

Trigger di database

- 10.1 **Tipi di trigger**
- 10.2 **Creazione di trigger**
- 10.3 **Tabelle mutanti**
- 10.4 **Sommario**

Il quarto tipo di blocco PL/SQL con nome è il trigger. I trigger condividono molte delle caratteristiche dei sottoprogrammi (esaminati nei due capitoli precedenti), ma presentano significative differenze nel modo in cui sono creati e chiamati. Nel presente capitolo si vedrà come creare diversi tipi di trigger e si descriveranno alcune possibili applicazioni.

10.1 Tipi di trigger

I trigger sono simili alle procedure o alle funzioni, per il fatto che sono blocchi PL/SQL con nome con sezioni di dichiarazione, eseguibile e di gestione di eccezioni. Come package e tipi oggetto (che verranno descritti nei Capitoli da 14 a 16), i trigger devono essere memorizzati come oggetti standalone nel database e non possono essere locali per un blocco o package. Come si è visto negli ultimi due capitoli, una procedura viene eseguita esplicitamente da un altro blocco attraverso una chiamata di procedura, che può anche passare argomenti. D'altra parte, un trigger viene eseguito implicitamente ogni qualvolta si verifica l'evento di attivazione e un trigger non accetta argomenti. L'atto di eseguire un trigger è noto come *attivazione* del trigger. L'evento di attivazione può essere un'operazione DML (INSERT, UPDATE o DELETE) su una tabella di database o su certi tipi di viste; oppure un evento di sistema, come l'avvio o la chiusura di un database e certi tipi di operazioni DDL. Si vedranno gli eventi di attivazione in dettaglio più avanti nel presente capitolo.

I trigger possono essere utilizzati in molte situazioni, tra cui:

- mantenere complesse limitazioni di integrità non possibili attraverso limitazioni dichiarative abilitate alla creazione della tabella;
- verificare informazioni in una tabella registrando le modifiche effettuate e chi le ha effettuate;

- segnalare automaticamente ad altri programmi che devono essere effettuate azioni quando vengono apportate modifiche a una tabella;
- pubblicare informazioni su diversi eventi in un ambiente publish/subscribe.

Esistono tre tipi principali di trigger: trigger DML, instead-of e di sistema presentati nei paragrafi successivi. Si vedranno ulteriori dettagli più avanti nel paragrafo “Creazione di trigger”.

NOTA *Oracle ammette la scrittura di trigger in PL/SQL o in altri linguaggi che possono essere chiamati come routine esterne. Per ulteriori informazioni consultare il paragrafo “Corpi di trigger” più avanti nel capitolo (oltre al Capitolo 12 per ulteriori informazioni sulle routine esterne in generale).*

Trigger DML

Un *trigger DML* viene attivato da una dichiarazione DML e il tipo di dichiarazione determina il tipo di trigger DML. I trigger DML possono essere definiti per operazioni INSERT, UPDATE, o DELETE. Possono essere attivati prima o dopo l’operazione su una riga. I trigger DML possono agire su tutte le righe o solo su alcune. Agiscono su un sottoinsieme di righe quando sono definiti come trigger a livello di dichiarazione. La differenza sta nel fatto che un trigger a livello di dichiarazione utilizza una clausola WHEN per calcolare dove si sta verificando un tipo specifico di cambiamento. Inserendo la condizione in una clausola WHEN, si elimina l’esecuzione del trigger se la condizione non si verifica.

Come esempio, si supponga di voler tracciare statistiche su diverse categorie, compreso il numero di libri nel database e il prezzo medio in ogni categoria. I risultati verranno memorizzati nella tabella `category_stats`:

```
-- Disponibile online come parte di of tables.sql
CREATE TABLE category_stats (
  category      VARCHAR2(20),
  total_books   NUMBER,
  average_price NUMBER
);
```

Per tenere aggiornata `category_stats`, è possibile creare un trigger su `books` che aggiornerà `category_stats` ogni qualvolta viene modificata `books`. Il trigger `UpdateCategoryStats`, mostrato di seguito, effettua questa operazione. Dopo ogni operazione DML su `books`, viene eseguito il trigger. Il corpo del trigger interroga `books` e aggiorna `category_stats` con le statistiche correnti.

```
-- Disponibile online come UpdateCategoryStats.sql
CREATE OR REPLACE TRIGGER UpdateCategoryStats
/* Tiene aggiornata la tabella category_stats con le modifiche fatte
   alla tabella books. */
```

```

AFTER INSERT OR DELETE OR UPDATE ON books
DECLARE
  CURSOR c_Statistics IS
    SELECT category,
           COUNT(*) total_books,
           AVG(price) average_price
    FROM books
    GROUP BY category;
BEGIN
  /* Prima cancella da category_stats. Questo azzerera' le
     statistiche ed e' necessario per considerare la cancellazione
     di tutti i libri in una data categoria */
  DELETE FROM category_stats;

  /* Ora scorre in ogni categoria e inserisce la riga adatta in
     category_stats. */
  FOR v_StatsRecord in c_Statistics LOOP
    INSERT INTO category_stats (category, total_books, average_price)
      VALUES (v_StatsRecord.category, v_StatsRecord.total_books,
              v_StatsRecord.average_price);
  END LOOP;
END UpdateCategoryStats;
/

```

Un trigger di dichiarazione può essere attivato da più tipi di dichiarazione di attivazione. Per esempio, `UpdateCategoryStats` viene attivato con dichiarazioni `INSERT`, `UPDATE` e `DELETE`. L'evento di attivazione specifica una o più operazioni DML che devono attivare il trigger.

Trigger instead-of

I trigger *instead-of* possono essere definiti solo su viste (relazionali od oggetto). A differenza di un trigger DML, che viene eseguito oltre all'operazione DML, un trigger *instead-of* viene eseguito in luogo della dichiarazione DML che lo ha attivato. I trigger *instead-of* devono essere a livello di riga. Per esempio, si consideri la vista `books _ authors`:

```

-- Disponibile online come parte di of_insteadOf1.sql
CREATE OR REPLACE VIEW books_authors AS
  SELECT b.isbn, b.title, a.first_name, a.last_name
  FROM books b, authors a
  WHERE b.author1 = a.id
        OR b.author2 = a.id
        OR b.author3 = a.id;

```

È illegale inserire direttamente in questa vista, poiché è un'unione di due tabelle e l'operazione `INSERT` richiede che entrambe le tabelle sottostanti siano modificate, come mostra la sessione `SQL*Plus` di seguito:

```
-- Disponibile online come parte di of instead0f1.sql
INSERT INTO books_authors (isbn, title, first_name, last_name)
VALUES ('72230665', 'Oracle Database 10g PL/SQL Programming', 'Joe', 'Blow');
```

L'output illustra il fallimento dell'inserimento nella vista:

```
INSERT INTO books_authors (isbn, title, first_name, last_name)
*
ERROR at line 1:
ORA-01779: cannot modify a column which maps to a non key-preserved table
```

Tuttavia, è possibile creare un trigger instead-of che effettua l'operazione corretta per un INSERT, vale a dire aggiornare le tabelle sottostanti:

```
-- Disponibile online come parte di of instead0f2.sql
CREATE OR REPLACE TRIGGER InsertBooksAuthors
  INSTEAD OF INSERT ON books_authors
DECLARE
  v_Book books%ROWTYPE;
  v_AuthorID authors.id%TYPE;
BEGIN
  -- Trova l'ID del nuovo autore
  BEGIN
    SELECT id
      INTO v_AuthorID
     FROM authors
    WHERE first_name = :new.first_name
          AND last_name = :new.last_name;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      -- Nessun autore trovato, ne crea uno nuovo
      INSERT INTO authors (id, first_name, last_name)
        VALUES (author_sequence.NEXTVAL, :new.first_name, :new.last_name)
        RETURNING ID INTO v_AuthorID;
  END;

  SELECT *
    INTO v_Book
   FROM books
  WHERE isbn = :new.isbn;

  -- Scopre se il libro ha già 1 o 2 autori e aggiorna
  -- di conseguenza
  IF v_Book.author2 IS NULL THEN
    UPDATE books
      SET author2 = v_AuthorID
     WHERE isbn = :new.isbn;
  ELSE
    UPDATE books
```

```

        SET author3 = v_AuthorID
        WHERE isbn = :new.isbn;
    END IF;
END InsertBooksAuthors;
/

```

Con il trigger `InsertBooksAuthors`, la dichiarazione `INSERT` va a buon fine ed effettua l'operazione corretta.

