Java - SQL

Prima parte: SQL

Giorno 01: 28.04.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer

Obiettivi specifici della prima parte del corso

Questa prima parte è progettata per dotarvi delle competenze essenziali nel campo dei database SQL. Attraverso una serie di lezioni teoriche e pratiche, miriamo a trasformarvi da principianti a utenti competenti di SQL:

- ***Comprensione Fondamentale***
- ***Familiarità con i DBMS***
- ***Progettazione di Database***
- ***Padronanza di SQL***
- ***Esperienza Pratica***
- ***Problemi Realistici***

Al termine di questo corso, sarete in grado di comprendere e utilizzare efficacemente SQL nei vostri progetti o nel vostro contesto lavorativo.

Benefici e le Applicazioni Pratiche dell'Apprendimento SQL

Benefici dell’apprendimento SQL

- Domanda di Mercato
- Versatilità Professionale
- Base per l'Apprendimento Avanzato

Applicazioni Pratiche

- Gestione di Database
- Analisi Dati
- Automatizzazione e Ottimizzazione

Con queste competenze SQL, avrete gli strumenti per esplorare nuove opportunità e affrontare sfide complesse nel campo dei dati.

Panoramica della prima parte del corso

- Introduzione ai Database
- Modello E-R e Relazionale
- Linguaggio SQL
- Implementazione di Database Relazionali
- Testing e Manutenzione

Che cosa è Un **DATABASE**?

Che cosa è un DATABASE?

Definizione:

Un database è un sistema organizzato per la raccolta, la memorizzazione, la gestione e la restituzione di informazioni. Consente l'organizzazione e l'accesso efficiente ai dati

Tipi di Database

- Relazionali
- Non Relazionali (NoSQL)
- Distribuiti
- Centralizzati

Importanza dei Database

Fondamentali per la gestione delle informazioni in tutti i settori: aziendale, educativo, scientifico, tecnologico

Importanza e applicazioni pratiche di un Database

Punti chiave

- Efficienza operativa.
- Supporto decisionale basato sui dati
- Sicurezza e integrità dei dati.

Applicazioni pratiche

- Gestione dei clienti in un'azienda
- Analisi finanziaria in una banca
- Ricerca e sviluppo in un laboratorio
- ……

La comprensione dei database è cruciale nell'era digitale per la gestione efficace dei dati e il supporto alle operazioni aziendali.

Esempi pratici di Database

Database nel mondo aziendale

Database dei clienti: gestione delle informazioni sui clienti, storico degli acquisti, fatture, pagamenti, preferenze, …

Database nel settore educativo

Database degli studenti: informazioni personali, risultati accademici, iscrizioni ai corsi, …

Database in Sanità

Database pazienti: dati sanitari, cronologia delle visite, prescrizioni mediche, …

Questi esempi mostrano come i database siano fondamentali in diversi settori per organizzare e accedere efficacemente ai dati

Che cosa è Un **DBMS**?

Che cosa è un DBMS (Database Management System)?

Definizione:

*Un DBMS è un **software** che permette la creazione, la gestione e l'interazione con i database. Funge da interfaccia tra i database e gli utenti o le applicazionii*

Funzioni principali di un DBMS

- Creazione e modifica di schemi di database
- Supporto per le query e le relazioni fra tabelle
- Implementazione di misure di sicurezza e backup
- Definizione e gestione utenti, ruoli e accesso al database

ESEMPI:

mySQL, mariaDB, PostgreSQL, Oracle, Microsoft SQL Server, MongoDB, SAP Hana …

Comprendere i DBMS è essenziale per sfruttare appieno le potenzialità dei database in qualsiasi settore

Approfondimento sui DBMS

Cosa fa un DBMS

Un DBMS non solo memorizza dati, ma gestisce anche operazioni complesse, assicura l'integrità e la sicurezza dei dati e facilita l'interazione degli utenti con i database

Tipologie di DBMS

DBMS relazionali, DBMS non-relazionali (NoSQL), DBMS distribuiti, DBMS gerarchici.

Un esempio pratico

Come un DBMS facilita la gestione di database in un ambiente aziendale, educativo o di ricerca

Comprendere i DBMS è fondamentale per qualunque professionista che lavori con i dati

Ruoli e funzioni dei DBMS

Ruolo del DBMS

I DBMS fungono da intermediari tra gli utenti e i database, facilitando la creazione, manipolazione e gestione dei dati.

Punti Chiave:

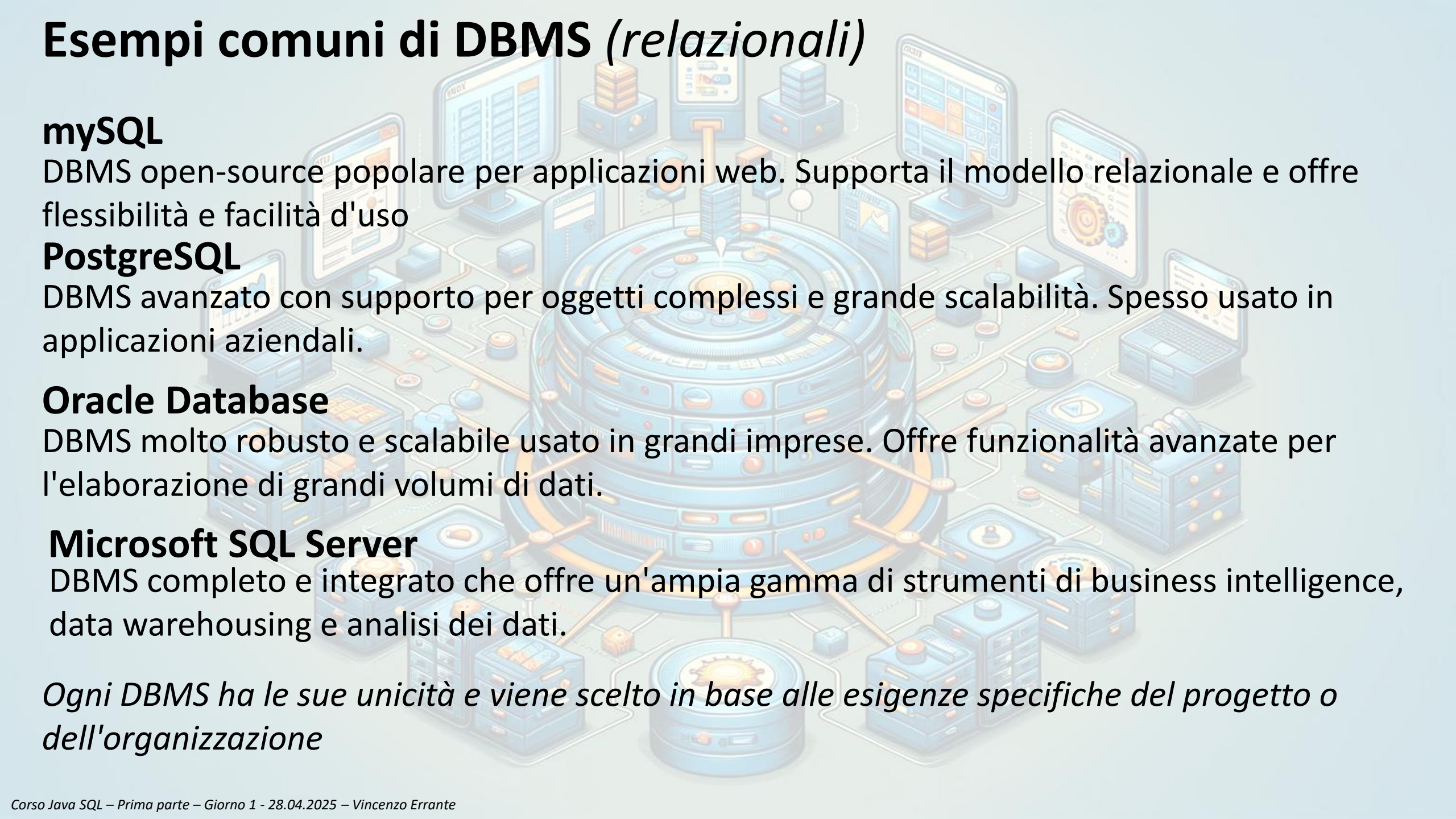
- Interfaccia utente semplice
- Gestione efficiente delle **transazioni**
- Controllo dell'accesso e sicurezza dei dati

Funzioni specifiche di un DBMS

Controllo della ridondanza dei dati, backup e ripristino, integrità dei dati, sicurezza dei dati, gestione delle transazioni e supporto per linguaggi di query come SQL

I DBMS sono essenziali per mantenere l'ordine e l'efficienza nei moderni sistemi informativi.

Esempi comuni di DBMS *(relazionali)*



mySQL
DBMS open-source popolare per applicazioni web. Supporta il modello relazionale e offre flessibilità e facilità d'uso

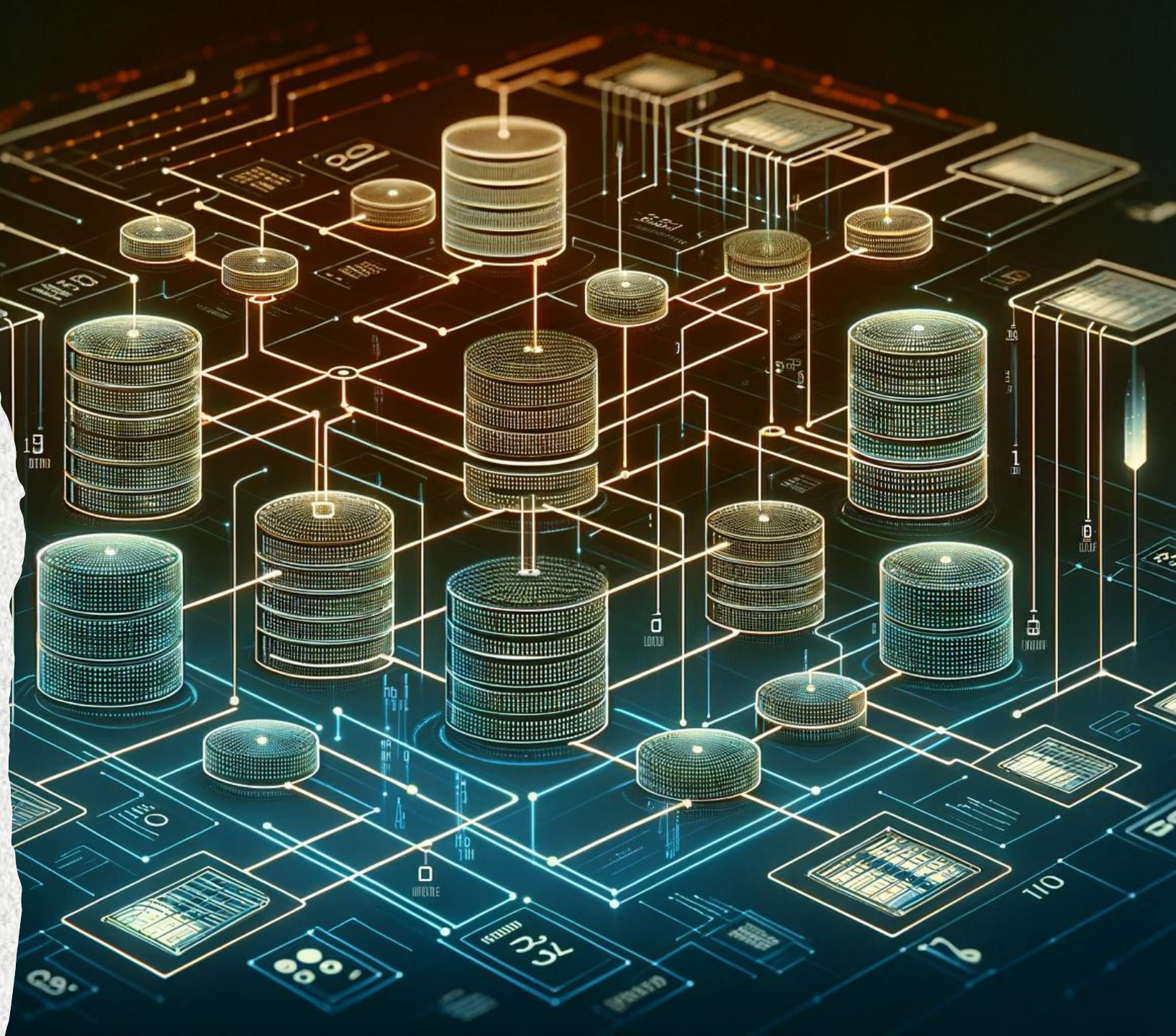
PostgreSQL
DBMS avanzato con supporto per oggetti complessi e grande scalabilità. Spesso usato in applicazioni aziendali.

