Basi di dati

Capitolo 5:

SQL: caratteristiche evolute

Vincoli di integrità generici: check

- Specifica di vincoli di ennupla (e anche vincoli più complessi, non sempre supportati)

check (Condizione)

Vincoli di integrità generici: check

Il vincolo CHECK assicura che tutti i valori in una colonna soddisfino determinate condizioni.

Se il valore inserito soddisfa il vincolo di check allora inserisce una nuova riga o se ne aggiorna una esistente.

Check, esempio

```
create table Imp
(
Matricola integer primary key,
Cognome character(20),
Nome character(20),
Sesso character not null check (sesso in ('M','F')) ,
Stipendio integer check (Stipendio > 0) ,
Superiore integer,
check (Stipendio <= (select Stipendio
                     from Imp J
                     where Superiore = J.Matricola) )
)
```

non supportata

Check, esempio 2

```
create table Impiegato
(
Matricola character(6),
Cognome character(20),
Nome character(20),
Sesso character not null check (sesso in (‘M’,‘F’))
Stipendio integer,
Ritenute integer,
Netto integer,
Superiore character(6),
check (Netto = Stipendio - Ritenute )
)
```

ok

Check, esempio 3

```
insert into Imp values (
1 , 'Rossi', 'Mario', '', 100, 20, 80 );

insert into Imp values (
2 , 'Neri', 'Mario', 'M', 100, 10, 80 );

insert into Imp values (
3 , 'Rossini', 'Luca', 'M', 70, 20, 50 )
```

Check

Dalla pagina del manuale di MySQL: http://dev.mysql.com/doc/refman/5.0/en/create-table.html

"The **CHECK** clause is parsed but ignored by all storage engines."

Vincoli di integrità generici: asserzioni

- Specifica vincoli a livello di schema

```
create assertion NomeAsser check ( Condizione )

create assertion AlmenoUnImpiegato
    check (1 <= (  select count(*)
                   from Impiegato ))
```

impone che sia sempre presente almeno una riga in *Impiegato*
non supportata

Vincoli di integrità generici: asserzioni

- Questi vincoli, chiamati asserzioni, sono elementi dello "schema", che vengono normalmente utilizzati per specificare "restrizioni" che coinvolgono più tuple o più tabelle
- Alcune asserzioni potrebbero naturalmente richiedere che la tabella "coinvolta" contenga dei dati al proprio interno

Vincoli di integrità generici: asserzioni

- Non tutti i DBMS prevedono tutti i tipi di vincoli, in particolare le asserzioni non sono generalmente supportate
- Quasi tutti i DBMS si limitano a gestire i vincoli che possono essere verificati esaminando una sola tupla
- Motivazione: efficienza della valutazione

Viste

Le viste possono essere considerate come tabelle virtuali. Generalmente, una tabella è caratterizzata da una serie di definizioni ed è il luogo in cui vengono fisicamente memorizzati i dati. Anche una vista dispone di una serie di definizioni, generata nella parte superiore di una o più tabelle o viste, ma non rappresenta il luogo in cui i dati vengono fisicamente memorizzati.

Viste - Sintassi

```
create view NomeVista [ ( ListaAttributi ) ] as SelectSQL
[ with [ local | cascaded ] check option ]

create view ImpiegatiAmmin
      (Nome, Cognome, Stipendio) as
    select Nome, Cognome, Stipendio
    from Impiegato
    where Dipart = 'Amministrazione' and
        Stipendio > 10
```

Creazione di una Vista

create view NomeVista [(ListaAttributi)] as SelectSQL
[with [local | cascaded] check option]

ESEMPIO:

Tabella ***Customer***

Nome	Tipo di Dato
First_Name	char(50)
Last_Name	char(50)
Address	char(50)
City	char(50)
Country	char(25)
Birth_Date	datetime

Creazione di una Vista

create view NomeVista [(ListaAttributi)] as SelectSQL
[with [local | cascaded] check option]

ESEMPIO:

Tabella ***Customer***

Nome	Tipo di Dato
First_Name	char(50)
Last_Name	char(50)
Address	char(50)
City	char(50)
Country	char(25)
Birth_Date	datetime

creiamo una vista denominata ***V_Customer***, in cui sono contenute solo le colonne First_Name, Last_Name e Country da questa tabella.

