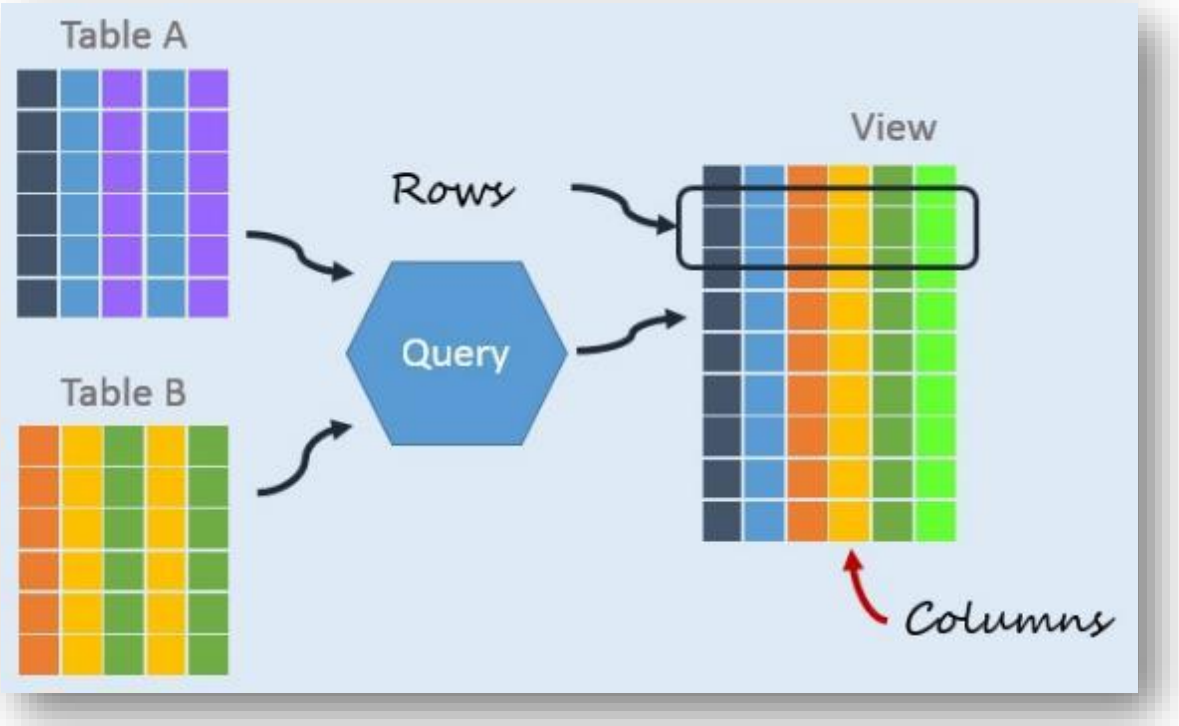
SQL

Structured Query Language



VIEW

viste utente

VIEW (Viste utente)

- le VIEW sono ***tabelle virtuali*** il cui contenuto (colonne e righe) è definito da una query
- le VIEW sono normalmente utilizzate per:
  - analizzare, semplificare e personalizzare la visualizzazione del database per un utente
  - come meccanismo di sicurezza grazie al quale è possibile consentire agli utenti di accedere ai dati tramite una vista, senza concedere loro le autorizzazioni di accesso alle tabelle di base sottostanti
  - fornire un'interfaccia compatibile con le versioni precedenti tramite la quale è possibile emulare una tabella precedente il cui schema è stato modificato

