

# SQL: aspetti avanzati

---

PROF. DIOMAIUTA CRESCENZO

UNIVERSITÀ DEGLI STUDI DELLA CAMPANIA «*LUIGI VANVITELLI*»

# Vincoli di integrità generici - Check

- La clausola **check** permette di specificare vincoli generici
- Specifica di vincoli di ennupla (e anche vincoli più complessi, non sempre supportati)

```
CREATE TABLE NomeTabella(  
    NomeAttributo Dominio [check (Condizione)]  
    .....  
    [check (Condizione)]  
)
```

- Le **condizioni utilizzabili** sono le stesse che possono apparire come argomento della clausola **where** di un'interrogazione SQL
- La **condizione contenuta** nel vincolo di check deve essere sempre verificata affinché la base di dati sia *corretta*.

# Check: Esempio

- Impiegato (Nome, Cognome, Dipart, StipendioAnn)
- Dipartimento (Nome, Città)

```
CREATE TABLE Impiegato(  
    Nome VARCHAR(20),  
    Cognome VARCHAR(20) check (Cognome like 'c%'), Dipart  
    VARCHAR(20),  
    StipendioAnn INT check (StipAnn >= 1 AND StipendioAnn <= 999)  
)
```

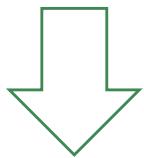
- Nella creazione della tabella **Impiegato**, vengono accettate solo tuple in cui il valore dell'attributo **Cognome** inizia con la 'c' e in cui lo **Stipendio** è compreso tra 1 e 999

# Check: Attenzione

- La condizione contenuta nel vincolo di check viene valutata immediatamente dopo l'inserimento \ modifica di una tupla nella tabella in cui il vincolo è definito

↳ Valutata dopo o anche con Rollback?

```
INSERT INTO Impiegato(Nome,Cognome,Dipart,StipendioAnn)  
VALUES ('Paolo', 'Cesarano', 'Produzione', 700)
```

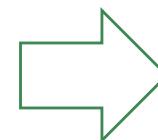


L'inserimento va a buon fine, perché soddisfa tutti i vincoli (compresi quelli di check) specificati per la tabella.

# Check: Attenzione

- In caso di violazione del vincolo, l'operazione di modifica\inserimento che ha causato tale violazione viene "disfatta" dal sistema (Rollback parziale).

```
INSERT INTO Impiegato(Nome,Cognome,Dipart,StipendioAnn)  
VALUES ('Andrea', 'Milani', 'Amministrazione', 800)
```



L'inserimento della tupla viene effettuato senza verificare i vincoli di check specificati per la tabella.

Successivamente viene verificato se l'inserimento ha violato qualche vincolo. Dato che il valore 'Milani' non inizia con la 'c', un vincolo di check è stato violato. La tupla viene eliminata dalla tabella



Rollback parziale

A diagram showing a database table with four columns: Nome, Cognome, Dipart, and StipendioAnn. A large red 'X' is drawn across the entire row corresponding to the inserted tuple ('Andrea', 'Milani', 'Amministrazione', 800).

Nome	Cognome	Dipart	StipendioAnn
Andrea	Milani	Amministrazione	800

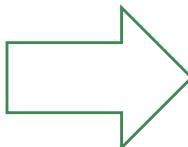
# Check

---

- Il vincolo di *check* è un costrutto potente :
  - la condizione specificata nel vincolo può riferirsi ad altri attributi della stessa tabella oppure ad attributi di tabelle differenti.
  - tutti i vincoli predefiniti (PRIMARY KEY, UNIQUE, FOREIGN KEY, ...) possono essere espressi attraverso il vincolo di *check*.

# Check

```
CREATE TABLE Impiegato(  
Nome VARCHAR(20),  
Cognome VARCHAR (20),  
Dipart VARCHAR (20),  
StipendioAnn INT,  
check (Cognome like 'c%' OR  
        Dipart in (SELECT Nome  
                  FROM Dipartimento  
                ))
```



Nella creazione della tabella **Impiegato**, vengono accettate solo tuple in cui il valore dell'attributo **Cognome** inizia con la 'c' oppure in cui il dipartimento a cui afferisce l'impiegato è contenuto nella tabella **Dipartimento**.

## ATTENZIONE

Quando un vincolo di **check** coinvolge due o più tabelle, possono emergere comportamenti non desiderati

# Check

---

```
CREATE TABLE Impiegato(  
    Nome VARCHAR (20),  
    Cognome VARCHAR (20),  
    Dipart VARCHAR (20),  
    StipendioAnn INT,  
    check ( (SELECT count(distinct Dipart) FROM Impiegato) +  
            (SELECT count(Nome) FROM Dipartimento) < 100  
        )  
)
```

## ATTENZIONE

Se nella tabella **Dipartimento** viene inserito un numero di Dipartimenti superiore a 100, il vincolo rimane soddisfatto, poiché il controllo è effettuato solo quando si inseriscono tuple nella tabella **Impiegato**...questa è un'anomalia.

# Asserzioni

---

- Per evitare eventuali anomalie causate dai vincoli di *check* su più tabelle, si possono utilizzare le *asserzioni*.
- Le asserzioni rappresentano dei vincoli non associati a nessun attributo o tabella in particolare, ma appartengono direttamente allo schema della Base di Dati.

```
CREATE ASSERTION NomeAsserzione  
    check (Condizione)
```

- Le asserzioni permettono di esprimere vincoli particolari, che coinvolgono più tabelle o vincoli che richiedono che una tabella abbia una cardinalità minima\massima.

# Asserzioni: Esempio

```
CREATE ASSERTION NonPiùDi100(  
    check ((SELECT count(distinct Dipart) FROM Impiegato) +  
           (SELECT count(Nome) FROM Dipartimento) < 100  
    )  
)
```

Ad esempio, se il numero di valori contenuti nell'attributo *Nome* della tabella *Dipartimento* supera il valore 100 e la tabella *Impiegato* è vuota , il vincolo **non è soddisfatto**.

```
CREATE ASSERTION AlmenoUna(  
    check (1 <= (SELECT count(*)  
                  FROM Impiegato)  
    )  
)
```

Esempio: Imporre che nella tabella *Impiegato* sia sempre presente almeno una riga

# Asserzioni: Esempio

Esempio: Costruire un vincolo che assicuri che tutti i supervisori abbiano età > 30

Impiegato (ID, Nome, Età, Salario)

Dipartimento (ID, Fondo, SupervisoreID)

```
CREATE TABLE Dipartimento(  
    ID INT PRIMARY KEY,  
    Fondo REAL,  
    SupervisoreID INT,  
    FOREIGN KEY (SupervisoreID) REFERENCES Impiegato  
)
```

Meglio usare un'asserzione, in modo da effettuare il controllo ogni volta che viene aggiornata una delle due tabelle

```
CREATE ASSERTION EtàSupervisore (  
    check(30 < (SELECT I.Età  
        FROM Impiegato I, Dipartimento D  
        WHERE I.ID = D. SupervisoreID  
    ))  
)
```

# Politiche di controllo nei vincoli di integrità

---

- Ogni vincolo di integrità, definito tramite *check* o tramite *asserzione*, è associato ad una **politica di controllo** che specifica se il vincolo è **immediato** o **differito**.
- I vincoli **immediati** (*valore di default*) – sono verificati immediatamente dopo ogni operazione che coinvolge le tabelle presenti nel vincolo di *check* o nell'*asserzione*.
  - Quando un vincolo immediato non è soddisfatto, l'operazione di modifica che ha causato la violazione è stata appena eseguita e il sistema può disfarla. Questo modo di procedere è chiamato **Rollback Parziale**.
  - Tutti i vincoli predefiniti (PRIMARY KEY, UNIQUE, FOREIGN KEY, NOT NULL) sono verificati in modo immediato.

