessariamente rispettano i vincoli

Query

La query sono operazioni compiute su una o più tabelle per recuperare, inserire, aggiornare o eliminare dati. In genere possono restituire una tabella derivata come risultato. Le query terminano con un "`;`".

Schema e istanza

- **lo schema** definisce come i dati sono organizzati nel database (tabelle, colonne, tipi di dato delle tabelle, relazioni tra tabelle, vincoli, ecc.). Uno schema non contiene dati e molto difficilmente cambia nel tempo. Gli schemi in SQL sono raggiuppati in "catalog" che in SQL sono anche detti "database"
- **l'istanza** è il contenuto effettivo del database in un dato momento, si riferisce ai dati memorizzati e cambia frequentemente

5.2 Linguaggio SQL

Introduzione

Il linguaggio SQL (Structured Query Language) è un linguaggio standard per la gestione e manipolazione di database relazionali attraverso i DBMS. SQL è un linguaggio dichiarativo, ovvero l'utente specifica cosa vuole ottenere senza dover specificare come esattamente eseguire l'operazione. Implementa lo schema relazionale e l'algebra relazionale. È composto da due sotto-linguaggi:

- **DDL (Data Definition Language)**: per definire e modificare la struttura del database
- **DML (Data Manipulation Language)**: per manipolare i dati all'interno delle tabelle

Esistono vari livelli di implementazione di SQL in base alle funzioni offerte:

- **entry SQL**: simile a SQL-89, include le funzionalità di base per la gestione dei dati
- **intermediate SQL**: con funzionalità più complesse implementate nella maggior parte dei DBMS
- **full SQL**: include tutte le funzionalità definite dallo standard SQL che non è detto siano implementate di base in qualsiasi DBMS

Sintassi di SQL

- **alfabeto**: i simboli validi per scrivere query SQL sono lettere (A-Z, a-z), numeri (0-9) e alcuni caratteri speciali; per scrivere le lettere accentate si usano sequenze di escape (`U&'d\0061 = a`)
- **token**: sono le unità lessicali delle query in SQL, si dividono in:
  - **identificatori**: nomi di oggetti del database (tabelle, colonne, vincoli, ecc.), sono case-insensitive e possono essere regular (iniziano con una lettera e contengono lettere, numeri e _) o delimited (racchiusi tra doppi apici, diventano case-sensitive e possono contenere qualsiasi carattere, es. `"Column-1&2"`)
  - **keywords**: parole riservate di SQL (`SELECT`, `FROM`, `WHERE`, ecc.), sono case-insensitive
  - **literals**: valori costanti (numeri, stringhe, date, ecc.), sono case-sensitive e possono essere scritti racchiusi tra apici singoli (`'Mario Rossi'`)
- **separatori**: caratteri che separano i token (white space) o commenti (`--` o `/* ... */`)

5.3 Data types in SQL

Dati built-in

- `CHARACTER(n)`, `CHAR(n)`: stringa di lunghezza fissa di esattamente n caratteri (n > 0)
- `CHARACTER VARYING(n)`, `VARCHAR(n)`: stringa di lunghezza variabile fino a n caratteri (n > 0)
- `BINARY(n)`: stringa di lunghezza fissa di esattamente n byte (n > 0)
- `BINARY VARYING(n)`, `VARBINARY(n)`: stringa di lunghezza variabile fino a n byte (n > 0).
- `NUMERIC(p [, s])`: numero a precisione arbitraria con p cifre totali e s cifre nella parte frazionaria. È particolarmente raccomandato per memorizzare importi monetari e altre quantità dove è richiesta esattezza, ad esempio denaro.
- `SMALLSERIAL`, `SERIAL`, `BIGSERIAL` [PostgreSQL only]: intero auto-incrementante
- `INTEGER`: intero con segno (in PostgreSQL: 4 bytes integer $\in [-2147483648, +2147483647]$)
- `SMALLINT`: piccolo intero con segno, (in PostgreSQL: 2 bytes integer $\in [-32768, +32767]$)
- `BIGINT`: grande intero con segno (in PostgreSQL: 8 bytes integer $\in [-9.22 \cdot 10^{18}, +9.22 \cdot 10^{18}]$)
- `REAL`: decimale con segno, (in PostgreSQL, 4 bytes floating point con 6 cifre di precisione)
- `DOUBLE PRECISION`: decimale con segno a doppia precisione (in PostgreSQL: 8 bytes floating point con 15 cifre di precisione)
- `BOOLEAN`: valore logico booleano
- `DATE`: data (giorno, mese, anno)
- `TIME [WITH TIMEZONE | WITHOUT TIMEZONE]`: orario del giorno, con o senza fuso orario
- `TIMESTAMP [WITH TIMEZONE | WITHOUT TIMEZONE]`: data e orario, con o senza fuso orario
- `INTERVAL x [TO y]`: intervallo di tempo