CREATE VIEW


```
CREATE VIEW <nome_vista> AS
(SELECT <lista_campi>
 FROM <lista_tabelle>
 WHERE <condizione>);
```

esempio

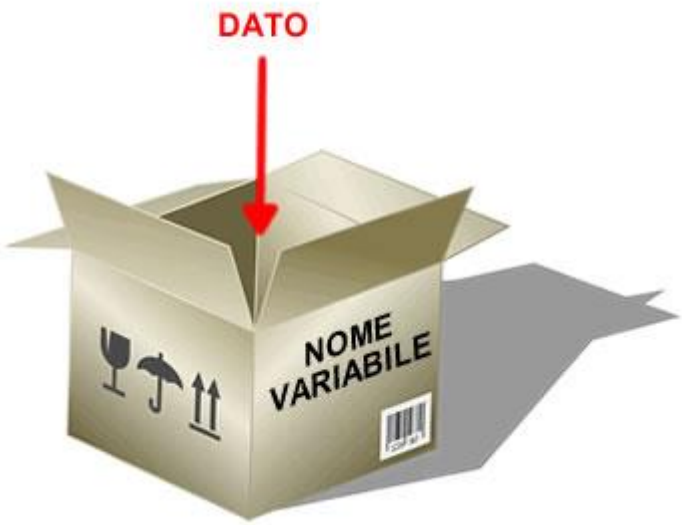

```
-- Creazione vista
CREATE VIEW StudentiConClasse AS
SELECT studente.cognome, classe.descrizione
FROM studente INNER JOIN classe
     ON studente.IDclasse = classe.ID
ORDER BY studente.cognome;

-- Utilizzo vista
SELECT * FROM StudentiConClasse;
```

esempio


```
CREATE VIEW filmConGenere AS
(SELECT film.titolo, genere.nome
        FROM film INNER JOIN genere
        ON film.IDgenere=genere.ID);

SELECT * FROM filmConGenere;
```



VARIABILI

- è possible memorizzare ***valori*** in variabili
- le variabili sono globali alle istruzioni SQL
- gli identificatori iniziano con @

- ***esempio***:

```
SELECT @numeroFilm:= count(*) FROM film;
```

recupero indice ultimo inserimento

A. Ferrari

- nel caso di tabelle con ***chiavi primarie in autoincremento*** dopo l'inserimento di una nuova entità può essere necessario recuperare il valore dell'indice che è stato assegnato e assegnarlo a una variabile
- la funzione **`LAST_INSERT_ID()`** restituisce il valore che è stato assegnato alla chiave primaria
- ***esempio***:
- **`SET @cod_attore = LAST_INSERT_ID();`**

INDICI

indici

- un indice è una ***struttura dati*** aggiuntiva realizzata allo scopo di ***velocizzare*** l'accesso ai dati di una tabella
- l'indice costruito su un ***campo*** consente di ***ridurre*** l'insieme dei dati da leggere per completare la ricerca su quel campo
- la creazione di un indice provoca un ***aumento*** dell'uso della memoria di massa
- gli indici rendono inoltre più ***lente*** le operazioni di inserimenti e modifica (update) dei dati
  - necessario aggiornare anche gli indici

creazione indice

- *sintassi differente mariadb e mysql* ☹
- *mariadb*

```
create index <nome_indice>
on <tabella>.<campo>
```

