07-11-2022

SICUREZZA ACCESSO

Una base di dati è una **risorsa condivisa** fra utenti. Servono strumenti per poter specificare chi e come si può accedere e cosa può usare del Database.

- I privilegi ("**Diritti di Accesso**")possono essere rivolti a intere tabelle oppure anche ad altri tipi di risorse, quali singoli attributi, viste o domini. Essi servono per decidere se un utente ha la possibilità di *fare qualcosa*, leggere, scrivere, eliminare
- Un utente predefinito **_system** (amministratore della base di dati) **ha tutti i privilegi**
- **Chi crea una risorsa**, automaticamente, **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, in particolare si può concedere un privilegio ricevuto a terzi

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**)

Concessione e revoca

- Concessione di privilegi:
  - `grant < Privileges | all privileges > on Resource to Users [ with grant option ]`
  - dove `grant opion` specifica se il privilegio può essere trasmesso a terzi: per esempio `grant select on Department to Stefano`
- Revoca di privilegi:
  - `revoke Privileges on Resource from Users [ restrict | cascade ]`
  - `restrict` = voglio bloccare la rimozione al privilegio se tale privilegio è stato concesso
  - `cascade` = voglio rimuovere i privilegi all'utente e a tutti gli utenti che hanno ricevuto lo stesso privilegio da questo utente.

Autorizzazioni

> L'idea di base è che **bisogna nascondere** le risorse alle quali non si ha accesso, senza sospetti.

Per esempio --> Se si vuole accedere a una tabella segreta e non si ha il permesso, il messaggio di errore deve essere "*la tabella non esiste*" e **NON** *"Non hai permesso di accedere a tale tabella".

- Come si fa a far accedere un utente solo ad alcuni record? Usando una vista
- **Estensioni di SQL**: Concetto di **ruolo**, **cui si associano privilegi** (anche articolati), poi concessi agli utenti attribuendo il ruolo (tipo su *Discord*).

TRANSAZIONE

Una transazione è un insieme di operazioni che si può considerare **indivisibile** ("*atomico*"), corretto anche in presenza di concorrenza e con effetti definitivi

- Il risultato, dopo la transazione, è un risultato definitivo, (memorizzato nel DB) e lo stato fra prima/dopo la transazione si ha uno stato valido nel DB.

Transazione di tipo "**ACIDe**"

- **A**tomicità, gruppo di istruzioni indivisibile, viene eseguito interamente
  - La sequenza di operazioni sulla base di dati viene eseguita **per intero o per niente**
  - Se un'operazione, durante una transazione complessa, fallisce, allora fallisce tutta la transazione.

  > `*Esempio*: trasferimento di fondi da un conto A ad un conto B: o si fanno il prelevamento da A e il versamento su B (entrambe le transazioni) o nessuno dei due`

- **C**onsistenza
  - 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*")
- **I**solamento, le istruzioni eseguite devono essere corrette anche in presenza di utenti concorrenti.
  - L'effetto di transazioni concorrenti deve essere coerente (ad esempio "*equivalente*" all'esecuzione separata).
  - *Esempio*: se due assegni emessi sullo stesso conto corrente vengono incassati contemporaneamente si deve evitare di trascurarne uno.
- **D**urabilità (persistenza)
  - 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.

Transazione 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)

Una transazione si conclude:

- `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

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

DB ATTIVI

Una base di dati si dice attia se contiene trigger.
Un **trigger** è formato da una serie di regole attive svolte dal DB a seguito di qualche evento che viene scatenato dalle operazioni dell'utente.

Concetto di trigger

Esso utilizza il paradigma: **Evento-Condizione-Azione**:

- Evento, ovvero quando avviene **insert, delete, uptade**:
  - Se la condizione è vera → l'azione è eseguita
  - Se la condizione è falsa → l'azione non è eseguita
  - Quando accade l'evento, il trigger è **ATTIVATO**
- Condizione, un predicato che indica se il trigger deve essere eseguito:
  - Quando la condizione viene valutata, il trigger è **CONSIDERATO**
- Azione, ovvero una procedura SQL:

- Quando l'azione viene eseguita, il trigger è detto **ESEGUITO**

Caratteristiche trigger

- **nome**
- **target** (tabella controllata)
- **modalità** (**before** o **after**):
  - fa riferimento all'evento. `before` viene eseguito prima che l'azione sia stata eseguita sulla tabella. `Before insert` vuol dire che trigger eseguito prima che l'inserimento è compiuto. *Viceversa* con `after insert`.
- **evento** (`insert`, `delete` o `update`)
- **granularità** (`statement-level` o `row-level`): indica quante volte il trigger viene eseguito.
  - `Statement`: ogni operazione esegue il trigger una sola volta ed essi sono **più efficienti** (una volta per operazione) **ma più complessi da scrivere**.
  - row-level: opera, invece, su tutte le righe. Sono **meno efficienti** ma **più semplici da scrivere**
- **alias** dei valori o tabelle di transizione:
  - Nella **tabella di transizione** le righe di transizione (**row level**) oppure colonne di transizione(statement level) rappresentano lo stato prima e dopo l'operazione.
- **azione**
- **timestamp** di creazione

Sintassi trigger SQL

```
create trigger TriggerName
        { before | after }
        { insert | delete | update [of Column] } on Table
        [referencing
                {[old_table [as] OldTableAlias]
                 [new_table [as] NewTableAlias] } |
                {[old [row] [as] OldTupleName]
                 [new [row] [as] NewTupleName] }]
        [for each { row | statement }]
        [when Condition]
        --SQLStatements
```

- `referencing` è lo *statement level*
  - si hanno **tabelle virtuali** `old_table` e `new_table` e contengono i record **prima/dopo** l'evento considerato.