UUID types

Gli UUID o Universally Unique Identifier sono identificatori univoci universali a 128 bit generati utilizzando uno dei diversi algoritmi standard nel modulo "uuid-ossp". Hanno la proprietà di avere bassissima probabilità di collisione. Sono spesso usati come identificatori in database distribuiti dove risulterebbe troppo costoso mantenere un contatore centralizzato per generare chiavi primarie univoche.

Range types

I Range types rappresentano un intervallo di valori di un tipo di dato specifico. In PostgreSQL sono disponibili i seguenti range types:

- `int4range`: range of `integer`
- `int8range`: range of `bigint`
- `numrange`: range of numeric
- `tsrange`: range of `timestamp without time zone`
- `tstzrange`: range of `timestamp with time zone`
- `daterange`: range of `date`

JSON types

I JSON types sono utilizzati per salvare dati in formato JSON (non relazionale). Sono spesso usati in database non relazionali chiamati NoSQL database. In PostgreSQL esistono due tipi di dati JSON:

- `json`: memorizza i dati in formato testo esattamente come sono stati inseriti, sono più veloci da inserire, ma più lenti da processare
- `jsonb`: memorizza i dati in un formato binario decomposto che li rende più lenti da inserire a causa dell'overhead di conversione, ma significativamente più veloci da processare, poiché non è necessaria una nuova analisi

Enumerated Types

```
-- crea un enumerated type con nome e valori specificati
CREATE TYPE <enum_name> AS ENUM ('<value1>', '<value2>', ...)
-- es. crea un enum "mood" con i valori 'happy', 'sad' e 'neutral'
CREATE TYPE mood AS ENUM ('happy', 'sad', 'neutral');

-- elimina un enumerated type con il nome specificato
DROP TYPE <enum_name> [CASCADE | RESTRICT];
-- es. elimina l'enum "mood"
DROP TYPE mood;
```

L'opzione `CASCADE` elimina anche gli oggetti che dipendono dal tipo, mentre `RESTRICT` impedisce l'eliminazione se il tipo è referenziato da altri oggetti. Il comportamento di default è `RESTRICT`.

Domain Types

```
-- crea un domain type basato su un tipo di dato esistente con vincoli opzionali
CREATE DOMAIN <domain_name> AS <data_type> [<constraint_name> <constraint_definition>];
-- es. crea un domain type basato su INTEGER con vincolo di essere positivo
CREATE DOMAIN positive_int AS INTEGER CHECK (VALUE > 0);
```

I vincoli possono essere:

- `[DEFAULT NOT NULL | NULL]`: per specificare se il valore di default è `NULL` o se è necessario sempre specificarne il valore
- `CHECK (expression)`: per specificare una condizione che i valori devono soddisfare

```
-- elimina un domain type con il nome specificato
DROP DOMAIN <domain_name> [CASCADE | RESTRICT];
-- es. elimina il domain type "positive_int"
DROP DOMAIN positive_int; -- elimina il domain type "positive_int"
```

L'opzione `CASCADE` elimina anche gli oggetti che dipendono dal tipo, mentre `RESTRICT` impedisce l'eliminazione se il tipo è referenziato da altri oggetti. Il comportamento di default è `RESTRICT`.

Null values

I valori nulli possono rappresentare tre situazini diverse:

- un valore indefinito
- un valore non disponibile (non è stato ancora assegnato)
- un valore sconosciuto (non è noto)

Gli attributi che costituiscono la chiave primaria o altri attributi obbligatori non possono assumere valori nulli ed è necessario indicare il vincolo `NOT NULL` durante la creazione della tabella.