NOTA *Per come è scritto, `InsertBooksAuthors` non ha alcun controllo di errore. Questo problema verrà corretto più avanti nel capitolo nel paragrafo “Creazione di trigger instead-of”.*

Trigger di sistema

Un *trigger di sistema* viene attivato quando si verifica un evento di sistema, come l'avvio o la chiusura di un database, non quando si verifica un'operazione DML su una tabella. Un trigger di sistema può anche essere attivato su operazioni DDL, come la creazione di tabelle. Per esempio, si supponga di voler registrare ogni qualvolta viene creato un dizionario di dati. È possibile farlo creando una tabella, come di seguito:

```

-- Disponibile online come parte di of LogCreations.sql
CREATE TABLE ddl_creations (
    user_id      VARCHAR2(30),
    object_type  VARCHAR2(20),
    object_name  VARCHAR2(30),
    object_owner VARCHAR2(30),
    creation_date DATE);

```

Quando la tabella è disponibile, è possibile creare un trigger di sistema per registrare le informazioni relative. Il trigger `LogCreations` fa esattamente questo: dopo ogni operazione `CREATE` sullo schema corrente, registra in `ddl_creations` informazioni sull'oggetto appena creato.

```

-- Disponibile online come parte di of LogCreations.sql
CREATE OR REPLACE TRIGGER LogCreations
AFTER CREATE ON SCHEMA
BEGIN
    INSERT INTO ddl_creations (user_id, object_type, object_name,
                               object_owner, creation_date)
    VALUES (USER, ORA_DICT_OBJ_TYPE, ORA_DICT_OBJ_NAME,
            ORA_DICT_OBJ_OWNER, SYSDATE);
END LogCreations;
/

```

10.2 Creazione di trigger

Indipendentemente dal tipo, tutti i trigger vengono creati utilizzando la stessa sintassi. La sintassi generale per la creazione di un trigger è:

```
CREATE [OR REPLACE] TRIGGER nome_trigger
  {BEFORE | AFTER | INSTEAD OF} evento_attivante
  [clausola_referenziale]
  [WHEN condizione_trigger]
  [FOR EACH ROW]
  corpo_trigger;
```

dove *nome_trigger* è il nome del trigger, *evento_attivante* specifica l'evento che attiva il trigger (può anche comprendere una tabella o vista specifica) e *corpo_trigger* è il codice principale per il trigger. *clausola_referenziale* viene utilizzata per fare riferimento ai dati nella riga che viene modificata da un nome diverso. *condizione_trigger* nella clausola WHEN, se presente, è la prima ad essere valutata e il corpo del trigger viene eseguito solo quando tale condizione viene calcolata come TRUE. Si vedranno ulteriori esempi di diversi tipi di trigger nei paragrafi successivi.

NOTA *Il corpo del trigger non può superare 32 K. Se esiste un trigger che supera tali dimensioni, è possibile ridurlo spostando parte del codice in package o procedure memorizzate compilati separatamente, e chiamandoli dal corpo del trigger. In generale è buona norma mantenere i corpi dei trigger di dimensioni ridotte, a causa della frequenza con cui vengono eseguiti.*

Il corpo del trigger è un blocco PL/SQL, che deve contenere almeno una sezione eseguibile. Come per qualsiasi altro blocco, le sezioni di dichiarazione e di gestione delle eccezioni sono opzionali. Se esiste una sezione di dichiarazione, tuttavia, *corpo_trigger* deve cominciare con la parola chiave DECLARE. Questa situazione è diversa da quella di un sottoprogramma, dove la parola chiave DECLARE non è presente.

Creazione di trigger DML

Un trigger DML viene attivato da un'operazione INSERT, UPDATE o DELETE su una tabella di database. Può essere attivato prima o dopo l'esecuzione della dichiarazione e può essere attivato una volta per riga interessata o una volta per dichiarazione. La combinazione di tali fattori determina il tipo di trigger. Esiste un totale di 28 possibili tipi: (3 dichiarazioni + 4 combinazioni di dichiarazioni) x 2 tempi x 2 livelli. Per esempio, tutti quelli indicati di seguito sono tipi di trigger DML validi:

- prima di UPDATE a livello di dichiarazione;
- dopo INSERT a livello di riga;
- prima di DELETE a livello di riga.

La Tabella 10.1 riassume le diverse opzioni. Un trigger può anche essere attivato per più tipi di dichiarazioni DML su una data tabella, per esempio INSERT e UPDATE. Qualsiasi codice nel trigger verrà eseguito insieme alla dichiarazione di attivazione stessa, come parte della stessa transazione.

Una tabella può avere qualsiasi numero di trigger definiti su di essa, compresi più tipi DML. Per esempio, è possibile definire due trigger dopo DELETE a livello di dichiarazione. Tutti i trigger dello stesso tipo verranno attivati in sequenza (per ulteriori informazioni sull'ordine di attivazione di trigger, consultare il paragrafo successivo).

L'evento di attivazione per un trigger DML specifica il nome della tabella (e della colonna) su cui verrà attivato il trigger. Un trigger può anche essere attivato su una colonna di una tabella annidata. Per ulteriori informazioni sulle tabelle annidate consultare il Capitolo 6.

Tabella 10.1 Tipi di trigger DML.

CATEGORIA	VALORI	COMMENTI
Dichiarazione	INSERT, DELETE o UPDATE	Definisce quale tipo di dichiarazione DML causa l'attivazione del trigger.
Tempo	BEFORE o AFTER	Definisce se il trigger viene attivato prima o dopo l'esecuzione della dichiarazione.
Livello	Riga o dichiarazione	Se il trigger è a livello di riga, viene attivato una volta per ciascuna riga interessata dalla dichiarazione di attivazione. Se il trigger è a livello di dichiarazione, viene attivato una volta, prima o dopo la dichiarazione. Un trigger a livello di riga è identificato dalla clausola FOR EACH ROW nella definizione di trigger.

Ordine di attivazione di trigger DML

I trigger sono attivati quando viene eseguita la dichiarazione DML. L'algoritmo per l'esecuzione di una dichiarazione DML è indicato qui di seguito.

1. Esecuzione dei trigger before a livello di dichiarazione, se presenti.
2. Per ogni riga interessata dalla dichiarazione:
 - esecuzione dei trigger before a livello di riga, se presenti;
 - esecuzione della dichiarazione stessa;
 - esecuzione dei trigger after a livello di riga, se presenti.
3. Esecuzione dei trigger after a livello di dichiarazione, se presenti.

Per illustrare l'algoritmo, si supponga di creare tutti i quattro tipi di trigger UPDATE sulla tabella books, before e after, a livello di dichiarazione e di riga. Verranno anche creati tre trigger prima della riga e due trigger dopo la dichiarazione, come di seguito:

```
-- Disponibile online come parte di of firingOrder.sql
CREATE SEQUENCE trig_seq
  START WITH 1
  INCREMENT BY 1;

CREATE OR REPLACE PACKAGE TrigPackage AS
  -- Contatore globale da utilizzare nei trigger
  v_Counter NUMBER;
END TrigPackage;
/

CREATE OR REPLACE TRIGGER BooksBStatement
  BEFORE UPDATE ON books
BEGIN
  -- Prima azzerata il contatore.
  TrigPackage.v_Counter := 0;

  INSERT INTO temp_table (num_col, char_col)
    VALUES (trig_seq.NEXTVAL,
      'Before Statement: counter = ' || TrigPackage.v_Counter);

  -- Ora lo incrementa per il trigger successivo.
  TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
END BooksBStatement;
/

CREATE OR REPLACE TRIGGER BooksAStatement1
  AFTER UPDATE ON books
BEGIN
  INSERT INTO temp_table (num_col, char_col)
    VALUES (trig_seq.NEXTVAL,
      'After Statement 1: counter = ' || TrigPackage.v_Counter);

  -- Incrementa per il trigger successivo.
  TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
END BooksAStatement1;
/

CREATE OR REPLACE TRIGGER BooksAStatement2
  AFTER UPDATE ON books
BEGIN
  INSERT INTO temp_table (num_col, char_col)
    VALUES (trig_seq.NEXTVAL,
      'After Statement 2: counter = ' || TrigPackage.v_Counter);

  -- Incrementa per il trigger successivo.
  TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
END BooksAStatement2;
/

CREATE OR REPLACE TRIGGER BooksBRow1
  BEFORE UPDATE ON books
```

```

FOR EACH ROW
BEGIN
  INSERT INTO temp_table (num_col, char_col)
    VALUES (trig_seq.NEXTVAL,
      'Before Row 1: counter = ' || TrigPackage.v_Counter);

  -- Incrementa per il trigger successivo.
  TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
END BooksBRow1;
/

CREATE OR REPLACE TRIGGER BooksBRow2
  BEFORE UPDATE ON books
  FOR EACH ROW
  BEGIN
    INSERT INTO temp_table (num_col, char_col)
      VALUES (trig_seq.NEXTVAL,
        'Before Row 2: counter = ' || TrigPackage.v_Counter);

    -- Incrementa per il trigger successivo.
    TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
  END BooksBRow2;
/

CREATE OR REPLACE TRIGGER BooksBRow3
  BEFORE UPDATE ON books
  FOR EACH ROW
  BEGIN
    INSERT INTO temp_table (num_col, char_col)
      VALUES (trig_seq.NEXTVAL,
        'Before Row 3: counter = ' || TrigPackage.v_Counter);

    -- Incrementa per il trigger successivo.
    TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
  END BooksBRow3;
/

CREATE OR REPLACE TRIGGER BooksARow
  AFTER UPDATE ON books
  FOR EACH ROW
  BEGIN
    INSERT INTO temp_table (num_col, char_col)
      VALUES (trig_seq.NEXTVAL,
        'After Row: counter = ' || TrigPackage.v_Counter);

    -- Incrementa per il trigger successivo.
    TrigPackage.v_Counter := TrigPackage.v_Counter + 1;
  END BooksARow;
/