# Politiche di controllo nei vincoli di integrità

---

- I **vincoli differiti** sono verificati solo al termine dell'esecuzione di una serie di operazioni (che costituisce una *transazione*).
- Il **controllo differito** viene utilizzato per gestire casi in cui non è possibile costruire uno stato consistente della base di dati con una singola modifica.
  - Si pensi, ad esempio, ad una tabella **Impiegato** con un attributo *not null Dipart* associato ad un vincolo di integrità referenziale verso la tabella **Dipartimento**, e la tabella **Dipartimento** ha un attributo *Direttore not null* associato ad un vincolo di integrità referenziale verso la tabella **Impiegato**.
  - Se entrambi i vincoli fossero immediati, non sarebbe possibile modificare lo stato iniziale vuoto delle due tabelle, in quanto ogni singolo comando di inserimento di tuple non rispetterebbe il vincolo di integrità referenziale.
  - Il **controllo differito** permette di gestire agevolmente questa situazione.

# Politiche di controllo nei vincoli di integrità

---

- Quando si verifica una violazione di un vincolo **differito** al termine della transazione, non c'è modo di individuare l'operazione che ha causato la violazione.
- Diventa perciò necessario disfare l'intera sequenza di operazioni che costituiscono la transazione. In questo caso si esegue un **rollback**.
- Grazie ai meccanismi di controllo *immediato\differto*, l'esecuzione di un comando di modifica dell'istanza della base di dati che soddisfa tutti i vincoli, produrrà una nuova istanza **consistente** della base di dati (cioè, che soddisfa a sua volta tutti i vincoli).
- ***set constraints* [NomeVincoli | all] *immediate***
- ***set constraints* [NomeVincoli | all] *deferred***

# Viste

---

- SQL permette di specificare *tabelle virtuali* in cui le righe non sono esplicitamente memorizzate nella base di dati, ma sono calcolate quando necessario → le *viste*

```
CREATE VIEW NomeVista [ (ListaAttributi ) ] AS SelectSQL
[with [local | cascaded ]check option]
```

- Le viste si definiscono associando un *nome* ed una *lista di attributi* al risultato dell'esecuzione di un'interrogazione.
- L'interrogazione interna (che può contenere anche altre viste) deve restituire un insieme di attributi pari a quelli contenuti nello schema della vista, nello stesso ordine.

# Viste: Esempio

---

Impiegato (Nome, Cognome, Dipart, StipendioAnn)

Dipartimento (Nome, Città)

**ESEMPIO:** Definire una vista ImpiegatiAmm che contiene tutti gli impiegati del dipartimento Amministrazione con uno stipendio superiore a 10

```
CREATE VIEW ImpiegatiAmm(  
    Nome, Cognome, Dipart, StipendioAnn ) as  
        SELECT Nome,Cognome,Dipart,StipendioAnn  
        FROM Impiegato  
        WHERE Dipart = 'Amministrazione' AND StipendioAnn > 10
```

# Viste: Esempio

Impiegato (Nome, Cognome, Dipart, SipendioAnn)

Dipartimento (Nome, Città)

**ESEMPIO:** Definire una vista ImpiegatiAmmPov definita a partire dalla vista ImpiegatiAmm, che contiene tutti gli impiegati amministrativi con uno stipendio compreso tra 10 mila e 50 mila euro

```
CREATE VIEW ImpiegatiAmmPov as SELECT *
FROM ImpiegatiAmm
WHERE StipendioAnn < 50
With check option
```

**check option** permette modifiche, ma solo a condizione che la tupla continui ad appartenere alla vista (**non posso modificare lo stipendio portandolo a 60**)

# Viste: Esempio

**ESEMPIO:** Estrarre il numero medio di uffici per ogni dipartimento

```
CREATE VIEW DipartUffici(NomeDip, NroUffici) as  
SELECT Dipart, count(distinct Ufficio)  
FROM Impiegato  
Group by Dipart
```

```
SELECT avg(NroUffici)  
FROM DipartUffici
```

```
select avg(count(distinct Ufficio))  
from Impiegato  
Group by Dipart
```

**SCORRETTA!**

L' interrogazione è scorretta in quanto la sintassi SQL non permette di combinare in cascata la valutazione di diversi operatori aggregati.

# Viste: Esempio

Moto (Targa, Cognome, Marca, Nazione, Tasse)

Proprietario (Nome, Targa)

**ESEMPIO:** Estrarre per ogni cliente le tasse che devono essere pagate per tutte le moto possedute, ipotizzando che vi sono più proprietari per una moto, l'ammontare delle tasse viene equamente diviso tra i proprietari

```
CREATE VIEW TasseInd (Targa, Tassa) as  
    SELECT Targa, Tasse/count(*)  
    FROM Moto join Proprietario  
        on Moto.Targa = Proprietario.Targa  
    Group by Targa, Tasse
```

```
SELECT Nome, sum(Tassa)  
FROM Proprietario join TasseInd  
    on Proprietario.Targa = TasseInd.Targa  
Group by Nome
```

# Viste aggiornabili

---

- Sintatticamente, posso pensare di modificare una vista
- I sistemi impongono una serie di restrizioni
  - Come regola, bisogna immaginare che una vista è modificabile se e solo se le modifiche possono essere applicate univocamente alle tabelle sottostanti.
  - Un'operazione di modifica\cancellazione nella vista deve soddisfare tutti i vincoli di integrità della tabella sottostante, altrimenti non viene eseguita.
- Queste restrizioni relative alla politica aggiornamenti sono FONDAMENTALI per rendere l'aggiornamento alla tabella sottostante non ambiguo.
  - A ciascuna riga della tabella di base corrisponderà una sola riga della vista (le viste mantengono i duplicati).

# Viste aggiornabili: Esempio

```
INSERT INTO ImpiegatiAmm (Nome,Cognome,Dipart,StipendioAnn)  
VALUES('Marco','Grigi','Amministrazione',40)
```

La tupla viene inserita correttamente nella vista **ImpiegatiAmm** e nella relazione sottostante **Impiegato**

```
INSERT INTO ImpiegatiAmm (Nome,Cognome,Dipart,StipendioAnn)  
VALUES('Marco','Grigi','Amministrazione',7)
```

La tupla viene inserita correttamente nella relazione sottostante **Impiegato**, ma non nella vista **ImpiegatiAmm**, che non accetta Stipendi < 10

«Gli inserimenti che non soddisfano le condizioni con cui è stata definita una vista possono essere disabilitati attraverso la clausola **with check option**»

# Funzioni scalari

---

- Oltre alle funzioni aggregate, SQL mette a disposizione diverse funzioni scalari che possono essere usate all'interno delle espressioni.
  - Ricevono come argomento una o più espressioni
- 
- Funzioni temporali: servono per la gestione di informazioni temporali
    - Current\_date
    - Current\_time
    - Current\_timestamp
  - Funzioni di manipolazione di stringhe: trasformano il contenuto di stringhe
    - Char\_length
    - Lower
    - Upper
    - substring

# Funzioni scalari

---

- **Funzioni di conversione di dominio:** permettono di **convertire un valore di un dominio in un altro dominio** (non tutte le conversioni sono ammesse)
  - Cast
    - Cast (Data as char(10))
- **Funzioni per la formattazione dell'output:** controllano l'aspetto della query, indentazione ed altre caratteristiche
- **Funzioni matematiche:** si applicano ad espressioni numeriche
  - Abs: valore assoluto
  - Sqrt: radice quadrata
- **Funzioni di accesso al Sistema Operativo:** permettono di **accedere ai servizi dell'ambiente ospite**

