

Vincoli di Integrità

Approccio procedurale

Antonella Poggi

Dipartimento di informatica e Sistemistica
Università di Roma "La Sapienza"

Progetto di Applicazioni Software
Anno accademico 2008-2009

Questi lucidi sono stati prodotti sulla base del materiale preparato per il corso di Progetto di Applicazioni Software da A. Calì e da D. Lembo.

Transazioni e vincoli

- Uno stato di un database si dice **consistente** se esso soddisfa tutti i vincoli di integrità
- Esistono due nozioni di consistenza:
 1. consistenza **a livello di transazione**: richiede che i dati siano consistenti alla fine di ciascuna transazione, ma è possibile che all'interno della transazione i dati attraversino stati inconsistenti
 2. consistenza **a livello di statement**: richiede che dopo ogni statement i dati siano in uno stato consistente. Questa nozione implica la precedente (è più restrittiva)

Transazioni e vincoli in MySQL

Il fatto che in MySQL non sia possibile differire il controllo dei vincoli (ed anche dei trigger) implica che la consistenza sia sempre verificata a livello di statement.

Questo può creare problemi, visto che noi saremmo invece interessati ad una consistenza a livello di transazione.

In particolare, in presenza di cicli di integrità referenziale (inclusioni o foreign key), ci troviamo nella situazione di non riuscire a popolare lo schema.

Transazioni e vincoli in MySQL (Esempio)

```
CREATE TABLE persona (  
  cod_fis VARCHAR(20) PRIMARY KEY,  
  nome VARCHAR(20),  
  cognome VARCHAR(20)  
);
```

```
CREATE TABLE residenza (  
  persona VARCHAR(20) PRIMARY KEY,  
  citta VARCHAR(20)  
);
```

Per semplicità omettiamo la definizione della tabella `citta`

```
ALTER TABLE persona ADD CONSTRAINT fk_persona_residenza  
    FOREIGN KEY(cod_fis) REFERENCES residenza(persona);
```

```
ALTER TABLE residenza ADD CONSTRAINT fk_residenza_persona  
    FOREIGN KEY(persona) REFERENCES persona(cod_fis);
```

Avendo definito i vincoli dopo la creazione delle tabelle abbiamo risolto un primo problema relativo al ciclo di foreign key che si verifica in fase di definizione dello schema: la definizione di ciascun vincolo richiede che sia già stata creata la tabella a cui si fa riferimento.

Non riusciamo però a popolare lo schema. Infatti eseguendo il comando

```
INSERT INTO persona VALUES  
( 'MRORSS72M21M' , 'Mario' , 'Rossi' );
```

otteniamo

```
ERROR 1452 (23000): Cannot add or update a child row:  
a foreign key constraint fails (`myDB/persona`,  
CONSTRAINT `fk_persona_residenza` FOREIGN KEY (`cod_fis`)  
REFERENCES `residenza` (`persona`))
```

Analogamente se facciamo eseguire prima il comando

```
INSERT INTO residenza VALUES ( 'MRORSS72M21M' , 'Roma' );
```

Transazioni e vincoli in MySQL (Esempio)

Non possiamo risolvere il problema mantenendo il ciclo, ma siamo costretti a “spezzare” il ciclo e gestire gli inserimenti in maniera transazionale in modo che non violino il vincolo.

Assumiamo di aver cancellato il FK da Persona verso residenza.

Una soluzione possibile prevede l'uso di una stored procedure (e quindi non potrà essere una soluzione esclusivamente dichiarativa).

Transazioni e vincoli in MySQL (Esempio)

```
DELIMITER |
CREATE PROCEDURE trans_insert(cf VARCHAR(20),
    p VARCHAR(20), n VARCHAR(20), c VARCHAR(20))
BEGIN
    INSERT INTO residenza VALUES (cf,c);
    INSERT INTO persona VALUES (cf,p,n);
END |
```

```
DELIMITER ;
CALL trans_insert
('MRORSS72M21B9M','mario','rossi','roma');
```

Non tutti i nostri problemi sono però risolti.