```

Ora si supponga di emettere la seguente dichiarazione UPDATE:

```
-- Disponibile online come parte di of firingOrder.sql
UPDATE books
  SET category = category
  WHERE category = 'Oracle Ebusiness';
```

Questa dichiarazione interessa tre righe. I trigger a livello di dichiarazione before e after vengono eseguiti ciascuno una volta e i trigger a livello di riga before e after vengono eseguiti ciascuno tre volte. È possibile utilizzare la seguente query per selezionare da temp_table:

```
-- Disponibile online come parte di of firingOrder.sql
SELECT *
FROM temp_table
ORDER BY num_col;
```

Si vedrà il seguente output:

```
NUM_COL CHAR_COL
-----
1 Before Statement: counter = 0
2 Before Row 3: counter = 1
3 Before Row 2: counter = 2
4 Before Row 1: counter = 3
5 After Row: counter = 4
6 Before Row 3: counter = 5
7 Before Row 2: counter = 6
8 Before Row 1: counter = 7
9 After Row: counter = 8
10 Before Row 3: counter = 9
11 Before Row 2: counter = 10
12 Before Row 1: counter = 11
13 After Row: counter = 12
14 After Statement 2: counter = 13
15 After Statement 1: counter = 14
```

Ogni trigger, quando viene attivato, vede le modifiche effettuate dai trigger precedenti, oltre alle modifiche effettuate dalla dichiarazione sul database fino a quel momento. Questo si può vedere dal valore di contatore stampato da ciascun trigger (per ulteriori informazioni sull'utilizzo di variabili di package consultare il Capitolo 9).

L'ordine con cui sono attivati i trigger dello stesso tipo non è definito. Nell'esempio precedente, ciascun trigger vedrà le modifiche effettuate dai trigger precedenti. Se l'ordine è importante, combinare tutte le operazioni in un solo trigger.

NOTA *Quando si crea uno snapshot log per una tabella, Oracle creerà automaticamente un trigger after riga per la tabella, che aggiornerà il log dopo ogni dichiarazione DML. È necessario saperlo, se su tale tabella deve essere creato un ulteriore trigger after riga. Esistono anche ulteriori*

limitazioni su trigger e snapshot (noti in Oracle9i come viste materializzate). Per ulteriori informazioni, consultare la documentazione Oracle Database Advanced Replication.

Identificatori di correlazione in trigger a livello di riga

Un trigger a livello di riga viene attivato una volta per ogni riga elaborata dalla dichiarazione di attivazione. All'interno del trigger, è possibile accedere ai dati nella riga attualmente in fase di elaborazione. Questo viene effettuato attraverso due identificatori di correlazione, `:old` e `:new`. Un *identificatore di correlazione* è uno speciale tipo di variabile bind PL/SQL. I due punti davanti a ciascun identificatore indicano che sono variabili bind, nel senso di variabili host utilizzate in PL/SQL embedded, e indica che non sono normali variabili PL/SQL. Il compilatore PL/SQL le tratterà come record di tipo `triggering_table%ROWTYPE`, dove `triggering_table` è la tabella per cui è definito il trigger. Quindi, un riferimento come:

```
:new.campo
```

sarà valido solo se `campo` è un campo nella tabella di attivazione. I significati di `:old` e `:new` sono descritti nella Tabella 10.2. Sebbene sintatticamente siano trattati come record, in realtà non lo sono (come discusso più avanti nel paragrafo “Pseudorecord”); per questo motivo `:old` e `:new` sono anche noti come *pseudorecord*.

NOTA *L'identificatore `:old` è indefinito per dichiarazioni INSERT e `:new` è indefinito per dichiarazioni DELETE. Il compilatore PL/SQL non genererà un errore se si utilizza `:old` in un INSERT o `:new` in un DELETE, ma i valori di campo di entrambi saranno NULL.*

Oracle definisce un ulteriore identificatore di correlazione, `:parent`. Se il trigger è definito su una tabella annidata, `:old` e `:new` fanno riferimento alle righe nella tabella annidata, mentre `:parent` fa riferimento alla riga corrente della tabella genitore. Per ulteriori informazioni, consultare la documentazione Oracle.

Tabella 10.2 Gli identificatori di correlazione `:old` e `:new`.

DICHIARAZIONE DI ATTIVAZIONE	:OLD	:NEW
INSERT	Indefinito: tutti i campi sono NULL	Valori che verranno inseriti quando la dichiarazione è completa.
UPDATE	Valori originali per la riga prima dell'aggiornamento	Nuovi valori che verranno aggiornati quando la dichiarazione è completa.
DELETE	Valori originali prima della cancellazione della riga	Indefinito: tutti i campi sono NULL.

Utilizzo di :old e :new. Il trigger `GenerateAuthorID`, mostrato di seguito, utilizza `:new`. È un trigger prima di `INSERT` e lo scopo è compilare il campo `ID` di `authors` con un valore generato dalla sequenza `author_sequence`.

```
-- Disponibile online come parte di of GenerateAuthorID.sql
CREATE OR REPLACE TRIGGER GenerateAuthorID
  BEFORE INSERT OR UPDATE ON authors
  FOR EACH ROW
BEGIN
  /* Compila il campo ID di authors con il valore successivo da
  author_sequence. Poiche' ID e' una colonna in authors, :new.ID
  e' un riferimento valido. */
  SELECT author_sequence.NEXTVAL
  INTO :new.ID
  FROM dual;
END GenerateAuthorID;
/
```

In realtà `GenerateAuthorID` modifica il valore di `:new.ID`. Questa è una delle caratteristiche utili di `:new`: quando la dichiarazione viene eseguita, verranno utilizzati i valori in `:new`. Con `GenerateAuthorID` è possibile emettere una dichiarazione `INSERT` come la seguente:

```
-- Disponibile online come parte di of GenerateAuthorID.sql
INSERT INTO authors (first_name, last_name)
  VALUES ('Lolita', 'Lazarus');
```

senza generare un errore. Anche se non è stato specificato un valore per la chiave primaria di colonna `ID` (necessario), il trigger lo fornirà. Di fatto, se si specifica un valore per `ID`, sarà ignorato, perché il trigger lo modifica. Se si emette

```
-- Disponibile online come parte di of GenerateAuthorID.sql
INSERT INTO authors (ID, first_name, last_name)
  VALUES (-7, 'Zelda', 'Zoom');
```

la colonna `ID` verrà popolata da `author_sequence.NEXTVAL`, invece che contenere `-7`.

Come risultato, non è possibile modificare `:new` in un trigger a livello di riga `after`, perché la dichiarazione è già stata elaborata. In generale, `:new` viene modificato solo in un trigger a livello di riga `before`, e `:old` non viene mai modificato, da esso si legge solamente.

I record `:new` e `:old` sono validi solo in trigger a livello di riga. Se si cerca di fare riferimento a uno di essi in un trigger a livello di dichiarazione, si otterrà un errore di compilazione. Poiché un trigger a livello di riga viene eseguito una volta, anche se vengono elaborate molte righe dalla dichiarazione, `:old` e `:new` non hanno significato. A quale riga farebbero riferimento?