Oracle Database
DBMS molto robusto e scalabile usato in grandi imprese. Offre funzionalità avanzate per l'elaborazione di grandi volumi di dati.

Microsoft SQL Server
DBMS completo e integrato che offre un'ampia gamma di strumenti di business intelligence, data warehousing e analisi dei dati.

Ogni DBMS ha le sue unicità e viene scelto in base alle esigenze specifiche del progetto o dell'organizzazione

Introduzione ai **DB Relazionali**



Che cosa è un Database Relazionale?

Definizione:

Un database relazionale organizza i dati in tabelle (o relazioni), che possono essere collegate tra loro tramite chiavi. Questo consente di strutturare, memorizzare e recuperare dati in modo efficiente.

Caratteristiche

Facilità di uso, flessibilità, indipendenza dei dati, supporto per operazioni complesse di query e transazioni

ESEMPI:

Gestione dei clienti, sistemi di inventario, applicazioni bancarie, sistemi di prenotazione

I database relazionali sono la spina dorsale di molte applicazioni moderne e sono essenziali per la gestione efficace dei dati.

Approfondimento sui DB Relazionali

Struttura di un Database Relazionale

I database relazionali sono costituiti da tabelle, con ogni tabella rappresentante un tipo di entità. Le righe (record) rappresentano istanze di quell'entità e le colonne rappresentano attributi.

Relazioni tra tabelle

Le tabelle sono collegate tra loro tramite chiavi (primarie e esterne), che stabiliscono le relazioni e consentono *l'integrità referenziale*

Normalizzazione

Processo di organizzazione dei dati per ridurre la ridondanza e migliorare l'integrità.
Include diversi 'forme normali' da seguire

Una comprensione approfondita di questi concetti è essenziale per progettare e gestire database relazionali efficacemente

Vantaggi dei Database Relazionali

Facilità di utilizzo

L'interfaccia tabellare dei database relazionali è intuitiva e facile da comprendere, anche per non esperti.

Flessibilità

I database relazionali permettono modifiche ai dati e alla struttura senza interrompere le operazioni esistenti

Integrità e sicurezza dei dati

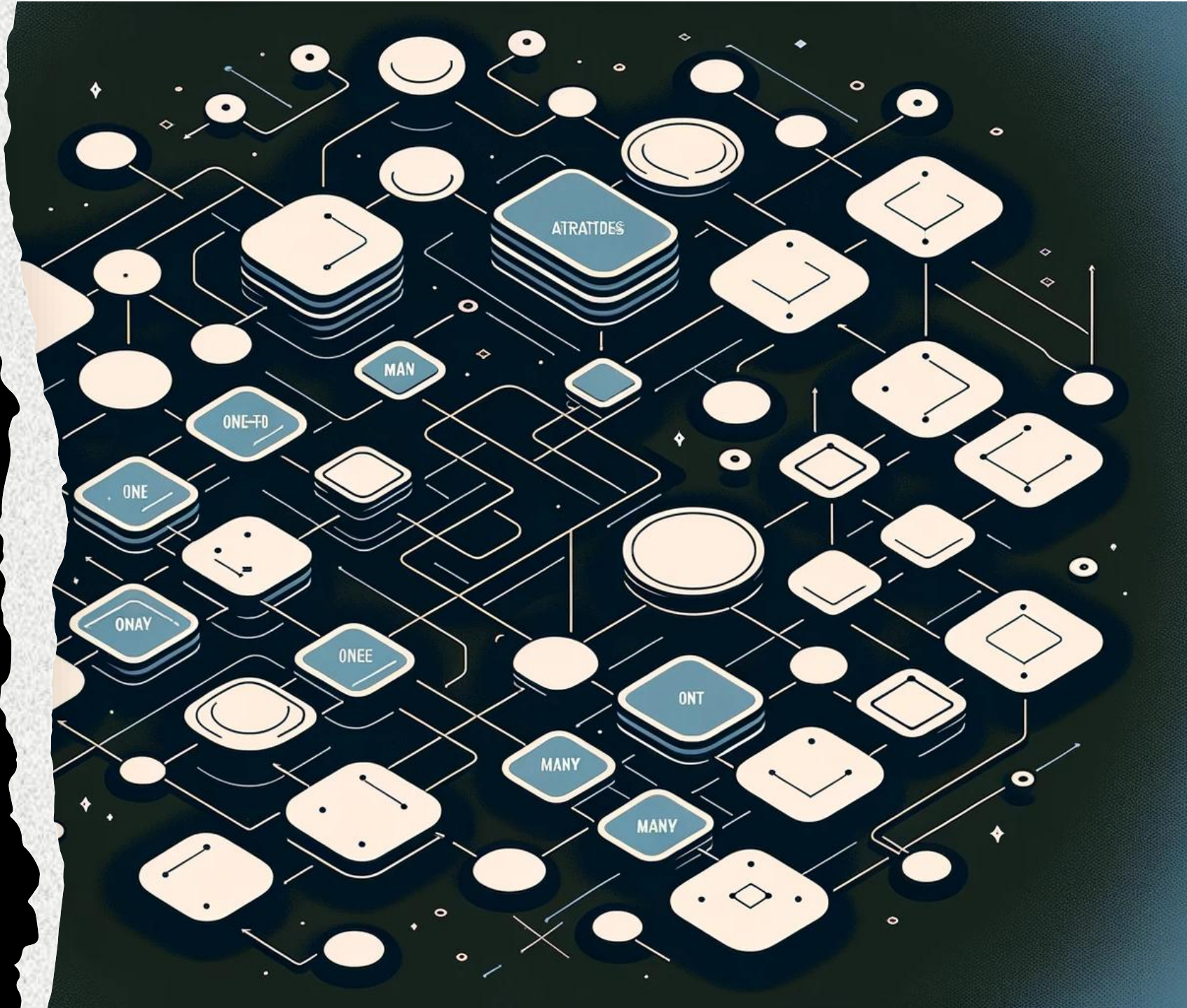
Forniscono strumenti robusti per mantenere l'integrità dei dati e garantire la sicurezza delle informazioni

Supporto per query complesse

Capacità di gestire query complesse e avanzate, essenziale per l'analisi dei dati e il reporting.

Questi vantaggi rendono i database relazionali una scelta popolare per una vasta gamma di applicazioni

Il modello E-R (Entità/Relazione)



Transizione al Modello Relazionale

Il modello ER

Il modello Entità-Relazione (E-R) è spesso il punto di partenza nella progettazione del database. Queste entità e relazioni sono poi tradotte in tabelle nel modello relazionale

Regole di conversione

Regole standard per trasformare entità in tabelle, relazioni in chiavi esterne, e attributi in colonne

Esempio di conversione

Esame (vedi esempio)

Questa capacità di transizione è fondamentale per costruire database relazionali che riflettano accuratamente i requisiti e le strutture del mondo reale.

Introduzione al Modello E-R

Iniziamo con una breve storia del modello Entità-Relazione, comunemente noto come modello E-R. Sviluppato da **Peter Chen nel 1976.**

Il modello E-R è diventato uno standard per la progettazione concettuale dei database.

Il suo obiettivo principale è quello di fornire un modo intuitivo e visuale di descrivere dati e relazioni in un sistema.

Questo modello è ampiamente usato per la progettazione di basi di dati e per garantire che tutte le parti di un sistema di database siano accuratamente e comprensivamente rappresentate.

Componenti principali del Modello E-R

Entità

Un'entità può essere definita come un oggetto o un concetto che esiste nel mondo reale e che è distinguibile da altri oggetti o concetti.

Ad esempio, in un database di una scuola, **Studente** può essere un'entità. Le entità sono rappresentate nei diagrammi **E-R come rettangoli**.

Ogni entità ha delle proprietà chiamate **attributi**, che ne descrivono le caratteristiche.

Ad esempio, per l'entità **Studente**, gli attributi potrebbero includere *Matricola*, *Nome*, e *Data di Nascita*.

Componenti principali del Modello E-R

Attributi

Gli attributi sono dettagli che descrivono un'entità.

Esistono diversi tipi di attributi: semplici, composti, multi-valore e derivati.

- Un attributo **semplice** non può essere diviso ulteriormente; un esempio potrebbe essere *Data di Nascita* di uno studente.
- Un attributo **composto**, come *Nome*, potrebbe essere suddiviso in *Nome* e *Cognome*.
- Un attributo **multi-valore**, come *Numero di Telefono*, può avere più valori per la stessa entità.
- un attributo **derivato** è calcolato da altri attributi, come l'*età* che può essere derivata dalla data di nascita.

Componenti principali del Modello E-R

Relazioni

Le Relazioni sono l'aspetto più cruciale del modello E-R.
Una relazione definisce come **due** (o più) **entità** sono associate tra loro.