- Per trigger *row-level* si ha `old` e `new` e rappresentano la riga prima/dopo l'evento
  N.B: In caso di `insert` si ha solo `new` perchè la *old* è inesistente, in caso di `delete` si ha solo `old` perchè la *new* è inesistente. (per logica)
- old e new hanno la stessa struttura della tabella *Table*(cioè di quella indicata). Pertanto posso usarle come variabili che contengono il record che uso (es: `old.nome`, `new.cognome`)

Esempio:

```
-- Definire il trigger che limita l'aumento del salario di un impiegato ad
un massimo del 20%
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
```

- Non si può fare un aumento più elevato del 20%. Il trigger agisce prima di scrivere il nuovo dato. Se fallisce allora il nuovo dato non viene scritto

```
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
```

- Agisce dopo aver scritto il nuovo dato. -> se il trigger fallisce allora avrò un nuovo dato indesiderato.

Granularità degli eventi

Esistono 2 tipi per la granularità del trigger:

1. Di default è `for each statement`. In questo caso il trigger viene considerato ed eseguito solo una volta per ogni comando (statement) che lo ha attivato e non dipende dal numero di tuple modificate.
2. `for each row`. In questo caso il trigger viene considerato e possibilmente eseguito una volta per ogni tupla modificata.
   Esempio

```
create trigger AccountMonitor
after update on
Account for each row
when new.Total > old.Total
insert
        into Payments -- nella tabella Payments
        values
                (new.AccNumber,new.Total-old.Total) into Payments
```

Sintassi trigger DB2

```
CREATE TRIGGER CheckDecrement
AFTER UPDATE OF Salary ON Employee
FOR EACH ROW WHEN (NEW.Salary < OLD.Salary * 0.97)
BEGIN
        update Employee
        set Salary=OLD.Salary*0.97
         where RegNum = NEW.RegNum;
END;
```

La struttura `BEGIN-END` serve per contenere le istruzioni da eseguire.

Ordine di esecuzione dei trigger (conflitto)

Quando vi sono più trigger associati allo stesso evento (in conflitto), ovvero che un trigger è associato **allo stesso evento**, vengono eseguiti come segue:

1. Per primi i ***BEFORE*** trigger (*statement-level* e poi *row-level*)
2. Poi viene eseguita la modifica e **verificati i vincoli di integrità**
3. Infine sono eseguiti gli ***AFTER*** trigger (*row-level* e poi *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)

Ordine di esecuzione dei trigger (ricorsione)

In SQL:1999 i trigger 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:

1. Lo stato del TEC includente viene salvato e quello del TEC incluso viene eseguito. Ciò può accadere ricorsivamente
2. Alla fine dell'esecuzione di un TEC incluso, lo stato di esecuzione del TEC includente viene ripristinato e la sua esecuzione ripresa
3. L'esecuzione **termina correttamente** in uno "**stato quiescente**". Questo stato sta a significare che non ci sono altri TEC aperti. In caso di ciclo, non si arriva mai alla chiusura dei TEC
4. L'esecuzione **termina in errore quando si raggiunge una data profondità** di ricorsione dando luogo ad una eccezione di **non-terminazione**
5. 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, quindi non viene annullata tutta l'esecuzione ma annullata fino al punto che ha causato l'errore. E fino a quel punto è tutto memorizzato.

> TEC : Memorizzate le info prima, dopo l'operazione e tutti i valori delle tabelle prima/dopo (update) o solo dopo (insert) o solo prima (delete).

Sintassi trigger in Oracle

In Oracle 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

Esempio:

```
create trigger TriggerName
        { before | after } event [, event [,event ]]
        [[referencing
                [old [row] [as] OldTupleName]
                [new [row] [as] NewTupleName] ]
        for each { row | statement } [when Condition]]
--PL/SQLStatements
Event ::= { insert | delete | update [of Column] } on Table
```

Conflitti in Oracle

Quando ci sono conflitti, essi vengono eseguiti nel modo seguente:

1. Quando molti trigger sono associati allo stesso evento, ORACLE segue il seguente schema:
    1. Per primi, i *BEFORE* statement-level trigger
    2. Poi, i *BEFORE* row-level trigger
    3. Poi viene eseguita la modifica e verificati i vincoli di integrità
    4. Poi, gli *AFTER* row-level trigger
    5. Infine, gli *AFTER* statement-level trigger
2. 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)
3. "**Mutating table exception**": scatta se la catena di trigger attivati da un before trigger T cerca di modifcare lo stato della tabella target di T

Esempio:

```
--Evento: update of QtyDisponibile in Magazzino
--Condizione: Quantità sotto soglia e mancanza ordini esterni
--Azione: insert of OrdiniEsterni
create trigger Riordino
after update of QtyDisponibile on Magazzino
for each row
when (new.QtyDisponibile < new.QtySoglia)
declare X number;
        select count(*) into X
        from OrdiniEsterni
        where Parte = new.Parte;
if X = 0 then insert into OrdiniEsterni
        values(new.Parte,new.QtyRiordino,sysdate)
end if;
end;
```