# Funzioni condizionali

---

- **Coalesce:** accetta una sequenza di espressioni e restituisce il primo valore diverso da **null**
  - Esempio: Estrarre i nomi, i cognomi e i dipartimenti cui afferiscono gli impiegati, usando la stringa 'Ignoto' nel caso in cui non si conosca il dipartimento

```
select Nome, Cognome, coalesce(Dipart,'Ignoto')  
from Impiegato
```

- **Nullif:** accetta una espressione e un valore costante; se l'espressione è pari al valore costante restituisce **NULL**, altrimenti restituisce il valore dell'espressione
  - Esempio: Estrarre i nomi, i cognomi e i dipartimenti cui afferiscono gli impiegati, restituendo il valore nullo per il dipartimento quando Dipart possiede il valore 'Ignoto'

```
Select Nome, Cognome, nullif (Dipart, 'Ignoto')  
from Impiegato
```

# Funzioni condizionali

- **Case:** permette di specificare strutture condizionali il cui risultato dipende dalla valutazione del contenuto delle tabelle. Esistono due varianti:

## Case Espressione

```
when Valore then EsprRisultato  
{when Valore then EsprRisultato}  
[ else EsprRisultato]  
end
```

Restituisce risultati diversi data una certa espressione

## Case when Condizione then Espressione

```
{when Condizione then Espressione}  
[ else Espressione]  
end
```

Ammette la valutazione di predici SQL generici

# Funzioni condizionali: Esempio 1

- **Esempio:** Calcolare l'ammontare delle tasse di circolazione, in base al tipo di veicolo e con immatricolazione dopo il 1975.
- **Veicolo(Targa, Tipo, Anno, Kwatt, Lunghezza, NAssi)**

```
Select Targa  
case Tipo  
    when 'Auto' then 2.58*KWatt  
    when 'Moto' then (22.00 + 1.00 *KWatt )  
    else null  
end as Tassa  
From Veicolo  
Where Anno>1975
```

# Funzioni condizionali: Esempio 2

- **Esempio:** Modificare lo stipendio di un impiegato, sulla base dei valori assunti dalle colonne Dipart e Ufficio

```
Update Impiegato
Set Stipendio =
case
    when (Dipart = 'Amministrazione' and Ufficio = 10)
        then Stipendio * 1.1
    when (Dipart = 'Amministrazione' and Ufficio <> 10)
        then Stipendio * 1.2
    when Dipart = 'Produzione'
        then Stipendio * 1.15
    else Stipendio
end
```

Senza l'istruzione case non è possibile effettuare un'operazione di questo tipo con una sola istruzione

# Stored Procedures

- Sono procedure che vengono normalmente **memorizzate** all'interno del Database come **parte dello schema**.
  - Supportate dalla maggior parte dei DBMS
  - Una volta definita essa è utilizzabile come se facesse parte dell'insieme dei comandi SQL predefiniti
  - Insieme di istruzioni SQL memorizzate nel DBMS, cui è associato un nome univoco.
  - Può ricevere parametri in input, può restituire più di un valore in output. Il corpo contiene istruzioni SQL

```
PROCEDURE nomeProcedura (:NomeParametro TipoParametro,...)
```

```
BEGIN
```

```
[dichiarazione di variabili locali]
```

```
[istruzioni SQL]
```

```
END;
```

# Stored Procedures

- Esempio: La procedura permette di assegnare all'attributo **Città** il valor **:NuovaCittà**, per tutte le righe di Dipartimento e Impiegato in cui l'attributo vale **:VecchiaCittà**

```
PROCEDURE CambiaCittaATutti (:NuovaCitta varchar(20),
                               :VecchiaCitta varchar(20))

BEGIN
    Update Dipartimento
    Set Città = :NuovaCitta;
    Where Città = :VecchiaCittà;
    Update Impiegato
    Set Città = :NuovaCitta;
    Where Città = :VecchiaCittà;
END;
```

È possibile utilizzare all'interno della procedura anche il controllo *if-then-else*

# Trigger e basi di dati attive

---

- SQL-3 fornisce un meccanismo per rendere la base di dati in grado di reagire ad eventi definiti dall'admin tramite opportune azioni (*trigger*)
- Una BD con questa capacità si dice *attiva*
- I trigger sono delle particolari stored procedures
- Le regole seguono il paradigma **Evento Condizione Azione** (E-C-A)

# Trigger e basi di dati attive

---

Diversi usi possibili:

- **Implementazione delle regole di business non esprimibili** attraverso i costrutti dichiarativi dell'SQL
- **Segnalazione automatica ad altri programmi** per effettuare determinate operazioni quando una tabella viene modificata
- **Inserimento di valori opportuni in particolari colonne al momento di un inserimento o di una modifica**
- **Rendere globali alcuni controlli affidandone l'esecuzione al DBMS**

# Il paradigma E-C-A

---

- Paradigma: Evento-Condizione-Azione
  - Quando un evento si verifica
  - Se la condizione è vera
  - Allora l'azione è eseguita
- Questo modello consente computazioni reattive

# Il paradigma E-C-A

---

## □ Evento

- Una richiesta di modifica dello stato della base di dati: **INSERT, DELETE, UPDATE**
- Quando avviene l'evento, il trigger viene *attivato*

## □ Condizione

- Un predicato booleano che identifica se l'azione del trigger deve essere eseguita
- Quando la condizione viene valutata il trigger è *considerato o valutato*

## □ Azione

- Una sequenza di operazioni che devono essere eseguite o una procedura
- Quando l'azione è eseguita anche il trigger è *eseguito*

↳ Azioni sempre col database

# Trigger DML e Trigger DDL

---

- I trigger si dividono in:
- Trigger DML: è la tipologia più comune, l'innesto avviene a seguito di una INSERT, UPDATE o DELETE. Possono essere eseguiti prima o subito dopo l'innesto. La condizione è espressa nella clausola WHEN.
- Trigger DDL: sono detti anche trigger di sistema, non reagiscono ad eventi di manipolazione, ma ad eventi di sistema, come l'avvio o chiusura di un database, creazione o cancellazione di tabelle, ecc. Usati come misura di sicurezza nei database.

# Definizione e uso dei trigger

---

I Triggers:

- In genere i trigger fanno riferimento ad una tabella detta target e risponde ad eventi relativi a tale tabella
- I valori di tale tabella vengono “congelati” prima e dopo l’evento in due strutture dette di transizione.

↳ deve consentire lo stato prima e dopo un’operazione.

ES: inserimento ha il dopo ma non il prima.  
update ha entrambi

# Sintassi dei trigger

- Lo standard **SQL:1999 (SQL-3)** sui trigger è stato fortemente **influenzato da DB2 (IBM)**; gli altri sistemi non seguono lo standard (esistono dagli anni 80')
- Ogni trigger è caratterizzato da:
  - **Nome** (*unico nello schema di appartenenza*)
  - **target** (*tabella controllata*) *Tabella sulla quale vengono inserite condizioni*
  - **modalità** (*before o after*) *Trigger before o trigger after*
  - **evento** (*insert, delete o update*) *(caso DML)*
  - **granularità** (*statement-level o row-level*) *Se è a livello di riga/tupla o a livello di set/tabella*
  - **alias** dei valori o tabelle di transizione
  - **azione** *Cosa deve fare*
  - **timestamp** di creazione *Dà orario nell'esecuzione*
    - ↳ Un trigger potrebbe innescare un altro

# Sintassi dei trigger

Modo

```
create trigger NomeTrigger  
    ← {Before | AFTER} Evento DML Attivante  
    on TabellaTarget (su quale tabella)  
    [referencing Referenza] ⇒ Per usare variabili old e new  
    [for each Livello] (granularità)  
    [when (PredicatoSQL)] Condizione  
    StatementProceduraleSQL ⇒ Cosa dovo fare ?
```