Transazioni e vincoli in MySQL (Esempio)

Abbiamo un comportamento transazionale? Cosa succede se ci sono degli errori negli inserimenti (ad esempio dovuti a duplicati di chiave)?

Per risolvere questi problemi modifichiamo il codice come segue:

```
CREATE PROCEDURE trans_insert(cf VARCHAR(20),
    p VARCHAR(20), n VARCHAR(20), c VARCHAR(20))
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        ROLLBACK;
        SET AUTOCOMMIT = 1;
        SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
    END;
    SET AUTOCOMMIT = 0;
    SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
    INSERT INTO residenza VALUES (cf,c);
    INSERT INTO persona VALUES (cf,p,n);
    COMMIT;
    SET AUTOCOMMIT = 1;
    SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
END
```

Transazioni e vincoli in MySQL (Esempio)

- Abbiamo assunto che lo stato di default per l'autocommit sia 1, e per il livello di isolamento sia REPEATABLE READ (alla fine, autocommit e isolamento sono impostati nuovamente a tali valori)
- Abbiamo inoltre assunto che la procedura **non** sia invocata all'interno di una transazione più complessa: infatti l'esecuzione del commit o del rollback causerebbe anche il commit o il rollback di altri statement precedentemente eseguiti
- Si noti infine che la procedura non segnala il suo esito a chi la ha invocata. Per risolvere questo problema, bisogna fare uso di opportuni parametri di output per notificare all'utente se l'inserimento ha avuto successo o meno (lasciato come esercizio)

Transazioni e vincoli in MySQL (Esempio)

- Resta però il problema di garantire che non si eseguano inserimenti direttamente sulla tabella `persona`! Questo problema è difficilmente risolvibile se per qualsiasi utente è possibile interagire con il database attraverso un interprete di comandi SQL
- Ci sono casi particolari in cui il problema è risolvibile con l'uso di trigger (cf. esercitazione)
- Se l'accesso avviene solo attraverso client che offrono esclusivamente funzionalità cablate nel codice, l'integrità è garantita (assunzione semplificatrice)

Approccio procedurale alla implementazione dei vincoli

- Come già detto, i vincoli implementati tramite gli strumenti del DBMS possono essere implementati in maniera procedurale attraverso l'uso di User Defined Function (UDF), Stored Procedure o Trigger
- Trigger, Stored Procedure e UDF vengono specificati in modo **procedurale**, dato che con essi definiamo delle **operazioni** da eseguire perchè il vincolo sia rispettato.
- È bene comunque che l'analisi dei vincoli produca una specifica **dichiarativa**, in generale più chiara e rigorosa, e più vicina all'SQL, che è il nostro più diretto strumento per l'implementazione dei vincoli.

User Defined Function (esempio)

Sia data la seguente relazione

```
UTENTE (ID_UTENTE, NOMINATIVO, DATA_NASCITA,  
        RECAPITO_TEL, INDIRIZZO_EMAIL)
```

Sia dato il seguente vincolo:

l'indirizzo di e-mail di un utente è una stringa formata da simboli alfanumerici e dai simboli '@', '-', '_' e '.'

Si assuma di aver realizzato la seguente stored function (di cui omettiamo il corpo).

```
create function verificaEMail(email varchar(30))  
  returns int
```

che restituisce 1 se la stringa passata come argomento verifica il vincolo, 0 altrimenti.

