Documento a cura di Simone Remoli.

EVENTI TEMPORIZZATI

Permettono di attivare del codice SQL periodicamente.

Un esempio:

```
set global event_scheduler = on;

create event if not exists `cleanup`
    on schedule
      every 2 day
       on completion preserve
    comment 'Remove old tuples'
    do
      delete from `entity` where `created_at` < (NOW() - interval 2 day)
```

Crea un evento chiamato **cleanup** che:

- Viene eseguito **ogni 2 giorni**.
- **Elimina le tuple** dalla tabella entity che sono più vecchie di 2 giorni rispetto alla data attuale (NOW()).
- Utilizza **ON COMPLETION PRESERVE**, il che significa che l'evento non viene eliminato dopo l'esecuzione, ma rimane nel database, altrimenti lo eseguo dopo due giorni e l'evento temporizzato poi sparisce.

  Dopo il "**do**" si inseriscono i comandi che devono essere eseguiti dal DBMS quando questo evento prende il controllo.

All'inizio **abilita l'event scheduler** nel database MySQL. Se non è attivo, gli eventi definiti non verranno eseguiti.

Pianificazione con TIMESTAMP.

Puoi specificare quando deve iniziare l'evento:

```
CREATE EVENT start_at_midnight
ON SCHEDULE
EVERY 1 DAY
STARTS TIMESTAMP '2025-03-01 00:00:00'
DO
UPDATE users SET active = 0 WHERE last_login < NOW() - INTERVAL 1 YEAR;
```

Questo evento disattiva gli utenti inattivi da più di un anno, eseguendosi a mezzanotte.

```
CREATE EVENT one_time_event
ON SCHEDULE
AT TIMESTAMP '2025-03-10 12:00:00'
DO
INSERT INTO notifications (message) VALUES ('Backup completato');
```

Questo evento viene eseguito **una sola volta** alla data e ora specificata.

Esempio da mettere nel do.

Eliminare record più vecchi di 7 giorni:

```
DELETE FROM logs WHERE created_at < NOW() - INTERVAL 7 DAY;
```

Nota: Per utilizzare gli eventi, il gestore degli eventi MySQL (**event scheduler**) deve essere attivo.
Puoi verificare il suo stato con:

```
SHOW VARIABLES LIKE 'event_scheduler';
```

Il risultato deve restituire ON.

FUNZIONI CONDIZIONALI.

```
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
```

La funzione **COALESCE()** viene utilizzata per ritornare un valore di default **non NULL** tra quelli specificati quando l'esatta cella è nulla.
La funzione **CASE** è simile a un'istruzione **SWITCH-CASE** e consente di eseguire operazioni condizionali.

Che fa la prima parte di questo codice?

Se **Dipart** ha un valore, restituisce quel valore.
Se **Dipart** è **NULL**, restituisce **'Ignoto'**.

Che fa la seconda parte di questo codice?

Se il valore di **Tipo** è **'Auto'**, calcola la tassa come **2.58 * KWatts**.
Se **Tipo** è **'Moto'**, calcola la tassa come **22.00 + 1.00 * KWatts**.
Se **Tipo** non è né **'Auto'** né **'Moto'**, restituisce **NULL**.
Filtra i veicoli con **Anno > 1975**.

***KWatts** è probabilmente una colonna della tabella **Veicolo**.*
Se non esiste, la query darà errore.
Puoi verificare se esiste con **DESC Veicolo;**

L'operazione **2.58 * KWatts (o tutte le altre)** viene assegnata alla colonna derivata **Tassa**, ma solo per alcuni valori della colonna **Tipo**. Vediamo in dettaglio.
Supponiamo che la tabella **Veicolo** contenga questi dati:

Targa	Tipo	KWatts	Anno
ABC123	Auto	50	1980
XYZ789	Moto	30	1990
LMN456	Camion	200	1995

Risultato della query:

Targa	Tassa
ABC123	129.00
XYZ789	52.00
LMN456	NULL

TRIGGER

Un **trigger** è come una **macchina automatica** che si attiva da sola quando succede qualcosa.

ESEMPIO FACILE:

Se prendi l'ultima fetta di torta, il forno si accende automaticamente per cuocerne un'altra.

DIFFERENZA TRA TRIGGER E STORED PROCEDURE.

Nel database: un **trigger** si attiva automaticamente quando avviene un'operazione come un inserimento, una modifica o una cancellazione.
Si attiva automaticamente.
Nel database: una **stored procedure** è una sequenza di istruzioni SQL che devi chiamare manualmente.
Si esegue manualmente con un comando CALL.
La attiva un utente o un programma esterno.