*Ad esempio, un'entità **Studente** può essere associata all'entità **Corso** tramite una relazione **Iscrizione** .*

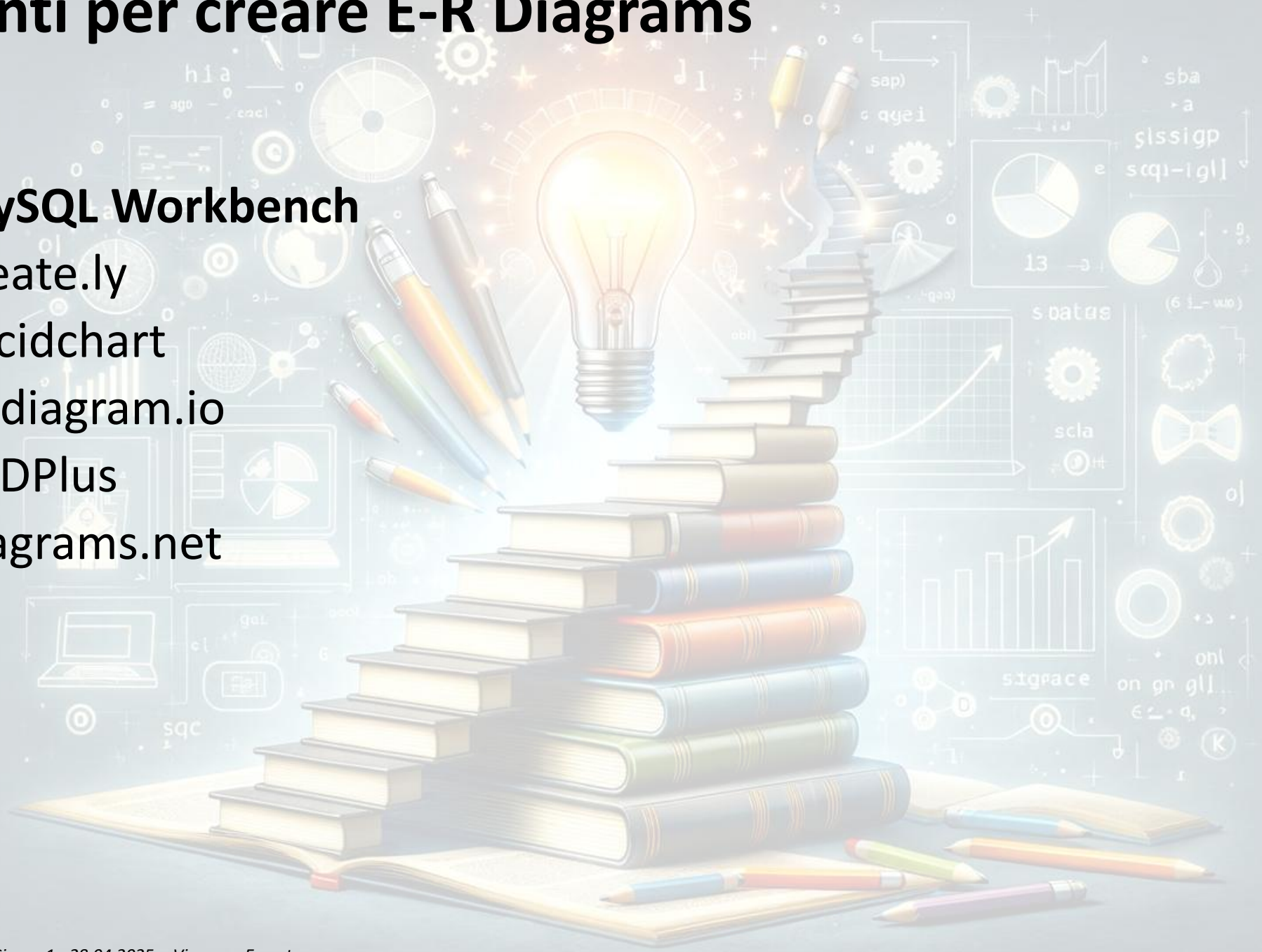
Le relazioni sono rappresentate come **rombi** nei diagrammi E-R (o con dei **simboli alle terminazioni delle frecce** che legano le entità).

Esistono diversi tipi di relazioni: **uno-a-uno, uno-a-molti e molti-a-molti**.

- una relazione **uno-a-uno** potrebbe esistere tra *Studente* e *Carta d'identità (valida)*
- una relazione **uno-a-molti** potrebbe esistere tra *Studente* e *Corso*, poiché uno studente può iscriversi a un solo corso, ma a un corso possono iscriversi più studenti
- una relazione **molti a-molti** potrebbe esistere tra *Materia* e *Corso*, poiché una materia può essere presente in più corsi e un corso e costituito da più materie

Strumenti per creare E-R Diagrams

- **MySQL Workbench**
- create.ly
- Lucidchart
- dbdiagram.io
- ERDPlus
- diagrams.net
- ...



Java - SQL

Prima parte: SQL

Giorno 02: 29.04.2025

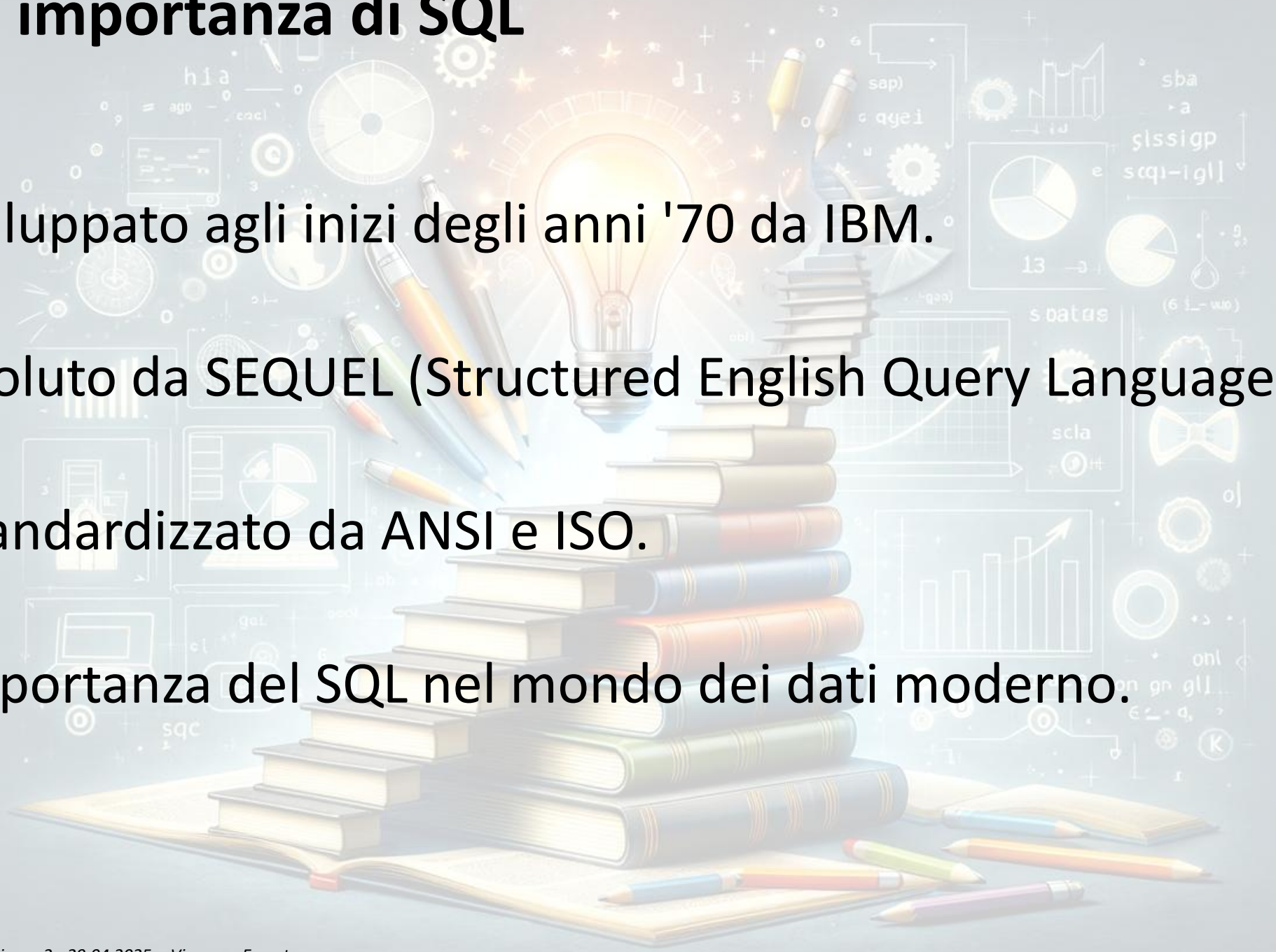
Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer

Introduzione a **SQL**

Fondamenti e applicazioni pratiche

Storia e importanza di SQL



- Sviluppato agli inizi degli anni '70 da IBM.
- Evoluto da SEQUEL (Structured English Query Language).
- Standardizzato da ANSI e ISO.
- Importanza del SQL nel mondo dei dati moderno.

I Pilastri di SQL

DDL ***(Data Definition Language)***

Tutti i comandi SQL relativi alla struttura dei dati.

Esempi: creazione di tabelle, definizione di indici.

DML ***(Data Manipulation Language)***

Tutti i comandi SQL si manipolazione dei dati.

Esempi: inserimento, aggiornamento e cancellazione.

DQL ***(Data Query Language)***

Tutti i comandi SQL relativi all’interrogazione e recupero dei dati.

Esempi: recupero di dati specifici, aggregazioni, filtri.

I comandi DDL

DDL (Data Definition Language) si occupa di "descrivere" e "definire" la struttura dei dati

CREATE

Creazione di nuove tabelle e database, indici, e altri oggetti

ALTER

Aggiunta, rimozione o modifica di colonne in una tabella o indice esistente.

DROP

Rimozione di tabelle o database o altri oggetti

I comandi DDL: CREATE

Usato per creare nuove tabelle e altri oggetti all'interno del database

Sintassi di base:

```
CREATE TABLE nome_tabella (colonna1 tipo_dato, colonna2 tipo_dato, ...);
```

Esempio: Creazione di una tabella 'studenti' con campi per id, nome, cognome e email.

```
CREATE TABLE studente (
  id INT AUTO_INCREMENT PRIMARY KEY,
  nome VARCHAR(50) NOT NULL,
  cognome VARCHAR(50) NOT NULL,
  email VARCHAR(100) UNIQUE
);
```

Il comando CREATE è il primo passo nella definizione della struttura

I comandi DDL: ALTER TABLE

Usato per modificare la struttura di una tabella esistente

Permette di aggiungere, eliminare o modificare **colonne**

Esempio:
Aggiunta di una nuova colonna 'data_di_nascita' alla tabella 'studenti'.

```
ALTER TABLE studente
ADD COLUMN data_nascita DATETIME;
```

ALTER è essenziale per l'evoluzione e l'adattamento della struttura del database

I comandi DDL: DROP

Usato per eliminare tabelle dal database

Rimuove completamente la struttura e i dati.

Esempio:
Eliminare la tabella studenti

```
DROP TABLE studenti;
```

Il comando DROP deve essere usato con cautela poiché è un'azione irreversibile

I comandi DML

DML (Data Manipulation Language) è usato per inserire, modificare, e cancellare i dati all'interno di una base di dati relazionali

INSERT

Sintassi per l'inserimento di dati, inclusi esempi pratici.

UPDATE

Aggiornamento dei dati con condizioni specifiche.