In generale i valori null non sono uguali tra di loro e non sono uguali a nessun altro valore perché in assenza di informazione possono essere tutto e niente.

```
-- errato: non si possono confrontare i valori NULL con l'operatore di uguaglianza
SELECT * FROM table WHERE column = NULL;

-- corretto: si usano gli operatori IS NULL e IS NOT NULL
... WHERE column IS NULL;
... WHERE column IS NOT NULL;
```

5.4 Data definition Language (DDL)

Creazione ed eliminazione di database

```
-- crea un nuovo database con il nome specificato
CREATE DATABASE <database_name> [OWNER <username>] [ENCODING <encoding_name>];
-- es. crea il database "Example" con codifica UTF-8
CREATE DATABASE Example ENCODING 'UTF-8';

-- elimina un database con il nome specificato
DROP DATABASE <database_name>;
-- es. elimina il database "Example"
DROP DATABASE Example;
```

È possibile, in fase di creazione, specificare opzionalmente il proprietario del database e la codifica dei caratteri da utilizzare.

Creazione ed eliminazione di schema

```
-- crea uno schema con il nome specificato
CREATE SCHEMA <schema_name> [AUTHORIZATION <username>];
-- es. crea uno schema chiamato "my_schema"
CREATE SCHEMA my_schema;

-- elimina uno schema con il nome specificato
DROP SCHEMA <schema_name> [CASCADE | RESTRICT];
-- es. elimina lo schema "my_schema"
DROP SCHEMA my_schema;
```

L'opzione `CASCADE` elimina anche tutti gli oggetti all'interno dello schema, mentre `RESTRICT` impedisce l'eliminazione se lo schema contiene oggetti. Il comportamento di default è `RESTRICT`.

Creazione di tabelle

```
-- crea una nuova tabella
CREATE TABLE <schema_name>.<table_name> (
    <column_name> <data_type> [<default_value>] [<column_constraint>],
    [<column_name> <data_type> [<default_value>] [<column_constraint>], ...]
    [, <table_constraint>, ...]
);
```

Se non viene specificato lo schema in cui inserirla, viene usato lo schema `public` di default. Le colonne devono avere i rispettivi tipi di dato e possono essere indicati altri vincoli o constraint sulle colonne (valore di default, vincoli, ecc.) o sulla tabella. I constraint di colonna possono essere:

- `NOT NULL`: la colonna non può contenere valori nulli
- `CHECK (expression)`: la colonna deve soddisfare una condizione specifica
- `DEFAULT <constant> | niladic-function | NULL`: valore di default che viene utilizzato se non ne viene specificato uno durante l'inserimento. Le `niladic-function` sono ad esempio `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`, `LOCALTIME`, ...

I constraint di tabella possono essere:

- `PRIMARY KEY (<column_name>, ...)`: specifica una chiave primaria su una o più colonne
- `UNIQUE (<column_name>, ...)`: specifica una chiave candidata su una o più colonne
- `FOREIGN KEY (<column_name>, ...) REFERENCES <ref_table>(<ref_column>, ...)`: definisce una chiave esterna che fa riferimento a un'altra tabella, è possibile indicare azioni opzionali per la cancellazione `ON DELETE` e l'aggiornamento `ON UPDATE` dei dati referenziati (ad esempio `CASCADE`, `SET NULL`, `SET DEFAULT`, `RESTRICT`, `NO ACTION`)

Se i constraint di tabella coinvolgono una singola colonna (ad esempio chiave primaria di una singola colonna), è possibile definirli direttamente nella definizione della colonna come constraint di tabella. Se invcene coinvolgono più colonne, devono necessariamente essere definiti come constraint di tabella.

```
-- es. crea la tabella "students" nello schema "my_schema"
CREATE TABLE my_schema.students (
    badge INTEGER PRIMARY KEY,      -- badge deello studente, chiave primaria
    name VARCHAR(100) NOT NULL,     -- nome dello studente, colonna non nulla
    surname VARCHAR(100) NOT NULL, -- cognome dello studente, colonna non nulla
    dob DATE DEFAULT NULL,          -- data di nascita, valore di default NULL
    degree VARCHAR(50)              -- corso di laurea
    school FOREIGN KEY REFERENCE my_schema.schools(school_id) -- chiave esterna che fa
    riferimento alla colonna "school_id" della tabella "schools"
);

-- oppure si definiscono i constraint di colonna separatamente in constraint di tabella
CREATE TABLE my_schema.students (
    badge INTEGER,      -- chiave primaria non indicata come constraint di colonna
    school VARCHAR(50), -- chiave esterna non indicata come constraint di colonna
    ...
    PRIMARY KEY (badge) -- chiave primaria definita come constraint di tabella
    FOREIGN KEY (school) REFERENCES my_schema.schools(school_id) -- chiave esterna
    definita come constraint di tabella
);
```

Eliminazione di tabelle

```
-- elimina una tabella con il nome specificato
DROP TABLE <table_name> [CASCADE | RESTRICT];
-- es. elimina la tabella "students"
DROP TABLE students; -- elimina la tabella "students"
```

L'opzione `CASCADE` elimina anche gli oggetti che dipendono dalla tabella, mentre `RESTRICT` impedisce l'eliminazione se la tabella è referenziata da altri oggetti. Il comportamento di default è `RESTRICT`.