Pseudorecord. Sebbene `:new` e `:old` siano sintatticamente trattati come record `triggering_table%ROWTYPE`, in realtà non lo sono. Come risultato, le operazioni che normalmente sarebbero valide sui record non lo sono per `:new` e `:old`. Per esempio, non possono essere assegnati come interi record. Possono essere assegnati solo i singoli campi al loro interno, come illustrato nell'esempio di seguito:

```
-- Disponibile online come pseudoRecords.sql
CREATE OR REPLACE TRIGGER TempDelete
  BEFORE DELETE ON temp_table
  FOR EACH ROW
DECLARE
  v_TempRec temp_table%ROWTYPE;
BEGIN
  /* Questa non e' un'assegnazione legale, poiche' :old non e' davvero
     un record. */
  v_TempRec := :old;
  /* E' possibile fare la stessa cosa, tuttavia, assegnando
     singolarmente i campi. */
  v_TempRec.char_col := :old.char_col;
  v_TempRec.num_col := :old.num_col;
END TempDelete;
/
```

Inoltre `:old` e `:new` non possono essere passati a procedure o funzioni che prendano argomenti di `triggering_table%ROWTYPE`. A causa di questo comportamento lo script `pseudoRecords.sql` fallirà con il seguente messaggio di errore:

```
LINE/COL ERROR
-----
6/16      PLS-00049: bad bind variable 'OLD'
```

Clausola REFERENCING. Se si desidera, è possibile utilizzare la clausola `REFERENCING` per specificare un nome diverso per `:old` e `:new`. Tale clausola si trova dopo l'evento di attivazione, prima della clausola `WHEN`, con la sintassi

```
REFERENCING [OLD AS vecchio_nome] [NEW AS nuovo_nome]
```

Nel corpo del trigger, è possibile utilizzare `:vecchio_nome` e `:nuovo_nome` invece di `:old` e `:new`. Si noti che gli identificatori di correlazione non hanno due punti nella clausola `REFERENCING`. Ciò che segue è una versione alternativa del trigger `GenerateAuthorID`, che utilizza `REFERENCING` per fare riferimento a `:new` come `:new_author`:

```
-- Disponibile online come parte di of GenerateStudentID.sql
CREATE OR REPLACE TRIGGER GenerateAuthorID
  BEFORE INSERT OR UPDATE ON authors
  REFERENCING new AS new_author
  FOR EACH ROW
```

```
BEGIN
  /* Compila il campo ID di authors con il valore successivo da
     author_sequence. Poiche' ID e' una colonna in authors, :new.ID
     e' un riferimento valido. */
  SELECT author_sequence.NEXTVAL
     INTO :new_author.ID
     FROM dual;
END GenerateAuthorID;
/
```

La clausola WHEN

La clausola WHEN è valida solo per trigger a livello di riga. Se presente, il corpo del trigger sarà eseguito solo per le righe che soddisfano la condizione specificata dalla clausola WHEN. La clausola WHEN ha la seguente sintassi:

```
WHEN condizione_trigger
```

dove *condizione_trigger* è un'espressione booleana, calcolata per ogni riga. Si può fare riferimento ai record :new e :old anche all'interno di *condizione_trigger*, ma come per REFERENCING, qui non vengono utilizzati i due punti, validi solo nel corpo del trigger. Per esempio, il corpo del trigger CheckPrice viene eseguito solo se il prezzo di un dato libro è superiore a \$ 49,99:

```
-- Disponibile online come parte di of CheckPrice1.sql
CREATE OR REPLACE TRIGGER CheckPrice
  BEFORE INSERT OR UPDATE OF price ON books
  FOR EACH ROW
  WHEN (new.price > 49.99) BEGIN
  /* Qui va il corpo del trigger. */
  NULL;
END;
/
```

CheckPrice potrebbe anche essere scritto come di seguito:

```
CREATE OR REPLACE TRIGGER CheckPrice
  BEFORE INSERT OR UPDATE OF price ON books
  FOR EACH ROW
BEGIN
  IF :new.price > 49.99 THEN
    /* Qui va il corpo del trigger. */
    NULL;
  END IF;
END;
/
```

Predicati di trigger: INSERTING, UPDATING e DELETING

Il trigger UpdateCategoryStats esaminato in precedenza nel capitolo è un trigger INSERT, UPDATE e DELETE. All'interno di un trigger di questo tipo (che verrà attivato per diversi tipi di dichiarazioni DML) si trovano tre funzioni

booleane che possono essere utilizzate per determinare il tipo di operazione. Tali predicati sono INSERTING, UPDATING e DELETING; il loro comportamento è descritto nella Tabella 10.3.

Tabella 10.3 Comportamento dei predicati

PREDICATO	COMPORTAMENTO
INSERTING	TRUE se la dichiarazione di attivazione è un INSERT; FALSE negli altri casi.
UPDATING	TRUE se la dichiarazione di attivazione è un UPDATE; FALSE negli altri casi.
DELETING	TRUE se la dichiarazione di attivazione è un DELETE; FALSE negli altri casi.

NOTA *Esistono ulteriori funzioni che possono essere chiamate nel corpo di un trigger, analoghe ai predicati di trigger. Per ulteriori informazioni consultare il paragrafo “Funzioni di attributo di eventi” più avanti nel capitolo.*

Il trigger LogInventoryChanges utilizza i predicati per registrare tutte le modifiche effettuate alla tabella inventory. Oltre alle modifiche, registra l'utente che le ha effettuate. I record sono mantenuti nella tabella inventory_audit, che ha il seguente aspetto:

-- Disponibile online come parte di of logInventoryChanges1.sql

```
CREATE TABLE inventory_audit (
  change_type      CHAR(1) NOT NULL,
  changed_by       VARCHAR2(8) NOT NULL,
  timestamp        DATE NOT NULL,
  old_isbn         CHAR(10),
  new_isbn         CHAR(10),
  old_status       VARCHAR2(25),
  new_status       VARCHAR2(25),
  old_status_date  DATE,
  new_status_date  DATE,
  old_amount       NUMBER,
  new_amount       NUMBER
);
```

LogInventoryChanges viene creato con:

-- Disponibile online come parte di of logInventoryChanges1.sql

```
CREATE OR REPLACE TRIGGER LogInventoryChanges
  BEFORE INSERT OR DELETE OR UPDATE ON inventory
  FOR EACH ROW
  DECLARE
    v_ChangeType CHAR(1);
  BEGIN
    /* Usa 'I' per un INSERT, 'D' per DELETE e 'U' per UPDATE. */
    IF INSERTING THEN
      v_ChangeType := 'I';
    ELSIF UPDATING THEN
      v_ChangeType := 'U';
```

```

ELSE
  v_ChangeType := 'D';
END IF;

/* Registra tutte le modifiche fatte a inventory in
   inventory_audit. Usa SYSDATE per generare il timestamp, e
   USER per restituire lo userid dell'utente corrente. */
INSERT INTO inventory_audit
  (change_type, changed_by, timestamp,
   old_isbn, old_status, old_status_date, old_amount,
   new_isbn, new_status, new_status_date, new_amount)
VALUES
  (v_ChangeType, USER, SYSDATE,
   :old.isbn, :old.status, :old.status_date, :old.amount,
   :new.isbn, :new.status, :new.status_date, :new.amount);
END LogInventoryChanges;
/

```

La seguente dichiarazione di aggiornamento aggiorna due righe e illustra il comportamento di LogInventoryChanges:

```

-- Disponibile online come parte di of logInventoryChanges2.sql
UPDATE inventory
SET amount = 2000
WHERE isbn IN ('72223049', '72223855');

```

È possibile interrogare la tabella Inventory_Audit per vedere il trigger in azione.

```
SELECT change_type, old_amount, new_amount FROM inventory_audit;
```

L'output della query è mostrato di seguito:

```

C OLD_AMOUNT NEW_AMOUNT
- - - - -
U      1,000      2,000
U      1,000      2,000

```

I trigger vengono molto utilizzati per l'auditing, come in LogInventoryChanges. Mentre l'auditing a livello di LogInventoryChanges è disponibile come parte del database, i trigger ammettono registrazioni più personalizzate e flessibili. LogInventoryChanges potrebbe essere modificato, per esempio, per registrare le modifiche effettuate solo da certe persone. Potrebbe anche controllare se gli utenti hanno i permessi per effettuare modifiche e sollevare un errore (con RAISE _APPLICATION_ERROR) se non li hanno.

Creazione di trigger instead-of

A differenza dei trigger DML, che vengono attivati oltre alle operazioni INSERT, UPDATE o DELETE (prima o dopo di esse), i trigger instead-of (come implica il nome, che significa “invece di”) vengono attivati in luogo di un’operazione DML e la sostituiscono. Inoltre i trigger instead-of possono essere definiti solo su viste, mentre i trigger DML sono definiti su tabelle. I trigger instead-of sono utilizzati in due casi:

- per consentire di modificare una vista altrimenti non modificabile;
- per modificare le colonne di una colonna di tabella annidata in una vista.

Il primo caso verrà descritto nel presente paragrafo. Per ulteriori informazioni sulle tabelle annidate, consultare il Capitolo 6.

Viste modificabili e non modificabili

Una vista modificabile è una vista su cui è possibile emettere una dichiarazione DML. In generale, una vista è modificabile se non contiene alcuni dei seguenti elementi:

- operatori set (UNION, UNION ALL, MINUS);
- funzioni aggregate (SUM, AVG ecc.);
- clausole GROUP BY, CONNECT BY o START WITH;
- l’operatore DISTINCT;
- unioni.

Tuttavia esistono alcune viste che contengono unioni e che sono modificabili. In generale, una vista unione è modificabile se l’operazione DML su di essa modifica solo una tabella di base per volta e se la dichiarazione DML soddisfa le condizioni indicate nella Tabella 10.4 (per ulteriori informazioni sulle viste unioni modificabili e non modificabili consultare la documentazione Oracle Database Concepts). Se una vista non è modificabile, è possibile scrivere un trigger instead-of su di essa che effettua l’operazione corretta, consentendole di essere modificata. Un trigger instead-of può anche essere scritto su una vista modificabile, se è necessaria ulteriore elaborazione.

