

# Basi di Dati Attive



Per approfondimenti:

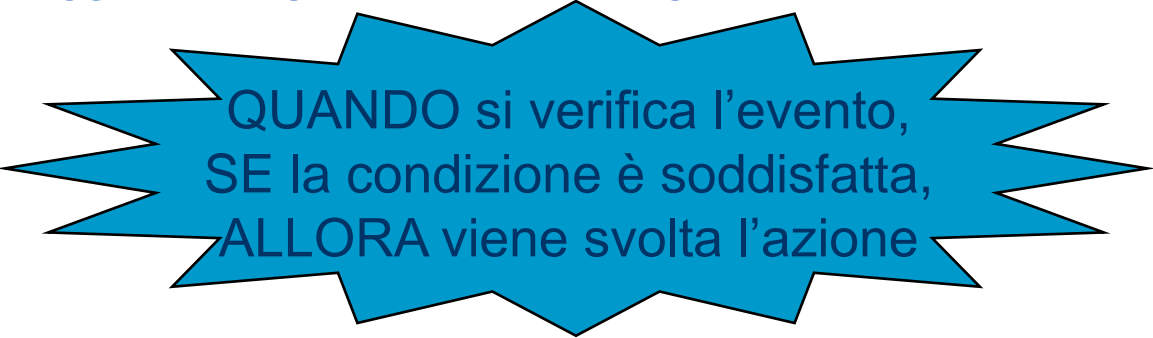
- Atzeni, Ceri, Paraboschi e Torlone. Basi di dati – II edizione: pp 463-476
- Manuale ORACLE 11g – Concepts 8.2
- Manuale ORACLE 11g – PL/SQL Reference 9

# ***Una base di dati si dice attiva quando dispone di un sottosistema integrato per definire e gestire regole di produzione.***

- ❑ Le regole (o *trigger*) seguono il *paradigma evento-condizione-azione (ECA)*:
  - reagiscono a *eventi*;
  - valutano una *condizione*;
  - eventualmente eseguono una *reazione*.
- ❑ Comportamento **reattivo** invece che **passivo**: alternarsi tra esecuzione di transazioni (lanciate dagli utenti) e regole (lanciate dal sistema).
- ❑ *Indipendenza della conoscenza*: la conoscenza relativa al comportamento reattivo viene sottratta al programma applicativo e codificata, sotto forma di regole, nello schema del database.
- ❑ Gestione di vincoli di integrità, calcolo dati derivati, gestione eccezioni, ...

# I trigger nei sistemi relazionali

- ❑ La definizione dei trigger fa parte del DDL
- ❑ I trigger fanno riferimento a una tabella (*target*)
  - Gli eventi sono primitive SQL di manipolazione (*insert*, *delete*, *update*)
  - La condizione è un predicato booleano
  - L'azione è una sequenza di primitive SQL, talvolta arricchite da un linguaggio di programmazione integrato nel DBMS (es. PL/SQL)



QUANDO si verifica l'evento,  
SE la condizione è soddisfatta,  
ALLORA viene svolta l'azione

- ❑ Altre caratteristiche che possono essere definite
  - granularità di tupla (*row-level*) o di primitiva (*statement-level*)
  - modalità *immediata* (after o before) o *differita* (dopo l'esecuzione del COMMIT)
  - trigger *in cascata*

# Sintassi SQL-99

```
CREATE TRIGGER <nome>
{BEFORE|AFTER|INSTEAD OF} <evento>
[REFERENCING <variabili>]
[FOR EACH {ROW | STATEMENT}]
[WHEN (<condizione>)]
<sequenza di comandi SQL>
```

```
<evento>: INSERT ON <tabella> |
DELETE ON <tabella> |
UPDATE [OF <lista_colonne>] ON <tabella>
```

**<variabili>**:

```
OLD AS <nome tuple vecchie>
NEW AS <nome tuple nuove>
OLD TABLE AS <nome tabella vecchia>
NEW_TABLE AS <nome tabella nuova>
```

