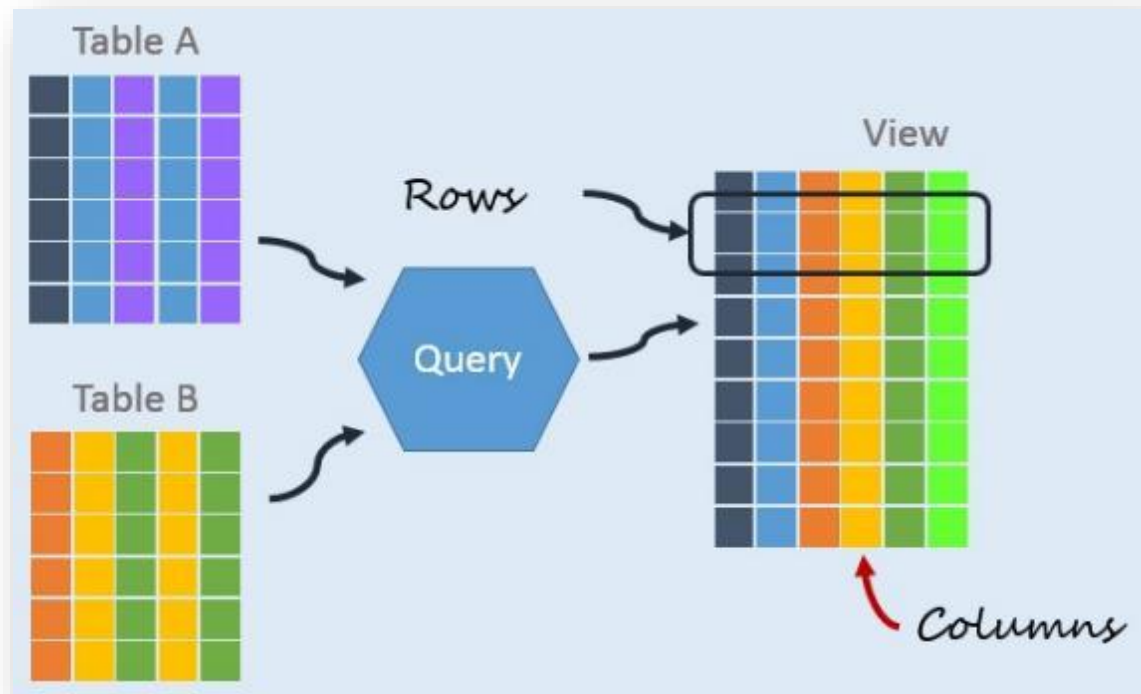




SQL

Structured Query Language



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

-- Creazione vista

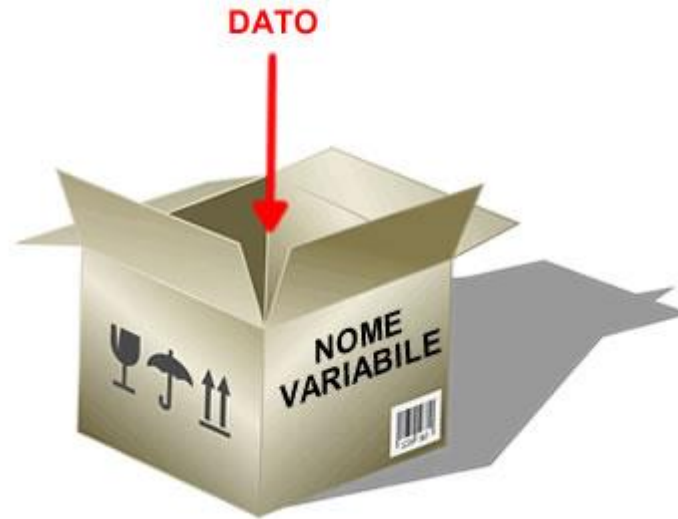
```
CREATE VIEW StudentiConClasse AS  
SELECT studente.cognome, classe.descrizione  
FROM studente INNER JOIN classe  
      ON studente.IDclasse = classe.ID  
ORDER BY studente.cognome;
```