La Tabella 10.4 si riferisce a tabelle con chiave preservata. Una tabella è con *chiave preservata* se, dopo un’unione con un’altra tabella, le chiavi nella tabella originale sono anche chiavi nell’unione risultante.

Tabella 10.4 Viste unione modificabili.

OPERAZIONE DML	AMMESSA SE
INSERT	La dichiarazione non fa riferimento, implicitamente o esplicitamente, alle colonne di una tabella con chiave non preservata.
UPDATE	Le colonne aggiornate mappano su colonne di una tabella con chiave preservata.
DELETE	Esiste esattamente una tabella con chiave preservata nell’unione.

Esempio instead-of

Si consideri la vista `books_authors` vista in precedenza nel presente capitolo:

```
-- Disponibile online come parte di of insteadOf1.sql
CREATE OR REPLACE VIEW books_authors AS
  SELECT b.isbn, b.title, a.first_name, a.last_name
     FROM books b, authors a
     WHERE b.author1 = a.id
        OR b.author2 = a.id
        OR b.author3 = a.id;
```

Come si è visto in precedenza, è illegale effettuare un `INSERT` in questa vista. È anche illegale effettuare `UPDATE` o `DELETE` dalla vista. Questo è vero in parte, perché è possibile definire un comportamento diverso per ciascuna operazione DML sulla vista. Si supponga, tuttavia, che abbiano i significati indicati nella Tabella 10.5:

Tabella 10.5 Significati delle operazioni DML sulle viste nel caso in esempio.

OPERAZIONE	SIGNIFICATO
INSERT	Aggiorna la riga contenente il libro, in modo da inserire l'autore fornito. Questo darà come risultato un aggiornamento di <code>author2</code> o <code>author3</code> . Se l'autore non esiste, lo aggiunge alla tabella <code>authors</code> utilizzando prima <code>author_sequence</code> per generare l'ID dell'autore.
UPDATE	Uguale al caso <code>INSERT</code> , ad eccezione del fatto che se l'autore viene cambiato potrebbe essere modificato <code>author1</code> , <code>author2</code> o <code>author3</code> .
DELETE	Aggiorna la riga contenente il libro per eliminare l'autore fornito. Questo potrebbe dare come risultato un aggiornamento in una qualsiasi delle colonne di autore.
Tutte	Per tutte le operazioni, ammette modifiche solo ai campi <code>first_name</code> e <code>last_name</code> di <code>books_authors</code> . Modifiche a <code>isbn</code> o <code>title</code> devono essere effettuate sulla tabella <code>base</code> .

Il trigger `InsteadBooksAuthors`, mostrato di seguito, rispetta le seguenti regole e consente di effettuare operazioni DML correttamente su `books_authors`. Questa è una versione più completa del trigger `InsertBooksAuthors` visto nei paragrafi introduttivi del presente capitolo e contiene anche la gestione di errori. Si noti che una parte della gestione di errori è effettuata dalle limitazioni sulla tabella `books` stessa e non nel trigger.

```
-- Disponibile online come parte di of InsteadBooksAuthors.sql
CREATE OR REPLACE TRIGGER InsteadBooksAuthors
  INSTEAD OF INSERT OR UPDATE OR DELETE ON books_authors
  FOR EACH ROW
DECLARE

  v_Book books%ROWTYPE;
  v_NewAuthorID authors.ID%TYPE;
  v_OldAuthorID authors.ID%TYPE;

-- Funzione locale che restituisce l'ID dei nuovi autori.
-- Se i nomi e i cognomi non esistono in authors
-- viene generato un nuovo ID da author_sequence.
```

```

FUNCTION getID(p_FirstName IN authors.first_name%TYPE,
              p_LastName IN authors.last_name%TYPE)
RETURN authors.ID%TYPE IS
  v_AuthorID authors.ID%TYPE;
BEGIN
  -- Controlla che siano specificati nome e cognome
  IF p_FirstName IS NULL or p_LastName IS NULL THEN
    RAISE_APPLICATION_ERROR(-20004,
      'Both first and last name must be specified');
  END IF;

  -- Usa un blocco annidato per intercettare l'eccezione NO_DATA_FOUND
  BEGIN
    SELECT id
      INTO v_AuthorID
    FROM authors
    WHERE first_name = p_FirstName
      AND last_name = p_LastName;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      -- Nessun autore trovato, ne crea uno nuovo
      INSERT INTO authors (id, first_name, last_name)
        VALUES (author_sequence.NEXTVAL, p_FirstName, p_LastName)
        RETURNING ID INTO v_AuthorID;
  END;

  -- Ora v_AuthorID contiene l'ID corretto e puo' essere restituito.
  RETURN v_AuthorID;
END getID;

-- Funzione locale che restituisce l'identificatore di riga da
-- ISBN o titolo.
FUNCTION getBook(p_ISBN IN books.ISBN%TYPE,
                p_Title IN books.title%TYPE)
RETURN books%ROWTYPE IS

  v_Book books%ROWTYPE;
BEGIN
  -- Controlla che sia fornito almeno uno tra isbn o titolo
  IF p_ISBN IS NULL AND p_Title IS NULL THEN
    RAISE_APPLICATION_ERROR(-20001,
      'Either ISBN or title must be specified');
  ELSIF p_ISBN IS NOT NULL AND p_Title IS NOT NULL THEN
    -- Entrambi specificati, quindi usa titolo e ISBN in query
    SELECT *
      INTO v_Book
    FROM books
    WHERE isbn = p_ISBN
      AND title = p_Title;
  ELSE
    -- Uno solo specificato, quindi usa titolo o ISBN in query
    SELECT *
      INTO v_Book

```

```
        FROM books
        WHERE isbn = p_ISBN
           OR title = p_Title;
    END IF;
    RETURN v_Book;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20002,
            'Could not find book with supplied ISBN/title');
    WHEN TOO_MANY_ROWS THEN
        RAISE_APPLICATION_ERROR(-20003,
            'ISBN/title must match a single book');
    END getBook;

BEGIN /* Inizio corpo trigger principale */
    IF INSERTING THEN
        -- Prende info su libro e autore
        v_Book := getBook(:new.ISBN, :new.title);
        v_NewAuthorID := getID(:new.first_name, :new.last_name);

        -- Controlla che non ci siano doppi autori
        IF v_Book.author1 = v_NewAuthorID OR
           v_Book.author2 = v_NewAuthorID THEN
            RAISE_APPLICATION_ERROR(-20006,
                'Cannot have duplicate authors');
        END IF;

        -- Vede se il libro ha gia' 1 o 2 autori e
        -- aggiorna di conseguenza
        IF v_Book.author2 IS NULL THEN
            UPDATE books
            SET author2 = v_NewAuthorID
            WHERE ISBN = v_Book.ISBN;
        ELSIF v_Book.author3 IS NULL THEN
            UPDATE books
            SET author3 = v_NewAuthorID
            WHERE ISBN = v_Book.ISBN;
        ELSE
            -- Troppi autori, non puo' inserire
            RAISE_APPLICATION_ERROR(-20005,
                v_Book.title || ' already has 3 authors');
        END IF;

    ELSIF UPDATING THEN
        -- Prima controlla che i campi ISBN o title non siano
        -- modificati.
        IF (:new.ISBN != :old.ISBN OR
            :new.title != :old.title) THEN
            RAISE_APPLICATION_ERROR(-20007,
                'Cannot modify ISBN or title in books_authors');
        END IF;
    END IF;
END;
```

```

— Prende info su libro e autore
v_Book := getBook(:new.ISBN, :new.title);
v_NewAuthorID := getID(:new.first_name, :new.last_name);
v_OldAuthorID := getID(:old.first_name, :old.last_name);

-- Vede quale modificare tra author1, author2 o author3
-- e aggiorna di conseguenza
IF v_Book.author1 = v_OldAuthorID THEN
  UPDATE books
    SET author1 = v_NewAuthorID
    WHERE ISBN = v_Book.ISBN;
ELSIF v_Book.author2 = v_OldAuthorID THEN
  UPDATE books
    SET author2 = v_NewAuthorID
    WHERE ISBN = v_Book.ISBN;
ELSE
  UPDATE BOOKS
    SET author3 = v_NewAuthorID
    WHERE ISBN = v_Book.ISBN;
END IF;
ELSE

-- Prende info su libro e autore
v_Book := getBook(:old.ISBN, :old.title);
v_OldAuthorID := getID(:old.first_name, :old.last_name);

-- Vede quale modificare tra author1, author2 o author3
-- e aggiorna di conseguenza. Si noti che se questo risulta
-- nell'eliminazione di tutti gli autori dalla tabella la limitazione
-- NOT NULL su author1 solleva un errore.
IF v_Book.author1 = v_OldAuthorID THEN
  -- Imposta author1 = author2, author2 = author3
  v_Book.Author1 := v_Book.Author2;
  v_Book.Author2 := v_Book.Author3;
ELSIF v_Book.author2 = v_OldAuthorID THEN
  -- Imposta author2 = author 3
  v_Book.Author2 := v_Book.Author3;
ELSE
  -- Elimina author3
  v_Book.Author3 := NULL;
END IF;
UPDATE BOOKS
  SET author1 = v_Book.Author1,
      author2 = v_Book.Author2,
      author3 = v_Book.Author3
  WHERE ISBN = v_Book.ISBN;
END IF;
END InsteadBooksAuthors;
/