User Defined Function (esempio)

Utilizziamo la precedente funzione per imporre il vincolo

```
ALTER TABLE UTENTE ADD CONSTRAINT verifica_mail  
CHECK(verificaEMail(INDIRIZZO_EMAIL)=1);
```

User Defined Function

- Le UDF sono un potente strumento per estendere proceduralmente SQL
- tramite il meccanismo appena visto, possiamo implementare facilmente qualsiasi vincolo di stato
- il vincolo precedente sarebbe piuttosto complicato da implementare solo tramite l'uso di SQL
- In generale, le UDF possono aiutarci nei casi in cui sia particolarmente complicato implementare un vincolo esclusivamente utilizzando SQL, oppure se il DBMS usato non consente di utilizzare i costrutti di SQL necessari all'implementazione del vincolo

User Defined Function in MySQL

- **Questo però non è il caso di MySQL**, per il quale non è ammesso l'uso di UDF all'interno di una condizione di una clausola `CHECK` (sappiamo in verità che l'uso della clausola `CHECK` non è affatto supportato)
- Nei casi in cui avremmo usato una UDF, MySQL ci forza ad usare un **Trigger**
- Si noti però che **il trigger non è uno strumento che nasce per la gestione di vincoli di stato**, in quanto ha sempre bisogno di un'evento innescante, e quindi, implicitamente, di una transizione da uno stato ad un altro!!

Trigger in MySQL (esempio)

Per l'esempio visto prima, abbiamo bisogno di realizzare un (row-level) trigger, il quale nei casi di inserimento e di aggiornamento dell'indirizzo e-mail innesca una stored procedure che effettua il controllo dell'email.

Creiamo il trigger per il caso di inserimento dei dati

```
CREATE TRIGGER tgr_email  
BEFORE INSERT ON UTENTE  
FOR EACH ROW  
CALL Proc_VerificaEMail(NEW.indirizzo_email);
```

Analogamente per il caso di aggiornamento (UPDATE)

Nota

Da un trigger non è possibile restituire un risultato. Nell'esempio appena visto, abbiamo perciò usato una procedura (`Proc_VerificaEMail`) invece della funzione menzionata in precedenza (`VerificaEMail`).

Il comportamento del trigger sarà quello di consentire l'inserimento se il controllo sul formato dell'email ha successo, bloccarlo in caso contrario.

Le slide che seguono spiegano come è possibile, in generale, ottenere questo comportamento in MySQL.

Operazioni critiche

Per implementare un vincolo per mezzo di un trigger, la prima cosa da fare è identificare le **operazioni critiche** che possono portare alla sua violazione

- Tipicamente operazioni di inserimento/aggiornamento/cancellazione
- Se una operazione non può violare in alcun modo il vincolo (per esempio, la cancellazione nel caso del vincolo sull'email delle slide precedenti), è inopportuno identificare tale operazione come critica, giacché si farebbero controlli inutili che porterebbero ad un peggioramento delle prestazioni.

Trigger e vincoli

I trigger sono basati sul modello

evento-condizione-azione, e possono essere usati per implementare vincoli come segue:

- L'evento è l'operazione critica che lancia l'esecuzione del trigger
 - ⇒ quando viene catturato all'interno di una transazione, l'esecuzione è **differita** al termine della transazione, usando la modalità apposita prevista nello standard
 - ⇒ altrimenti, conviene definire il trigger in maniera tale che la condizione sia verificata prima (opzione BEFORE) che sia eseguita l'operazione critica che l'ha innescato (l'uso dell'opzione AFTER richiederebbe di disfare l'esito dell'operazione in caso di violazione)

- La condizione è la **negazione del vincolo di integrità**
- L'azione dipende dalla politica di reazione alla violazione del vincolo; può essere:
 - una notifica di errore e il lancio di una eccezione, se la politica scelta è quella di impedire l'esecuzione dell'operazione
 - l'esecuzione di una transazione costituita da più operazioni che insieme portano al soddisfacimento del vincolo (per es. una cancellazione in cascata in caso di violazione di un vincolo di inclusione a seguito di una delete).