- mysql

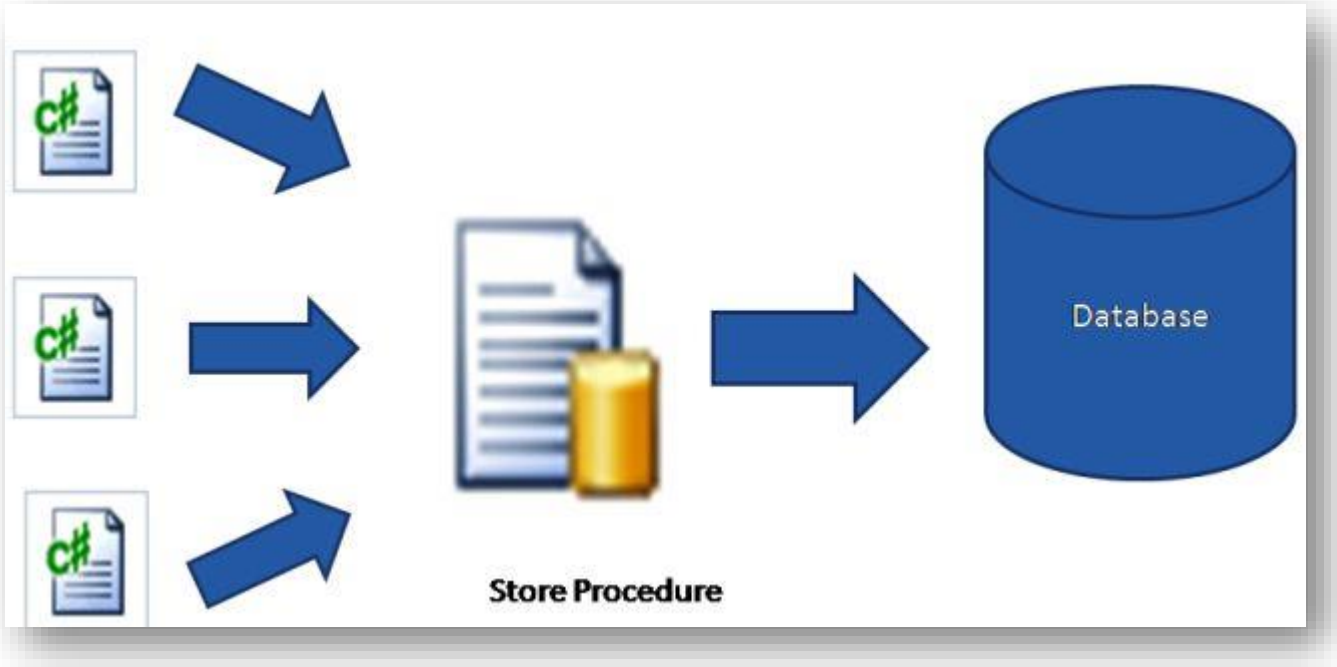
```
alter table <tabella>.<campo>
add index <nome_indice>
```

- *esempio:*

```
create index idx_attore_nome on cinema.attore(nome);
```

- *eliminazione:*

```
alter table cinema.attore
drop index idx_attore_nome;
```



stored procedure

stored procedure

- le stored procedures sono presenti in MySQL dalla versione 5.0
- sono ***gruppi di istruzioni*** SQL (o scritte in altri linguaggi) ***memorizzati*** nel motore database e utilizzabili dai client che accedono al database
- ogni procedura è identificata da un nome ed è attribuita ad uno specifico database

stored procedure: vantaggi

- aumento della ***velocità*** di esecuzione del codice SQL e quindi delle performance generali delle applicazioni (sono compilate una sola volta)
- aumento della ***leggibilità*** e della ***portabilità*** del codice e quindi della scalabilità delle applicazioni

sintassi


```
CREATE PROCEDURE nome ([parametro[,...]])
[SQL SECURITY { DEFINER | INVOKER }] corpo
//parametri:
[ IN | OUT | INOUT ] nomeParametro tipo
```

- ogni procedura può avere uno o più ***parametri***, ciascuno dei quali è formato da un nome, un tipo di dato e l'indicazione se trattasi di parametro di input, di output o entrambi (se manca l'indicazione, il parametro è considerato di input)
- la clausola SQL SECURITY stabilisce se, al momento dell'esecuzione, la procedura utilizzerà i permessi dell'utente che la sta eseguendo o quelli dell'utente che l'ha creata (il default è DEFINER)

esempio


```
DELIMITER $$
CREATE PROCEDURE nomeProc
      (IN param1 INT, OUT param2 INT)
SELECT COUNT(*) INTO param2 FROM tabella
WHERE campo1 = param1; $$
```