```

NOTA *La clausola FOR EACH ROW è opzionale per un trigger instead-of. Tutti i trigger instead-of sono a livello di riga, che sia presente o meno la clausola.*

InsteadBooksAuthors utilizza predicati di trigger per determinare l'operazione DML da effettuare e per intraprendere l'azione adatta. La Figura 10.1 contiene i contenuti originali di books, authors e books _ authors per l'ISBN 72223855, Oracle 9i New Features. Si supponga di emettere quindi la seguente dichiarazione INSERT:

```
-- Disponibile online come parte di of InsteadBooksAuthors.sql
INSERT INTO books_authors(ISBN, title, first_name, last_name)
VALUES ('72223855', 'Oracle 9i New Features', 'Esther', 'Elegant');
```

Il trigger fa in modo che books sia aggiornata in modo da riflettere il nuovo autore e sia inserita una nuova riga in authors (l'ID di autore per Esther Elegant potrebbe essere diverso, secondo il valore di author _ sequence). La Figura 10.2 illustra la situazione che si verifica dopo l'INSERT. Ora, si supponga di emettere la seguente dichiarazione UPDATE:

```
-- Disponibile online come parte di of InsteadBooksAuthors.sql
UPDATE books_authors
SET first_name = 'Rose', last_name = 'Riznit'
WHERE ISBN = '72223855'
AND last_name = 'Elegant';
```

books					authors		
ISBN	Title	Author1	Author2	Author3	ID	First_Name	Last_Name
72223855	Oracle 9i New Features	38			38	Robert	Freeman

books_authors			
ISBN	Title	First_Name	Last_Name
72223855	Oracle 9i New Features	Robert	Freeman

Figura 10.1 Contenuti originali di books, authors e books_authors per ISBN 72223855.

La Figura 10.3 illustra la situazione dopo UPDATE. La tabella books è stata nuovamente aggiornata, ed è stata inserita un'altra riga in authors. Infine, si supponga di emettere la seguente dichiarazione DELETE:

```
— Disponibile online come parte di of InsteadBooksAuthors.sql
DELETE FROM books_authors
WHERE ISBN = '72223855'
AND last_name = 'Riznit';
```

books					authors		
ISBN	Title	Author1	Author2	Author3	ID	First_Name	Last_Name
72223855	Oracle 9i New Features	38	1000		38	Robert	Freeman
					1000	Esther	Elegant

books_authors			
ISBN	Title	First_Name	Last_Name
72223855	Oracle 9i New Features	Esther	Elegant
72223855	Oracle 9i New Features	Robert	Freeman

Figura 10.2 Contenuti dopo INSERT.

Ora la tabella books si trova nuovamente nella sua situazione originale, insieme a books_authors. Ma ci sono ancora le due righe aggiuntive in authors, come mostrato nella Figura 10.4.

books					authors		
ISBN	Title	Author1	Author2	Author3	ID	First_Name	Last_Name
72223855	Oracle 9i New Features	38	1001		38	Robert	Freeman
					1000	Esther	Elegant
					1001	Rose	Riznit

books_authors			
ISBN	Title	First_Name	Last_Name
72223855	Oracle 9i New Features	Rose	Riznit
72223855	Oracle 9i New Features	Robert	Freeman

Figura 10.3 Contenuti dopo UPDATE.

books					authors		
ISBN	Title	Author1	Author2	Author3	ID	First_Name	Last_Name
72223855	Oracle 9i New Features	38			38	Robert	Freeman
					1000	Esther	Elegant
					1001	Rose	Riznit

books_authors			
ISBN	Title	First_Name	Last_Name
72223855	Oracle 9i New Features	Robert	Freeman

Figura 10.4 Contenuti dopo DELETE.

Creazione di trigger di sistema

Come si è visto nei paragrafi precedenti, trigger DML e instead-of vengono attivati su (o invece di) eventi DML, vale a dire dichiarazioni INSERT, UPDATE o DELETE. I trigger di sistema, invece, vengono attivati su due diversi tipi di eventi: DDL o di database. Gli eventi DDL comprendono dichiarazioni CREATE, ALTER o DROP, mentre gli eventi di database comprendono avvio/shutdown del server, logon/logoff di un utente e un errore di server. La sintassi per la creazione di un trigger di sistema è la seguente:

```
CREATE [OR REPLACE] TRIGGER [schema.]nome_trigger
{BEFORE | AFTER}
{elenco_eventi_ddl | elenco_eventi_database}
ON {DATABASE | [schema.]SCHEMA}
[clausola_when]
corpo_trigger;
```

dove *elenco_eventi_ddl* è uno o più eventi DDL (separati dalla parola chiave OR) ed *elenco_eventi_database* è uno o più eventi di database (separati dalla parola chiave OR).

La Tabella 10.6 descrive gli eventi DDL e di database, insieme alle tempistiche ammesse (BEFORE o AFTER). Non è possibile creare un trigger di sistema instead-of.

NOTA Per poter creare un trigger di sistema è necessario avere il privilegio di sistema `ADMINISTER DATABASE TRIGGER`. Per ulteriori informazioni consultare il paragrafo “Privilegi di trigger” più avanti nel capitolo.

Tabella 10.6 Eventi DDL e di database di sistema.

EVENTO	TEMPISTICHE AMMESSE	DESCRIZIONE
STARTUP	AFTER	Attivato quando viene avviata un'istanza.
SHUTDOWN	BEFORE	Attivato quando viene chiusa un'istanza. Questo evento potrebbe non essere attivato se il database viene chiuso in modo non normale (come in un annullamento di shutdown).
SERVERERROR	AFTER	Attivato quando si verifica un errore.
LOGON	AFTER	Attivato dopo che un utente si è collegato con successo al database.
LOGOFF	BEFORE	Attivato all'inizio del logoff di un utente.
CREATE	BEFORE, AFTER	Attivato prima o dopo la creazione di un oggetto schema.
DROP	BEFORE, AFTER	Attivato prima o dopo l'eliminazione di un oggetto schema.
ALTER	BEFORE, AFTER	Attivato prima o dopo la modifica di un oggetto schema.
TRUNCATE	BEFORE, AFTER	Attivato prima o dopo l'emissione di una dichiarazione TRUNCATE.

(segue)

Tabella 10.6 Eventi DDL e di database di sistema. *(continua)*

EVENTO	TEMPISTICHE AMMESSE	DESCRIZIONE
DDL	BEFORE, AFTER	Attivato prima o dopo la maggior parte delle dichiarazioni DDL. Questo evento non verrà attivato per dichiarazioni ALTER DATABASE, CREATE CONTROLFILE o CREATE DATABASE, né verrà attivato per DDL emesse attraverso un'interfaccia procedurale, quale AQ.
ANALYZE	BEFORE, AFTER	Attivato prima o dopo l'emissione di ANALYZE STATEMENT.
ASSOCIATE STATISTICS	BEFORE, AFTER	Attivato prima o dopo l'emissione di una dichiarazione ASSOCIATE STATISTICS.
DISASSOCIATE STATISTICS	BEFORE, AFTER	Attivato prima o dopo l'emissione di una dichiarazione DISASSOCIATE STATISTICS.
AUDIT	BEFORE, AFTER	Attivato prima o dopo l'emissione di una dichiarazione AUDIT.
NOAUDIT	BEFORE, AFTER	Attivato prima o dopo l'emissione di una dichiarazione NOAUDIT.
COMMENT	BEFORE, AFTER	Attivato prima o dopo l'emissione di una dichiarazione COMMENT.
GRANT	BEFORE, AFTER	Attivato prima o dopo l'emissione di una dichiarazione GRANT.
REVOKE	BEFORE, AFTER	Attivato prima o dopo l'emissione di una dichiarazione REVOKE.
RENAME	BEFORE, AFTER	Attivato prima o dopo l'emissione di una dichiarazione RENAME.
SUSPEND*	AFTER	Attivato dopo la sospensione di una dichiarazione SQL a causa di una condizione di mancanza di spazio. In questo caso il trigger può correggere la situazione, quindi la dichiarazione può essere nuovamente emessa.

*Questo evento è disponibile in Oracle9i e successivi.