# I trigger in ORACLE

- ❑ Possono eseguire azioni che contengono codice PL/SQL
- ❑ Supportano granularità row- e statement-level
- ❑ Supportano solo la modalità immediata (before, after e instead of)
- ❑ Supportano anche eventi basati su:
  - Comandi DDL (es. create, alter, drop)
  - Eventi (es. logon, logoff, shutdown, startup, servererror)

# Sintassi ORACLE (di base)

```
CREATE TRIGGER <nome>
{BEFORE|AFTER} <eventi>
ON <tabella>
<blocco PL/SQL>
```



statement-level

```
CREATE TRIGGER <nome>
{BEFORE|AFTER} <eventi>
ON <tabella>
[REFERENCING <referimenti>]
```



row-level

**FOR EACH ROW**

```
[WHEN (<condizione>)] <blocco PL/SQL>
```

```
<eventi> ::= INSERT|DELETE|UPDATE [OF <colonne>]
<referimenti> ::= OLD AS <nome vecchio valore>|
                  NEW AS <nome nuovo valore>
```

# Esempio ORACLE

```
create or replace TRIGGER riordine
AFTER UPDATE OF P_QTADISP ON LA_Prodotti
FOR EACH ROW
WHEN (NEW.P_QTADISP<NEW.P_livelloMinimo)

DECLARE
    x NUMBER;
BEGIN
    SELECT COUNT(*) INTO x
    FROM LA_ordini
    WHERE O_CODP=:NEW.P_COD;
    IF x=0 THEN
        INSERT INTO LA_ordini
            VALUES (:NEW.P_COD, :NEW.P_QTARIORDINO, SYSDATE);
    END IF;
END;
```

# (Alcuni) vincoli sui trigger

## ❑ Tutti i trigger:

- Non possono eseguire i comandi COMMIT e ROLLBACK
- Perché un trigger è parte di una operazione più ampia

## ❑ I *before* trigger:

- Possono modificare i valori assegnati alle variabili `new`, ma non possono contenere comandi SQL che provochino una modifica allo stato della base di dati
- Non possono essere specificati su una vista