- crea una procedura chiamata 'nomeProc' nel database in uso
  - la procedura usa un parametro in input e uno in output, entrambi interi
  - effettua il conteggio delle righe in tabella in cui il valore di campo1 corrisponde al primo parametro
  - il risultato della query viene memorizzato nel secondo parametro attraverso la clausola INTO

delimiter

- il comando DELIMITER serve per modificare il normale delimitatore delle istruzioni, che sarebbe il punto e virgola
- la stored procedure contiene più istruzioni, al suo interno il punto e virgola viene utilizzato più volte
  - di conseguenza, se vogliamo riuscire a memorizzare la procedura, dobbiamo comunicare al server che il delimitatore è un altro; in caso contrario, al primo punto e virgola penserebbe che la nostra CREATE sia terminata
- classici delimitatori: // o $$

stored procedure: esecuzione

A. Ferrari

```
CALL nomeProc (5, @a);
SELECT @a;
```

- con CALL si effettua la chiamata della procedura passando il valore 5 come parametro di input e la variabile **@a** come parametro di output, nel quale verrà memorizzato il risultato
- l'istruzione SELECT successiva visualizza il valore della variabile **@a** dopo l'esecuzione

- nell'esempio la stored procedure conteneva una istruzione **`SELECT`**
- è possibile creare procedure che contengono sintassi complesse comprendenti più istruzioni: in pratica, dei veri e propri script, con la possibilità di controllare il flusso attraverso vari costrutti **`(IF, CASE, LOOP, WHILE, REPEAT, LEAVE, ITERATE)`**

esempio

- definizione di procedura che restituisce il numero di fatture relative a un cliente (codice)

```
DELIMITER //
CREATE PROCEDURE NumFattureCliente
(IN codCliente INT, OUT numFatture INT)
      SELECT COUNT(Fatture.cod) INTO numFatture
      FROM Fatture WHERE Fatture.cod = codCliente;
//
```

- chiamata (esecuzione) della procedura [cliente con codice 3]

```
CALL NumFattureCliente (3, @num);
SELECT @num;
```

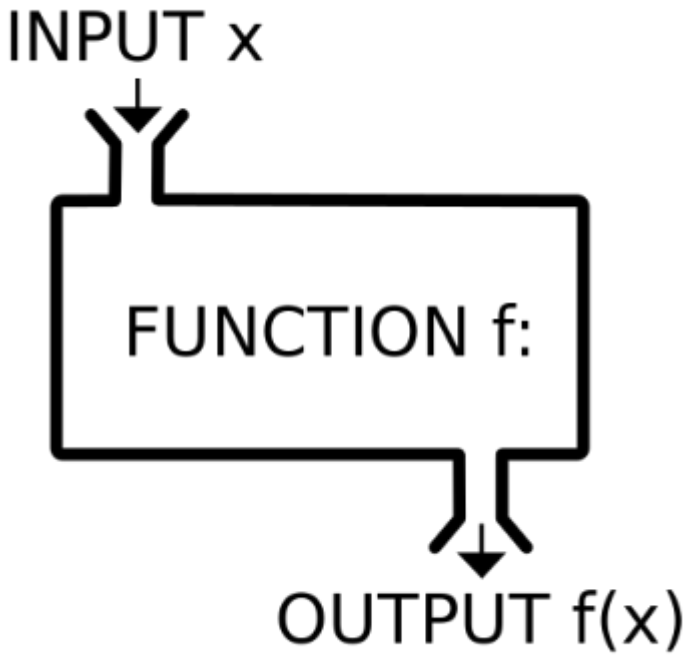
esempio

- definizione procedura che elimina cliente e relative fatture

```
DELIMITER //
CREATE PROCEDURE EliminaFattureECliente
(IN codCliente INT)
BEGIN
 DELETE FROM Fatture WHERE Fatture.cod = codCliente;
 DELETE FROM Clienti WHERE Clienti.cod = codCliente;
END
//
```

- esecuzione della procedura

```
CALL EliminaFattureECliente (3);
```



stored function

stored function

- le stored functions sono simili alle stored procedures
- restituiscono un valore e non possono restituire result set

A SQL result set is a set of rows from a database, as well as metadata about the query such as the column names, and the types and sizes of each column.