Modifica di tabelle

```
-- modifica una tabella esistente
ALTER TABLE <table_name> <
    ADD COLUMN <column_definition> |                 -- aggiunge una nuova colonna
    DROP COLUMN <column_name> [RESTRICT | CASCADE] | -- rimuove una colonna
    ALTER COLUMN <column_name> <SET DEFAULT <new_default>> | -- cambia il valore di def.
    ALTER COLUMN <column_name> <DROP DEFAULT> |      -- rimuove il valore di default
    ADD CONSTRAINT <constraint_definition> |         -- aggiunge un nuovo vincolo
    DROP CONSTRAINT <constraint_name>                -- rimuove un vincolo esistente
>;
```

5.5 Data Manipulation Language - DML

Inserimento di dati

```
-- inserisce una nuova riga in una tabella, se non vengono specificate le colonne, si
    assume che i valori siano forniti per tutte le colonne e nell'ordine in cui le
    colonne sono state definite
INSERT INTO <table_name> [(<column_name> ...)] VALUES (<value1>, ...);
-- es. inserimento di una nuova riga nella tabella "employees" esplicitando le colonne
INSERT INTO employees (name, dob, degree, salary)
    VALUES ('John Doe', '1990-01-01', 'Computer Science', 50000);
-- es. inserimento senza specificare le colonne
INSERT INTO employees
    VALUES ('Jane Smith', '1985-05-15', 'Mathematics', 60000);
```

Eliminazione di dati

```
-- elimina righe da una tabella, opzionalmente filtrate da una condizione
DELETE FROM <table_name> [WHERE <condition>];
-- es. elimina tutte le righe della tabella "employees"
DELETE FROM employees;
-- es. elimina le righe dalla tabella "employees" dove il salario e' inferiore a 30000
DELETE FROM employees WHERE salary < 30000;
```

Aggiornamento o modifica dei dati

```
-- aggiorna i valori delle colonne in una tabella
UPDATE <table_name> SET <column_name> = <expr.> | NULL | DEFAULT, ... [WHERE <expr.>];
-- aggiorna il salario dei dipendenti nella tabella "employees" aumentando del 10% per
    quelli con salario inferiore a 60000
UPDATE employees SET salary = salary * 1.1 WHERE salary < 60000;
```

5.6 Query

Struttura generale

```
SELECT [DISTINCT | ALL] <column_list> | *
    FROM <table_name> [AS <alias>]
    [JOIN <table_name> [AS <alias>] ON <join_condition>]
    [WHERE <condition>]
    [GROUP BY <column_list> HAVING <condition>]
    [ORDER BY <column_name> [ASC | DESC], ...];
```

- `SELECT`: specifica le colonne da recuperare, può essere usato `DISTINCT` per eliminare i duplicati o `ALL` per includerli tutti (default), se si vuole recuperare tutte le colonne si usa l'asterisco `*`, eventualmente si possono usare funzioni di aggregazione (opportunamente rinominate)
- `FROM`: specifica la tabella da cui recuperare i dati, può essere usato un alias per riferirsi alla tabella attraverso un altro nome (utile nelle join)
- `JOIN`: unisce le tuple di una o più tabelle basandosi su una condizione di join, per facilitare la scrittura delle condizioni di join si utilizzano gli alias
- `WHERE`: filtra le righe in base a una condizione
- `GROUP BY`: raggruppa le righe in gruppi in base ai valori di una o più colonne, eventualmente si possono applicare funzioni di aggregazione sui gruppi ed è possibile filtrare i gruppi attraverso la clausola `HAVING`
- `ORDER BY`: ordina i risultati in base ai valori di una o più colonne, se specificati più ordinamenti su più colonne, vanno indicati in ordine di priorità decrescente