## ➤ I trigger di tipo *after*

- Non possono essere specificati su una vista

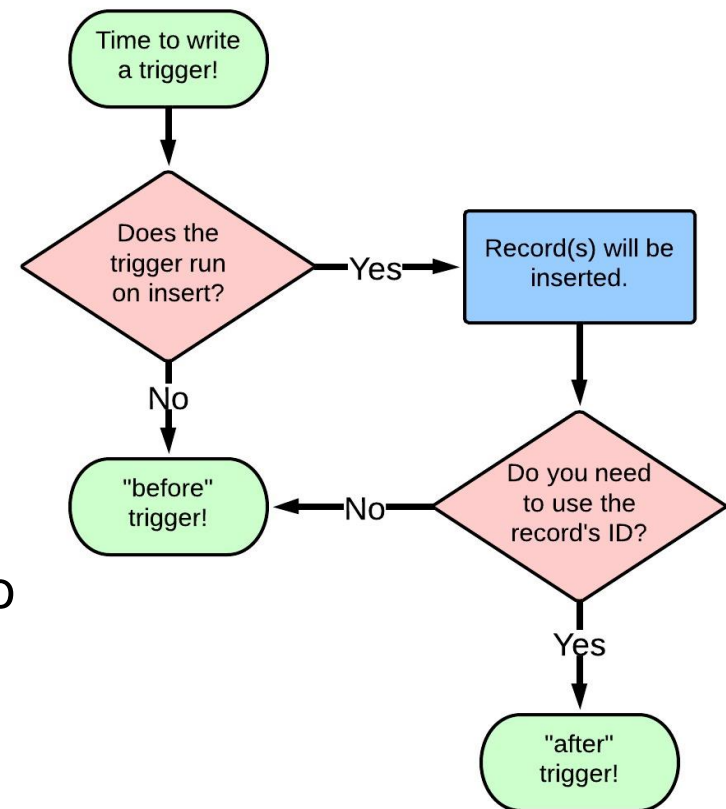
## ❑ I trigger di tipo *instead of*

- Possono essere specificati **solo** su una vista



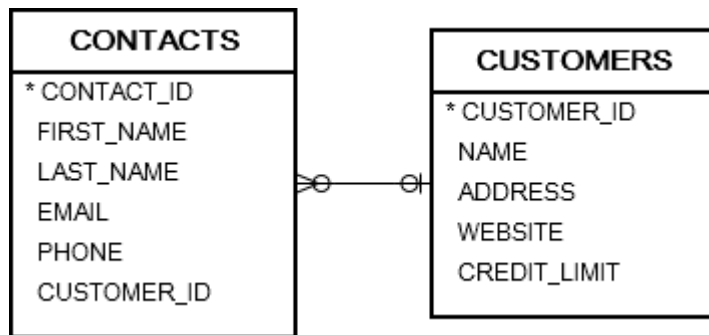
# After o Before?

- ❑ il 95% dei trigger è di tipo **before**
- ❑ Se è necessario apportare modifiche a un record in un **after** trigger è necessario eseguire un'istruzione DML.
- ❑ In un **before** trigger si può evitare, modificando i valori delle variabili :New
- ❑ Di solito, si usa un **after** trigger se è necessario accedere a un ID nel caso di un inserimento.
- ❑ Si usa un **after** trigger quando l'operazione non deve essere modificata, ma deve determinare azioni su altri dati (es. logging)



# Trigger INSTEAD OF

- ❑ In Oracle, è possibile creare un trigger INSTEAD OF solo per una vista (Non per una tabella).
- ❑ Un trigger INSTEAD OF può servire per gestire l'aggiornamento di una vista **che non può essere modificata direttamente tramite istruzioni DML (SQL Error: ORA-01779).**
- ❑ Esempio:



```
CREATE VIEW vw_customers AS
SELECT name, address, website,
       credit_limit, first_name,
       last_name, email, phone
FROM customers c INNER JOIN
contacts co
ON (c.customer_id=co.customer_id);
```

# Trigger INSTEAD OF

```
CREATE OR REPLACE TRIGGER new_customer_trg
  INSTEAD OF INSERT ON vw_customers
  FOR EACH ROW
DECLARE
  l_customer_id NUMBER;
BEGIN
  -- insert a new customer first
  INSERT INTO customers(name, address, website,
credit_limit)
  VALUES(:NEW.NAME, :NEW.address, :NEW.website,
:NEW.credit_limit)
  RETURNING customer_id INTO l_customer_id;

  -- insert the contact
  INSERT INTO contacts(first_name, last_name, email,
phone, customer_id)
  VALUES(:NEW.first_name, :NEW.last_name, :NEW.email,
:NEW.phone, l_customer_id);
END;

insert into vw_products values (78, 'Prova', 6, 29, 'Beverages');
```

# Progettazione dei trigger

- ❑ La progettazione di un singolo trigger è solitamente semplice. Bisogna identificare:
  - semantica di esecuzione
  - evento
  - condizione (opzionale)
  - azione
- ❑ La comprensione delle interazioni reciproche tra i trigger è più complessa:
  - l'azione di un trigger può essere l'evento di un trigger differente (esecuzione a cascata)
  - Se si verificano inneschi reciproci è possibile un'esecuzione infinita

# Concorrenza tra trigger

- Un comando SQL sul DB può scatenare più di un trigger. In questo caso la priorità dei trigger è definita da:

1. La sua natura secondo l'algoritmo:

1. esegui i **before-trigger stat-level**;
2. per ogni tupla nella tabella target:
  - 2.1 esegui i **before-trigger row-level**;
  - 2.2 esegui la modifica della tupla  
e i controlli di integrità referenziale (tupla);
  - 2.3 esegui gli **after-trigger row-level**;
3. esegui i controlli di integrità referenziale globali;
4. esegui gli **after-trigger stat-level**.

2. La data di creazione

3. Un **ordinamento** specifico di attivazione: se su un evento di attivazione (ad esempio INSERT) sono definiti più trigger è possibile specificare la sequenza di innesco usando l'opzione **FOLLOWS** o **PRECEDES**.

