

## Cap. 7 -Trigger e loro uso

## SOMMARIO

- Introduzione ..... 3
- Definizione standard di trigger ..... 10
- Uso dei trigger e integrità referenziale ..... 18

## Comportamento attivo delle BD

- Si realizza disponendo di un componente del DBMS per la gestione di regole che scattano su determinati **E**venti transazionali, in maniera **C**ondizionata, sviluppando una prescritta **A**zione – regole **E-C-A** dette anche “**trigger**”.
- I DBMS, in tal caso, hanno comportamento “**reattivo**” (in contrasto con “**passivo**”): eseguono non solo le transazioni utente ma anche le regole.
- le regole “mettono a fattor comune” parte dell’applicazione che diventa così “condivisa” con tutte le applicazioni: si parla di “**indipendenza della conoscenza**” in quanto la conoscenza “reattiva” viene sottratta alle applicazioni e codificata nei trigger.
- tutti i DBMS commerciali hanno i **trigger** (standardizzati da SQL:2003).

## 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 che identifica le situazioni in cui è necessaria l’applicazione del trigger
  - Quando si valuta la condizione il trigger viene **considerato**
- **Azione**
  - Un generico comando di modifica o una stored procedure
  - Quando si elabora l’azione il trigger viene **eseguito**

## Trigger

- definiti con istruzioni DDL (`create trigger`)
  - basati sul paradigma ECA:
    - evento**: modifica dei dati, specificata con `insert`, `delete`, `update`
    - condizione** (opzionale): predicato SQL
    - azione**: sequenza di istruzioni SQL (o estensioni, ad esempio PL/SQL in Oracle)
- ogni trigger fa riferimento ad una tabella (**target**) e risponde ad eventi relativi a tale tabella
- i valori di tale tabella vengono "congelati" prima e dopo l'evento nelle due tabelle di transizione **old table** e **new table**.

## Trigger: modo e granularità

- Modo di esecuzione**
  - after**: il trigger è considerato ed eseguito dopo l'applicazione sulla base dati dell'azione che lo ha attivato
  - before**: il trigger è considerato ed eseguito prima della applicazione alla base dati dell'azione che lo ha attivato
- Granularità**
  - di tupla (**row - level**): attivazione **per ogni tupla** – identificata con `old [row]` e `new [row]` - **della tabella target coinvolta nell'operazione**
  - di operazione (**statement - level**): una sola attivazione – identificata con `old [table]` e `new [table]` **per ogni istruzione SQL**, con riferimento a tutte le ennuple della tabella target coinvolte ("set-oriented")

## Trigger, esempio: dichiarazione

```
create trigger Reorder
after update of QtyAvbl on WAREHOUSE
for each row
when (new.QtyAvbl < new.QtyLimit)
declare X number;
begin
  select count(*) into X
  from PENDINGORDERS
  where Part = new.Part;
  if X = 0
  then
    insert into PENDINGORDERS
    values(new.Part,new.QtyReord,sysdate)
  end if;
end;
```

## Trigger, esempio: tabella target e transazioni attivanti

WAREHOUSE	Part	QtyAvbl	QtyLimit	QtyReord
	1	200	150	100
	2	780	500	200
	3	450	400	120

Viene eseguita la transazione

```
T1: update WAREHOUSE
      set QtyAvbl = QtyAvbl - 70
      where Part = 1
```

E successivamente

```
T2: update WAREHOUSE
      set QtyAvbl = QtyAvbl - 60
      where Part <= 3
```

## Trigger, esempio: esecuzione

- La esecuzione di T1 causa un'esecuzione del trigger "Reorder" che porta nella tabella **PENDINGORDERS** la tupla (1,100,<oggi>);
- La successiva esecuzione di T2 causa l'esecuzione del trigger "Reorder" per tre volte con soddisfacimento della condizione di scatto per le parti 1 e 3.
  - L'azione relativa alla parte 1 non ha effetto perchè
 

```
select count(*) into X
from PENDINGORDERS
where Part = new.Part;
```

 valuta X=1 poiché è già presente la parte considerata;
  - L'azione relativa alla parte 3 porta nella tabella **PENDINGORDERS** la nuova tupla (3,120,<oggi>)