Ordine di valutazione delle clausole

```
FROM --> JOIN --> WHERE --> GROUP BY --> HAVING --> SELECT --> ORDER BY
```

Condizioni ed espressioni delle clausole

Le condizioni nelle varie clausole possono utilizzare operatori di confronto (`=`, `<>`, `<`, `>`, `<=`, `>=`) e di appartenenza (`IN`, `NOT IN`). È possibile combinare più condizioni utilizzando operatori logici (`AND`, `OR`, `NOT`). Inoltre, esiste l'operatore `LIKE` per confrontare stringhe con pattern che possono includere i caratteri jolly: il `%` rappresenta una sequenza di zero o più caratteri, mentre il `_` rappresenta un singolo carattere

```
-- 'J_n%' <=> 'J' + 1 carattere + 'n' + qualsiasi sequenza di caratteri
SELECT * FROM employees WHERE name LIKE 'J_n%';
```

Selection in SQL - clausola `WHERE`

Le selection selezionano le righe di una tabella che soddisfano una certa condizione e vengono implementate in SQL attraverso la clausola `WHERE`. Ad esempio:

```
SELECT * FROM employees WHERE age > 30;
```

Projection in SQL - clausola `SELECT`

Le projection selezionano le colonne di una tabella e vengono implementate in SQL attraverso la clausola `SELECT`. Ad esempio:

```
SELECT name, age FROM employees;
```

Si osserva che, siccome in SQL le tabelle derivate possono contenere righe duplicate, le projection in SQL non corrispondono a quelle dell'algebra relazionale, a meno che non si usi la clausola `DISTINCT`.

Rename in SQL - clausola `AS`

Le rename in SQL vengono implementate attraverso la clausola `AS` che permette di assegnare un alias a una tabella o a una colonna nella query. Ad esempio:

```
SELECT e.name AS employee_name, e.dob AS date_of_birth FROM employees AS e;
```

Set Operators

I set operators combinano i risultati di due o più query. In SQL non è necessario che le tabelle siano compatibili all'unione (come in algebra relazionale), ma è sufficiente che le colonne siano dello stesso numero e che abbiano tipi di dato compatibili. Inoltre in SQL le tabelle derivate da set operators non contengono righe duplicate, a meno che non venga specificata l'opzione `ALL`. In SQL sono disponibili i seguenti operatori di insieme tra query:

- `query1 UNION query2`: unisce i risultati di due query
- `query1 INTERSECT query2`: restituisce le righe comuni a due query
- `query1 EXCEPT query2`: restituisce le righe presenti nella prima query ma non nella seconda

Aggregate Functions

Le aggregate functions permettono di eseguire funzioni su un insieme di valori di determinati attributi. Le funzioni di aggregazione più comuni in SQL sono:

- `COUNT(<column_name>)`: conta il numero di righe in un gruppo
- `SUM(<column_name>)`: calcola la somma dei valori in un gruppo
- `AVG(<column_name>)`: calcola la media dei valori in un gruppo
- `MIN(<column_name>)`: trova il valore minimo in un gruppo
- `MAX(<column_name>)`: trova il valore massimo in un gruppo

In alternativa al `<column_name>` è possibile usare l'asterisco `*` per indicare di agire su tutte le colonne.

Grouping

Il raggruppamento in SQL viene implementato attraverso la clausola `GROUP BY` che permette di raggruppare le righe in base ai valori di una o più colonne. È possibile inoltre filtrare i gruppi utilizzando la clausola `HAVING` che specifica una condizione che i gruppi devono soddisfare. Inoltre è possibile utilizzare funzioni di aggregazione per calcolare valori sui gruppi. Ad esempio, per calcolare il salario medio per ogni grado di istruzione:

```
SELECT degree, AVG(salary) AS average_salary FROM employees GROUP BY degree;
```

Le colonne elencate nella clausola `GROUP BY` devono essere presenti anche nella clausola `SELECT`. Inoltre è sempre consigliato rinominare le colonne derivate dalle funzioni di aggregazione per facilitare la lettura dei risultati. Si nota che la clausola `HAVING` viene valutata prima della clausola `SELECT` e di conseguenza prima della rename, per cui se si vuole filtrare in base ad una colonna generata con funzione di aggregazione, bisogna usare l'espressione originale e non l'alias. Ad esempio, per trovare i gradi di istruzione con salario medio superiore a 60000:

```
SELECT degree, AVG(salary) AS average_salary FROM employees
    GROUP BY degree HAVING AVG(salary) > 60000;
```