# Concorrenza tra trigger

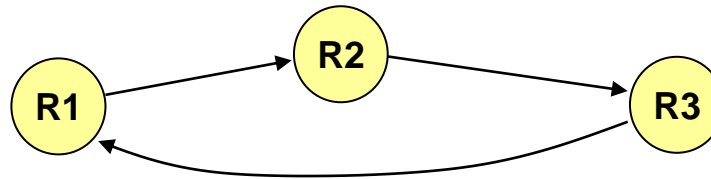
- ❑ Se le azioni svolte dai trigger causano l'attivazione di altri trigger, l'esecuzione del trigger corrente è sospesa e vengono considerati gli altri trigger attivati.
- ❑ I DBMS operano in un contesto di isolamento definito a livello transazionale. L'esecuzione delle regole ha le stesse proprietà di isolamento della transazione che le scatena.
- ❑ L'atomicità della transazione fa sì che in caso di fallimento vengono annullati tutti gli effetti prodotti, compresi quelli dei trigger.

# Proprietà delle regole attive

- ❑ Progettare le singole regole è semplice, più complesso è prevedere il comportamento collettivo di insiemi complessi di regole.
  - Un insieme di regole garantisce la **terminazione** quando, per ogni transazione che ne scatena l'esecuzione, l'esecuzione termina producendo uno stato finale (incluso abort)
  - Un insieme di regole garantisce la **confluenza** quando, per ogni transazione che ne scatena l'esecuzione, l'esecuzione termina producendo un unico stato finale, indipendentemente dall'ordine di esecuzione delle regole non esplicitamente priorizzate
  - Un insieme di regole garantisce il **determinismo delle osservazioni** quando, per ogni transazione che ne scatena l'esecuzione, l'esecuzione è confluyente e tutte le azioni visibili svolte sono identiche e prodotte nello stesso ordine.
- ❑ Mentre la **terminazione** è una caratteristica essenziale, **confluenza** e **determinismo** possono essere trascurate specie in presenza di varie soluzioni equivalenti di uno stesso problema applicativo.

# Analisi delle regole

- Il **grafo di attivazione** viene usato per valutare la proprietà di terminazione:
  - A ogni regola corrisponde un nodo.
  - Un arco da R1 a R2 indica che R1 contiene una primitiva che fa scattare R2.



- In presenza di cicli, un'esecuzione può non terminare.
- Solo alcuni cicli corrispondono effettivamente a situazioni di non terminazione.

```
create or replace TRIGGER controllaSalari AFTER UPDATE OF
salario ON impiegati
DECLARE
AvgSal number;
BEGIN
SELECT AVG(salario) into AvgSal FROM impiegati;
if AvgSal>100 then
    UPDATE impiegati SET salario = salario*0.9;
end if;
end;
```



# Analisi delle regole

(2)

## □ Tecniche semantiche:

- Per ogni arco  $(R_i, R_j)$  nel grafo, se la condizione di  $R_j$  è sicuramente falsa dopo l'esecuzione di  $R_i$ , allora si può eliminare  $(R_i, R_j)$  dal grafo.
- In realtà, è sufficiente garantire che i nuovi dati prodotti da  $R_i$  non soddisfino la condizione di  $R_j$ . In questo caso,  $R_i$  non contribuisce alla ripetuta esecuzione di  $R_j$ , e si può eliminare  $(R_i, R_j)$  dal grafo.

# Applicazioni

## ❑ Regole interne:

- integrità
- dati derivati
- dati replicati
- versioni
- sicurezza e privacy
- logging

## ❑ Regole esterne:

- business rules
- allertatori

# Gestione dell'integrità

```
dipartimenti(codDip, nomeDip, Direttore:impiegati,...)  
impiegati(codImp, nomeImp, salario,...DipNum:dipartimenti)
```

```
FOREIGN KEY(DipNum) REFERENCES dipartimenti(codDip)  
ON DELETE SET NULL  
ON UPDATE CASCADE
```

- ❑ Le operazioni che possono violare il vincolo di integrità
- ❑ referenziale sono:

```
INSERT INTO impiegati  
DELETE FROM dipartimenti  
UPDATE TO impiegati.codDip  
UPDATE TO dipartimenti.codDip
```