## Sintassi SQL:2003 dei trigger

- Lo standard propone una sintassi simile a quella offerta da IBM DB2; ogni trigger è caratterizzato da:
  - Nome del trigger
  - nome della tabella che viene monitorata
  - modo di esecuzione (BEFORE o AFTER)
  - l'evento monitorato (INSERT, DELETE o UPDATE)
  - granularità (statement-level o row-level)
  - nomi e alias per "transition tables" e "transition rows"
  - eventuale condizione
  - l'azione
  - il timestamp di creazione

## Sintassi SQL:2003 dei trigger

```
create trigger NomeTrigger
{before | after}
{ insert | delete | update [of Colonne] } on
Tabella
[referencing
  {[old table [as] AliasTabellaOld]
  [new table [as] AliasTabellaNew] } |
  {[old [row] [as] NomeTuplaOld]
  [new [row] [as] NomeTuplaNew] }]
[for each { row | statement } ]
[when Condizione]
ComandiSQL
```

## Esecuzione di un singolo trigger

- Modo di esecuzione:
  - BEFORE
    - Il trigger viene considerato ed eventualmente eseguito prima che venga applicata sulla base di dati l'azione che lo ha attivato
    - Di norma viene utilizzata questa modalità quando si vuole verificare la correttezza di una modifica, prima che la stessa venga applicata
  - AFTER
    - Il trigger viene considerato ed eventualmente eseguito dopo che è stata applicata sulla base di dati l'azione che lo ha attivato
    - È il modo più comune, adatto a quasi tutte le applicazioni
    - È più semplice da utilizzare correttamente

## Granularità degli eventi

- Modo statement level (modo di default)
  - Il trigger viene considerato ed eventualmente eseguito una volta sola per ogni comando che lo ha attivato, indipendentemente dal numero di tuple modificate
  - È il modo più vicino all'approccio tradizionale dei comandi SQL, che sono di norma set-oriented
- Modo row-level (opzione `for each row`)
  - Il trigger viene considerato ed eventualmente eseguito una volta per ciascuna tupla che è stata modificata dal comando
  - Consente di scrivere i trigger in modo più semplice
  - Può essere meno efficiente

## Clausola referencing

- Il formato dipende dalla granularità
  - Per il modo row-level, si hanno **due variabili di transizione** (*transition variables*) `old` e `new`, che rappresentano rispettivamente il valore precedente e successivo alla modifica della tupla che si sta valutando
  - Per il modo statement-level, si hanno **due tabelle di transizione** (*transition tables*) `old table` e `new table`, che contengono rispettivamente il valore vecchio e nuovo di tutte le tuple modificate
- Le variabili `old` e `old table` **non** sono utilizzabili in trigger il cui evento è `insert`
- Le variabili `new` e `new table` **non** sono utilizzabili in trigger il cui evento è `delete`
- Le variabili e le tabelle di transizione sono importanti per realizzare i trigger in modo efficiente

## Esempio di trigger row-level

```
create trigger MonitoraConti
after update on CONTI
for each row
when (old.NomeConto =
new.NomeConto and
new.Totale > old.Totale)
insert values
(new.NomeConto,new.Totale-
old.Totale)
into SINGOLIVERSAMENTI
```

## Esempio di trigger statement-level

```
create trigger
ArchiviaFattureCanc
after delete on FATTURE
referencing old table as
SETOLDFATTURE
insert into FATTURECANCELLATE
(select *
from SETOLDFATTURE)
```

## Esecuzione: 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**
  - Vengono eseguiti i trigger **BEFORE row-level**
  - Si applica la modifica e si verificano i vincoli di integrità definiti sulla base di dati
  - Vengono eseguiti i trigger **AFTER row-level**
  - Vengono eseguiti i trigger **AFTER statement-level**
- Se vi sono più **trigger della stessa categoria**:
  - l'ordine di esecuzione viene scelto dal sistema in un modo che dipende dall'implementazione.