CREATE VIEW V_Customer
AS SELECT First_Name,
Last_Name, Country
FROM Customer;

Creazione di una Vista

Si dispone di una vista denominata ***V_Customer*** caratterizzata dalla seguente struttura:

Vista ***V_Customer***

Nome	Tipo di Dato
First_Name	char(50)
Last_Name	char(50)
Country	char(25)

Utilizzo di una Vista

Una vista può anche essere utilizzata per applicare le unioni a due tabelle.

In questo caso, gli utenti visualizzano solo una vista anziché due tabelle e l'istruzione SQL che devono utilizzare diviene molto più semplice.

Utilizzo di una Vista

Si supponga di disporre delle seguenti due tabelle:

Tabella *Store_Information*

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

Tabella *Geography*

Region_Name	Store_Name
East	Boston
East	New York
West	Los Angeles
West	San Diego

e che si desideri creare una vista in cui sono visualizzate le informazioni relative alle vendite per regione.

Utilizzo di una Vista

Codice vista:

```
CREATE VIEW V_REGION_SALES
AS SELECT A1.Region_Bame REGION, SUM(A2.Sales) SALES
FROM Geography A1, Store_Information A2
WHERE A1.Store_Name = A2.Store_Name
GROUP BY A1.Region_Name;
```

Utilizzo di una Vista

In questo modo si dispone di una vista, ***V_REGION_SALES***, definita per memorizzare i record delle vendite realizzate in base alla regione.

Se vogliamo trovare il contenuto di questa vista, basta usare una select:

```
SELECT * FROM V_REGION_SALES;
```

risultato:

REGION	SALES
East	700
West	2050

Interrogazioni sulle viste

- Possono fare riferimento alle viste come se fossero relazioni di base

```
select * from ImpiegatiAmmin
```

equivale a (e viene eseguita come)

```
select Nome, Cognome, Stipendio
from Impiegato
where Dipart = 'Amministrazione' and
      Stipendio > 10
```


Aggiornamenti sulle viste

- Ammessi (di solito) solo su viste definite su una sola relazione
- Alcune verifiche possono essere imposte

Esempio di Viste

```
create view ImpiegatiAmminPoveri as
    select *
    from ImpiegatiAmmin
    where Stipendio < 50
    with check option
```

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

Esempio di Viste

```
create view ImpiegatiAmminPoveri as
    select *
    from ImpiegatiAmmin
    where Stipendio < 50
    with check option

update ImpiegatiAmminPoveri
    set stipendio = 60
    where nome = 'Paola'
```


Un'interrogazione non standard

- Interrogazione scorretta

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

- Con una vista

```
create view DipartUffici(NomeDip,NroUffici) as
    select Dipart, count(distinct Ufficio)
    from Impiegato
    group by Dipart;

select avg(cast(NroUffici as decimal(5,2))) as NumeroMedioUffici
from DipartUffici
```

Ancora sulle viste

- La nidificazione nella having non è ammessa in alcuni sistemi

```
select Dipart
from Impiegato
group by Dipart
having sum(Stipendio) >= all
  (select sum(Stipendio)
   from Impiegato
   group by Dipart)
```


Soluzione con le viste

```
create view BudgetStipendi(Dip,TotaleStipendi) as
    select Dipart, sum(Stipendio)
    from Impiegato
    group by Dipart

select Dip
from BudgetStipendi
where TotaleStipendi =(select max(TotaleStipendi)
                       from BudgetStipendi)
```


Viste ricorsive

- Per ogni persona, trovare tutti gli antenati, avendo
  Paternita (Padre, Figlio)
- Serve la ricorsione; in Datalog:

  Discendenza (Antenato: p, Discendente: f) ←
  Paternita (Padre: p, Figlio: f)

  Discendenza (Antenato: a, Discendente: d) ←
  Paternita (Padre: a, Figlio: f) ,
  Discendenza (Antenato: f, Discendente: d)

Viste ricorsive in SQL:1999

```
with Discendenza(Antenato,Discendente) AS
(       select Padre, Figlio
        from Paternita
    union all
        select D.Antenato, Figlio
        from Discendenza D, Paternita
        where D.Discendente = Padre)
select *
from Discendenza
```


Utilità delle Viste

- Per nascondere certe modifiche dell'organizzazione logica dei dati.
- Per **proteggere** i dati
  - Es. si può dare ad un utente accesso solo ad una parte limitata/aggregata dei dati
- Per offrire visioni diverse degli stessi dati senza ricorrere a **duplicazioni**
- Per rendere **più semplici**, o per rendere **possibili**, alcune interrogazioni

Funzioni scalari

- Funzioni a livello di ennupla che restituiscono singoli valori
- Temporali
  - current_date, extract(year from …)
- Manipolazione stringhe
  - char_length, lower
- Conversione
  - cast
- Condizionali
  - …

Funzioni condizionali

- Case, coalesce, nullif

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

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
```


Procedure

Procedure e trigger permettono di scrivere codice per implementare funzionalità e controlli sui dati direttamente nella base di dati.

E' possibile fornire agli utenti della base di dati stessa un ulteriore strato di astrazione verso la struttura logica dei dati e assicurarsi che alcuni controlli essenziali siano sempre eseguiti e non delegati al programma client che si interfaccia con il DBMS.

Procedure (vantaggi)

Creare procedure per manipolare i dati e associarle ai dati nel DBMS è di solito utile per fornire uno strato di astrazione più completo sulla rappresentazione delle informazioni nel database.
Grazie all'uso delle procedure è possibile

- evitare che un programmatore implementi scorrettamente le operazioni di aggiornamento dei dati (soprattutto nel caso coinvolgano piu' tabelle)
- migliorare le prestazioni del database, perch`e l'interazione tra il programma client e la base di dati diminuisce sensibilmente se delle operazioni complesse vengono gestite completamente dal DBMS

Procedure

Una procedura può essere composta da due tipi di istruzioni:

- **dichiarative**: le istruzioni SQL viste finora (CREATE, UPDATE, SELECT), con speciali estensioni per gestire la memorizzazione dei valori estratti dalla base di dati in variabili
- **procedurali**: costrutti comuni dei linguaggi di programmazione, come IF-THEN-ELSE e WHILE-DO.

Procedure - sintassi

Sintassi:

```
CREATE [OR REPLACE] PROCEDURE nome_procedura
   [ (nome_parametro tipo_parametro, … ) ]
IS
   [sezione_dichiarativa]
BEGIN
   executable_section
[EXCEPTION
   sezione_eccezioni]
END [nome_procedura];
```


Procedure - sintassi

Sintassi:

```
CREATE [OR REPLACE] PROCEDURE nome_procedura
   [ (nome_parametro tipo_parametro, … ) ]
IS
   [sezione_dichiarativa]
BEGIN
   executable_section
[EXCEPTION
   sezione_eccezioni]
END [nome_procedura];
```

La sintassi delle procedure non è oggetto di standardizzazione.

Quella descritta di seguito è la sintassi ORACLE/PLSQL

Procedure - sintassi

La clausola opzionale or replace ri-crea la procedura/funzione.

Una procedura può essere cancellata utilizzando il comando **drop procedure**

I tipi di dati validi includono tutti i tipi di dati

Esempio di Procedura

la seguente procedura viene utilizzata per incrementare lo stipendio di tutti gli impiegati che lavorano nel dipartimento fornito dal parametro della procedura. La percentuale dell'aumento di stipendio è data anch'essa da un parametro.