# Sintassi dei trigger DML

```
create trigger NomeTrigger
{ before | after }
{ insert | delete | update [of Column] } on TabellaTarget
[referencing
{[old table [as] VarTuplaOld]
  [new table [as] VarTuplaNew] } |
  prima dell'oper.
  {[old [row] [as] VarTabellaOld]
    [new [row] [as] VarTabellaNew] }
  [for each { row | statement }]
  [when Condizione]
  StatementProceduraleSQL
```

*grado di rigore di scrittura*

*Valore vecchio*

*grado di rigore a livello di riga*

*grado di rigore a livello di riga*

Modo: before, after

Evento: insert, delete, update

Consente di introdurre nomi di variabili

Livello: row (tupla) o statement (primitiva)

# Sintassi dei trigger DDL

---

```
create trigger NomeTrigger           ↑ Evento DDL
{Before | AFTER} Evento DDL Attivante | <Elenco Eventi Database>
[ on Database_schema] Questo serve schema del database.
[when (Condizione_Trigger)]
<Corpo Trigger>
```

# Modo di esecuzione

---

## **Before** *Condizionamento*

- Il trigger è considerato e possibilmente eseguito **prima dell'evento** (i.e., la modifica del database)
- I trigger before non possono modificare lo stato del database; possono al più condizionare i valori “new” in modalità row-level (set t.new=expr) *es. Consigliano se c'è una violaz. per esempio*
- Normalmente questa modalità è usata quando si vuole verificare una modifica prima che essa avvenga e “modificare la modifica”

## **After** *Risultante*

- Il trigger è considerato e eseguito **dopo l'evento**
- È la modalità più comune, adatta alla maggior parte delle applicazioni

# Esempio trigger con before e after

- Agisce prima dell'update e della verifica di integrità (*Condizionare*)

```
create trigger LimitaAumenti  
before update of Stipendio on Impiegato  
for each row (dove essere verificata per ogni riga)  
when (new.Stipendio > old.Stipendio * 1.2)  
set new.Stipendio = old.Stipendio * 1.2
```

Condiziona i valori usati da un'operazione di modifica

- Agisce dopo l'update (*Reagire*)

```
create trigger LimitaAumenti2  
after update of Stipendio on Impiegato  
for each row (gli posso mettere una riga)  
when (new.Stipendio > old.Stipendio * 1.2)  
update Impiegato  
set new.Stipendio = old.Stipendio * 1.2  
where ImpNum = new.ImpNum
```

Con il trigger posso correggere  
check invec no.

Reagisce ad una modifica dello stipendio, **che viene svolta in ogni caso**, e può violare i vincoli di integrità

# Granularità dei trigger

---

- **Modalità statement-level** (di default, opzione **for each statement**)
  - Il trigger viene considerato e possibilmente eseguito solo una volta per ogni istruzione (statement) DML specificata nell'evento che lo ha attivato, indipendentemente dal numero di tuple modificate
- **Modalità row-level** (opzione **for each row**)
  - Il trigger viene considerato e possibilmente eseguito una volta per ogni tupla coinvolta nell'operazione indicata nell'evento

# Referencing

---

- Dipende dalla granularità
  - Se la modalità è row-level, ci sono due *variabili di transizione* (**old** e **new**) che rappresentano il *valore precedente o successivo alla modifica di una tupla*
  - Se la modalità è statement-level, ci sono due *tabelle di transizione* (**old table** e **new table**) che contengono i valori precedenti e successivi delle tuple modificate dallo statement
  - **old** e **old table** non sono presenti con l'evento **insert**
  - **new** e **new table** non sono presenti con l'evento **delete**

# Esempi di Trigger

**Componenti** (id\_comp, nome\_componente, fornitore, costo\_unitario)

**Dipartimenti** (id\_dip, nome\_dipartimento, localita, provincia)

**Personale** (matricola, id\_dip\*, nominativo, data\_nascita, qualifica, stipendio)

**Prodotti** (id\_prod, id\_dip\*, nome\_prodotto, prezzo)

**Composizione** (id\_prod\*, id\_comp\*, unita\_comp)

```
CREATE TRIGGER Aggiorna_Costo_Componente
AFTER UPDATE
ON Componenti
FOR EACH ROW
BEGIN
    IF NEW.costo_unitario <> OLD.costo_unitario
        INSERT INTO CronologiaCostiComponenti
        VALUES (OLD.id_comp, OLD.nome_componente, OLD.costo_unitario,
                CURRENT_DATE());
END IF;
END;
```

Il seguente trigger attiva, quando viene aggiornato il costo di un componente, l'inserimento dei dati di tale componente in una tabella chiamata **CronologiaCostiComponenti** per registrare l'avvenuto aggiornamento:

# Esempi di Trigger

**Componenti** (id\_comp, nome\_componente, fornitore, costo\_unitario)

**Dipartimenti** (id\_dip, nome\_dipartimento, localita, provincia)

**Personale** (matricola, id\_dip\*, nominativo, data\_nascita, qualifica, stipendio)

**Prodotti** (id\_prod, id\_dip\*, nome\_prodotto, prezzo)

**Composizione** (id\_prod\*, id\_comp\*, unita\_comp)

```
CREATE TRIGGER Elimina_Impiegato
BEFORE DELETE
ON Personale
FOR EACH ROW
INSERT INTO CronologiaPersonale
VALUES (OLD.matricola, OLD.id_dip, OLD.nominativo, 'Eliminato', CURRENT_DATE());
```

Il seguente *trigger* attiva, quando viene cancellata una riga della tabella *Personale*, l'inserimento dei dati dell'impiegato cancellato in una opportuna tabella chiamata **CronologiaPersonale** che prevede la registrazione della matricola, codice di dipartimento, nominativo, tipo di operazione e data dell'operazione

# Esempi di Trigger

**Componenti** (id\_comp, nome\_componente, fornitore, costo\_unitario)

**Dipartimenti** (id\_dip, nome\_dipartimento, localita, provincia)

**Personale** (matricola, id\_dip\*, nominativo, data\_nascita, qualifica, stipendio)

**Prodotti** (id\_prod, id\_dip\*, nome\_prodotto, prezzo)

**Composizione** (id\_prod\*, id\_comp\*, unita\_comp)

```
CREATE TRIGGER Nuovo_Prodotto
BEFORE INSERT ON Prodotti
FOR EACH ROW
BEGIN
    IF NEW.prezzo < 0 THEN
        SET NEW.prezzo = 0;
    END IF;
    IF NEW.prezzo > 250.00 THEN
        SET NEW.prezzo = 250.00;
    END IF;
END;
```

Il seguente trigger attiva, quando viene inserito un nuovo prodotto, il controllo sul prezzo di vendita ponendo il prezzo uguale a 0 se è stato inserito un valore negativo oppure ponendo lo stesso uguale a 250 se è stato inserito un valore maggiore

# Conflitti tra trigger

---

- Se vi sono più trigger associati allo stesso evento, SQL:2003 prescrive questa politica di gestione:
  - Vengono eseguiti i trigger BEFORE statement-level, se presenti
  - Vengono eseguiti i trigger BEFORE row-level, se presenti
  - Esecuzione dell'evento di innesco del trigger
  - Vengono eseguiti i trigger AFTER row-level, se presenti
  - Vengono eseguiti i trigger AFTER statement-level, se presenti
- Se vi sono più trigger della stessa categoria:
  - l'ordine di esecuzione è definito in base al loro timestamp di creazione (i trigger più vecchi hanno priorità più alta) .

# Esecuzione ricorsiva

Ad ogni trigger ha TEC, un gestore del contesto.