Trigger e vincoli (in Mysql)

- In MySQL non è possibile usare la modalità DEFERRED per richiedere che il trigger sia eseguito al termine della transazione
⇒ vedremo più avanti come fare quando si deve passare da stati inconsistenti all'interno di una stessa transazione
- In MySQL non è possibile definire più trigger innescati dallo stesso evento
⇒ è necessario implementare tutti i controlli di integrità da farsi prima dell'inserimento con un trigger unico (nell'ordine in cui si vuole siano eseguiti)

Confronto con i vincoli di foreign key

- La gestione dei vincoli di foreign key da parte del DBMS può essere “simulata” con un trigger così caratterizzato:
 - Evento: inserimento/cancellazione/aggiornamento
 - Condizione: valore da inserire non presente nella tabella referenziata/ valore da cancellare presente nella tabella che referencia/ valore da modificare presente nella tabella che referencia

- azione:
 - * notifica di errore e il lancio di una eccezione, se l'opzione scelta è RESTRICT
 - * cancellazione/aggiornamento in cascata, se l'opzione scelta è ON DELETE/UPDATE CASCADE
 - * assegnazione a NULL o a DEFAULT, se l'opzione scelta è ON DELETE/UPDATE SET NULL/DEFAULT

Uso delle eccezioni per impedire l'esecuzione di un'operazione critica

- Per impedire l'esecuzione di un'operazione, è prassi comune fare uso di eccezioni
- Per gestire le eccezioni, lo standard prevede solamente delle “non-core features” che ancora non sono di fatto implementate interamente da nessun DBMS
- Per lanciare un'eccezione, si ricorre pertanto all'esecuzione di un'operazione che genera sempre un'eccezione, in modo tale da bloccare l'esecuzione dell'operazione critica e di conseguenza la violazione del vincolo

Esempi di procedure che generano sempre un'eccezione (in Mysql)

- FETCH su un cursore vuoto:

```
CREATE PROCEDURE eccezione ()
BEGIN DECLARE c1 VARCHAR(14);
DECLARE crs CURSOR FOR (SELECT * FROM mysql.user
                        WHERE 0=1);

OPEN crs;
FETCH crs INTO c1;
END
```

- SELECT su una colonna che non esiste; astuzia: si nomina la colonna con la stringa di errore che vogliamo sia generata:

```
CREATE PROCEDURE eccezione ()  
    UPDATE `Vincolo violato` SET X=1;
```

Esempio

```
CREATE TABLE padre (  
cod_fis    VARCHAR(14) PRIMARY KEY,  
nome      VARCHAR(20),  
cognome   VARCHAR(20)  
);
```

```
CREATE TABLE paternita (  
padre VARCHAR(20),  
figlio VARCHAR(14),  
PRIMARY KEY(padre,figlio)  
);
```

Vogliamo implementare il vincolo

$\text{padre}[\text{nome}] \subseteq \text{paternita}[\text{padre}]$

che impone ad ogni padre di avere almeno un figlio (per semplicità omettiamo altri vincoli sullo schema).

Esempio

```
DELIMITER |

CREATE FUNCTION inclusione1 (nome VARCHAR(20)) RETURNS bool
BEGIN
DECLARE soddisfatto bool;
SELECT count(*) INTO soddisfatto FROM paternita
                                WHERE padre=nome;
RETURN soddisfatto;
END |

CREATE TRIGGER tgr_incl BEFORE INSERT ON padre
FOR EACH ROW
    IF inclusione1(NEW.nome)=FALSE
    THEN CALL eccezione();
    END IF |

DELIMITER ;
```

Esempio (Cont.)

Se ora proviamo ad inserire una tupla dentro a `padre` il cui attributo `nome` è un valore che non compare dentro alla tabella `paternità` otteniamo:

```
INSERT INTO padre VALUES ('XX4','Mario','Rossi');
```

```
ERROR 1146 (42S02):
```

```
Table 'lezione_vincoli.Vincolo violato' doesn't exist
```