-- Utilizzo vista

```
SELECT * FROM StudentiConClasse;
```

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

SELECT * FROM filmConGenere;
```



VARIABILI

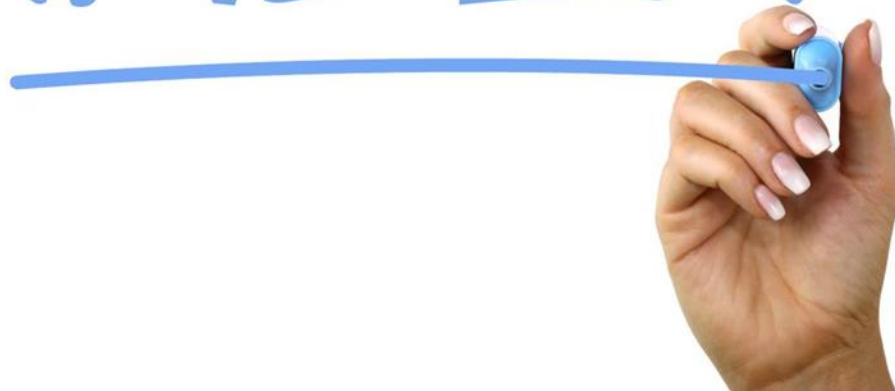
- è possibile memorizzare *valori* in variabili
- le variabili sono globali alle istruzioni SQL
- gli identificatori iniziano con @

○ *esempio:*

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


- 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();**

INDEX



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

- *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

- 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

- 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

```
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)

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

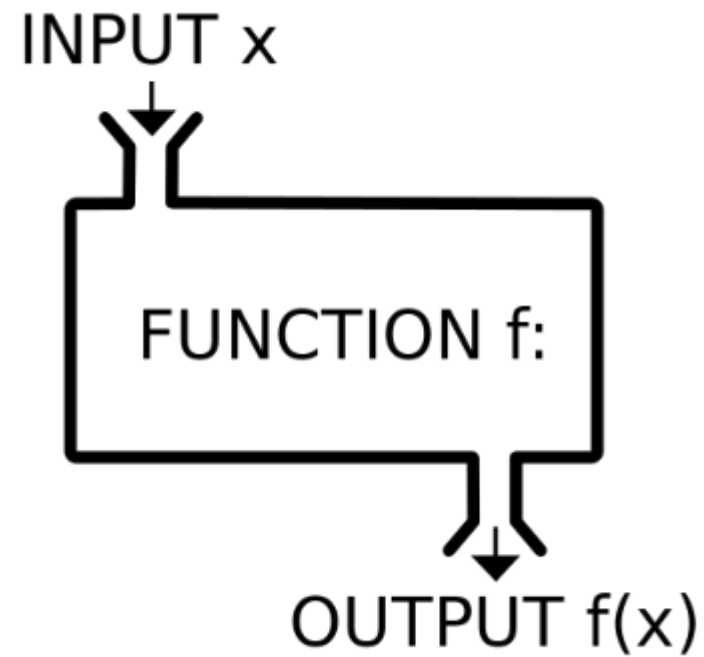
- il comando DELIMITER serve per modificare il normale delimitatore delle istruzioni, che sarebbe il punto e virgola
- la stored procedure contiene più istruzioni, al suo interno il punto e virgola viene utilizzato più volte
 - di conseguenza, se vogliamo riuscire a memorizzare la procedura, dobbiamo 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 \$\$

```
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**)



stored function

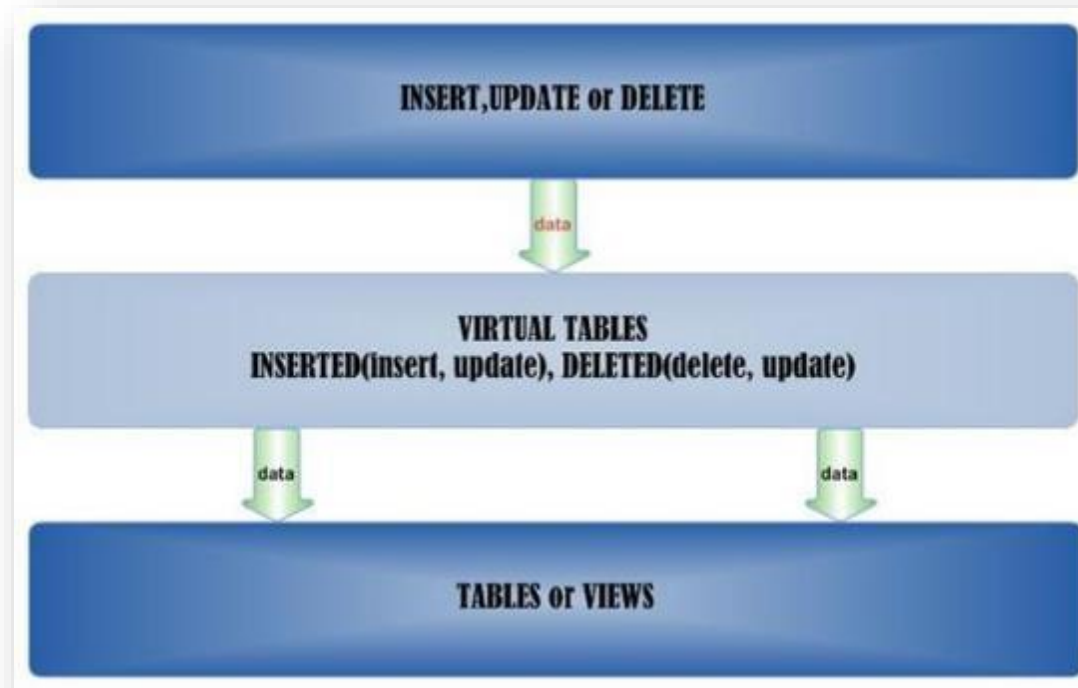
- le stored functions sono simili alle stored procedures
- restituiscono un valore e non possono restituire result set

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.