DELETE

Eliminazione di dati, con attenzione all'integrità dei dati e agli effetti

I comandi DML: INSERT

Usato per inserire nuovi dati nelle tabelle

Sintassi di base:

```
INSERT INTO nome_tabella (colonna1, colonna2, ...) VALUES (valore1, valore2, ...);
```

Esempio:

Inserimento di un nuovo studente nella tabella *studente*.

```
INSERT INTO studente (nome, cognome, email)
VALUES ('Mario', 'Rossi', 'mario.rossi@email.com');
```

INSERT è il modo per aggiungere nuovi record al tuo database.

I comandi DML: UPDATE

Usato per modificare i dati esistenti in una tabella

Sintassi di base:

```
UPDATE nome_tabella SET colonna1 = valore1, colonna2 = valore2 WHERE condizione;
```

Esempio:

Modifica email di uno studente nella tabella 'studenti’.

```
UPDATE studente
SET email = 'nuovo.email@email.com’
WHERE id = 1;
```

UPDATE permette di mantenere i dati del database aggiornati e accurati.

I comandi DML: DELETE

Usato per eliminare i record da una tabella

Sintassi di base:

```
DELETE FROM nome_tabella WHERE condizione;
```

Esempio:

Eliminazione di uno studente

```
DELETE FROM studenti
WHERE id = 1;
```

DELETE rimuove i dati in base alle condizioni specificate, essenziale per gestire i dati non più necessari

I comandi DDL: TRUNCATE TABLE

Usato per SVUOTARE le tabelle

Rimuove completamente i dati.

Esempio:
Svuotare la tabella studenti

```
TRUNCATE TABLE studenti;
```

Il comando Truncate deve essere usato con cautela poiché è un'azione irreversibile

Java - SQL

Prima parte: SQL

Giorno 03: 30.04.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer

I comandi Base DQL

DQL (Data Query Language) è utilizzato per interrogare (ovvero, recuperare) i dati dalla base di dati. Non modifica i dati, ma estrae informazioni

- **SELECT**: Struttura di base e variazioni (ad es. **SELECT DISTINCT**).
- Filtrare i dati con **WHERE**.
- Ordinare e raggruppare i risultati con **ORDER BY, GROUP BY**.
- Funzioni aggregate (es. **COUNT, AVG**).
- Limitare i risultati con **LIMIT**

Questo è un elenco base e non esaustivo dei comandi DQL

DQL: SELECT e SELECT DISTINCT

SELECT è Utilizzato per selezionare dati da una o più tabelle
SELECT DISTINCT elimina I duplicati, restituendo solo I valori unici

Esempio:
Selezionare tutti i titoli dei film e poi solo quelli unici

```
-- Selezione di tutti i titoli dei film da 'film’
SELECT title FROM film;

-- Selezione di titoli unici dei film da 'film’
SELECT DISTINCT title FROM film;
```

SELECT è il pilastro fondamentale per interrogare i dati.

DQL: Filtrare i dati con WHERE

WHERE viene utilizzato per filtrare i record in base a una o più condizioni permettendo di restringere il set di risultati

Esempio:

Selezionare tutti i titoli dei film e poi solo quelli unici

```
-- Selezione di film con rating 'PG-13’
SELECT title, rating FROM film WHERE rating = 'PG-13’;

-- Selezione di clienti da una specifica città
SELECT first_name, last_name FROM customer WHERE city_id = 5;
```

WHERE è essenziale per un'analisi dei dati mirata.

DQL: Comandi ORDER BY e GROUP BY

ORDER BY ordina i risultati in base a una o più colonne.
GROUP BY raggruppa i risultati in base a una o più colonne per aggregazione

Esempio:
Ordinare i film per anno di uscita e raggrupparli

```
-- Ordinamento dei film per anno di uscita
SELECT title, release_year FROM film ORDER BY release_year DESC;

-- Raggruppamento dei clienti per città e conteggio
SELECT city_id, COUNT(*) AS customer_count FROM customer GROUP BY city_id;
```

ORDER BY e GROUP BY offrono una vista strutturata dei dati.

DQL: Utilizzo delle Funzioni Aggregate

Le funzioni aggregate calcolano valori derivati da un insieme di dati.
Comunemente usate con GROUP BY

Esempio:
Contare i film e calcolare la media dei prezzi dei noleggi per categoria

```
-- Calcolo del numero totale di film
SELECT COUNT(*) AS total_films FROM film;

-- Media dei prezzi di noleggio per categoria
SELECT category_id, AVG(rental_rate) AS average_rental_price FROM film GROUP BY
category_id;
```

Le funzioni aggregate forniscono insights essenziali sui dati aggregati.

Clausola HAVING

Clausola HAVING: L'operatore di filtraggio post-aggregazione

HAVING
viene utilizzato per filtrare i risultati di una query che utilizza *GROUP BY*

A differenza di WHERE, HAVING **filtra valori aggregati**

Se WHERE è il cancello d'ingresso, HAVING è il controllo di sicurezza finale.

DQL: Limitare i Risultati con LIMIT

LIMIT viene utilizzato per restrizione del numero di record restituiti.
Fondamentale per grandi set di dati e per implementare la paginazione.

Esempio:
Selezionare i primi film per titolo e i primi clienti che spendono di più

```
-- Selezione dei primi 10 film per titolo
SELECT title FROM film ORDER BY title LIMIT 10;

-- Selezione dei primi 5 clienti con i più alti pagamenti
SELECT customer_id, amount FROM payment ORDER BY amount DESC LIMIT 5;
```

LIMIT è utile per gestire grandi quantità di dati e perottimizzare le query.

Operatore LIKE

LIKE: Trovare Corrispondenze di Pattern nei Dati

LIKE è utilizzato per cercare pattern di testo specifici

Definiscono il pattern i simboli

- **%** (qualsiasi sequenza di caratteri)
- _ (un singolo carattere) definiscono il pattern.

LIKE è la lente d'ingrandimento per esaminare i dettagli nei dati.

Funzioni di controllo di flusso

CASE e IF nella clausola SELECT in SQL

- **CASE**
  è utilizzata per implementare logica condizionale nelle query, simile a 'if-then-else' nei linguaggi di programmazione. Permette di selezionare diversi valori .

- **IF**
  In SQL, è un'altra forma di espressione condizionale che restituisce un valore se una condizione è vera, e un altro valore se è falsa. Simile a `CASE`, ma più limitato a condizioni singole.

Le funzioni di controllo del flusso aumentano la flessibilità delle tue query

Introduzione alle JOIN

Unire dati per informazioni complete

Le JOIN sono utilizzate per combinare le righe di due o più tabelle, basate su una **colonna relazionale** comune.

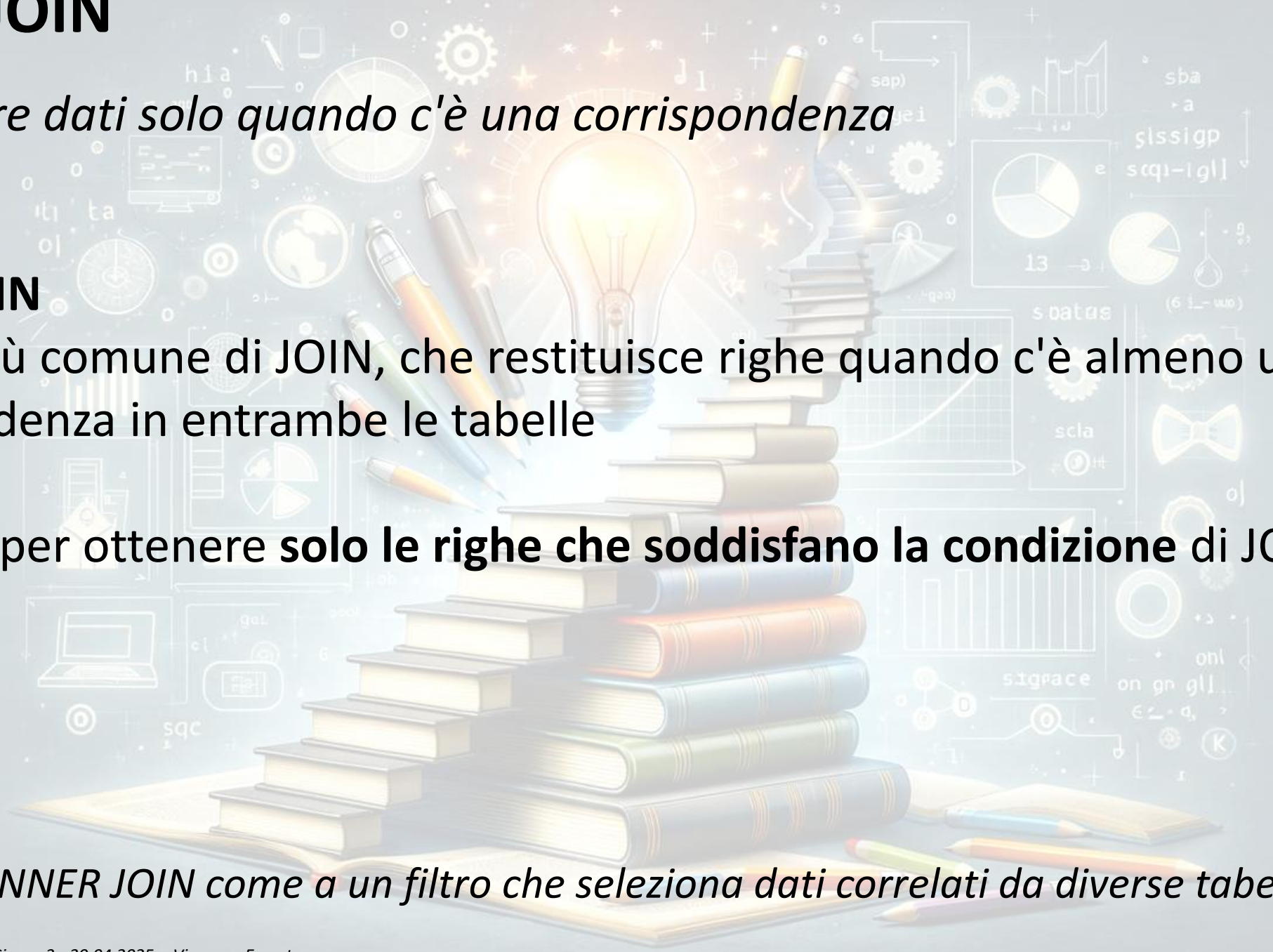
Esistono diversi tipi di JOIN:

- **INNER,**
- **LEFT/RIGHT OUTER,**
- **FULL OUTER, e**
- **CROSS UNION ALL**

Le JOIN sono essenziali per interrogazioni che richiedono un panorama completo dei dati.