Esempio di TRIGGER (automatico):

```
CREATE TRIGGER traccia_prezzo
BEFORE UPDATE ON Prodotti
FOR EACH ROW
INSERT INTO StoricoPrezzi (id_prodotto, vecchio_prezzo, nuovo_prezzo, data_modifica)
VALUES (OLD.id, OLD.prezzo, NEW.prezzo, NOW());
```

Se qualcuno cambia il prezzo di un prodotto, il database salva automaticamente il vecchio prezzo in una tabella di storico.

Esempio di STORED PROCEDURE (manuale):

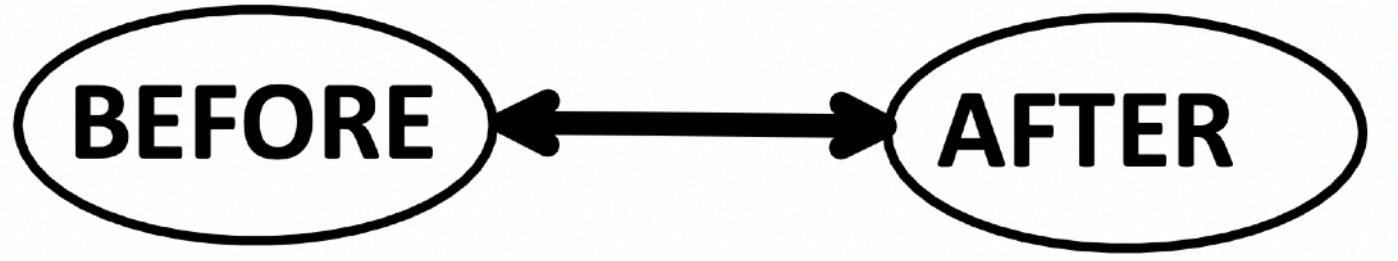
```
CREATE PROCEDURE aggiorna_prezzo(IN prodotto_id INT)
BEGIN
     UPDATE Prodotti
     SET prezzo = prezzo * 1.10
     WHERE id = prodotto_id;
END;
```

Vuoi aggiornare il prezzo di un prodotto e aumentarlo del 10%, ma solo quando decidi tu.
Questa procedura **NON si esegue da sola**. Devi chiamarla tu con:
`CALL aggiorna_prezzo(5);`

Il trigger scatta da solo, la stored procedure devi eseguirla manualmente.

Quindi, ricapitolando i trigger si attivano quando vengono scatenati determinati **EVENTI**.

TIPI DI EVENTI.



I **trigger** si attivano automaticamente in risposta a determinati eventi (INSERT, UPDATE, DELETE) su una tabella del database. Possono scattare **prima** o **dopo** l'evento.

Before: Prima che l'evento venga eseguito.
I trigger before possono condizionare i valori che si stanno per inserire nel database.
After: Dopo che l'evento è stata eseguito.

GRANULARITÀ DEGLI EVENTI.

Modalità statement-level (**for each** statement) o Modalità row-level (opzione **for each row**).

All'interno dei **Trigger**, ho la possibilità di operare sia a livello della granularità dell'intero **result set**, sia con la precisione della singola tupla coinvolta.

Nel caso di un **trigger a livello di istruzione (statement-level)**, se viene eseguito un **UPDATE** su 100 tuple, il **DBMS** considera l'intero insieme delle 100 tuple modificate, senza trattarle singolarmente.
MySQL non permette di osservare tutti i dati insieme.

MySQL permette la modalità row-level.

Nel caso di un **trigger a livello di riga (row-level)**, se l'operazione di **inserimento, aggiornamento o eliminazione** coinvolge **100 righe**, il **trigger** verrà attivato **100 volte**, una per ciascuna tupla interessata, permettendo così di osservare nel dettaglio il modo in cui ogni singola tupla viene **inserita, aggiornata o eliminata.**

ESEMPI DI TRIGGER.

```
CREATE TRIGGER prevent_negative_price
BEFORE INSERT ON Prodotti
FOR EACH ROW
BEGIN
    IF NEW.prezzo < 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Il prezzo non può essere
negativo!';
    END IF;
END;
```

Se qualcuno prova ad aggiungere un prodotto con prezzo negativo, MySQL blocca l'operazione.
In un **trigger**, **NEW** è una parola chiave che rappresenta i **nuovi valori** che stanno per essere inseriti o aggiornati in una riga della tabella.
"NEW" contiene i dati della riga appena inserita o modificata!
Funziona nei trigger INSERT e UPDATE, perché in questi casi ci sono nuovi dati che vengono salvati nella tabella.

NEW non è disponibile in **DELETE**, perché non ci sono nuovi dati, quando elimini una riga da una tabella, quella riga **non esiste più nel database** dopo l'operazione.