- In SQL:1999 i trigger sono associati ad un “Trigger Execution Context”
- L’azione di un trigger può produrre eventi che attivano altri trigger, che verranno valutati con un nuovo TEC interno:
  - Lo stato del TEC includente viene salvato e quello del TEC incluso viene eseguito. Ciò può accadere ricorsivamente
  - Alla fine dell’esecuzione di un TEC incluso, lo stato di esecuzione del TEC includente viene ripristinato e la sua esecuzione ripresa
- L’esecuzione termina correttamente in uno “stato quiescente”
- L’esecuzione termina in errore quando si raggiunge una data profondità di ricorsione dando luogo ad una eccezione di non-terminazione (**rollback parziale di S**)

# Triggers in Oracle

- Presenta differenze sintattiche e semantiche rispetto allo standard SQL-3

```
create trigger NomeTrigger
{ before | after } Evento [,evento[, evento ]]
[[referencing
    [old [row] [as] VarTuplaOld]
    [new [row] [as] VarTuplaNew] ]
for each { row | statement } [when Condizione]]
BloccoPL/SQL
```

- Sono consentiti **eventi multipli**,
- Non sono previste variabili per le tabelle,
- I **before trigger** possono prevedere update
- La **condizione** è presente solo con trigger row-level
- l'**azione** è un programma PL/SQL

**Evento** ::= {insert|delete|update [of Attributo] } on Tabella

# Esempio di Trigger in Oracle

```
create trigger Riordino
after update of QtaDisponibile on Magazzino
when (new.QtaDisponibile < new.QtaSoglia)
for each row
declare
    X number;
begin
    select count(*) into X
    from OrdiniEsterni
    where Parte = :new.Parte;
    if X = 0 => Controllo booleano
    then
        insert into OrdiniEsterni
        values(:new.Parte,:new.QtaRiordino,sysdate)
    end if;
end;
```

Questo trigger genera un nuovo ordine ogni volta che la quantità disponibile scende al di sotto di una specifica soglia di riordino

**Evento:** update of *QtaDisponibile* in Magazzino

**Condizione:** Quantità sotto soglia e mancanza ordini esterni

**Azione:** insert of *OrdiniEsterni*

# Esempio di Trigger in Oracle

- Il trigger ha granularità a livello di tupla
- Viene considerato immediatamente dopo ogni modifica di *QtaDisp*.
- La condizione viene valutata tupla per tupla confrontando *QtaDisp* e *QtaSoglia* dopo la modifica di *QtaDisp*

Magazzino			
Parte	QtaDisp	QtaSoglia	QtaRiord
1	200	150	100
2	780	500	200
3	450	400	120

OrdiniEsterni

Parte	QtaRiord	Data

# Esempio di Trigger in Oracle

- Si consideri la seguente transazione, attivata il giorno 10/10/2013

T1: **update Magazzino**  
**set QtaDisp = QtaDisp - 70**  
**where Part = 1**

↓ inferiori a 150, segnala.  
Invia al warehouse

Magazzino

Parte	QtaDisp	QtaSoglia	QtaRiord
1	200	150	100
2	780	500	200
3	450	400	120

T1 causa l'attivazione, valutazione ed esecuzione del trigger Riordino, inserendo la tupla nella tabella **OrdiniPendenti**

Y. chi working

OrdiniEsterni

Parte	QtaRiord	Data
1	100	10-10-2013

# Esempio di Trigger in Oracle

- Supponiamo successivamente venga eseguita un'altra transazione

T2: update Magazzino  
set QtaDisp = QtaDisp - 60  
where Part <= 3



Il trigger è eseguito relativamente a tutte le parti, e la condizione è verificata per le parti 1 e 3. Ma l'azione alla parte 1 non ha effetto, quindi si ha il solo inserimento della nuova tupla relativa alla parte 3

Magazzino

Parte	QtaDisp	QtaSoglia	QtaRiord
1	200	150	100
2	780	500	200
3	450	400	120



OrdiniPendenti

Parte	QtaRiord	Data
1	100	10-10-2013
3	120	10-10-2013

# Proprietà dei trigger

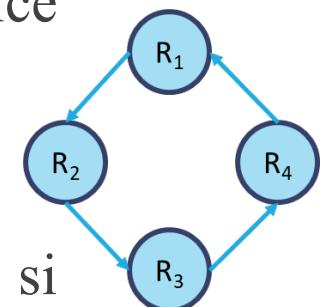
---

- E' importante garantire che l'interferenza tra trigger in una qualunque loro attivazione non produca comportamenti anomali
- Vi sono tre proprietà classiche:
  - **Terminazione:** per un qualunque stato iniziale e una qualunque transazione, si produce uno stato finale (stato quiescente)
  - **Confluenza:** L'esecuzione dei trigger termina e produce un unico stato finale, indipendente dall'ordine di esecuzione dei trigger
  - **Determinismo delle osservazioni:** I trigger sono confluenti e producono verso l'esterno (messaggi, azioni di display) lo stesso effetto
- La terminazione è la proprietà essenziale

# Analisi delle regole

---

- Si usa una rappresentazione delle regole detta grafo di attivazione (triggering):
  - Un nodo per ogni trigger (*regola*)
  - Un arco dal nodo  $t_i$  al nodo  $t_j$  se l'esecuzione dell'azione di  $t_i$  può attivare il trigger  $t_j$  (ciò può essere dedotto con una semplice analisi sintattica)
- Se il grafo è aciclico, l'esecuzione termina
  - Non possono esservi sequenze infinite di triggering
- Se il grafo ha cicli, esso può avere problemi di terminazione: lo si capisce guardando i cicli uno per uno.



# Esempio di regola

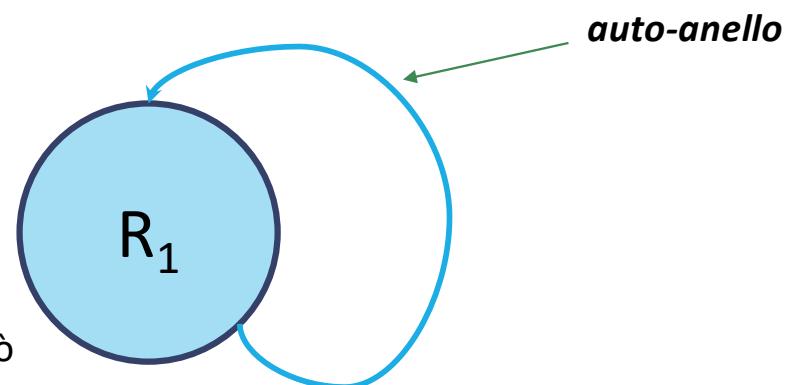
- Consideriamo la regola *ControllaStipendi* (scritta in DB2) che riduce lo stipendio di tutti gli impiegati quando la media degli stipendi supera una certa soglia

```
create trigger ControllaStipendi  
after update of Stipendio on Impiegato  
then update Impiegato  
  set Stipendio = 0.9 * Stipendio  
  where (select avg(Stipendio) from Impiegato) > 100
```

1.1

Attenzione. Cambiando la regola si ha la non terminazione

Il grafo presenta un *ciclo* (la regola può riattivarsi). Qualunque sia la transazione iniziale, l'esecuzione della regola prima o poi termina, poiché lo stipendio via via si riduce, rendendo *falsa* la condizione



# Caratteristiche evolute delle BD attive

---

- **Eventi di sistema e comandi DDL**
  - **Sistema:** servererror, shutdown, etc.
  - **DDL:** **Modifiche di autorizzazioni**
- **Eventi temporali** (anche periodici)
  - Ad esempio «ogni venerdì sera» oppure «alle 15:45 del 07/05/200»
  - Sono di interesse in diverse applicazioni
  - È difficile integrarli perché operano in un contesto transazionale autonomo
  - Si può realizzare anche con software esterno al DBMS