## Impiego dei trigger

- Per la descrizione di regole **esterne** (o regole aziendali)
  - Esprimono conoscenza di tipo applicativo
- Per la descrizione di regole **interne** alla base di dati
  - Trigger generati dal sistema e non visibili all'utente; funzionalità trattata: **gestione della integrità referenziale**.

## Integrità referenziale: esempio

- Si consideri il seguente schema di base dati:
 

IMPIEGATI ( Matricola, Nome, Salario,  
NDip: DIPARTIMENTI)

DIPARTIMENTI(NroDip, Manager: IMPIEGATI)

e le strategie di riparazione per le violazioni dei vincoli di integrità referenziale tra NDip di IMPIEGATI e NroDip di DIPARTIMENTI

## Integrità referenziale: possibili violazioni

- Es: 

```
CREATE TABLE IMPIEGATI
(
  ... ..
  FOREIGN KEY(NDip) REFERENCES DIPARTIMENTI(NroDip)
  ON DELETE SET NULL,
  ... ..
);
```
- Le operazioni che possono violare questo vincolo sono:
  - INSERT in IMPIEGATI
  - UPDATE di IMPIEGATI.NDip
  - UPDATE di DIPARTIMENTI.NroDip
  - DELETE in DIPARTIMENTI
- Il vincolo è espresso come predicato nella parte condizione

## INSERT (UPDATE) nella tabella IMPIEGATI

- Evento:** inserimento (modifica) in IMPIEGATI
- Condizione:** il nuovo valore di N<sub>dip</sub> non è tra quelli contenuti nella tabella DIPARTIMENTI
- Azione:** si inibisce l'inserimento (modifica), segnalando un errore

```
CREATE TRIGGER ControllaDipImpiegato
BEFORE INSERT ON IMPIEGATI
FOR EACH ROW
WHEN (not exists select * from DIPARTIMENTI
      where NroDip = NEW.NDip)
BEGIN
    raise_application_error(-20000, 'Dipartimento
non valido');
END;
(Per la update di Ndip in IMPIEGATI il trigger cambia solo nella
parte evento: BEFORE UPDATE OF Ndip ON IMPIEGATI)
```

## UPDATE nella tabella DIPARTIMENTI

- Evento:** modifica dell'attributo N<sub>roDip</sub> in DIPARTIMENTI
- Condizione:** il vecchio valore di N<sub>roDip</sub> è tra quelli contenuti nella tabella IMPIEGATI
- Azione:** si modifica anche N<sub>dip</sub> in IMPIEGATI

```
CREATE TRIGGER ControllaModificaDipartimento
AFTER UPDATE OF NroDip ON DIPARTIMENTI
FOR EACH ROW
WHEN(exists select * from IMPIEGATI
      where NDip = OLD.NroDip)
BEGIN
    update IMPIEGATI set NDip = NEW.Nrodip
    where NDip = OLD.NroDip;
END;
```

## DELETE nella tabella DIPARTIMENTI

- Evento:** cancellazione in DIPARTIMENTI
- Condizione:** il valore di N<sub>roDip</sub> che si intende cancellare è tra quelli contenuti nella tabella IMPIEGATI
- Azione:** si pongono a Null i valori N<sub>dip</sub> di IMPIEGATI pari al valore N<sub>roDip</sub> di DIPARTIMENTI che si intende cancellare

```
CREATE TRIGGER ControllaCancDipartimento
AFTER DELETE ON DIPARTIMENTI
FOR EACH ROW
WHEN(exists select * from IMPIEGATI
      where NDip = OLD.NroDip)
BEGIN
    UPDATE IMPIEGATI
    SET Ndip=NULL
    WHERE Ndip = OLD.NroDip;
END;
```