```
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


```
-- 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');
```



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**

CREATE

```
[DEFINER = { utente | CURRENT_USER }]
```

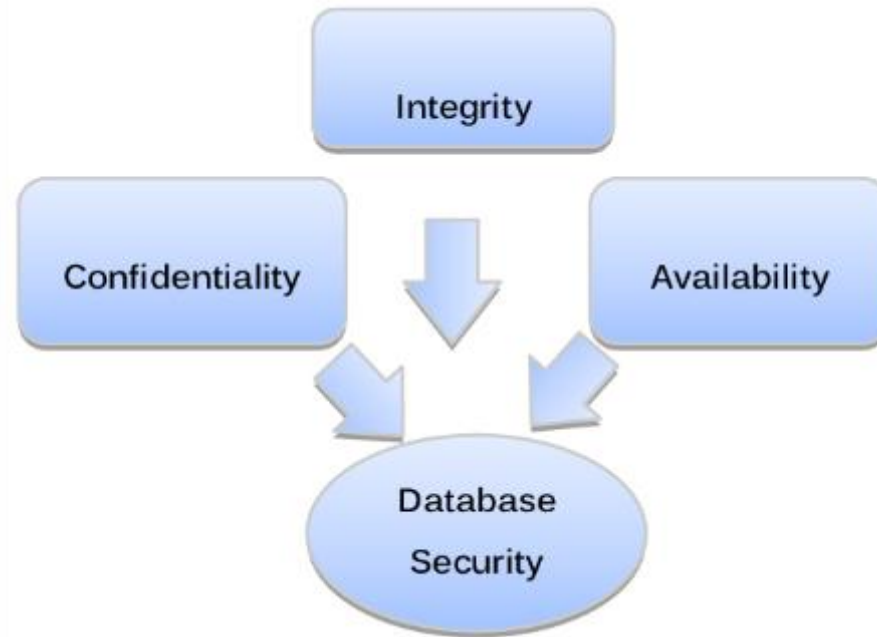
```
TRIGGER nome tipo
```

```
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

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

- 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 specifici

- **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**

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

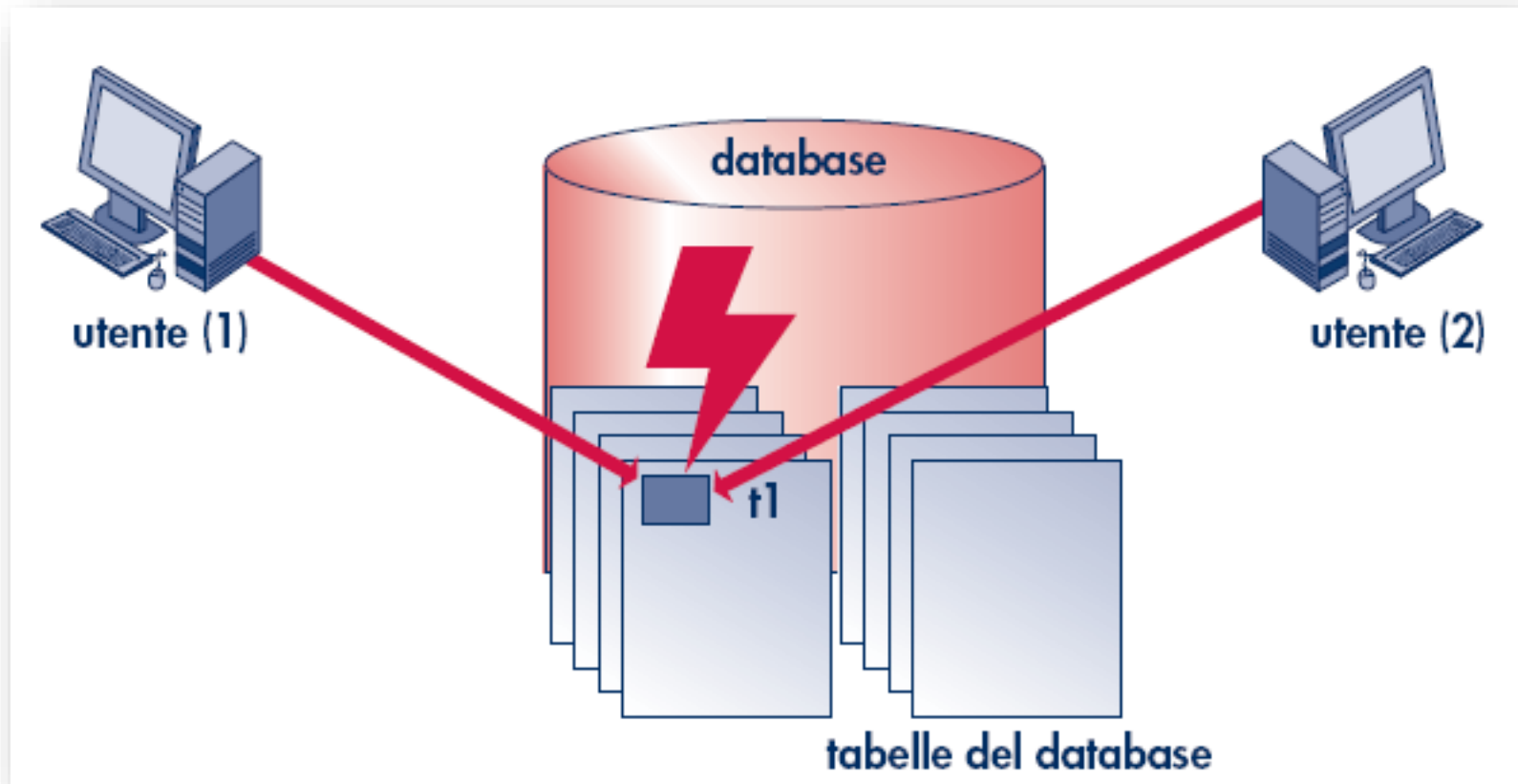
RBAC

(Role-based access control)

- gli utenti sono associati a uno o più *ruoli* (gruppi) le risorse sono accessibili solo a ruoli specificati