Trigger di database e di schema

Un trigger di sistema può essere definito a livello del database o a livello di schema. Un trigger a livello di database verrà attivato ogniqualvolta si verifica l'evento di attivazione, mentre un trigger a livello di schema verrà attivato solo quando l'evento di attivazione si verifica per lo schema specificato. Le parole chiave DATABASE e SCHEMA determinano il livello per un dato trigger di sistema. Se lo schema non è specificato con la parola chiave SCHEMA, per impostazione predefinita corrisponde allo schema che possiede il trigger. Per esempio, si supponga di creare il seguente trigger mentre si è connessi come UserA:

NOTA *Questi esempi richiedono l'esistenza di UserA, UserB ed Example nel database. Se non sono presenti, eseguire createUser.sql prima di eseguire DatabaseSchema1.sql. Per ulteriori dettagli consultare DatabaseSchema1.sql. Inoltre è necessario eseguire lo script DatabaseSchema1.sql come utente system o come utente con privilegi di ruolo DBA. Lo script crea l'utente example e concede i necessari privilegi di sistema.*

```
-- Disponibile online come parte di of DatabaseSchema1.sql
CREATE OR REPLACE TRIGGER LogUserAConnects
  AFTER LOGON ON SCHEMA
BEGIN
  INSERT INTO example.temp_table
    VALUES (1, 'LogUserAConnects fired!');
END LogUserAConnects;
/
```

LogUserAConnects registrerà in temp_table ogni connessione di UserA al database. È possibile fare la stessa cosa per UserB creando quanto segue, mentre si è connessi come UserB:

```
-- Disponibile online come parte di of DatabaseSchema.sql
CREATE OR REPLACE TRIGGER LogUserBConnects
  AFTER LOGON ON SCHEMA
BEGIN
  INSERT INTO example.temp_table
    VALUES (2, 'LogUserBConnects fired!');
END LogUserBConnects;
/
```

Infine è possibile creare il seguente trigger mentre si è connessi come example. LogAllConnects registrerà tutte le connessioni al database, perché è un trigger a livello di database.

```
-- Disponibile online come parte di of DatabaseSchema1.sql
CREATE OR REPLACE TRIGGER LogAllConnects
  AFTER LOGON ON DATABASE
BEGIN
  INSERT INTO example.temp_table
    VALUES (3, 'LogAllConnects fired!');
END LogAllConnects;
/
```

Ora è possibile connettersi al database come UserA, UserB ed Example e vedere gli effetti dei diversi trigger. Il trigger after-logout sullo schema si attiva per primo, seguito dal trigger after-logout sul database.

```
-- Disponibile online come parte di of DatabaseSchema1.sql
connect UserA/UserA
connect UserB/UserB
connect example/example
```

Una query SQL*Plus formattata sulla tabella temporanea consente di vedere la sequenza dei trigger attivati.

```
COL num_col FORMAT 9
COL char_col FORMAT A50
SELECT * FROM temp_table;
```

Il trigger di schema UserA è il primo record nella tabella, seguito dal trigger sul database after-logon. La terza e la quarta voce riflettono il comportamento per UserB.

L'utente Example non ha un trigger a livello di schema. La connessione a tale schema attiva solo il trigger after-logon sul database.

```

NUM_COL CHAR_COL
-----
      1 LogUserAConnects fired!
      3 LogAllConnects fired!
      2 LogUserBConnects fired!
      3 LogAllConnects fired!
      3 LogAllConnects fired!

```

LogAllConnects è stato attivato tre volte (una volta per le tre connessioni), mentre LogUserAConnects e LogUserBConnects sono stati attivati una sola volta, come previsto.

NOTA *I trigger STARTUP e SHUTDOWN sono rilevanti solo a livello di database. Non è illegale crearli a livello di schema, ma non verranno attivati.*

Funzioni di attributo di eventi

In un trigger di sistema, sono disponibili numerose funzioni di attributo di eventi. Simili ai predicati di trigger (INSERTING, UPDATING e DELETING), consentono al corpo di un trigger di prendere informazioni sull'evento di attivazione. Sebbene sia legale chiamare tali funzioni da altri blocchi PL/SQL (non necessariamente nel corpo di un trigger di sistema), non restituiranno sempre un risultato valido. Le funzioni di attributo di eventi sono descritte nella Tabella 10.7.

Il trigger LogCreations, visto all'inizio del presente capitolo, utilizza alcune funzioni di attributo. A differenza dei predicati di trigger, le funzioni di attributo di eventi sono funzioni PL/SQL standalone, con sinonimi pubblici definiti, e cominciano con ORA_.

Prima di eseguire il presente paragrafo, ci si deve connettere allo schema di esempio.

```

-- Disponibile online come parte di of LogCreations.sql
CREATE OR REPLACE TRIGGER LogCreations
  AFTER CREATE ON SCHEMA
BEGIN
  INSERT INTO ddl_creations (user_id, object_type, object_name,
                           object_owner, creation_date)
  VALUES (USER, ORA_DICT_OBJ_TYPE, ORA_DICT_OBJ_NAME,
          ORA_DICT_OBJ_OWNER, SYSDATE);
END LogCreations;
/

```

SUGGERIMENTO *Prima di Oracle8i, le funzioni di attributo di eventi, oltre ad avere nomi diversi, erano possedute da SYS e non avevano sinonimi definiti. Di conseguenza, dovevano avere il prefisso SYS per poter essere risolte. Sebbene questa sintassi sia ancora legale, si dovrebbe utilizzare la sintassi corrente.*

Utilizzare il comando di descrizione SQL*Plus per vedere elenco di argomenti, tipo e modalità di ORA _ DICT _ OBJ _ NAME _ LIST.

```
FUNCTION ora_dict_obj_name_list RETURNS BINARY_INTEGER
Argument Name                Type                In/Out Default?
-----
OBJECT_LIST                  DBMS_STANDARD      OUT
/
```

Alcune funzioni di attributo (quali ORA _ DICT _ OBJ _ NAME _ LIST) hanno parametri OUT di tipo ORA _ NAME _ LIST _ T. Tale tipo è definito come parte del package STANDARD come di seguito:

```
TYPE ORA_NAME_LIST_T IS TABLE OF VARCHAR2(64);
```

I parametri OUT sono descritti nella Tabella 10.7.

Tabella 10.7 Funzioni di attributo di eventi.

FUNZIONE ATTRIBUTO	TIPO RESTITUITO	EVENTI DI SISTEMA PER CUI E' APPLICABILE	DESCRIZIONE
ORA_CLIENT_IP_ADDRESS client	VARCHAR2	LOGON	Restituisce l'indirizzo IP del per un logon di database. Se il protocollo non è TCP/IP, questa funzione non è valida.
ORA_DATABASE_NAME	VARCHAR2(50)	Tutti gli eventi	Restituisce il nome del database.
ORA_DES_ENCRYPTED_ PASSWORD	VARCHAR2	ALTER	Per eventi ALTER USER, restituisce la password decifrata dell'utente.
ORA_DICT_OBJ_NAME	VARCHAR2(30)	ALTER, ANALYZE, ASSOCIATE STATISTICS, COMMENT, CREATE, DDL, DISASSOCIATE STATISTICS, DROP, GRANT, RENAME, REVOKE, TRUNCATE	Restituisce il nome dell'oggetto dizionario su cui si è verificata un'operazione DDL.
ORA_DICT_OBJ_NAME_LIST (elenco_nomi OUT ORA_NAME_LIST_T)	BINARY_INTEGER	ASSOCIATE STATISTICS, DISASSOCIATE STATISTICS	elenco_nomi conterrà un elenco di nomi di oggetti modificati dall'evento. Il valore restituito è la dimensione dell'array.
ORA_DICT_OBJ_OWNER	VARCHAR2(30)	ALTER, ANALYZE, ASSOCIATE STATISTICS, COMMENT, CREATE, DDL, DISASSOCIATE STATISTICS, DROP, GRANT, RENAME, REVOKE, TRUNCATE	Restituisce il possessore dell'oggetto dizionario su cui si è verificata un'operazione DDL.