Inserimento in caso di vincoli di foreign key ciclici in Mysql

- Abbiamo già visto che in caso di vincoli ciclici, una possibilità è quella di “spezzare” i ciclo, ovvero rinunciare ad implementare un vincolo di integrità a livello di DBMS e rimandarne la gestione alle applicazioni che accedono alla base di dati
 - soluzione concettualmente sbagliata
 - accettabile quando la base di dati è acceduta solamente da applicazione
 - non accettabile quando la base di dati può essere acceduta dalla linea di comando (caso dei progetti svolti per questo corso)

- Se non si spezza il ciclo, allora il popolamento delle tabelle coinvolte da vincoli ciclici viene impedito per fare in modo che il vincolo sia soddisfatto
 - ⇒ Problema: non si riesce a popolare le tabelle neanche attraverso l'uso di una procedura transazionale scritta appositamente per garantire la consistenza della base di dati (cf. slide precedenti)
 - ⇒ Soluzione: modificare la procedura transazionale in maniera tale che al suo interno: (i) disabiliti i vincoli (triggers e/o FOREIGN KEY), (ii) effettui l'inserimento, (iii) riabiliti i vincoli

Inserimento in caso di vincoli di foreign key ciclici in Mysql (Esempio)

Si mantengono i due vincoli di foreign key e si modifica la procedura transazionale vista in precedenza, per un inserimento consistente:

```
CREATE PROCEDURE trans_insert(cf VARCHAR(20),
                             p VARCHAR(20), n VARCHAR(20), c VARCHAR(20))
BEGIN
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
  BEGIN
    ROLLBACK;
    SET AUTOCOMMIT = 1;
    SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
  END;
```

```
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET foreign_key_checks=0;
INSERT INTO residenza VALUES (cf,c);
INSERT INTO persona VALUES (cf,p,n);
SET foreign_key_checks=1;
COMMIT;
SET AUTOCOMMIT = 1;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
END
```

Trigger e transazioni

- Quando i trigger sono innescati all'interno di **transazioni**, lo standard SQL prevede la possibilità di eseguirli in modalità immediata o differita
- Quando i trigger sono usati per implementare vincoli di integrità ed è necessario avere dati inconsistenti durante la transazione (che però al termine riporta il database in uno stato di consistenza), allora la modalità dovrebbe essere differita
⇒ e.g. caso di inserimenti in tabelle coinvolte in cicli di vincoli di inclusione, o cicli di foreign key ed inclusioni

Trigger e transazioni (in MySQL)

- Poiché in MySQL non esiste la modalità differita, per implementare alcuni vincoli che richiedono l'impostazione di transazione, può essere necessario disabilitare e poi riabilitare i trigger
→ per esempio per gestire operazioni critiche su tabelle su cui vertono cicli di vincoli di inclusione

Inserimenti in presenza di cicli di vincoli di inclusione (in Mysql)

- Si definiscono dei trigger che impediscono inserimenti nelle tabelle che violino i vincoli di inclusione
- In maniera analoga a quanto fatto per i vincoli di foreign key, si definisce una procedura transazionale per permettere inserimenti consistenti, e una variabile ad-hoc per controllare l'esecuzione del trigger
 - all'interno della a procedura transazionale per l'inserimento si usa una variabile locale che viene settata ad un valore diverso da NULL prima di eseguire l'inserimento e poi nuovamente a NULL
 - si definisce un trigger che a fronte di un

inserimento, controlla il valore di tale variabile, e se è pari a NULL (una variabile che non è mai stata inizializzata, di default, è pari a NULL), allora verifica se il vincolo è violato e, se lo è, lancia un'eccezione, altrimenti non fa nulla.