Clemmo

## Caratteristiche evolute delle BD attive

---

- Combinazioni booleane di eventi
  - consente di specificare più eventi per un trigger, in disgiunzione
- Clausola instead of: è alternativa a Before e After, non si esegue l'operazione che ha attivato l'evento, ma un'altra azione
- Definizione di priorità
  - Permette di specificare l'ordine di esecuzione dei trigger quando ce ne sono diversi attivati contemporaneamente

# Caratteristiche evolute delle BD attive

---

- attivazione e deattivazione dinamica (non è standard)
- Trigger organizzati in gruppi
  - I trigger si possono organizzare in gruppi ed ogni gruppo può essere attivato o disattivato separatamente

# Modalità di esecuzione

---

- **Modalità di esecuzione:** è il collegamento tra attivazione (evento) e considerazione/esecuzione (condizione e azione). Condizione e azione sono sempre valutate assieme
- **Immediate** (immediato): il trigger viene considerato ed eseguito con l'evento
- **Deferred** (differito): Il trigger è valutato al termine della transazione *fatto tutto e dopo affatto trigger*
  - Esempio: trigger che gestiscono vincoli di integrità che possono essere violati durante una transazione
- **Detached** (distaccato): Il trigger viene valutato in una transazione separata
  - Esempio: gestione di una variazione di valore di titoli della borsa

# Controllo dell'accesso

---

- In SQL è possibile specificare chi (utente) e come (lettura, scrittura, ...) può utilizzare la base di dati (o parte di essa)
- I privilegi (diritti di accesso) di solito fanno riferimento a tabella, ma anche altri tipi di risorse, come i singoli attributi, viste o domini
- Esiste un utente predefinito system (amministratore della base di dati) ha tutti i privilegi
- Chi crea una risorsa ha tutti i privilegi su di essa

# Creazione di un utente

---

- La **creazione di un utente** avviene con la seguente sintassi:

```
CREATE USER username IDENTIFIED BY password
```

- Una volta definito le credenziali di accesso bisogna stabilire i privilegi. Di norma un privilegio  $P$  è caratterizzato da una quintupla di informazioni

$$P = \langle R, U_1, U_2, A, T \rangle$$

# Privilegi

---

- **R**: la **risorsa** su cui è concesso il **privilegio**
- **U<sub>1</sub>**: l'**utente** che **concede** il **privilegio**
- **U<sub>2</sub>**: l'**utente** che **riceve** il **privilegio**
- **A**: l'insieme delle azioni che sono **permesse** sulla **risorsa**
- **T**: Se il **privilegio** può essere *trasmesso* o meno ad altri **utenti**

# Tipi di privilegi

---

- **create**: permette di **definire nuove istanze per una data risorsa**
- **drop**: permette di **rimuovere istanze per una data risorsa**
- **insert**: permette di **inserire un nuovo oggetto (ennupla) nella risorsa (tabella, viste)**
- **update**: permette di **aggiornare il valore di un oggetto** (tabelle, viste e attributi)
- **delete**: permette di **eliminare oggetti dalla risorsa** (tabelle e viste)
- **select**: permette di **leggere la risorsa** (tabelle, viste e attributi)
- **resource**: permette di **creare, modificare e rimuovere risorse da uno schema**
- **references**: permette la **definizione di vincoli di integrità referenziale** verso la risorsa (può limitare la possibilità di modificare la risorsa)
- **connect**: permette ad **un dato utente di connettersi al DBMS**
- **all privileges**: permette ad **un utente di ottenere tutti i privilegi**

→ Trigger, assert

# Comandi per concedere e revocare privilegi

## □ Concessione di privilegi:

`grant < Privileges | all privileges > on NomeRisorsa to username [ with grant option ]`

- **grant option** specifica se il privilegio può essere trasmesso ad altri utenti

Esempio: `grant select on Dipartimento to Stefano`

## □ Revoca di privilegi

`revoke Privileges on NomeRisorsa from username [ restrict | cascade ]`

- **Restrict** (default): impedisce che il comando sia eseguito se con questo privilegio l'utente ha creato nuove risorse oppure se è stato trasmesso ad altri utenti
- **Cascade**: si forza l'esecuzione del comando con la rimozione dei nuovi oggetti creati e la revoca dei privilegi trasmessi

# Ruoli in SQL-3

- Quando più utenti devono condividere gli stessi privilegi è possibile creare ruoli o profili
- Concetto di ruolo, cui si associano privilegi (anche articolati), poi concessi agli utenti attribuendo il ruolo (*contenitore di privilegi*). Vengono attribuiti tramite il comando **grant**

```
create role NomeRuolo  
grant privilegio1,...,privilegioN on NomeRisorsa1 to nomeruolo [ with grant option ];  
....  
grant privilegio1,...,privilegioN on NomeRisorsaN to nomeruolo [ with grant option ];  
grant nomeruolo to username
```

Per usufruire però dei privilegi è necessario che l'utente invochi un esplicito comando

```
set role NomeRuolo
```

# Ruoli in SQL-3: Esempio

- Si vogliono creare due profili utenti: uno di *giornalista* che permette ad un generico utente di modificare le informazioni presenti nella tabella dello schema e uno di *tifoso* che permette di leggere le sole informazioni presenti nella tabella **GIOCATORI**. Si vogliono creare 4 utenti: 2 tifosi e 2 giornalisti

```
create role giornalista;
grant connect, update ON CAMPIONATO.* to giornalista;
create role tifoso;
grant connect, select ON CAMPIONATO.GIOCATORI to tifoso;
create user sonia IDENTIFIED BY soniadip;
create user carla IDENTIFIED BY aldo;
create user enzo IDENTIFIED BY rossi;
create user antonio IDENTIFIED BY bianchi;
grant giornalista to sonia;
grant giornalista to carla;
grant tifoso to enzo;
grant tifoso to antonio;
```

modifisco Vai alle Tabelle nel Campionato

Tutto ciò che facciamo sul nostro database riguarda  
una transazione

# Transazioni

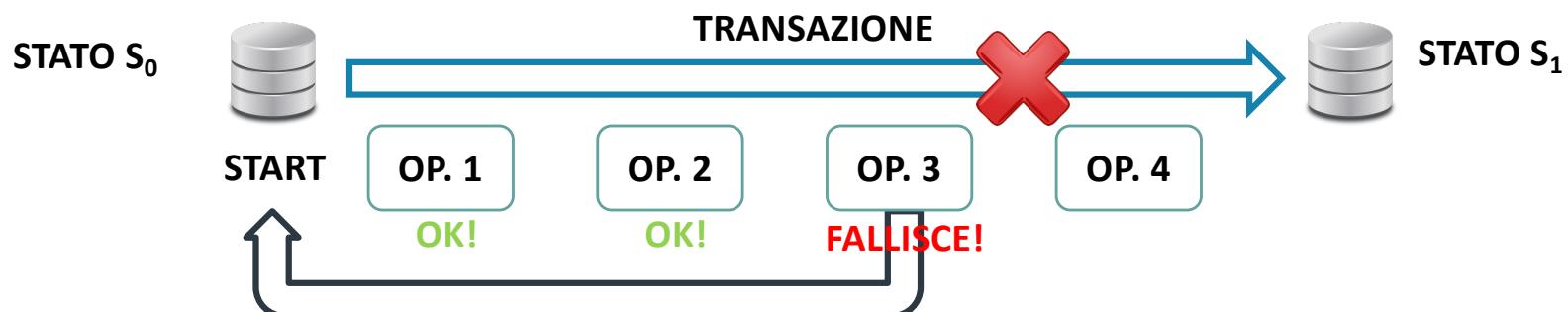
- Insieme ordinato di operazioni (di lettura e scrittura) da considerare indivisibile ("atomico"), corretto anche in presenza di concorrenza e con effetti definitivi
- Una transazione inizia al primo comando SQL dopo la "connessione" alla base di dati oppure alla conclusione di una precedente transazione (lo standard indica anche un comando **start transaction**, non obbligatorio, e quindi non previsto in molti sistemi)

```
start transaction; // begin transaction
update ContoCorrente
set Saldo = Saldo + 10
where NumConto = 12202;
update ContoCorrente
set Saldo = Saldo - 10
where NumConto = 42177;
commit work; oppure abort
```