```
create procedure raise_salary(dno number, percentage number DEFAULT 0.5) is
      cursor emp_cur (dept_no number) is
            select SAL from EMP where DEPTNO = dept_no
            for update of SAL;
      empsal number(8);
begin
      open emp_cur(dno);
      loop
            fetch emp_cur into empsal;
            exit when emp_cur%NOTFOUND;
            update EMP set SAL = empsal*((100+percentage)/100)
            where current of emp_cur;
      end loop;
      close emp_cur;
      commit;
end raise_salary
```


Esempio di Procedura

La procedura vista precedentemente può essere richiamata da una shell SQL*Plus utilizzando il commando:

execute raise_salary(10,3);

Se la procedura viene chiamata con il solo parametro 10, viene assunto il valore di default 0.5 come specificato nella lista di parametri nella definizione della procedura. Se la procedura viene chiamata dall'interno di un blocco PL/SQL, la parola chiave execute viene omessa.

Basi di dati attive

- Una base di dati che contiene regole attive (chiamate *trigger*)
- Presentazione:
  - Definizione dei trigger in SQL:1999
  - Definizione dei trigger in DB2 e Oracle
  - Problemi di progetto per applicazioni basate sull'uso dei trigger

Il concetto di trigger

Un costrutto **trigger** esegue un blocco di codice sul database quando si verifica un evento.

Dal punto di vista di SQL, i trigger sono molto simili alle procedure: si tratta di una sorta di procedura particolare che si attiva automaticamente dopo un determinato evento

Il codice di un trigger deve essere semplice e di rapida esecuzione, in quanto viene eseguito dal DBMS ad ogni manipolazione delle tabelle associate.

Il concetto di trigger

I trigger sono utilizzati per diversi scopi nella progettazione di un database:

- mantenere l'integrità referenziale tra le varie tabelle
- mantenere l'integrità dei dati della singola tabella
- monitorare i campi di una tabella ed eventualmente generare eventi ad hoc
- calcolare i valori di campi particolari ad ogni modifica dei dati associati

Il concetto di trigger

- Paradigma: Evento-Condizione-Azione
  - Quando un evento si verifica
  - Se la condizione è vera
  - Allora l'azione è eseguita
- Questo modello consente computazioni reattive
- Regole attive possono gestire:
  - Vincoli di integrità
  - Regole datalog
  - Regole di business
- Problema: è difficile realizzare applicazioni complesse

Evento-Condizione-Azione

- Evento
  - Normalmente una modifica dello stato del database: insert, delete, update
  - Quando accade l'evento, il trigger è *attivato*
- Condizione
  - Un predicato booleano che identifica se l'azione del trigger deve essere eseguita
  - Quando la condizione viene valutata, il trigger è *considerato*
- Azione
  - Una sequenza di update SQL o una procedura
  - Quando l'azione è eseguita anche il trigger è *eseguito*
- I DBMS forniscono tutti i componenti necessari. Basta integrarli.

Il concetto di trigger

Due livelli di granularita':

- di **tupla** (*Row Level*). L'attivazione avviene per ogni tupla coinvolta nell'operazione. Un comportamento orientato alle singole istanze. I trigger a livello di tupla vengono eseguiti una volta sola per ogni riga su cui agisce un'istruzione DML
- di **primitiva** (*Statement Level*). L'attivazione avviene una sola volta per ogni primitiva SQL facendo riferimento a tutte le tuple coinvolte nella primitiva, con un comportamento orientato agli insiemi.
  Es.: se una singola istruzione INSERT avesse inserito 500 righe in una tabella, un trigger a livello di istruzione su tale tabella verrebbe eseguito una sola volta.

SQL:1999, Sintassi

- 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
  - target (tabella controllata)
  - modalità (**`before`** o **`after`**)
  - evento (**`insert`**, **`delete`** o **`update`**)
  - granularità (statement-level o row-level)
  - alias dei valori o tabelle di transizione
  - azione
  - timestamp di creazione

SQL:1999, Sintassi

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

SQL:1999, Sintassi

`create trigger` *NomeTrigger*
Modo Evento **`on`** *TabellaTarget*
[**`referencing`** *Referenza*]
[**`for each`** *Livello*]
[**`when`** (*PredicatoSQL*)]
StatementProcedurale

Tipi di eventi (*modo*)

Le parole chiave BEFORE e AFTER specificano se attivare il trigger prima o dopo che l'evento indicato sia gestito dal DBMS

- BEFORE
  - 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)
  - Normalmente questa modalità è usata quando si vuole verificare una modifica prima che essa avvenga e "modificare la modifica"
- AFTER
  - Il trigger è considerato e eseguito dopo l'evento
  - E' la modalità più comune, adatta alla maggior parte delle applicazioni

Esempio "before" e "after"

- 1. "Conditioner" (agisce prima dell'update e della verifica di integrità)