# Gestione dell'integrità

## ❑ Semantica di esecuzione

- **dopo** che un'operazione di modifica o cancellazione su dipartimenti
  - Aggiornare la tabella impiegati
- **Livello di tupla**
  - esecuzione separata per ogni tupla della tabella dipartimenti
    - più semplice da implementare

## ❑ Evento

- **Update/delete on Dipartimenti**

## ❑ Condizione

- **Nessuna:** è sempre eseguito

## ❑ Azione

- Aggiorna nella tabella Impiegati le righe che contengono un riferimento al codDip modificato/cancellato

# Gestione dell'integrità

```
create or replace TRIGGER dipRef3
AFTER DELETE ON dipartimenti
FOR EACH ROW
BEGIN
UPDATE impiegati
SET DIPNUM = NULL
WHERE DIPNUM = :OLD.codDip;
END;
```

```
create or replace TRIGGER dipRef4
AFTER UPDATE OF codDip ON dipartimenti
FOR EACH ROW
BEGIN
UPDATE impiegati
SET DIPNUM = :NEW.codDip
WHERE DIPNUM = :OLD.codDip;
END;
```



# Gestione dei dati derivati

```
create table dipRicchi AS  
( SELECT DISTINCT DIPNUM FROM impiegati i  
WHERE i.salario > 100 );
```

## ❑ Semantica di esecuzione (2 possibili approcci)

- dopo che un'operazione di inserimento/modifica/cancellazione su impiegati
  - Aggiornare la tabella dipRicchi
- Livello di statement (Approccio refresh):
  - ricalcolo della vista ex-novo dalle tabelle sorgente dopo ogni update
- Livello di tupla (Approccio incrementale)
  - calcolo delle variazioni (tuple da inserire o eliminare dalla vista)

## ❑ Evento

- Insert/Update/delete on Impiegati

## ❑ Condizione

- solo approccio incrementale
- NEW.salario > 100

## ❑ Azione

- Aggiorna nella tabella dipRicchi

# Gestione dei dati derivati

```
create or replace TRIGGER refresh
AFTER DELETE OR INSERT OR UPDATE OF salario ON impiegati
BEGIN
DELETE FROM dipRicchi;
INSERT INTO dipRicchi
(SELECT DISTINCT DIPNUM FROM impiegati i
WHERE i.salario > 100);
end;
```

```
create or replace TRIGGER incremental
AFTER INSERT ON impiegati
FOR EACH ROW
WHEN (NEW.salario > 100)
BEGIN
INSERT INTO dipRicchi (DIPNUM) VALUES (:NEW.DIPNUM);
end;
```

# Regole aziendali e alertatori

- ❑ Gli **alertatori** si limitano nella parte azione ad emettere messaggi e avvisi in risposta ad eventi straordinari:
  - fluttuazione del valore dei titoli di borsa
  - diminuzione della quantità disponibile in magazzino
  
- ❑ Le **regole aziendali** esprimono le strategie di un'azienda nel perseguire i propri scopi
  - Riordinare automaticamente i prodotti sotto una certa soglia
  - Evitare che un dipendente abbia uno stipendio maggiore del suo responsabile
  
- ❑ Le regole attive sono particolarmente utili poiché esprimono le politiche reattive a livello di schema e quindi centralizzate rispetto alle diverse applicazioni



# Regole aziendali e allertatori

## ❑ Problema:

- Evitare che un dipendente abbia uno stipendio maggiore del suo responsabile: emettere un avviso in caso di salario troppo elevato

## ❑ Semantica di esecuzione

- **dopo** che un'operazione di modifica del salario sulla tabella impiegati
  - Se il salario è maggiore di quello del direttore emettere avviso
- Livello di tupla

## ❑ Evento

- Update on Impiegati

## ❑ Condizione

- nessuna

## ❑ Azione

- Se il salario è maggiore di quello del direttore emettere avviso

# Regole aziendali e allertatori

```
create or replace TRIGGER SalarioEccessivo
AFTER UPDATE OF salario ON impiegati
FOR EACH row
declare
    sal_max integer :=0;
begin
    select max(salario) into sal_max
    from impiegati
    where NomeImp =
        (select Direttore
         from Dipartimenti
         where CODDIP = :New.DipNum);
    if :New.salario > sal_max then
        raise_application_error(-20001, 'Salario troppo elevato ');
    end if;
end;
```