INNER JOIN

Recuperare dati solo quando c'è una corrispondenza



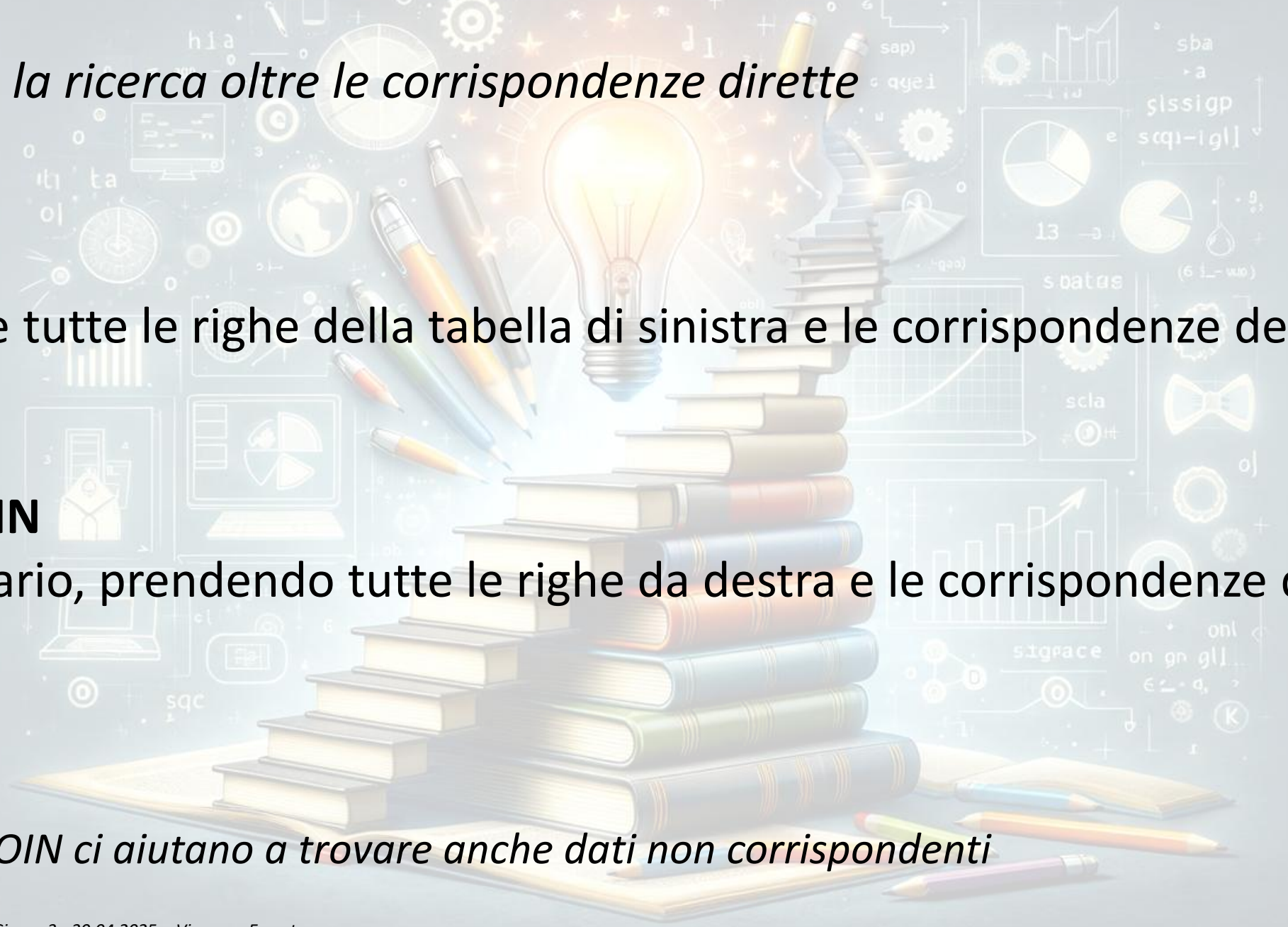
INNER JOIN
è il tipo più comune di JOIN, che restituisce righe quando c'è almeno una corrispondenza in entrambe le tabelle

Utilizzato per ottenere **solo le righe che soddisfano la condizione** di JOIN**.**

Pensare all'INNER JOIN come a un filtro che seleziona dati correlati da diverse tabelle.

LEFT e RIGHT OUTER JOIN

Estendere la ricerca oltre le corrispondenze dirette



LEFT JOIN

restituisce tutte le righe della tabella di sinistra e le corrispondenze della tabella di destra.

RIGHT JOIN

fa il contrario, prendendo tutte le righe da destra e le corrispondenze da sinistra.

Le OUTER JOIN ci aiutano a trovare anche dati non corrispondenti

Java - SQL

Prima parte: SQL

Giorni 04-06: 05-07.05.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer

FULL OUTER JOIN e CROSS JOIN

Combinazioni complete e prodotti cartesiani

FULL OUTER JOIN
combina i risultati sia di LEFT che di RIGHT JOIN.

CROSS JOIN
produce il prodotto cartesiano combinando ogni riga di una tabella con tutte le righe dell'altra.

Questi tipi di JOIN sono utili in casi specifici per analisi dettagliate o generazione di dati combinati.

Unire Risultati con UNION e UNION ALL

Combinare query mantenendo o duplicando i record

UNION

Combina i risultati di due o più query in un unico set di risultati, eliminando i duplicati.

UNION ALL

fa lo stesso, ma include tutti i duplicati, utile quando i duplicati sono necessari o l'eliminazione dei duplicati non è efficiente."

UNION e UNION ALL: Due modi per unire i dati, con o senza duplicati

Operatore IN

IN: Specificare un Elenco di Valori

IN permette di specificare una lista di valori possibili per una colonna.

Molto utile per filtri che richiedono molteplici corrispondenze

Utilizzato per effettuare il **semi_join**, anche se non è efficiente

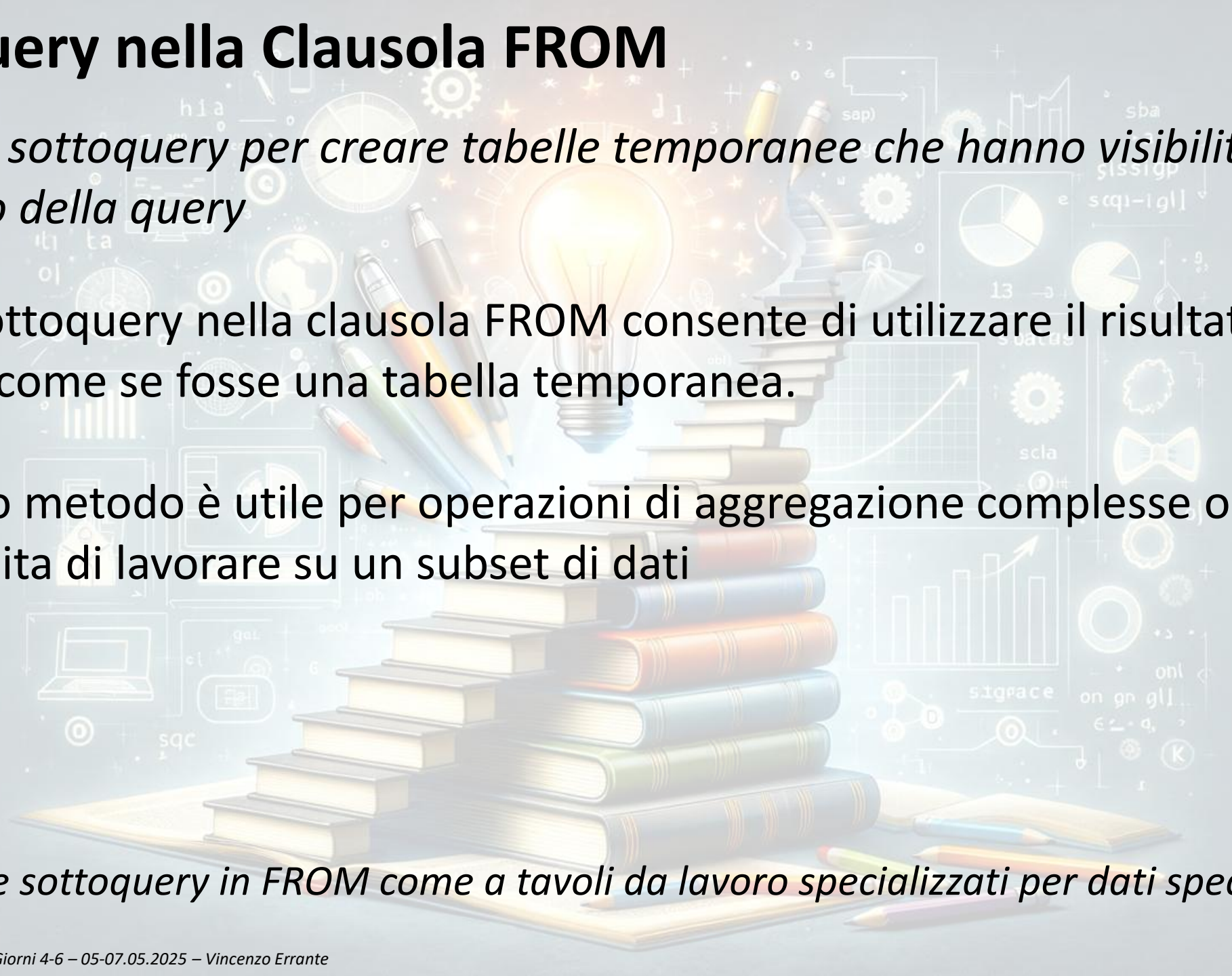
IN: il filtro multiplo per una ricerca precisa

Sottoquery nella Clausola FROM

Utilizzo di sottoquery per creare tabelle temporanee che hanno visibilità solo all'interno della query

- Una sottoquery nella clausola FROM consente di utilizzare il risultato di una query come se fosse una tabella temporanea.

- Questo metodo è utile per operazioni di aggregazione complesse o quando si necessita di lavorare su un subset di dati



Pensate alle sottoquery in FROM come a tavoli da lavoro specializzati per dati specifici.

Introduzione alle Viste (Views) in SQL

Una vista è una tabella virtuale basata sui risultati di una query SQL, ovvero di un subset di dati di una o più tabelle.

Si usano per:

- Semplificare query complesse e migliorare la leggibilità.
- Riutilizzare codice SQL e mantenere la logica di business in un unico posto.
- Aumentare la sicurezza limitando l'accesso ai dati.

Le viste sono potenti strumenti per organizzare, proteggere e semplificare l'interazione coi dati.

Creare una Vista in SQL

Utilizzo del comando **CREATE VIEW** per definire una nuova vista.
Specificare la query su cui si basa la vista

Esempio:
vista con i nomi e gli indirizzi email dei clienti che hanno un indirizzo emai

```
CREATE VIEW customer_email_list AS
SELECT first_name, last_name, email
  FROM customer
 WHERE email IS NOT NULL
```

Creare una vista è come impostare un filtro personalizzato sui nostri dati.

Utilizzo delle Tabelle Temporanee

Gestione efficace dei dati intermedi

- Le tabelle temporanee sono utilizzate per immagazzinare e manipolare insiemi di dati temporanei all'interno di sessioni di database.

- Sono particolarmente utili per operazioni complesse, dove si necessita di conservare i risultati intermedi.

- Le tabelle temporanee vengono automaticamente eliminate al termine della sessione di database.