sintassi

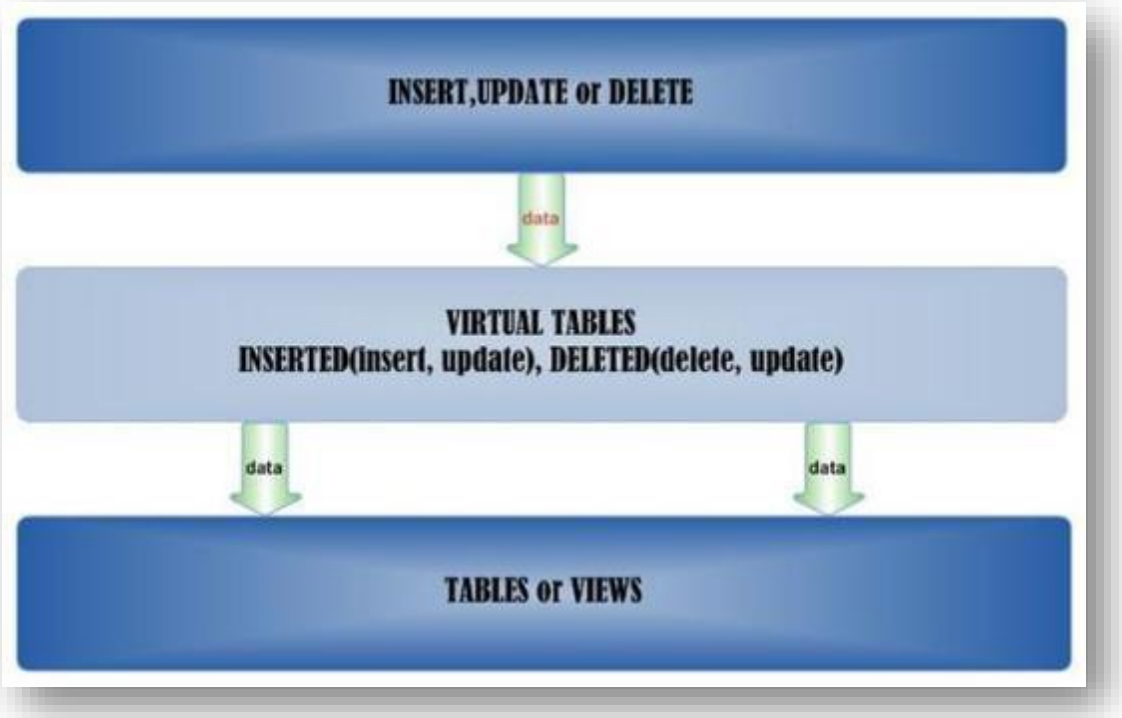

```
CREATE FUNCTION nome ([parametro[,...]])
RETURNS tipo
[SQL SECURITY { DEFINER | INVOKER }] corpo
//parametri:
nomeParametro tipo
```

- RETURNS specifica che tipo di dato la funzione restituisce
- i parametri sono solo in input

esempio function

A. Ferrari

```
-- Crea una funzione
DELIMITER //
CREATE FUNCTION TotaleFattureCliente (cod int)
RETURNS DECIMAL(10,2)
BEGIN
  DECLARE somma DECIMAL(10,2);
  SELECT SUM(importoFattura) INTO somma
         FROM Fatture WHERE Fatture.Cod = cod;
RETURN somma;
END
//
-- Richiama la funzione
SELECT TotaleFattureCliente(3);
```


INSERT,UPDATE or DELETE
data
VIRTUAL TABLES
INSERTED(insert, update), DELETED(delete, update)
data
data
TABLES or VIEWS

trigger

trigger

- procedure che vengono eseguite in maniera ***automatica*** al verificarsi di un determinato evento (esempio la cancellazione di un record di una tabella)
- permettono di specificare e mantenere ***vincoli di integrità*** anche complessi
- i trigger sono «nascosti», non è possibile attivarli esplicitamente e non hanno parametri
- quando definiamo un trigger, stabiliamo per quale ***evento*** deve essere attivato e se deve essere eseguito ***prima*** o ***dopo*** tale evento:
  - **`BEFORE INSERT`**
  - **`BEFORE UPDATE`**
  - **`BEFORE DELETE`**
  - **`AFTER INSERT`**
  - **`AFTER UPDATE`**
  - **`AFTER DELETE`**

sintassi


```
CREATE
    [DEFINER = { utente | CURRENT_USER }]
    TRIGGER nome tipo
    ON tabella FOR EACH ROW istruzioni
```