```
create trigger LimitaAumenti
before update of Salario on Impiegato
for each row
when (New.Salario > Old.Salario * 1.2)
set New.Salario = Old.Salario * 1.2
```

- 2. "Re-installer" (agisce dopo l'update)

```
create trigger LimitaAumenti
after update of Salario on Impiegato
for each row
when (New.Salario > Old.Salario * 1.2)
set New.Salario = Old.Salario * 1.2
```


Tipi di eventi (*evento*)

Dove *Evento* si riferisce alle parole chiave **`insert`**, **`delete`** oppure **`update`**

Granularità degli eventi

Espressa tramite la clausola **`for each`** ***Livello.***

- Modalità statement-level (di default, opzione **`for each statement`**)
  - Il trigger viene considerato e possibilmente eseguito solo una volta per ogni statement (comando) che lo ha attivato, indipendentemente dal numero di tuple modificate (eseguito solo una volta dopo l'attivazione del trigger).
- Modalità row-level (opzione **`for each row`**)
  - Il trigger viene considerato e possibilmente eseguito una volta per ogni tupla modificata (eseguito una volta per ogni riga della tabella di transizione associata al trigger.)
  - Scrivere trigger row-level è più semplice

Granularità degli eventi (`statement`)

Se il livello e' **`statement`**, le variabili si riferiscono alle porzioni della tabella target che subiscono la modifica.

`old_table as` VarTabellaOld | **`new_table as`** VarTabellaNEW

Granularità degli eventi (`row`)

Se il livello e' **`row`**, le variabili si riferiscono alle tuple che subiscono la modifica

`old as` VarTabellaOld | **`new as`** VarTabellaNEW

old fa riferimento alla tupla nello stato precedente alla modifica, mentre new fa riferimento alla tupla nello stato prodotto dalla modifica

Esempio di trigger

Trigger attivato prima di un evento (di tipo *before*):

```
create trigger LimitaAumenti1
before update of Stipendio on Impiegato
for each row
when (new.Stipendio > old.Stipendio * 1.2)
set new.Stipendio = old.Stipendio *1.2
```

Tale assegnamento avviene prima che la modifica della base di dati abbia luogo

Esempio di trigger

Trigger attivato prima di un evento (di tipo *after*):

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

In questo caso il trigger scatta dopo la modifica di stipendio.

Clausola `referencing`

- Dipende dalla granularità
  - Se la modalità è row-level, ci sono due *variabili di transizione* (**`old`** and **`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`** and **`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`**

Esempio di trigger row-level

```
create trigger AccountMonitor
  after update on Account
  for each row
  when new.Totale > old.Totale
  insert values
   (new.NumeroConto,
      new.Totale-old.Totale)
   into Pagamenti
```


Esempio di trigger statement-level

```
create trigger ArchiviaFattureCancellate
   after delete on Fattura
  referencing old_table as VecchieFatture
  insert into FattureCancellate
  (select *
   from VecchieFatture)
```


Esempio di trigger `before`