Le tabelle temporanee sono come quaderni di appunti: utili per annotare i dati importanti durante l'elaborazione e facilmente scartabili una volta finito.

WITH (Common Table Expression)

Semplificare le Query Complesse per migliorare la Leggibilità

La clausola **WITH**, nota anche come Common Table Expression (**CTE**), è uno strumento potente in SQL che permette di creare set di risultati temporanei utilizzabili all'interno di una query complessa.

Le **CTE** offrono un modo per suddividere le query complesse in parti più semplici, migliorando la leggibilità e facilitando la manutenzione del codice.
Supporta la ricorsione in SQL, permettendo di Eseguire query ricorsive in modo più semplice

```
WITH CustomerRentals AS (
  SELECT customer_id
       , COUNT(rental_id) tot_rents
    FROM rental
   GROUP BY customer_id
)
SELECT c.first_name
     , c.last_name
     , r.tot_rents
  FROM CustomerRentals r
 INNER JOIN customer c
    ON r.customer_id = c.customer_id
 ORDER BY r.num_rentals DESC
 LIMIT 10;
```

La clausola WITH rende le tue query SQL più pulite, organizzate e facili da comprendere

Java - SQL

Prima parte: SQL

Giorni 07-08: 08-09.05.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer

Introduzione alle Stored Procedure

Automatizzare le Operazioni con Stored Procedure

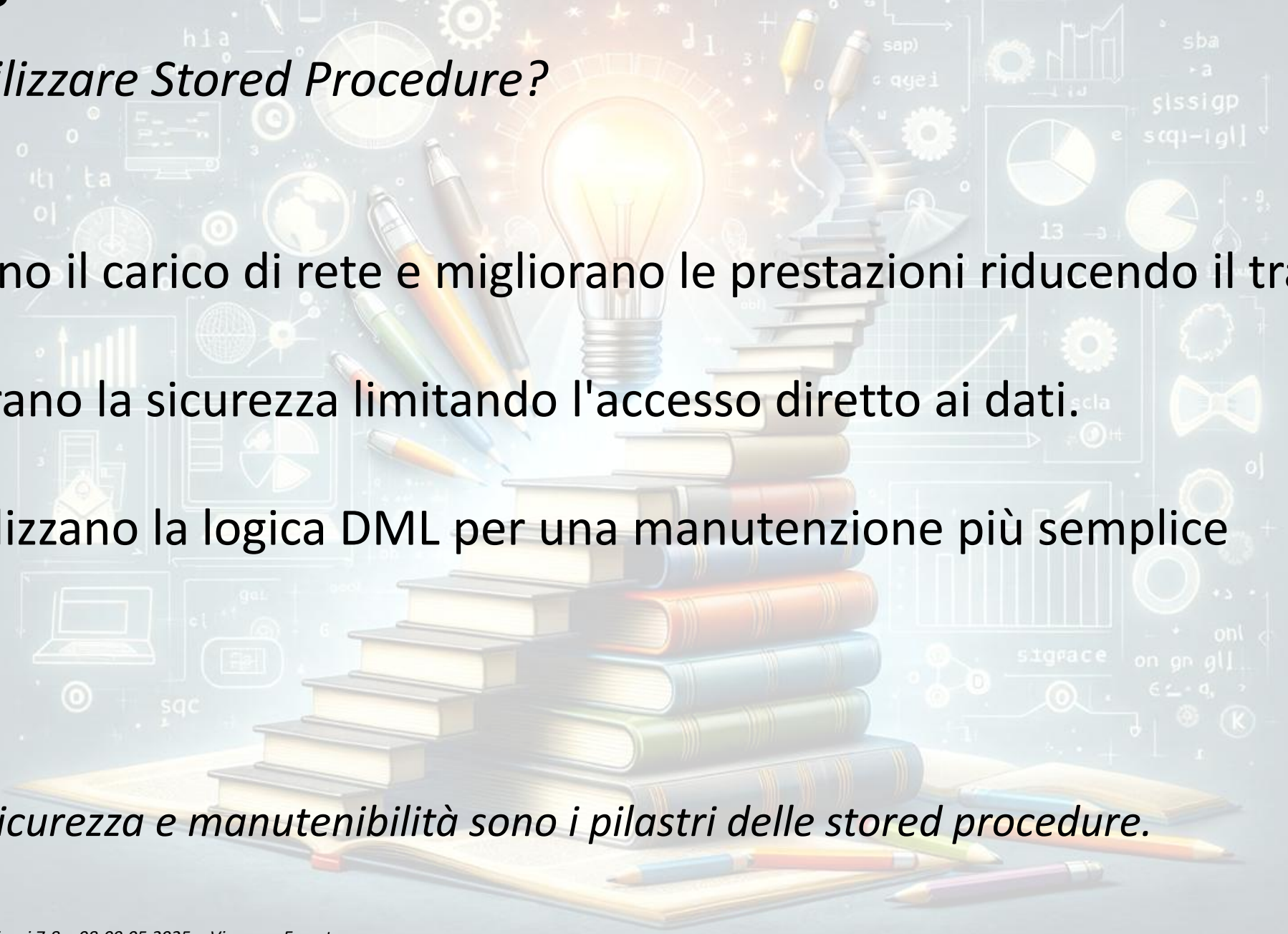
Una stored procedure è **una serie di istruzioni SQL (DML e DQL) che vengono salvate ed eseguite sul database**

Utili per automatizzare operazioni ricorrenti, garantire la consistenza dei dati, gestire transazioni e migliorare la sicurezza e l'efficienza.

Le stored procedure sono come ricette preimpostate per il tuo database

Vantaggi delle Stored Procedure

Perché Utilizzare Stored Procedure?



- Riducono il carico di rete e migliorano le prestazioni riducendo il traffico dati.
- Migliorano la sicurezza limitando l'accesso diretto ai dati.
- Centralizzano la logica DML per una manutenzione più semplice

Efficienza, sicurezza e manutenibilità sono i pilastri delle stored procedure.

Creare una Store Procedure

Utilizzo del comando **CREATE PROCEDURE** per definire una nuova stored procedure.

Esempio:
una procedura per inserire un nuovo film nel database.

```
  CREATE PROCEDURE AddNewFilm(title VARCHAR(255), description TEXT, release_year YEAR,
language_id INT, rental_duration INT, rental_rate DECIMAL(4,2), length INT,
replacement_cost DECIMAL(5,2), rating ENUM('G', 'PG', 'PG-13', 'R', 'NC-17'))
  BEGIN
    INSERT INTO film (title, description, release_year, language_id, rental_duration,
rental_rate, length, replacement_cost, rating)
    VALUES (title, description, release_year, language_id, rental_duration, rental_rate,
length, replacement_cost, rating);
  END
```

Introduzione alle transazioni

Gestione sicura e coerente dei dati

- Una transazione in SQL è una serie di operazioni che vengono trattate come un'unità singola
- Le transazioni assicurano che tutte le operazioni siano completate con successo oppure che nessuna venga eseguita (tutto o niente).
- Segue il principio **ACID**: Atomicità, Consistenza, Isolamento, Durabilità
- Le transazioni non sono nidificabili

Le transazioni sono essenziali per mantenere l'integrità del database

Comandi delle transazioni

START TRANSACTION

Definisce l'inizio di una transazione (se *Autocommit=1*)

COMMIT

conferma tutte le operazioni nella transazione.

ROLLBACK

Annulla tutte le operazioni, ritornando allo stato precedente,

SAVEPOINT

Permette di definire un punto intermedio nella transazione a cui è possibile tornare

Esempio Pratico di transazione

Utilizzo dei comandi **START TRANSACTION, COMMIT, ROLLBACK**

Esempio:

Aggiornare le informazioni di un cliente e registrare una transazione di pagamento

```
START TRANSACTION;
INSERT INTO payment (customer_id, staff_id, rental_id, amount, payment_date)
VALUES (1, 1, 1, 3.99, NOW());
UPDATE customer
   SET last_update = NOW()
 WHERE customer_id = 1;

-- Se tutto va bene
COMMIT;

-- In caso di errore
ROLLBACK;
```

Gestione delle Eccezioni in SQL

Assicurare la Robustezza con la Gestione delle Eccezioni

- Le **eccezioni** in SQL vengono usate per gestire errori o comportamenti inaspettati durante l'esecuzione delle **stored procedures**.
- Un **handler di eccezioni** cattura errori specifici o una classe di errori, permettendo di eseguire codice alternativo.
- Usate comunemente con transazioni per garantire l'integrità dei dati

Gli handler di eccezioni sono come reti di sicurezza che proteggono i vostri dati dagli errori.

Introduzione alle Funzioni in MySQL

Cosa sono le Funzioni in MySQL

Le funzioni in MySQL sono **blocchi di codice SQL** che possono essere salvati e riutilizzati per eseguire operazioni specifiche e restituire un valore singolo.
Queste funzioni permettono di incapsulare logiche complesse in modo che possano essere richiamate con facilità **in qualsiasi punto** del codice SQL.
Le funzioni sono simili alle procedure memorizzate (stored procedures), ma differiscono principalmente per il fatto che le funzioni restituiscono sempre un valore e **possono essere utilizzate in espression**i, mentre le stored procedures possono eseguire operazioni più complesse e non restituiscono necessariamente un valore.
Uso tipico: **Functions per DQL, Procedure per DML**

Le funzioni in MySQL sono particolarmente utili per riutilizzare il codice e mantenere una logica centralizzata.

Creazione di una Funzione

Sintassi di base

La sintassi di base per creare una funzione in MySQL è la seguente:

```
CREATE FUNCTION function_name (par1 tipo1, par2 tipo2,…)
RETURNS data_type
[ DETERMINISTIC/NOT DETERMINISTIC | NOSQL | READS SQL DATA ]
BEGIN
    -- corpo della funzione
    RETURN valore;
END;
```

Le direttive di una funzione possono essere deterministiche, sul linguaggio SQL, e altre direttive.

Esempio: Calcolare la Durata del Noleggio

Funzione per calcolare la durata del noleggio in giorni

Utilizziamo il database `sakila` per creare una funzione che calcola la durata del noleggio di un film in giorni.

```
CREATE FUNCTION rental_duration (rental_date DATE, return_date DATE)
RETURNS INT
NOT DETERMINISTIC
BEGIN
    RETURN DATEDIFF(ifnull(return_date,datetime), rental_date);
END;
```

Questa funzione utilizza la funzione DATEDIFF per calcolare la differenza in giorni tra due date.

Deterministic e Non-deterministic

Deterministicità delle Funzioni

Una funzione è deterministica se restituisce sempre lo stesso risultato per gli stessi input.

È non-deterministica se il risultato può variare.
La direttiva **DETERMINISTIC** o **NOT DETERMINISTIC** viene utilizzata per specificare questo comportamento.

Questo perché l'ottimizzatore utilizza la cache per le funzioni deterministiche (e quindi sono più veloci)

Le funzioni deterministiche possono essere ottimizzate meglio dal motore di MySQL.