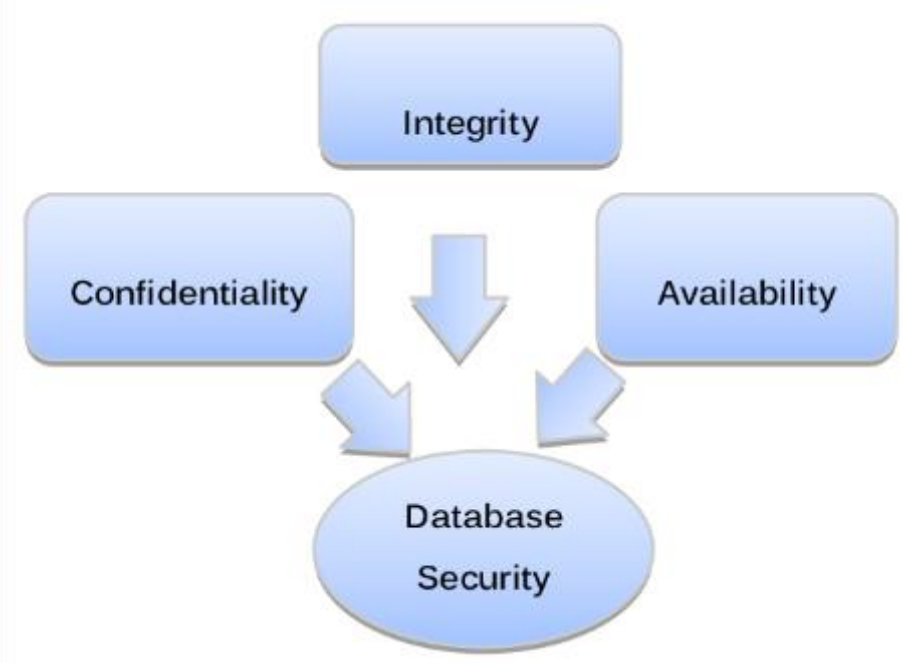
- il trigger è associato ad una tabella, ma fa parte di un database, per cui il suo nome deve essere univoco all'interno del db stesso

esempio


```
DELIMITER $$
CREATE TRIGGER CancellazioneStudenti
BEFORE DELETE ON Studenti
FOR EACH ROW
BEGIN
  INSERT StudentiAnnoPrecedente VALUES (OLD.nome,
   OLD.cognome, CURRENT_DATE());
END;
$$
```

NEW e OLD

- NEW indica che il nome di colonna si riferisce al nuovo valore della riga che sta per essere aggiornata
- NEW si può utilizzare in caso di INSERT e UPDATE
- OLD fa riferimento ai valori precedenti la modifica, e si può utilizzare in caso di UPDATE e DELETE
- la modifica attraverso l'istruzione SET è possibile solo per i valori NEW e solo nei trigger di tipo BEFORE



sicurezza nelle basi di dati

- una base di dati è sicura quando soddisfa i parametri:
  - regola l'***accesso*** ai dati protetti
  - evita la modifica o la manipolazione dei dati da parte di utenti ***non autorizzati***
  - è ***disponibile*** (nel momento in cui deve essere consultata è presente, ***consistente*** e ***coerente***)