Esempio: inserimenti in presenza di vincoli di inclusione

Si consideri la base di dati precedentemente definita, costituita dalle tabelle `padre` e `paternità` e dal vincolo di inclusione `padre[nome] ⊆ paternità[padre]`

Si supponga ora di avere anche il vincolo:

`paternità[a[padre]] ⊆ padre[nome]`

1. Definiamo due trigger per impedire inserimenti in `padre` e `paternità` che violino entrambi questi vincoli

```
CREATE TRIGGER tgr_incl1 BEFORE INSERT ON padre
FOR EACH ROW
    IF @DIS_TRIG IS NULL
        AND inclusione1(NEW.nome)=FALSE
    THEN CALL eccezione();
END IF;
```

```
CREATE TRIGGER tgr_incl2 BEFORE INSERT ON paternita
FOR EACH ROW
  IF @DIS_TRIG IS NULL
    AND inclusione2(NEW.padre)=FALSE
  THEN CALL eccezione();
END IF;
```

dove `inclusione1` e `eccezione` sono definite come
visto (`inclusione2` è analoga a `inclusione1`)

2. Definiamo una procedura `insert_inclusione` che permette degli inserimenti consistenti in padre e paternità

```
CREATE PROCEDURE insert_inclusione(cf VARCHAR(14),  
n VARCHAR(20), c VARCHAR(20))  
BEGIN  
DECLARE EXIT HANDLER FOR SQLEXCEPTION  
BEGIN  
ROLLBACK;  
SET @DIS_TRIG=NULL;  
SET AUTOCOMMIT = 1;  
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;  
END;
```

```
SET AUTOCOMMIT = 0;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET @DIS_TRIG=1;
INSERT INTO padre VALUES (cf,n,c);
INSERT INTO paternita(padre) VALUES (n);
SET @DIS_TRIG=NULL;
COMMIT;
SET AUTOCOMMIT = 1;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
END
```

Criteri generali per l'implementazione dei vincoli

- Usare SQL quando possibile, i trigger altrimenti
- La ciclicità di vincoli può porre problemi
 - nella definizione dichiarativa di vincoli ciclici di tipo FOREIGN KEY
 - ⇒ si usa il comando ALTER CONSTRAINT
 - nell'esecuzione di operazioni critiche in tabelle cu cui vertono vincoli ciclici
 - ⇒ si può “spezzare” il ciclo rinunciando a rappresentare un vincolo e si rimanda la sua gestione alle applicazioni che accedono alla base di dati
 - * soluzione concettualmente sbagliata

- * accettabile quando la base di dati è acceduta solamente da applicazione
 - * non accettabile quando la base di dati può essere acceduta dalla linea di comando (caso dei progetti svolti per questo corso)
- ⇒ soluzione da preferire: permettere l'inserimento attraverso una transazione che usa il livello massimo di isolamento
- * quando possibile: usare l'opzione DEFERRED
 - * altrimenti: disabilitare temporaneamente i vincoli/trigger

Vincoli in MySQL

- Utilizzare SQL quando possibile
 - costrutti NOT NULL, DEFAULT, ENUM per vincoli di stato
 - costrutti UNIQUE, PRIMARY KEY per vincoli di tabella
 - costrutto FOREIGN KEY per vincoli inter-tabella (integrità referenziale)
- Ricorrere a **trigger** negli altri casi
 - Se il vincolo è di tupla e di stato, allora si può sempre fare
 - Se più trigger devono essere associati allo stesso evento, raggruppare tutti i trigger in un unico trigger (si ricordi che MySQL non è in grado di

- gestire più trigger associati ad un evento)
- Il fatto che in MySQL possono essere definiti solo trigger la cui azione:
 - * deve essere eseguita per ogni tupla coinvolta dall'evento
 - * non modifica la tabella che li ha innescati
- condiziona l'implementazione dei vincoli
e.g. è possibile che l'unico modo di implementare alcuni vincoli sia attraverso l'uso di apposite procedure transazionali per eseguire operazioni critiche, all'interno delle quali sia necessario disabilitare/riabilitare i trigger

Osservazioni importanti

- In linea di principio, tutti i vincoli devono essere implementati a livello di DBMS
- Qualora, si scelga di non implementare alcuni vincoli a livello di DBMS, e si demandi il controllo di integrità alle applicazioni che interagiscono con la base di dati (client), è necessario documentare la base di dati e replicare il controllo su ogni applicazione

- Nell'implementazione di un vincolo è cruciale fare un'attenta analisi della politica di reazione alla sua violazione (cf. Esercitazione 3)
 - e.g. nelle slide, abbiamo sempre assunto di voler impedire un inserimento che viola un vincolo di inclusione; una politica alternativa potrebbe essere quella di inserire nella tabella a destra dell'inclusione una tupla che rende il vincolo soddisfatto
→ è sempre possibile farlo?