Opzionale

# Transazioni

- Conclusione di una transazione
  - **commit [work]**: le operazioni specificate a partire dall'inizio della transazione vengono eseguite sulla base di dati (salvati nella base dati)
  - **rollback [work]**: si rinuncia all'esecuzione delle operazioni specificate dopo l'inizio della transazione
- Molti sistemi prevedono una modalità **autocommit**, in cui ogni operazione forma una transazione

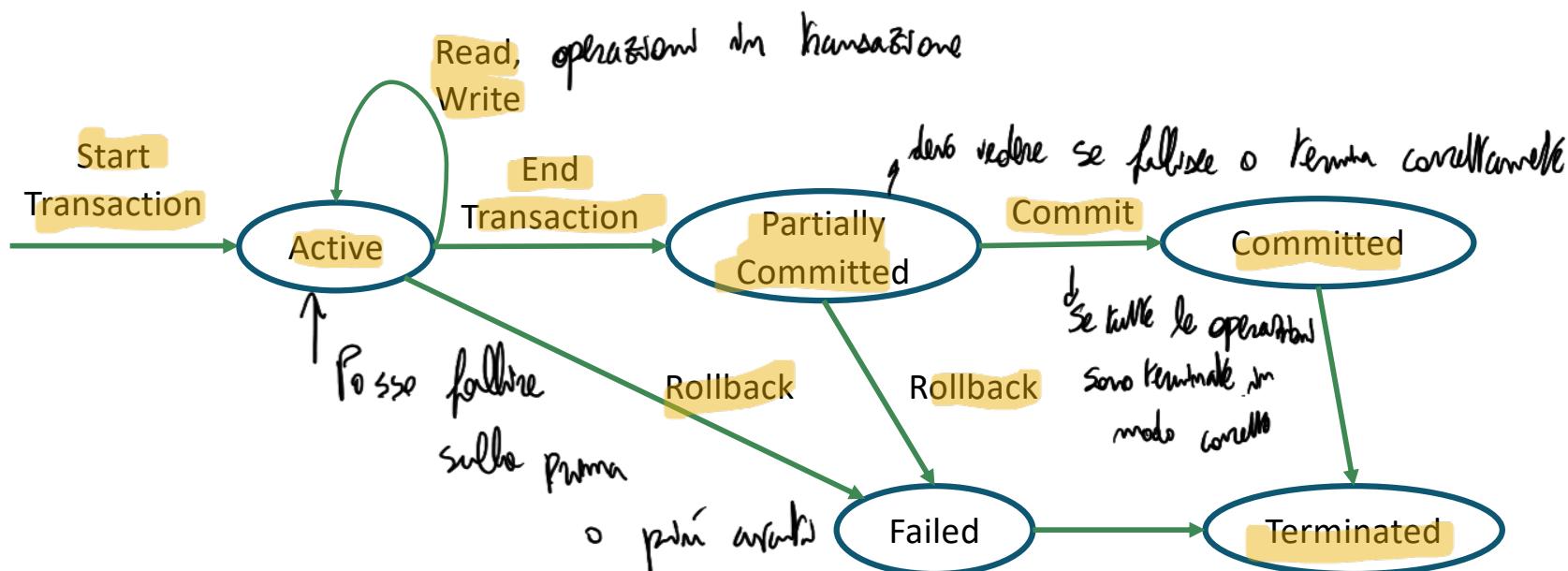


# Transazioni ben formate

---

- Transazione “*Ben Formata*”:
  - Inizia con **start transaction** (oppure **begin transaction**)
  - Termina con **end transaction**;
  - Viene eseguito un solo **Commit** o un solo **rollback**;
  - Non avvengono letture e/o scritture dopo il Commit/Rollback.
- Proprietà garantita dai sistemi.

# Stati di una transazione



# Proprietà delle transazioni

---

- Proprietà ("acide"):
  - **A**tomicità: Una transazione è una unità indivisibile
  - **C**onsistenza: Una transazione deve lasciare il DB in uno stato consistente con i vincoli.
  - **I**solamento: Una transazione deve agire in maniera indipendente dalle altre.
  - **D**urabilità (persistenza): Gli effetti di una transazione che ha effettuato il Commit non devono mai essere persi.

# Atomicità

- Il modello di esecuzione è **tutto o niente**.
- Una transazione può non andare a buon fine:
  - per decisione autonoma;
  - per decisione del DBMS;
  - per errori e/o guasti.
- L'atomicità ci assicura che, **indipendentemente dal momento e dai motivi dell'abort**, le eventuali azioni già effettuate vengono disfatte (operazione di undo).
- Durante l'esecuzione della transazione, le **varie operazioni non sono visibili al mondo esterno**.
- Dopo l'eventuale commit, le **operazioni effettuate sono tutte rese visibili al mondo esterno**.



# Consistenza

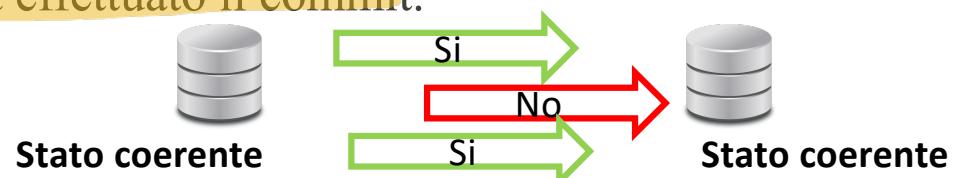
- Una transazione lascia il DB in uno stato consistente.
- SET CONSTRAINTS { ALL | name [, ...] }  
    { DEFERRED | IMMEDIATE }
- **Vincoli Immediati:**
  - se è violato un vincolo immediato, il corrispondente comando ritorna un codice di errore;
  - se tale errore è gestito dal codice della transazione, questa può provare a continuare.
- **Vincoli Ritardati:**
  - il controllo sui vincoli deferred avviene quando è effettuato il commit;
  - se qualche vincolo è violato, la transazione è uccisa in extremis.



# Isolamento

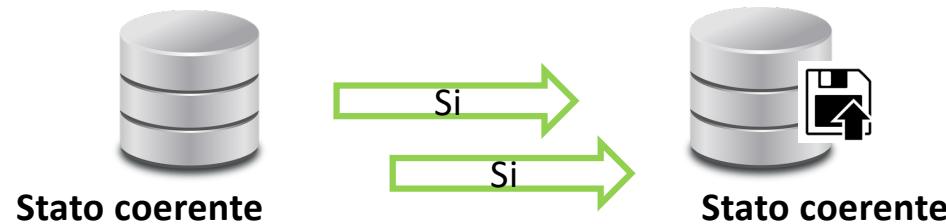
*Alcuni esempi*

- Per migliorare i tempi medi di risposta (TpS), è necessario eseguire più transazioni in maniera concorrente.
- La proprietà di isolamento garantisce che:
  - il risultato di un insieme di transazioni eseguite in maniera concorrente è in qualche modo equivalente a quello che si otterrebbe se le transazioni fossero eseguite una dopo l'altra.
  - venga evitato il Rollback a catena (l'abort di una transazione provoca l'abort di un'altra transazione e così via – *effetto domino*);
  - il Rollback a catena sarebbe particolarmente pericoloso qualora coinvolgesse transazioni che hanno già effettuato il commit.