Query con `JOIN`

Le join in SQL permettono di creare tabelle derivate che hanno come righe la combinazione di righe di due o più tabelle basate su una condizione di join. Le condizioni di join sono uguaglianze tra attributi, ovvero le nuove righe risultanti saranno date degli attributi della tabella di sinistra, altri attributi dati dalla tabella di destra e un attributo in comune tra le due. Esitono 4 tipi principali di join:

- `INNER JOIN`: restituisce solo le righe che hanno corrispondenza in entrambe le tabelle
- `LEFT JOIN`: restituisce tutte le righe della tabella di sinistra a cui vengono associate le righe corrispondenti della tabella di destra, se non ci sono corrispondenze, i valori della tabella di destra saranno `NULL`
- `RIGHT JOIN`: restituisce tutte le righe della tabella di destra a cui vengono associate le righe corrispondenti della tabella di sinistra, se non ci sono corrispondenze, i valori della tabella di sinistra saranno `NULL`
- `FULL JOIN`: restituisce tutte le righe di entrambe le tabelle, se non ci sono corrispondenze, i valori della tabella senza corrispondenza saranno `NULL`

È utile effettuare i rename delle tabelle coinvolte nella join per facilitare la scrittura delle condizioni di join. Inoltre è possibile avere più join concatenate. Ad esempio, per unire le tabelle "employees", "departments" e "projects" basandosi sull'attributo "department_id" e "project_id":

```
SELECT e.name, d.department_name
    FROM employees AS e
    -- inner join tra employees e departments
    INNER JOIN departments AS d ON e.department_id = d.department_id
    -- inner join tra il risultato delle join precedenti e projects
    INNER JOIN projects AS p ON d.project_id = p.project_id;
```

Generalized projection

La generalized projection in SQL permette di creare nuove colonne derivate da espressioni o funzioni. Ad esempio, per calcolare l'età dei dipendenti basandosi sulla loro data di nascita:

```
SELECT name, EXTRACT(YEAR FROM AGE(CURRENT_DATE, dob)) AS age FROM employees;
```

Nested Queries

È possibile annidare query all'interno di altre query per creare condizioni più complesse o per calcolare valori intermedi. Le query annidate si indicano tra parentesi tonde.Ad esempio, per trovare i dipendenti con un salario superiore alla media:

```
SELECT name, salary FROM employees WHERE salary > (SELECT AVG(salary) FROM employees);
```

Un esempio comune di utilizzo delle nested query è trovare il valore di un certo attributo associato al valore massimo di un altro attributo, ad esempio per trovare lo studente con il voto più alto:

```
SELECT name, grade FROM student WHERE grade = (SELECT MAX(grade) FROM student);
```

Viste

Le viste in SQL sono tabelle virtuali derivate da altre tabelle attraverso una query. Non sono memorizzate fisicamente nel database, ma vengono generate al momento dell'accesso. Le viste permettono di semplificare query complesse spezzandole in più fasi. Il fatto di essere sempre ricalcolate porta al vantaggio di avere sempre dati aggiornati, ma lo svantaggio di avere prestazioni inferiori rispetto ad utilizzare una query completa e unica (il cui calcolo è ottimizzato dal DBMS). Le viste possono essere:

- **online views**: vengono create e utilizzate direttamente dal database
- **materialized views**: vengono create e memorizzate fisicamente nel database, migliorando le prestazioni a scapito di possibile incoerenza dei dati se le tabelle sottostanti vengono aggiornate

Di seguito sono riportati i comandi per creare ed eliminare viste in SQL:

```
-- crea una vista basata su una query selezionata
CREATE [MATERIALIZED] VIEW <view_name> AS <select_query>;

-- crea una vista chiamata "high_salary_employees" che mostra i dipendenti con salario
    superiore a 70000
CREATE VIEW high_salary_employees AS
    SELECT name, salary FROM employees WHERE salary > 70000;

-- elimina una vista con il nome specificato
DROP VIEW <view_name> [CASCADE | RESTRICT];

-- elimina la vista "high_salary_employees"
DROP VIEW high_salary_employees;
```

L'opzione `CASCADE` elimina anche gli oggetti che dipendono dalla vista, mentre `RESTRICT` impedisce l'eliminazione se la vista è referenziata da altri oggetti. Il comportamento di default è `RESTRICT`.