- I vincoli di tupla (di stato o di transizione di stato), e vincoli sui singoli attributi in particolare, **vanno verificati generalmente anche a livello dell'applicazione**
 - siccome le transazioni su oggetti potrebbero essere lunghe e complesse, l'errore potrebbe essere rilevato dal DBMS solo dopo un insieme di operazioni che avrebbero potuto essere evitate.

Esempio

Riprendiamo l'esempio visto al termine della lezione sull'approccio procedurale all'implementazione dei vincoli:

Persona (cod_p, nome, salario, tipo)

Azienda (cod_a, nome)

Impiego (cod_p, cod_a)

Vincoli sullo schema logico

1. tipo deve essere "P" o "I" (pensionato o impiegato)
2. salario <= 180.000

3. solo le righe di `Persona` che hanno tipo "I" possono avere il valore dell'attributo `cod_p` che compare in `Impiego[cod_p]`
4. tutte le righe di `Persona` che hanno tipo "I" devono avere il valore dell'attributo `cod_p` che compare in `Impiego[cod_p]`
5. `Impiego[cod_p] \subseteq Persona[cod_p] (foreign key)`
6. `Impiego[cod_a] \subseteq Azienda[cod_a] (foreign key)`

Vincoli implementabili in maniera dichiarativa

1. $Persona[tipo] \in \{P, I\}$ Vincolo di stato tupla

→ Implementato tramite costrutto ENUM.

5. $Impiego[cod_p] \subseteq Persona[cod_p]$ Vincolo di stato inter-tabella

6. $Impiego[cod_a] \subseteq Azienda[cod_a]$ Vincolo di stato inter-tabella

→ implementati tramite costrutto FOREIGN KEY

Creazione tabelle – codice SQL

```
create table PERSONA(  
cod_p    varchar(10) primary key,  
nome     varchar(15) not null,  
cognome  varchar(15) not null,  
salario  int,  
tipo     enum ('I','P')  
);
```

```
create table AZIENDA(  
cod_a  varchar(10) primary key,  
nome   varchar(15) not null  
);
```

Creazione tabelle – codice SQL (cont.)

```
create table IMPIEGO(  
cod_p varchar(10),  
cod_a varchar(10),  
primary key (cod_p, cod_a)  
);  
alter table IMPIEGO add constraint fk_impiego_persona  
foreign key (cod_p) references PERSONA(cod_p)  
on delete cascade on update cascade;  
  
alter table IMPIEGO add constraint fk_impiego_azienza  
foreign key (cod_a) references AZIENDA(cod_a)  
on delete cascade on update cascade;
```

Vincoli che richiedono un approccio procedurale

2. $\text{Persona}[\text{salario}] \leq 180.000$ Vincolo di stato di tupla

3. $\forall cp, n, s, t, ca \text{ Persona}(cp, n, s, t), \text{ Impiego}(cp, ca)$
 $\Rightarrow t = \text{'I'}$ Vincolo di stato inter-tabella

4. $\forall cp, n, s \text{ Persona}(cp, n, s, \text{'I'})$
 $\exists ca | \text{ Impiego}(cp, ca)$ Vincolo di stato inter-tabella

Implementazione (cont.)

Per il vincolo **2. Persona[salario] \leq 180.000**, le operazioni critiche sono:

- l'inserimento in PERSONA
- l'update in PERSONA

```
delimiter |

create function check_salario(sal int) returns bool
begin
if sal >180000
then call eccezione();
end if;
end |

delimiter ;

create trigger tgr_salario_ins before insert
on PERSONA for each row
call check_salario(new.salario);

create trigger tgr_salario_up before update
on PERSONA for each row
call check_salario(new.salario);
```

Implementazione (cont.)