Funzione con READS SQL DATA

Uso della direttiva READS SQL DATA

La direttiva READS SQL DATA indica che la funzione legge dati dal database ma non modifica i dati. Questa è importante per ottimizzare e proteggere il database.

```
CREATE FUNCTION film_in_stock (p_film_id INT)
RETURNS INT
READS SQL DATA
BEGIN
    DECLARE cnt INT;
    SELECT COUNT(*) INTO cnt
      FROM inventory
     WHERE film_id = p_film_id;
    RETURN cnt;
END;
```

Questa funzione conta il numero di copie di un film presenti nel magazzino.

Funzione con NO SQL

Creazione di funzioni che non interagiscono con SQL

La direttiva **NO SQL** indica che la funzione non esegue alcuna operazione SQL.

Questa è utile per funzioni puramente computazionali.

```
CREATE FUNCTION add_numbers (a INT, b INT)
RETURNS INT
NO SQL
BEGIN
    RETURN a + b;
END;
```

Le funzioni NO SQL sono semplici e non influenzano le operazioni del database.

Esempio: Funzione per Calcolare il Profitto

Funzione che calcola il profitto basato sui pagamenti

Esempio di funzione che calcola il profitto totale ottenuto dai noleggi di un film.

```
CREATE FUNCTION film_profit (p_film_id INT)
RETURNS DECIMAL(10,2)
READS SQL DATA
BEGIN
   DECLARE profit DECIMAL(10,2);
   SELECT SUM(amount) INTO profit
     FROM payment
    INNER JOIN rental ON payment.rental_id = rental.rental_id
   WHERE rental.inventory_id IN (SELECT inventory_id FROM inventory WHERE film_id = p_film_id);
   RETURN profit;
END;
```

Questa funzione utilizza una sottoquery per calcolare il profitto totale di un film specifico.

Uso di Parametri nelle Funzioni

Passaggio e gestione dei parametri

I parametri sono utilizzati per passare valori alle funzioni. Possono essere di vari tipi di dati e devono essere specificati al momento della creazione della funzione.

```
CREATE FUNCTION get_actor_name (p_actor_id INT)
RETURNS VARCHAR(255)
READS SQL DATA
BEGIN
    DECLARE name VARCHAR(255);
    SELECT CONCAT(first_name, ' ', last_name) INTO name
    FROM actor
    WHERE actor_id = p_actor_id;
    RETURN name;
END;
```

Questo esempio mostra come concatenare il nome e il cognome di un attore utilizzando i parametri.

Errore e Gestione delle Eccezioni

Gestione degli errori nelle funzioni

MySQL consente **la gestione degli errori** nelle funzioni utilizzando il comando **DECLARE HANDLER**, ma con limitazioni:

Non è possibile modificare lo stato del database (ad esempio, eseguire operazioni come INSERT, UPDATE o DELETE).

Ad esempio, non è possibile aggiornare una tabella di log

```
CREATE FUNCTION safe_divide (a INT, b INT)
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
  -- Restituisce NULL in caso di errore
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
  RETURN NULL;

  -- Divisione, genera errore se b = 0
  RETURN a / b;
END;
```

Questa funzione restituisce NULL se il divisore è zero per evitare errori di divisione.

Testing e Debugging delle Funzioni

Come testare e fare debug delle funzioni

È importante testare le funzioni per garantire che funzionino correttamente.

Esempi di test includono:

```
SELECT rental_duration('2023-01-01', '2023-01-10’);
SELECT film_in_stock(1);

SELECT add_numbers(5, 10);

SELECT film_profit(1);

SELECT get_actor_name(1);

SELECT safe_divide(10, 2);
SELECT safe_divide(10, 0);
```

Testare le funzioni con vari input aiuta a garantire la loro correttezza e affidabilità.

Java - SQL

Prima parte: SQL

Giorno 09: 12.05.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer

Introduzione ai Trigger

Automatizzare le Azioni con i Trigger

- Un **trigger è un tipo di stored procedure** che viene automaticamente eseguita in risposta a determinati eventi su una tabella, come inserimenti, aggiornamenti o cancellazioni (Istruzioni DML).

- Utili per mantenere l'integrità dei dati, eseguire la manutenzione automatica del database, e registrare le modifiche ai dati

I trigger agiscono come guardiani automatici del tuo database

Tipi di Trigger

BEFORE vs AFTER Triggers

BEFORE TRIGGER

- Eseguito prima che l'operazione (**INSERT, UPDATE, DELETE**) sia eseguita.
- Utile per validare o modificare i dati prima che siano definitivamente scritti nel database.

AFTER TRIGGER

- Eseguito dopo che l'operazione è completata.
- Usato per eseguire operazioni di pulizia, aggiornare tabelle di log, o sincronizzare cambiamenti in tabelle separate.

Il **trigger** è sempre relativo a **una operazione** DML di **una sola tabella**

Scegliere il tipo di trigger in base all'azione desiderata e al momento dell'esecuzione

Sintassi di base

La sintassi di base per creare un trigger include la definizione del momento dell'attivazione (**BEFORE** o **AFTER**), il tipo di evento (**INSERT, UPDATE, DELETE**) e la tabella su cui opera

Esempio:

Creare un BEFORE INSERT trigger per validare i dati prima dell'inserimento in una tabella

```
CREATE TRIGGER check_mail_before_insert
BEFORE INSERT ON customer
FOR EACH ROW
BEGIN
    IF NEW.email IS NULL THEN
       SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Email cannot be null';
  END IF;
END;
```

Considerazioni Importanti sui Trigger

Uso Responsabile dei Trigger

- I trigger possono rendere il debugging più complesso, poiché operano in background e possono modificare i dati in modo non immediatamente evidente
- Un uso eccessivo dei trigger può influenzare le prestazioni del database, specialmente in sistemi con un alto volume di transazioni
- Importante documentare chiaramente tutti i trigger e le loro funzioni all'interno del database

Usare i trigger con consapevolezza per evitare effetti indesiderati

Java - SQL

Prima parte: SQL

Giorno 10: 13.05.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer

ETL: CREATE TABLE LIKE

Clonare la Struttura di una Tabella

Il comando CREATE TABLE LIKE crea una nuova tabella basandosi sulla struttura di una tabella esistente.
"Copierà la struttura delle colonne, gli indici e le definizioni, ma non i dati."

Esempio: Creare una tabella per il backup per i clienti.

```
-- Creare una nuova tabella basata su 'customer’
   CREATE TABLE customer_backup LIKE customer;
```

CREATE TABLE LIKE è un modo rapido per replicare la struttura di una tabella senza copiarne i contenuti

ETL: CREATE TABLE AS

Creare una Nuova Tabella dai Risultati di una Query

Il comando **CREATE TABLE AS** crea una nuova tabella popolandola con i risultati di una query.
Utilizzato per creare tabelle per reporting, analisi o backup di dati specifici.

Esempio: Creare una tabella contenente solo i clienti attivi.

```
-- Copiare clienti specifici in una nuova tabella
CREATE TABLE active_customers AS SELECT * FROM customer WHERE active = 1;
```

CREATE TABLE AS consente di trasformare i risultati delle query in nuove tabelle persistenti.

ETL: INSERT INTO .. SELECT

Inserire con un Singolo Comando partendo da una query

INSERT INTO .. SELECT , inserisce il contenuto di una query in una tabella. L'unica condizione da verificare è che i valori dei campi delle colonne della SELECT rispecchino il tipo dei campi indicati nella query dopo il nome della tabella.

Esempio: Inserire i dati da un backup precedente

```
-- Inserire massivamente dati in film da dati presenti su film_bkp
INSERT INTO film (title, description, release_year)
SELECT title, descritpion, release_year
  FROM film_bkp
 WHERE last_update < str_to_date('01012010','%D%M%Y');
```

INSERT INTO.. SELECT, permette di inserire massivamente dati partendo da una query

ETL: REPLACE INTO

Aggiornare o Inserire con un Singolo Comando

REPLACE INTO funziona come un INSERT, ma se trova una chiave primaria o univoca già esistente, prima elimina la riga esistente e poi inserisce la nuova. Adatto per situazioni in cui è necessario aggiornare i dati esistenti o inserire nuovi dati senza errori di chiavi duplicate.

Esempio: Aggiornare o inserire un record nella tabella film.

```
-- Aggiornare o inserire un record nella tabella film
REPLACE INTO film (film_id, title, description, release_year)
VALUES (1000, 'New Film', 'A new film description', 2023);
```

REPLACE INTO combina la flessibilità dell'inserimento con la sicurezza dell'aggiornamento

ETL: UPSERT

Combinare INSERT e UPDATE in un'unica Operazione

In MySQL, si realizza con il comando **INSERT ... ON DUPLICATE KEY UPDATE**.
Questo comando tenta di inserire una nuova riga; se una riga con la stessa chiave primaria o unica esiste già, esegue un'operazione di UPDATE sulla riga esistente.

Esempio: Aggiornare o inserire un record nella tabella film.

```
INSERT INTO customer (customer_id, first_name, last_name, email)
VALUES (100, 'John', 'Doe', 'john.doe@example.com') AS new_row
ON DUPLICATE KEY
UPDATE first_name=new_row.first_name, last_name=new_row.last_name, email=new_row.email;
```

A differenza di REPLACE INTO, che elimina e ricrea una riga, UPSERT aggiorna semplicemente i valori esistenti, mantenendo l'integrità dei dati e l'id originali. Questo lo rende più efficiente e sicuro per situazioni dove l'identità della riga è cruciale

Introduzione agli Indici

Velocizzare le Query con gli Indici

- Gli indici in SQL sono strutture dati che migliorano la velocità delle operazioni di ricerca sulle tabelle (*in generale delle Query*).
- Funzionano come gli indici di un libro: permettono al database di trovare rapidamente i dati senza dover scorrere tutta la tabella.
- Sono particolarmente utili in tabelle di grandi dimensioni e in query che richiedono frequenti operazioni di ricerca, filtro o ordinamento.

Gli indici sono come quelli di un libro, aiutano a trovare rapidamente ciò che stiamo cercando

Creare e Gestire Indici

Creare un indice:

CREATE INDEX nome_indice ON nome_tabella(nome_colonna1, [nome_colonna2] …);

- E’ importante considerare quali colonne indicizzare, basandosi sull'utilizzo nelle query.
- Gli indici su colonne frequentemente ricercate o filtrate migliorano le prestazioni.
- Gestire gli indici comprende la rimozione (**`DROP INDEX`**) quando non sono più necessari o influenzano negativamente le prestazioni.
- Eccessivi indici **possono rallentare le operazioni di inserimento e aggiornamento**.

Usare gli indici in modo strategico per migliorare l'efficienza.