# **Trigger (laboratorio)**

Prof. Alessandra Lumini

[Alessandra.lumini@unibo.it](mailto:Alessandra.lumini@unibo.it)

# Sintassi ORACLE

```
CREATE TRIGGER <nome>
{BEFORE|AFTER} <eventi>
ON <tabella>
<blocco PL/SQL>
```



statement-level

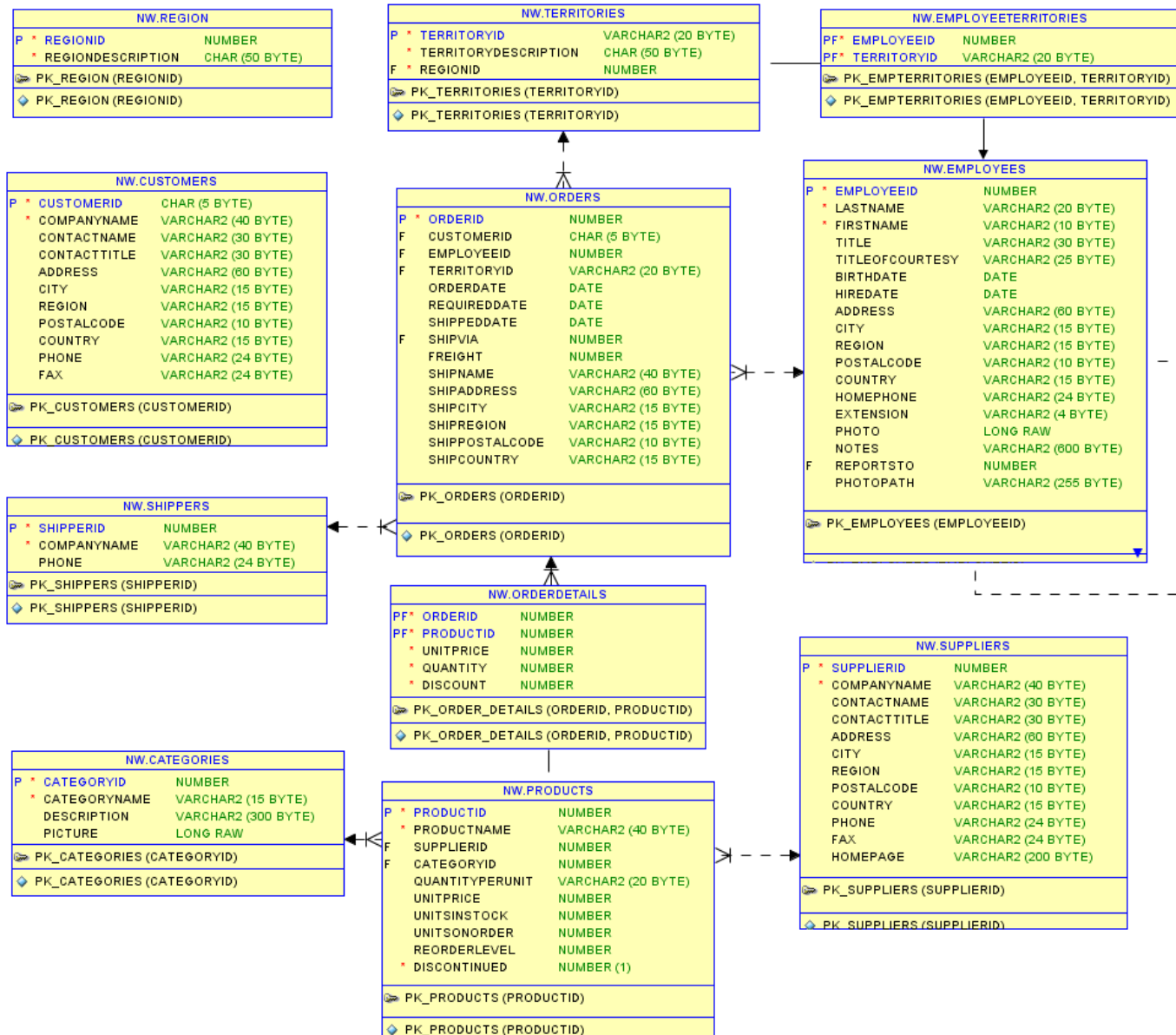
```
CREATE TRIGGER <nome>
{BEFORE|AFTER} <eventi>
ON <tabella>
[REFERENCING <referimenti>]
FOR EACH ROW
[WHEN (<condizione>)] <blocco PL/SQL>
```