Se provi ad eseguire i seguenti statement SQL

```
insert into persona values('AAA','mario','rossi',160000,'I');  
insert into persona values('BBB','marco','verdi',200000,'I');  
update persona set salario=salario+21000 where cod_p='AAA';
```

Implementazione (cont.)

Per il vincolo 3. $\forall cp, n, s, t, ca \text{ Persona}(cp, n, s, t), \text{Impiego}(cp, ca) \Rightarrow t = \text{'I'}$, le operazioni critiche sono:

- l'inserimento in `IMPIEGO`
- l'update del campo tipo in `PERSONA` (**per semplicità, prevediamo solo il passaggio da "I" a "P"**)

Implementazione (cont.)

Nel seguito ci occupiamo solo dell'update su `Persona[tipo]` (passaggio da "I" a "P"). Il resto dell'implementazione è lasciata per esercizio

- Quando una persona passa da impiegato a pensionato, occorre cancellare la riga corrispondente in `IMPIEGO`.
- Si noti che l'attributo `SALARIO` di `PERSONA` non è più significativo per la riga aggiornata (per rappresentare quest'ultima proprietà si può aggiungere un vincolo di stato su `PERSONA - esercizio`)

- **Per semplicità, assumiamo che non ci siano altri trigger specificati sull'update in Persona.** Quindi assumiamo che non sia definito il trigger `tgr_salario_up` visto prima. Ovviamente, se tale vincolo fosse definito, si potrebbe scrivere un unico trigger per entrambi i vincoli (esercizio).

Implementazione (cont.)

```
delimiter |

create procedure vaiInPensione(old_tipo char,
                               new_tipo char, cod varchar(10))
begin
  if ((old_tipo='I' or old_tipo='i') and
      (new_tipo='P' and new_tipo='p'))
  then delete from impiego where cod_p=cod;
  end if;
end|

delimiter ;

create trigger tgr_pensione after update
on PERSONA for each row
call vaiInPensione(old.tipo,new.tipo,new.cod_p);
```

Implementazione (cont.)

Per il vincolo 4. $\forall cp, n, s \text{ Persona}(cp, n, s, \text{'I'})$
 $\exists ca | \text{Impiego}(cp, ca)$, le operazioni per cui è necessario impostare una transazione sono:

- l'inserimento in PERSONA
- la cancellazione da IMPIEGO (**per semplicità, prevediamo che non sia possibile aggiornare l'impiego**)

Implementazione (cont.)

Inserimento in PERSONA (vincolo 4):

- se il tipo è “P”, tutto va bene;
- se il tipo è “I”, è obbligatorio inserire una riga in IMPIEGO, specificando l’azienda in cui la corrispondente persona è impiegata. **Tutto ciò in modo atomico (una transazione)**. I dati attraverseranno uno stato non consistente, ma saranno poi consistenti alla fine della transazione (si noti che il foreign key `fk_impiego_persona` ed il trigger che vogliamo definire creano un ciclo di dipendenze fra vincoli che dobbiamo considerare quando impostiamo la transazione che garantisce l’atomicità degli inserimenti - esercizio).

Implementazione (cont.)

Cancellazione da `IMPIEGO` (vincolo 4)

Rimangono “appese” le righe di `PERSONA` che erano impiegate nell’azienda cancellata.

Ci sono due approcci possibili, a seconda della politica di reazione alla violazione che si vuole seguire

Implementazione (cont.)

Approcci possibili:

- cancellazione a cascata da Persona
- impedire la cancellazione di aziende che abbiano almeno un impiegato (implementato **con trigger**)
⇒ in questo caso la clausola ON DELETE CASCADE è superflua sulla foreign key `Impiego[cod_a] ⊆ Azienda[cod_a]`