In un **trigger**, **OLD** è una parola chiave che rappresenta i **valori precedenti** di una riga **prima di un'operazione UPDATE o DELETE**. OLD non può essere usato in un trigger **INSERT**, perché prima dell'inserimento non esiste un valore precedente.

```
CREATE TRIGGER aggiorna_totale_ordini
AFTER INSERT ON Ordini
FOR EACH ROW
BEGIN
     UPDATE Clienti
     SET numero_ordini = numero_ordini + 1
     WHERE id = NEW.id_cliente;
END;
```

Ogni volta che viene aggiunto un nuovo ordine, vogliamo aggiornare il totale degli ordini di quel cliente. Dopo ogni nuovo ordine, il numero totale di ordini del cliente viene aggiornato automaticamente.

Troppi trigger possono rendere la base di dati **difficile da manutenere**, ma **non averli affatto** può portare a dati **inconsistenti.**

Evitare trigger annidati o dipendenti tra loro:

- Se un trigger A attiva B, che a sua volta attiva C, sarà **difficile da gestire.**
- **Meglio:** Usare una **stored procedure** richiamata da un solo trigger.

A volte una **vista** o una **foreign key con ON DELETE CASCADE** è meglio di un trigger.

```
CREATE TRIGGER log_eliminazione_utente
BEFORE DELETE ON Utenti
FOR EACH ROW
BEGIN
     INSERT INTO Log_Utenti (id_utente, nome,
data_eliminazione) VALUES (OLD.id, OLD.nome, NOW());
END;
```

Prima che un utente venga cancellato, il trigger salva il suo **id** e **nome** nella tabella **Log_Utenti.**

```
CREATE TRIGGER verifica_prezzo
BEFORE UPDATE ON Prodotti
FOR EACH ROW
BEGIN
     IF NEW.prezzo < OLD.prezzo * 0.8 THEN
     SIGNAL SQLSTATE '45000'
     SET MESSAGE_TEXT = 'Errore: il prezzo non può essere ridotto di oltre il 20%';
     END IF;
END;
```

Se qualcuno cerca di ridurre il prezzo più del 20%, MySQL blocca l'operazione.

```
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
```

Questo trigger si attiva prima che l'UPDATE venga eseguito. Se il **nuovo salario (NEW.Salario) supera il 20%** del vecchio salario (**OLD.Salario * 1.2**), allora **limita l'aumento** e imposta il nuovo salario al massimo consentito (**OLD.Salario * 1.2**).
In pratica, impedisce aumenti superiori al 20% prima che i dati vengano salvati nel database.

```
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
```

Quì new.salario è già scritto dentro la base di dati.
Agisce dopo l'update.

Utilizzo di Trigger per regole aziendali.

```
create trigger maggiorenni
before insert on `studenti` for each row
begin
    if New.eta < 18 then
            signal sqlstate '45000';
    end if;
end
```

Si attiva **prima** che venga inserito un nuovo record nella tabella **studenti**.
Controlla se **NEW.eta** (l'età del nuovo studente) è **minore di 18**.
Se l'età è inferiore a 18, genera un **errore SQLSTATE '45000'**, bloccando l'inserimento.

ATTENZIONE.

Quando un **trigger** in MySQL genera un errore con **SIGNAL SQLSTATE**, il client Java riceve un'**eccezione JDBC**.

Come Java capisce che un trigger ha generato un errore?

Supponiamo di avere il seguente **trigger su MySQL**:

```
CREATE TRIGGER maggiorenni
BEFORE INSERT ON studenti
FOR EACH ROW
BEGIN
    IF NEW.eta < 18 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Errore: lo studente deve essere maggiorenne!';
    END IF;
END;
```

Se qualcuno prova a inserire uno studente con meno di 18 anni, MySQL genera un errore SQLSTATE 45000.

Gestire l'errore da Java con **JDBC**

Ora vediamo come un programma Java può intercettare questo errore:

```java
public class TriggerErrorHandling {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/scuola";
        String user = "root";
        String password = "password";

        try (Connection conn = DriverManager.getConnection(url, user, password)) {
            String sql = "INSERT INTO studenti (nome, eta) VALUES (?, ?)";

            try (PreparedStatement stmt = conn.prepareStatement(sql)) {
                stmt.setString(1, "Luca");
                stmt.setInt(2, 17); // Studente minorenne

                stmt.executeUpdate();
                System.out.println("Studente inserito con successo!");
            }
        } catch (SQLException e) {
            // Controlliamo se l'errore proviene dal trigger (SQLSTATE 45000)
            if (e.getSQLState().equals("45000")) {
                System.err.println("⚠️ ERRORE DAL TRIGGER: " + e.getMessage());
            } else {
                System.err.println("❌ Errore SQL generico: " + e.getMessage());
            }
        }
    }
}
```

Quando un trigger genera un errore con **SIGNAL SQLSTATE**, il client JDBC lo vede come un'eccezione SQLException.
e.getSQLState() permette di distinguere gli errori provenienti dai trigger.
Gestire gli errori da trigger è utile per garantire integrità e sicurezza nei dati.