- ***DAC*** (Discretionary Access Control) – controllo discrezionale
  - il proprietario decide chi può accedere alle risorse
- ***MAC*** – a ogni risorsa viene associata una label (livello di privilegio che deve possedere l'utente per accedere)
- ***RBAC*** – gli utenti sono associati a uno o più ruoli (gruppi) le risorse sono accessibili solo a ruoli specificati

protezione e integrità dei dati

- ***autenticazione*** (login – password)
- ***tracciabilità*** – registrazione delle operazioni effettuate da un utente (file di ***log***)
- integrità mediante "***giornale***" (file di log database) che registra i dati coinvolti e le operazioni effettuate su questi
- ***checkpoint*** e ripristino
- ***backup***

DAC

- utente associato al DBMS e permessi relativi all'attività su un database
- **GRANT** (assegnazione diritti)
- **REVOKE** (revoca dei diritti)

RBAC
(Role-based access control)

A. Ferrari

- gli utenti sono associati a uno o più ***ruoli*** (gruppi) le risorse sono accessibili solo a ruoli specificati
- esempio: associo Utente1 al ruolo che gli permette l'accesso in lettura al database
  - **`EXEC sp_addrolemember db_datareader, Utente1`**
- accesso in scrittura:
  - **`EXEC sp_addrolemember db_datawriter, Utente1`**
- negazione dell'accesso in scrittura
  - **`EXEC sp_droprolemember db_datawriter, Utente1`**