# Durability (Persistenza)

- L'effetto di una transazione che ha effettuato il commit non deve mai essere perso.
- La persistenza fornisce meccanismi per rispondere ai malfunzionamenti hw/sw del sistema.



---

# ESERCIZI

# Esercizio 1

---

Dato il seguente schema

**AEROPORTO(Città, Nazione,NumPiste)**

**VOLO(IdVolo,GiornoSett,CittàPart,OraPart, CittàArr,OraArr,TipoAereo)**

**AEREO(TipoAereo,NumPasseggeri,QtaMerci)**

scrivere le seguenti interrogazioni SQL

1. Le città con un aeroporto di cui non è noto il numero di piste;
2. Le nazioni da cui parte e arriva il volo con codice AB123;
3. Il numero di voli internazionali che partono il sabato da Napoli;
4. Le città francesi da cui partono più di venti voli alla settimana diretti in Italia;
5. Il massimo numero di passeggeri che possono arrivare in un aeroporto italiano dalla Spagna la Domenica  
(se vi sono più voli, si devono sommare i passeggeri)

# Esercizio 1

---

Le città con un aeroporto di cui non è noto il numero di piste

```
select Città  
from AEROPORTO  
where NumPiste is NULL
```

Le nazioni da cui parte e arriva il volo con codice AB123

```
select A1.Nazione, A2.Nazione  
from AEROPORTO as A1 join VOLO on A1.Città=CittàArr  
join AEROPORTO as A2 on CittàPart=A2.Città  
where IdVolo= 'AB123'
```

# Esercizio 1

---

Il numero di voli internazionali che partono il sabato da Napoli

```
select count(*)
from VOLO join AEROPORTO on CittàArr=Città
where CittàPart = 'Napoli' and Nazione <> 'Italia' and GiornoSett= 'Sabato'
```

Le città francesi da cui partono più di venti voli alla settimana diretti in Italia

```
select CittàPart
from AEROPORTO as A1 join VOLO on A1.Città=CittàPart
join AEROPORTO as A2 on CittàArr=A2.Città
where A1.Nazione='Francia' and A2.Nazione= 'Italia'
group by CittàPart
Having count(*) >20
```

# Esercizio 1

Il massimo numero di passeggeri che possono arrivare in un aeroporto italiano dalla Spagna la domenica  
(se vi sono più voli, si devono sommare i passeggeri)

```
create view Passeggeri(Numero)
as select sum ( NumPasseggeri )
from AEROPORTO as A1 join VOLO on A1.Città=CittàPart
join AEROPORTO as A2 on A2.Città=CittàArr
join AEREO on VOLO.TipoAereo=Aereo.TipoAereo
where A1.Nazione='Spagna' and A2.Nazione='Italia' and GiornoSett='Domenica'
group by A2.Città

select max(Numero)
from Passeggeri
```

# Esercizio 2

---

Dato il seguente schema

```
DISCO(NroSerie, TitoloAlbum, Anno, Prezzo)
CONTIENE(NroSerieDisco, CodiceReg, NroProg)
ESECUZIONE(CodiceReg, TitoloCanz, Anno)
AUTORE(Nome, TitoloCanzone)
CANTANTE(NomeCantante, CodiceReg)
```

scrivere le seguenti interrogazioni SQL

1. I cantautori (persone che hanno cantato e scritto la stessa canzone) il cui nome inizia per ‘D’;
2. I cantanti del disco che contiene il maggior numero di canzoni;

# Esercizio 2

I cantautori (persone che hanno cantato e scritto la stessa canzone) il cui nome inizia per ‘D’

```
select NomeCantante  
from CANTANTE join ESECUZIONE on CANTANTE.CodiceReg=ESECUZIONE.CodiceReg  
join AUTORE on ESECUZIONE.TitoloCanz=AUTORE.TitoloCanzone  
where Nome=NomeCantante and Nome like 'd%'
```

```
create view DiscoNumerato (NroSerieDisco,NumCanzoni)  
as select NroSerieDisco , count(*)  
from CONTIENE  
group by NumSerieDisco
```

```
select NomeCantante  
from CANTANTE join CONTIENE on  
CANTANTE.CodiceReg=CONTIENE.CodiceReg  
join DiscoNumerato on CONTIENE.NroSerieDisco=DiscoNumerato.NroSerieDisco  
where NumCanzoni= (select max (NumCanzoni) from DiscoNumerato)
```

I cantanti del disco che contiene il maggior numero di canzoni

# Esercizio 3

Definire sulla tabella ***Impiegato*** il vincolo che il dipartimento ***Amministrazione*** abbia meno di 100 dipendenti, con uno stipendio medio superiore ai 40 mila €

```
check (100 >= ( select count(*)
                  from Impiegato
                  where Dipartimento='Amministrazione' )
       and 40000 <= ( select avg(Stipendio)
                      from Impiegato
                      where Dipartimento='Amministrazione'))
```

## Esercizio 4

Definire (con una opportuna notazione) su una relazione PAGHE (Matricola, StipLordo, Ritenute, StipNetto, OK) un vincolo che imponga che il valore di OK `e:

- zero se StipNetto è pari alla differenza fra StipLordo e Ritenute
- uno altrimenti.

(Verifica = 0 and (Netto = StipLordo-Tasse))

or

(Verifica = 1 and (Netto <> StipLordo-Tasse))

# Esercizio 5

Definire a livello di schema il vincolo che il massimo degli stipendi degli impiegati di dipartimenti con sede a Firenze sia minore dello stipendio di tutti gli impiegati del dipartimento Direzione.

```
create assertion ControlloSalari
check ( not exists( select *
                      from Impiegato join Dipartimento on
                                Impiegato.Dipartimento=Dipartimento.Nome
                     where Dipartimento.Città='Firenze' and
                           Stipendio > ( select min(Stipendio)
                                from Impiegato
                     where Dipartimento='Direzione' ))
```

# Esercizio 6

Dato lo schema relazionale

**IMPIEGATO (Nome, Salario, DipNum)**  
**DIPARTIMENTO (DipNum, NomeManager)**

Definire le seguenti regole attive in Oracle e DB2

1. una regola, che quando il dipartimento è cancellato, mette ad un valore di default (99) il valore di DipNum degli impiegati appartenenti a quel dipartimento;
2. una regola che cancella tutti gli impiegati appartenenti a un dipartimento quando quest'ultimo è cancellato;

# Esercizio 6

---

una regola, che quando il dipartimento è cancellato, mette ad un valore di default (99) il valore di DipNum degli impiegati appartenenti a quel dipartimento;

```
create trigger T1
after delete on DIPARTIMENTO
for each row
when (exists (select *
               from IMPIEGATO
               where DipNum=Old.DipNum))
update IMPIEGATO.DipNum = 99
```

# Esercizio 6

---

una regola che cancella tutti gli impiegati appartenenti a un dipartimento quando quest'ultimo è cancellato

```
create trigger T2
after delete on DIPARTIMENTO
for each row
when (exist (select *
              from IMPIEGATO
              where DipNum=Old.DipNum) )
delete from IMPIEGATO where DipNum=Old.DipNum
```

# Esercizio 7

---

Marco concede a Luigi e a Francesco l'autorizzazione di **select** e di concedere a loro volta l'autorizzazione

```
Marco: grant select on Table1 to Luigi,Francesco with grant option
```

Marco revoca l'autorizzazione di Francesco e tramite **cascade** anche di Luigi. Ora solo Marco ha autorizzazioni sulla tabella.

```
Marco: revoke select on Table1 from Francesco cascade
```