- esempio: associa 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`

- il database è una rappresentazione della realtà mediante un formalismo \Rightarrow ***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

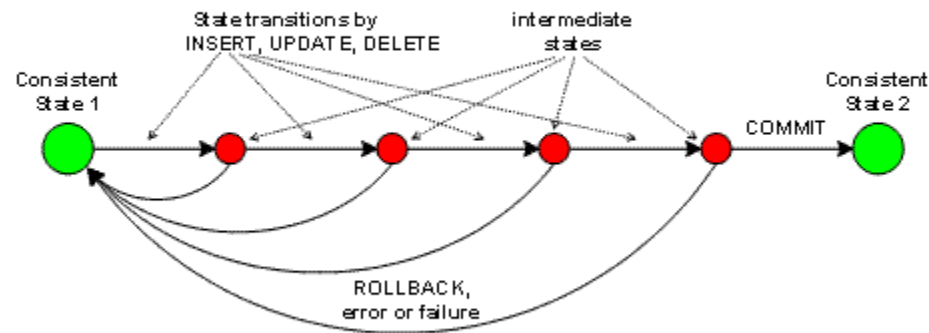
- accesso ***concorrente***: più utenti accedono a una stessa risorsa ***nello stesso istante***
- l'accesso concorrente è una delle cause principali dei problemi di inconsistenza delle basi di dati
- le soluzioni ai problemi di accesso concorrente sono basate su blocchi (***lock***) che operano come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



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

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

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



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