```
create trigger LimitaAumenti1
  before update of Stipendio on Impiegato
  for each row
  when (new.Stipendio > old.Stipendio * 1.2)
   set new.Stipendio = old.Stipendio * 1.2
```

con trigger di tipo **`before`**, assegnamento **`set`** avviene prima della modifica della base di dati.

Esempio di trigger `after`

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

in questo caso il trigger scatta dopo la modifica di stipendio. Se condizione e' vera il trigger viene attivato.

Triggers in DB2

- Seguono la sintassi e semantica di SQL:1999

```
CREATE TRIGGER [nome trigger] [activation time]
  [trigger event] ON [subject table] REFERENCING
  [object] AS [name] [granularity]
   MODE DB2SQL WHEN [condition]
   BEGIN ATOMIC [triggered action ]
   END
```


Triggers in DB2

- Seguono la sintassi e semantica di SQL:1999
- Esempio: Definire un trigger che, quando un dipartimento è cancellato, mette a NULL il valore di DipNum degli impiegati appartenenti a quel dipartimento

```
CREATE TRIGGER DB2Admin.DelDip
AFTER DELETE ON DB2Admin.Dipartimento
REFERENCING OLD AS OldRow OLD_TABLE AS OldTable
FOR EACH ROW MODE DB2SQL
WHEN ( EXISTS (SELECT * FROM Impiegato WHERE DipNum
  = OldRow.DipNum) )
BEGIN ATOMIC
UPDATE Impiegato
SET DipNum = null
WHERE DipNum = OldRow.DipNum;
END
```


Esecuzione di Trigger in conflitto

- Quando vi sono più trigger associati allo stesso evento (in conflitto) vengono eseguiti come segue:
  - Per primi i **`before`** triggers (*statement-level* e *row-level*)
  - Poi viene eseguita la modifica e verificati i vincoli di integrità
  - Infine sono eseguiti gli **`after`** triggers (*row-level* e *statement level*)
- Quando vari trigger appartengono alla stessa categoria, l'ordine di esecuzione è definito in base al loro timestamp di creazione (i trigger più vecchi hanno priorità più alta)

Modello di esecuzione ricorsivo

- In SQL:1999 i triggers sono associati ad un "Trigger Execution Context" (TEC)
- 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
- Se si verifica un errore o eccezione durante l'esecuzione di una catena di trigger attivati inizialmente da uno statement S, viene fatto un rollback parziale di S

Trigger in Oracle

- Si usa una sintassi differente: 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

`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

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

Conflitti tra i trigger in Oracle

I trigger in Oracle sono immediati e prevedono:

- opzione *before* e *after*
- granularita' a livello di tupla o primitiva

Quattro combinazioni disponibili:

before row
before statement
after row
after statement

Conflitti tra i trigger in Oracle

- L'esecuzione di una primitiva di **insert**, **delete** o **update** in SQL se inframezzata da trigger, vengono attivati col seguente ordine:

1. si eseguono trigger **before statement**
2. per ogni tupla in target coinvolta nella primitiva
   a. si esegue trigger **before row**
   b. si applica primitiva tupla
   c. si eseguono trigger **after row**
3. test per integrita' tabella
4. esecuzione trigger **after statement**

- Quando vari trigger appartengono alla stessa categoria, l'ordine di esecuzione è definito in base al loro timestamp di creazione (i trigger più vecchi hanno priorità più alta)
- "*Mutating table exception*": scatta se la catena di trigger attivati da un **before** trigger T cerca di modificare lo stato della tabella target di T

Esempio of Trigger in Oracle

Evento: **update** of QtaDisponibile in Magazzino
Condizione: Quantità sotto soglia e mancanza ordini esterni
Azione: **insert** of OrdiniEsterni

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


Proprietà formali 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
  - Univoca osservabilità: I trigger sono confluenti e producono verso l' esterno (messaggi, azioni di display) lo stesso effetto
- La terminazione è la proprietà principale

Analisi della terminazione

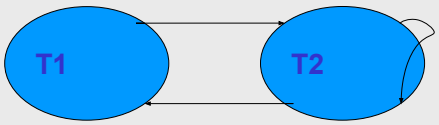
- Si usa una rappresentazione delle regole detta grafo di triggering:
  - Un nodo per ogni trigger
  - Un arco dal nodo t_i al nodo t_j se l'esecuzione dell'azione di t_i può attivare il trigger t_j (ciò può essere dedotto con una semplice analisi sintattica)
- Se il grafo è aciclico, l'esecuzione termina
  - 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 con due trigger