row-level

```
<evento> ::= INSERT|DELETE|UPDATE [OF <colonne>]
<referimento> ::= OLD AS <nome vecchio valore>|
                  NEW AS <nome nuovo valore>
```

# Schema DB



# Preparazione

- ❑ Copiare nel proprio schema le seguenti tabelle dell'utente NW:
- ❑ NW.Products, NW\_Orders. NW\_OrderDetails
- ❑ `create table NW_Products as select * from NW.Products;`
- ❑ `create table NW_Orders as select * from NW.Orders;`
- ❑ `create table NW_OrderDetails as select * from NW.OrderDetails;`

# Esempio 1

- A fronte dell'inserimento di un nuovo ordine, se il campo data è NULL assegna la data di oggi

```
create or replace TRIGGER trg_before_ord_insr_data
BEFORE INSERT
ON nw_orders
FOR EACH ROW
BEGIN
    if :NEW.ORDERDATE is NULL then
        :NEW.ORDERDATE := sysdate;
    end if;
END;
```

# Testing

- ❑ **ALTER TRIGGER** trg\_before\_ord\_insr\_data **ENABLE**;
- ❑ **INSERT INTO** nw\_orders **VALUES**  
('20000','BONAP','9','20852',null,null,null,'2','3  
8,28','Bon app''','12, rue des  
Bouchers','Marseille',null,'13008','France');
- ❑ **SELECT \* from** nw\_orders **WHERE** orderid=20000



# Esempio 2

- Data tracking: log dei cambiamenti di prezzo dei prodotti

```
create or replace TRIGGER trg_price
AFTER UPDATE OF unitprice
ON nw_products
FOR EACH ROW

BEGIN
IF (:NEW.unitprice != :OLD.unitprice)
THEN
    INSERT INTO nw_data_tracking
    values (nw_track_id.nextval,
:OLD.productid, :OLD.unitprice,
:NEW.unitprice, sysdate, user);
    END IF;
END;
```

```
CREATE SEQUENCE nw_track_id
START WITH 1
INCREMENT BY 1;

CREATE TABLE nw_data_tracking (
IDtracking INT PRIMARY KEY ,
IDprod INT NOT NULL ,
old_value INT NOT NULL ,
new_value INT NOT NULL ,
dateModified DATE NOT NULL,
userModified varchar2(20)
);
```

# Testing

- ❑ **ALTER TRIGGER** trg\_price **ENABLE**;
- ❑ **UPDATE** nw\_products  
  **SET** unitprice=unitprice+1  
  **WHERE** productid=1;
- ❑ **SELECT** \* **FROM** nw\_data\_tracking;

# Esempio 3

- ❑ Calcola alcune statistiche sugli ordini a fronte di variazioni nella tabella ordini
- ❑ Data la tabella riassuntiva order\_stats :

```
CREATE TABLE order_stats as
SELECT
    EXTRACT (year FROM orderdate) as year,
    count (Distinct customerid) as numCust,
    count (*) as numOrd
FROM
    nw_orders
GROUP BY EXTRACT (year FROM orderdate)
ORDER BY 1
```