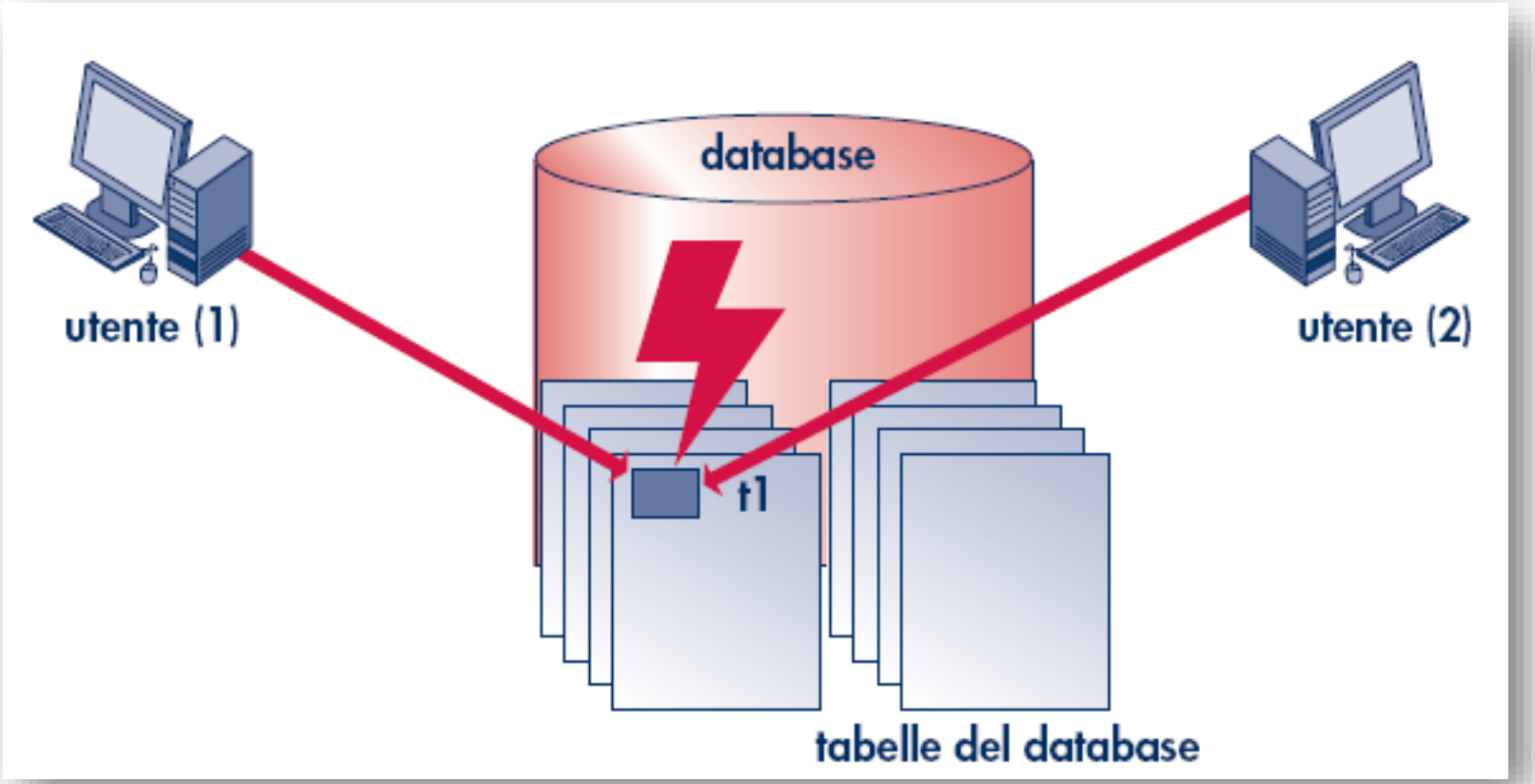
database consistente

- il database è una rappresentazione della realtà mediante un formalismo ⇒ ***modello***
- ***inconsistenza*** : ***sfasamento*** fra realtà e modello che la rappresenta (database)
  (es. il film "…" è stato prestato ma risulta presente nel database)
- ***consistenza*** : nessuna discrepanza tra la realtà fisica e il modello che la rappresenta

accesso concorrente

- accesso ***concorrente***: più utenti accedono a una stessa risorsa ***nello stesso istante***
- l'accesso concorrente è una delle cause principali dei problemi di inconsistenza delle basi di dati
- le soluzioni ai problemi di accesso concorrente sono basate su blocchi (***lock***) che operano come semafori e regolano il "traffico" verso le risorse

accesso concorrente



- una ***transazione*** è un insieme di operazioni
  - ***indivisibili*** (atomiche)
  - ***corrette*** anche in presenza di concorrenza
  - con effetti ***definitivi***

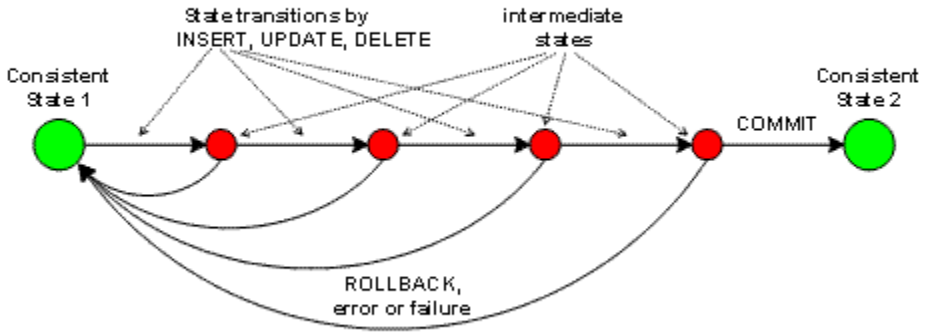
caratteristiche di una transazione (ACID)

A. Ferrari

- ***atomicità*** (***A****tomicity*) una transazione viene portata a termine completamente o non viene effettuata
- ***consistenza*** (***C****onsistency*) prima e dopo la transazione la base di dati è sempre in uno stato consistente
- ***isolamento*** (***I****solation*) il database non viene modificato finché la transazione non è conclusa (nessuno può vedere un risultato intermedio)
- ***permanenza*** (***D****urability*) una volta conclusa la transazione i dati sono in uno stato consistente e non possono essere ripristinati allo stato precedente

COMMIT - ROLLBACK

- transazione terminata con successo
  - **`COMMIT`**
- transazione abortita
  - **`ROLLBACK`**



- ***Lock*** in ***lettura*** (compatibile con altri lock in lettura)
- ***Lock*** in ***scrittura*** (blocco esclusivo della risorsa)