```
T1: create trigger CorreggiContributi
    after update of Stipendio on Impiegato
    referencing new table as NewImp
    update Impiegato
    set Contributi = Stipendio * 0.8
    where Matr in (select Matr
            from NewImp)

T2: create trigger ControllaSogliaBudget
       after update on Impiegato
       when 50000 < (select    (New.Stipendio
                     + New.Contributi)
            from Impiegato)
    update Impiegato
    set Stipendio = 0.9 * Stipendio
```

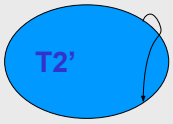
Grafo di triggering corrispondente



- Ci sono due cicli ma il sistema termina.
- Per renderlo non terminante basta cambiare il comparatore nella condizione di T2 oppure moltiplicare per un fattore più grande di 1 nella azione di T2.

Esempio di non terminazione

```
T2':   create trigger ControllaSogliaBudget
       after update on Impiegato
       when New.Stipendio < 50000
   update Stipendio
   set Stipendio = 0.9 * Stipendio
```



Aspetti evoluti delle basi di dati attive

- Modalità di esecuzione (immediata, differita, distaccata)
- Amministrazione delle regole (priorità, gruppi, attivazione e deattivazione dinamica
- Clausola “Instead-of”
- Altri eventi (di sistema, di utente, system-defined)
- Eventi complessi e calcolo degli eventi
- Una nuova categoria di sistema: “stream database”.

Modalità di esecuzione

- E’ il collegamento tra attivazione (evento) e considerazione/esecuzione (condizione e azione)
- Condizione e azione sono sempre valutate assieme
- Caso “immediato”: considerazione e esecuzione assieme all’evento
- Alternative:
  - Differito: il trigger è valutato alla fine della transazione
    - Esempio d’uso: trigger che gestiscono vincoli di integrità che possono essere violati durante una transazione
  - Distaccato: il trigger è valutato in un’altra transazione
    - Esempio d’uso: gestione di una variazione di valore di titoli della borsa

Priorità, attivazioni e gruppi

- Definizione di priorità:
  - Specifica l’ordine di esecuzione dei trigger quando molti di loro vengono attivati dallo stesso evento
  - SQL:1999 indica la priorità di differenti classi di trigger; all’ interno di una classe l’ordine dipende dall’ordine di creazione
- Attivazione/deattivazione dei trigger
  - Non è standard, ma è spesso disponibile
- Organizzazione dei trigger in gruppi
  - Alcuni sistemi consentono di raggruppare trigger e quindi di attivarli/deattivarli come gruppo

Clausola `instead of`

- Alternativa a **`before`** e **`after`**
- Viene eseguita una differente operazione rispetto a quella che ha attivato il trigger
- La semantica è piuttosto pericolosa (l’applicazione fa una cosa e il sistema ne fa un’altra)

  Es: ‘when updating the salary of X, instead update the salary of Y’
- Implementata in alcuni sistemi con limitazioni
  - In Oracle si può usare per ridirigere gli update dalle viste alle tabelle di base in caso di ambiguità

Controllo dell'accesso

- In SQL è possibile specificare chi (utente) e come (lettura, scrittura, …) può utilizzare la base di dati (o parte di essa)
- Oggetto dei privilegi (diritti di accesso) sono di solito le tabelle, ma anche altri tipi di risorse, quali singoli attributi, viste o domini
- Un utente predefinito _system (amministratore della base di dati) ha tutti i privilegi
- Il creatore di una risorsa ha tutti i privilegi su di essa

Privilegi

- Un privilegio è caratterizzato da:
  - la risorsa cui si riferisce
  - l'utente che concede il privilegio