# Es. 3 – trigger statement level

- Aggiorna la tabella a fronte di variazioni nella tabella ordini

```
CREATE OR REPLACE TRIGGER trg_stat_orders
AFTER INSERT OR DELETE OR UPDATE ON nw_orders
DECLARE
CURSOR c_statistics IS
SELECT EXTRACT(year FROM orderdate) as year, count (Distinct customerid)
as numCust,
        count (*) as numOrd FROM nw_orders
GROUP BY EXTRACT (year FROM orderdate);

BEGIN
    FOR v_statsRecord IN c_statistics LOOP
        UPDATE order_stats
            SET numCust = v_statsRecord.numCust, numOrd =
v_statsRecord.numOrd
            WHERE year = v_statsRecord.year;
        IF SQL%NOTFOUND THEN
            INSERT INTO order_stats(year, numCust, numOrd)
                VALUES (v_statsRecord.year, v_statsRecord.numCust,
v_statsRecord.numOrd);
        END IF;
    END LOOP;
END trg_stat_orders;
```

# Testing

- ❑ **ALTER TRIGGER** trg\_stat\_orders **ENABLE**;
- ❑ **INSERT INTO** nw\_orders **VALUES**  
(**'20001', 'BONAP', '9', '20852', null, null, null, '2', '3**  
**8,28', 'Bon app''', '12, rue des**  
**Bouchers', 'Marseille', null, '13008', 'France'**);
- ❑ **SELECT \* from** order\_stats;

# Stampe da trigger

- ❑ Non è una buona idea inserire istruzioni di stampa all'interno di un Trigger, ma se serve a fini di debug occorre prima abilitare l'output (da console)
  - `set serveroutput on;`
- ❑ Poi usare l'istruzione di stampa
  - `DBMS_OUTPUT.put_line(Message');`
- ❑ Gestione di messaggio di errore tramite:
  - `RAISE_APPLICATION_ERROR(-20001, 'Messaggio di errore');`

# Esercizi

## ❑ EX1: Gestione di dati derivati

- Definire una nuova tabella NW\_EmployeeRank (EmployeeID, NumOrders) che mantiene il conteggio del numero di ordini gestiti da ciascun dipendente
- Popolare la tabella con una query che conta il numero di ordini in base ai dati attualmente presenti
- Scrivere un trigger che a fronte dell'inserimento di un nuovo ordine in NW\_Orders aggiorna la tabella

## ❑ EX2: Regole aziendali

- Scrivere un trigger che verifica la disponibilità dei prodotti: deve valutare le quantità richieste di ogni prodotto all'interno di un ordine e verificare la quantità disponibile nella tabella NW\_Products: se la quantità ( $\text{UnitsInStock} + \text{UnitsOnOrder}$ ) non è sufficiente a evadere l'ordine il riordino deve essere classificato come urgente, se la quantità è sufficiente, ma l'evasione dell'ordine porta il prodotto sotto il livello di riordino (ReorderLevel) si richiede in riordino standard.

## ❑ EX3: Gestione dell'integrità

- Scrivere un trigger che a fronte della cancellazione di un ordine in NW\_Orders elimina tutti i dettagli ad esso relativi in NW\_OrderDetails (e aggiorna NW\_EmployeeRank )

# Testing

- ❑ **SELECT \* FROM NW\_EMPLOYEEERANK ;**
- ❑ **Insert into NW\_ORDERS values ('11078','BONAP','9','20852',to\_date('07-MAG-98','DD-MON-RR'),to\_date('04-GIU-98','DD-MON-RR'),null,'2','38,28','Bon app"',12,rue des Bouchers','Marseille',null,'13008','France');**
- ❑ **Insert into NW\_ORDERS values ('11079','RATTC','1','19713',to\_date('07-MAG-98','DD-MON-RR'),to\_date('04-GIU-98','DD-MON-RR'),null,'2','8,53','Rattlesnake Canyon Grocery','2817 Milton Dr.','Albuquerque','NM','87110','USA');**
- ❑ **SELECT \* FROM NW\_EMPLOYEEERANK ;**
- ❑ **Insert into NW\_orderdetails values ('11078','8','40','2','0,1');**
- ❑ **Insert into NW\_orderdetails values ('11078','10','31','1','0');**
- ❑ **Insert into NW\_orderdetails values ('11078','11','21','30','0,05');**
- ❑ **Insert into NW\_orderdetails values ('11078','20','81','100','0,04');**
- ❑ **delete from NW\_orders where orderid=11078;**
- ❑ **delete from NW\_orders where orderid=11079;**
- ❑ **SELECT \* FROM NW\_EMPLOYEEERANK ;**