Creazione e rimozione di un Indice

- Utilizzo del comando **CREATE INDEX** per creare un indice

  Esempio:
  Creare un indice sulla colonna **title** nella tabella **film**

```
CREATE INDEX idx_title ON film(title);
```

- Utilizzo del comando **DROP INDEX** per rimuovere un indice

  Esempio:
  Rimuovere l'indice **idx_title** dalla tabella **film**

```
DROP INDEX idx_title ON film;
```

Creazione di indici composti

Ottimizzazione creando Indici su più colonne

Esempio:

Creare un indice che copre **title** e **release_year** nella tabella **film**

```
CREATE INDEX idx_title_release_year ON film(title, release_year);
```

Un indice composto può essere utile per ottimizzare il piano di esecuzione

Manutenzione degli indici

Assicurare l'efficienza degli indici nel tempo

- **Monitoraggio degli Indici**: Verificare regolarmente l'utilizzo e le prestazioni degli indici.
- **Ricostruzione e Ristrutturazione**: Ricostruire o ristrutturare gli indici può essere necessario per ottimizzare le prestazioni, specialmente dopo grandi cambiamenti nei dati.
- **Pulizia degli Indici**: Rimuovere gli indici non utilizzati o inefficaci per ridurre il sovraccarico di spazio e migliorare le prestazioni di scrittura.
- **Bilanciamento**: Considerare il rapporto tra operazioni di lettura e scrittura nel database per decidere la strategia di indicizzazione.

La manutenzione degli indici è un processo continuo per garantire prestazioni ottimali del DB

Utilizzare gli indici in modo efficace

PRO

- **Miglioramento delle prestazioni delle query**: Riducono significativamente il tempo di ricerca, soprattutto in tabelle di grandi dimensioni.
- **Efficienza nei join**: Migliorano le prestazioni dei join, in particolare nelle query complesse che coinvolgono più tabelle.
- **Supporto alle clausole ORDER BY e GROUP BY**: Aiutano a velocizzare operazioni di ordinamento e raggruppamento.

CONTRO

- **Costo nelle operazioni di scrittura**: Ogni inserimento, aggiornamento o cancellazione richiede l'aggiornamento degli indici, rallentando queste operazioni.
- **Uso dello spazio su disco**: Gli indici occupano spazio aggiuntivo sul disco.
- **Complessità di manutenzione**: Richiedono una corretta manutenzione e una strategia di utilizzo ponderata

Gli indici sono alleati per le prestazioni delle query, ma devono essere utilizzati con giudizio

Ottimizzazione delle query

Scrivere Query Efficaci

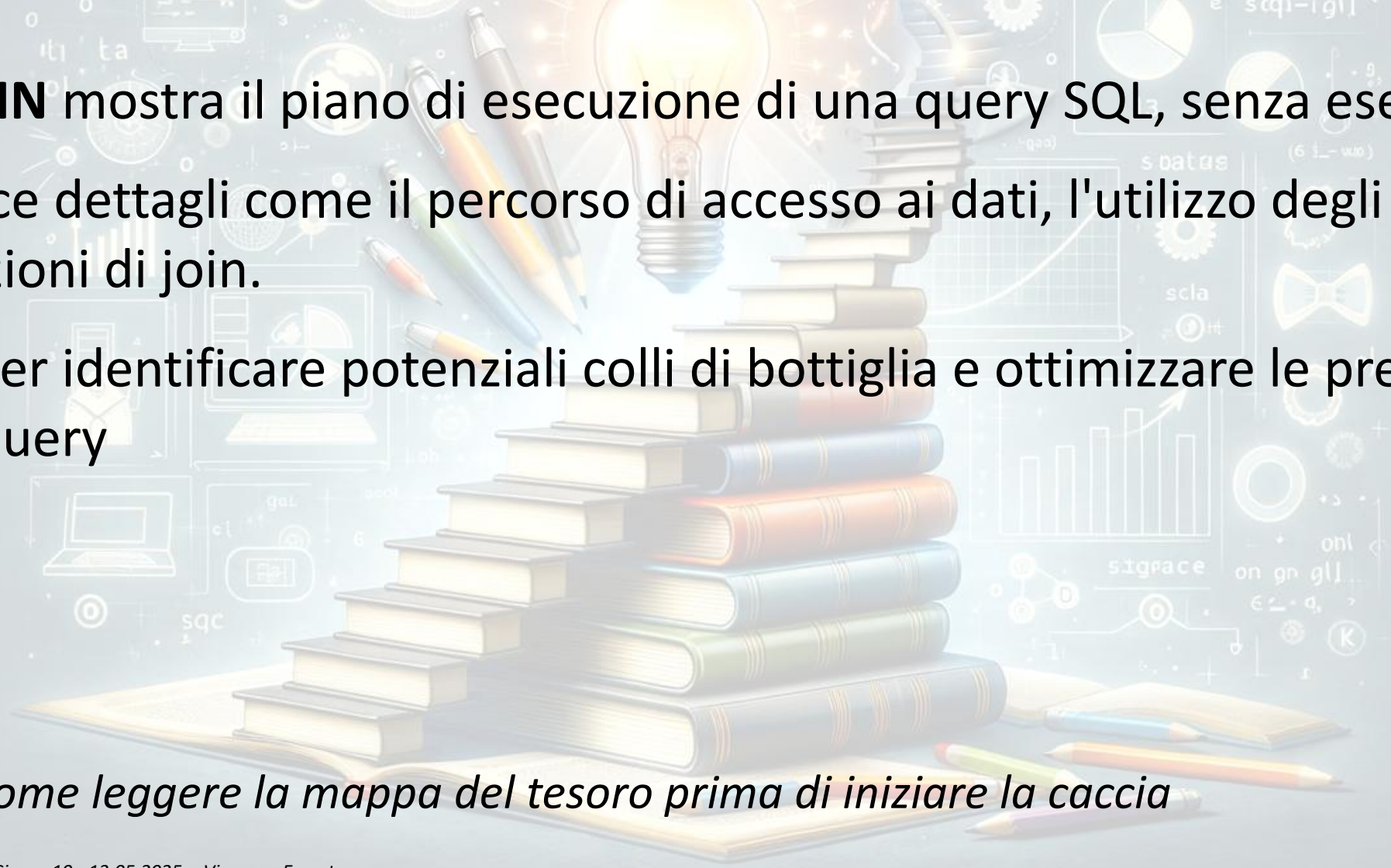
- Ottimizzare le query significa ridurre il tempo di esecuzione e migliorare l'efficienza nell'utilizzo delle risorse del database.
- Include tecniche come la scelta appropriata di indici, la ristrutturazione delle query per sfruttare meglio i **piani di esecuzione** del database e la riduzione delle operazioni onerose come **scan completi** di tabelle o join inefficienti.
- Usare **EXPLAIN** per analizzare il piano di esecuzione di una query, identificare i colli di bottiglia e apportare le modifiche necessarie.

Ottimizzare le query per ottenere il massimo dal database

Il Piano di esecuzione

Analizzare le Query Prima dell'Esecuzione

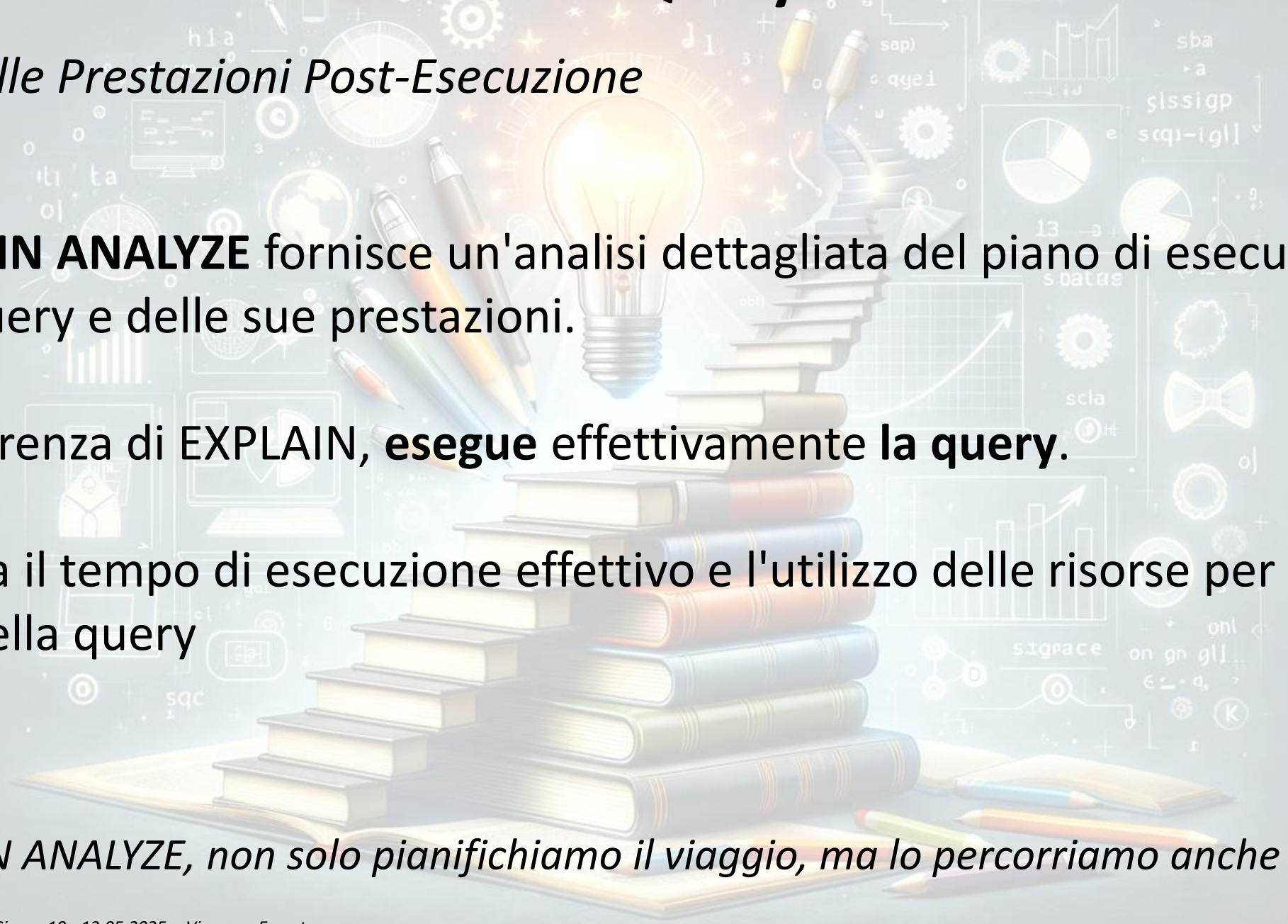
- **EXPLAIN** mostra il piano di esecuzione di una query SQL, senza eseguirla.
- Fornisce dettagli come il percorso di accesso ai dati, l'utilizzo degli indici e le operazioni di join.
- Utile per identificare potenziali colli di bottiglia e ottimizzare le prestazioni delle query



EXPLAIN è come leggere la mappa del tesoro prima di iniziare la caccia

Approfondimento dell’analisi Query

Analisi delle Prestazioni Post-Esecuzione



- **EXPLAIN ANALYZE** fornisce un'analisi dettagliata del piano di esecuzione di una query e delle sue prestazioni.
- A differenza di EXPLAIN, **esegue** effettivamente **la query**.
- Mostra il tempo di esecuzione effettivo e l'utilizzo delle risorse per ciascuna fase della query

Con EXPLAIN ANALYZE, non solo pianifichiamo il viaggio, ma lo percorriamo anche