  - l'utente che riceve il privilegio
  - l'azione che viene permessa
  - la trasmissibilità del privilegio

Tipi di privilegi offerti da SQL

- insert: permette di inserire nuovi oggetti (ennuple)
- update: permette di modificare il contenuto
- delete: permette di eliminare oggetti
- select: permette di leggere la risorsa
- references: permette la definizione di vincoli di integrità referenziale verso la risorsa (può limitare la possibilità di modificare la risorsa)
- usage: permette l'utilizzo in una definizione (per esempio, di un dominio)

grant e revoke

- Concessione di privilegi:
  grant < *Privileges* | all privileges > on *Resource*
  to *Users* [with grant option]
  - grant option specifica se il privilegio può essere trasmesso ad altri utenti
  grant select on Department to Stefano
- Revoca di privilegi
  revoke *Privileges* on *Resource* from *Users*
  [restrict | cascade]

Autorizzazioni, commenti

- La gestione delle autorizzazioni deve "nascondere" gli elementi cui un utente non può accedere, senza sospetti
- Esempio:
  - Impiegati non esiste (esiste Impiegati)
  - ImpiegatiSegreti esiste, ma l'utente non è autorizzato
- L'utente deve ricevere lo stesso messaggio

Autorizzazioni, commenti, 2

- Come autorizzare un utente a vedere solo alcune ennuple di una relazione?
  - Attraverso una vista:
    - Definiamo la vista con una condizione di selezione
    - Attribuiamo le autorizzazioni sulla vista, anziché sulla relazione di base

Autorizzazioni, ancora

- (Estensioni di SQL:1999)
- Concetto di ruolo, cui si associano privilegi (anche articolati), poi concessi agli utenti attribuendo il ruolo

Transazione

- Insieme di operazioni da considerare indivisibile ("atomico"), corretto anche in presenza di concorrenza e con effetti definitivi
- Proprietà ("acide"):
  - Atomicità
  - Consistenza
  - Isolamento
  - Durabilità (persistenza)

Le transazioni sono … atomiche

- La sequenza di operazioni sulla base di dati viene eseguita per intero o per niente:
  - trasferimento di fondi da un conto A ad un conto B: o si fanno il prelevamento da A e il versamento su B o nessuno dei due

Le transazioni sono … consistenti

- Al termine dell'esecuzione di una transazione, i vincoli di integrità debbono essere soddisfatti
- "Durante" l'esecuzione ci possono essere violazioni, ma se restano alla fine allora la transazione deve essere annullata per intero ("abortita")

Le transazioni sono … isolate

- L'effetto di transazioni concorrenti deve essere coerente (ad esempio "equivalente" all'esecuzione separata)
  - se due assegni emessi sullo stesso conto corrente vengono incassati contemporaneamente si deve evitare di trascurarne uno

I risultati delle transazioni sono durevoli

- La conclusione positiva di una transazione corrisponde ad un impegno (in inglese commit) a mantenere traccia del risultato in modo definitivo, anche in presenza di guasti e di esecuzione concorrente

Transazioni in SQL

- 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)
- Conclusione di una transazione
  - commit [work]: le operazioni specificate a partire dall'inizio della transazione vengono eseguite sulla base di 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

Una transazione in SQL

```
start transaction          (opzionale)
update ContoCorrente
    set Saldo = Saldo – 10
    where NumeroConto = 12345 ;
update ContoCorrente
    set Saldo = Saldo + 10
    where NumeroConto = 55555 ;
commit work;
```