(segue)

Tabella 10.7 Funzioni di attributo di eventi. (continua)

FUNZIONE ATTRIBUTO	TIPO RESTITUITO	EVENTI DI SISTEMA PER CUI E' APPLICABILE	DESCRIZIONE
ORA_DICT_OBJ_OWNER_LIST(elenco_nomi OUT ORA_NAME_LIST_T)	BINARY_INTEGER	ASSOCIATE STATISTICS, DISASSOCIATE STATISTICS	elenco_nomi conterrà un elenco dei possessori di oggetti modificati dall'evento. Il valore restituito è la dimensione dell'array.
ORA_DICT_OBJ_TYPE	VARCHAR2(20)	ALTER, ANALYZE, ASSOCIATE STATISTICS, COMMENT, CREATE, DDL, DISASSOCIATE STATISTICS, DROP, GRANT, RENAME, REVOKE, TRUNCATE	Restituisce il tipo di oggetto di dizionario su cui si è verificata un'operazione DDL.
ORA_GRANTEE(elenco_utenti OUT ORA_NAME_LIST_T)	BINARY_INTEGER	GRANT	elenco_utenti conterrà i concessionari per una dichiarazione GRANT. Il valore restituito è la dimensione dell'array.
ORA_INSTANCE_NUM	NUMBER	Tutti gli eventi	Restituisce il numero di istanza.
ORA_IS_ALTER_COLUMN (nome_colonna IN VARCHAR2)	BOOLEAN	ALTER	Per eventi ALTER TABLE, restituisce true se viene alterato nome_colonna.
ORA_IS_CREATING_NESTED_TABLE	BOOLEAN	CREATE	Restituisce true se l'evento corrente sta creando una tabella annidata.
ORA_IS_DROP_COLUMN (nome_colonna IN VARCHAR2)	BOOLEAN	DROP	Restituisce true se viene eliminata nome_colonna.
ORA_IS_SERVERERROR (num_errore IN BINARY_INTEGER)	BOOLEAN	SERVERERROR, SUSPEND	Restituisce true se num_errore è nello stack di errori.
ORA_LOGIN_USER	VARCHAR2(30)	Tutti gli eventi	Restituisce il nome di utente di login.
ORA_PARTITION_POS*	BINARY_INTEGER	CREATE	Per una dichiarazione CREATE TABLE, restituisce la posizione nel testo dove può essere inserita una clausola PARTITION.
ORA_PRIVILEGE_LIST(elenco_privilegi OUT ORA_NAME_LIST_T)	BINARY_INTEGER	GRANT, REVOKE	elenco_privilegi conterrà i privilegi concessi o revocati. Il valore restituito è la dimensione dell'array.
ORA_REVOKEE(elenco_utenti OUT ORA_NAME_LIST_T)	BINARY_INTEGER	REVOKE	elenco_utenti conterrà i revocanti per una dichiarazione REVOKE. Il valore restituito è la dimensione dell'array.
ORA_SERVER_ERROR(posizione IN BINARY_INTEGER)	NUMBER	SERVERERROR	Restituisce il numero di errore nella posizione data nello stack di errori. La posizione in cima allo stack è la posizione 1.
ORA_SERVER_ERROR_DEPTH*	BINARY_INTEGER	SERVERERROR	Restituisce il numero totale di errori nello stack di errori.
ORA_SERVER_ERROR_MSG (posizione IN BINARY_INTEGER)*	VARCHAR2	SERVERERROR	Restituisce il messaggio di errore nella posizione data nello stack di errori. La posizione in cima allo stack è la posizione 1.

(segue)

Tabella 10.7 Funzioni di attributo di eventi. *(continua)*

FUNZIONE ATTRIBUTO	TIPO RESTITUITO	EVENTI DI SISTEMA PER CUI E' APPLICABILE	DESCRIZIONE
ORA_SERVER_ERROR_NUM_PARAMS(posizione IN BINARY_INTEGER)*	BINARY_INTEGER	SERVERERROR	Restituisce il numero di parametri per il messaggio di errore nella posizione data. Un parametro viene inserito in un messaggio di errore utilizzando nel testo del messaggio di errore un formato stringa come "%s" o "%d". La posizione in cima allo stack è la posizione 1.
ORA_SERVER_ERROR_PARAM(posizione IN BINARY_INTEGER), param IN BINARY_INTEGER)*	VARCHAR2	SERVERERROR	Restituisce il valore sostituito per il parametro dato (1 è il primo parametro) nella posizione data nello stack di errori. La posizione in cima allo stack è la posizione 1.
ORA_SQL_TEXT(testo_sql OUT ORA_NAME_LIST_T*)	BINARY_INTEGER	Tutti gli eventi	Restituisce il testo della dichiarazione attivante. Se la dichiarazione è lunga, viene suddivisa in più elementi, con il valore restituito che specifica la dimensione dell'array.
ORA_SYSEVENT	VARCHAR2	Tutti gli eventi	Nome dell'evento di sistema che attiva il trigger.
ORA_WITH_GRANT_OPTION	BOOLEAN	GRANT	Restituisce true se i privilegi sono concessi con l'opzione grant.
SPACE_ERROR_INFO(numero_errore OUT NUMBER, tipo_errore OUT VARCHAR2, possessore_oggetto OUT VARCHAR2, nome_spazio_tabella OUT VARCHAR2, nome_oggetto OUT VARCHAR2, nome_sottooggetto OUT VARCHAR2)*	BOOLEAN	SERVERERROR, SUSPEND	Restituisce true se l'errore è relativo a una condizione di mancanza di spazio e i parametri sono compilati con informazioni sull'oggetto che ha causato l'errore.

*Questa funzione è disponibile in Oracle9iR1 e successivi.

Trigger di sistema e transazioni

Secondo l'evento di attivazione, cambia il comportamento transazionale di un trigger di sistema. Un trigger di sistema viene attivato come una transazione separata confermata al completamento con successo del trigger, oppure viene attivato come parte della transazione dell'utente corrente. I trigger STARTUP, SHUTDOWN, SERVERERROR e LOGON si attivano tutti come transazioni separate, mentre i trigger LOGOFF e DDL vengono attivati come parte della transazione corrente.

È importante notare, tuttavia, che il lavoro svolto dal trigger in genere verrà confermato in ogni caso. Nel caso di un trigger DDL, la transazione corrente (vale a dire la dichiarazione CREATE, ALTER o DROP) viene confermata automaticamente, confermando il lavoro nel trigger. Anche il lavoro in un trigger LOGOFF verrà confermato come parte della transazione finale nella sessione.

NOTA *Poiché in genere i trigger di sistema vengono confermati in ogni caso, la loro dichiarazione come autonomi non avrà alcun effetto.*

Trigger di sistema e clausola WHEN

Come i trigger DML, i trigger di sistema possono utilizzare la clausola WHEN per specificare una condizione sull'attivazione del trigger. Tuttavia esistono limitazioni sui tipi di condizioni che possono essere specificati per ogni tipo di trigger di sistema, vale a dire:

- i trigger STARTUP e SHUTDOWN non possono avere alcuna condizione;
- i trigger SERVERERROR possono utilizzare il test ERRNO per controllare solo un errore specifico;
- i trigger LOGON e LOGOFF possono controllare ID o nome utente con i test USERID o USERNAME;
- i trigger DDL possono controllare il tipo e il nome dell'oggetto da modificare e possono controllare ID o nome utente.

Altre questioni sui trigger

Nel presente paragrafo si vedranno altre questioni sui trigger. Tra queste il namespace per i nomi di trigger, diverse limitazioni sull'utilizzo dei trigger e diversi tipi di corpi di trigger. Il paragrafo si conclude con una discussione sui privilegi relativi ai trigger.

Nomi di trigger

Il namespace per i nomi di trigger è diverso da quello di altri sottoprogrammi. Un *namespace* è l'insieme di identificatori legali disponibili per l'uso come nomi di un oggetto. Procedure, package e tabelle condividono il medesimo namespace. Questo significa che, in uno schema di database, tutti gli oggetti nello stesso namespace devono avere nomi unici. Per esempio, è illegale dare lo stesso nome a una procedura e a un package.

I trigger, tuttavia, si trovano in un namespace separato. Questo significa che un trigger può avere lo stesso nome di una tabella o di una procedura. In uno schema, tuttavia, un dato nome può essere utilizzato per un solo trigger. Per esempio, è possibile creare un trigger chiamato *inventory* sulla tabella *inventory*, ma è illegale creare anche una procedura chiamata *inventory*. Quanto di seguito deve essere testato nell'account *UserA* a causa della dipendenza dalla tabella *inventory* creata in esso. In alternativa, per provarlo, è possibile eseguire lo script *tables.sql* in un altro schema.

```
-- Disponibile online come samename.sql
CREATE OR REPLACE TRIGGER inventory
  BEFORE INSERT ON inventory
BEGIN
  INSERT INTO temp_table (char_col)
```

```
VALUES ('Trigger fired!');
END inventory;
/
```

Se si cerca di creare una procedura chiamata `inventory` dopo la creazione del trigger con lo stesso nome, l'operazione fallirà perché cerca di occupare lo stesso namespace.

```
-- Disponibile online come samename.sql
CREATE OR REPLACE PROCEDURE inventory AS
BEGIN
  INSERT INTO temp_table (char_col)
  VALUES ('Procedure called!');
END inventory;
/
```

Il tentativo di creare la procedura solleverà il seguente errore:

```
CREATE OR REPLACE PROCEDURE inventory AS
*
ERROR at line 1:
ORA-00955: name is already used by an existing object
```

SUGGERIMENTO *Sebbene sia possibile utilizzare lo stesso nome per un trigger e una tabella, è sconsigliato: è meglio dare a ciascun trigger un nome unico che identifichi la sua funzione, oltre alla tabella su cui viene definito, o di anteporre ai trigger una sequenza di caratteri comune (quale TRG_).*

Limitazioni sui trigger

Il corpo di un trigger è un blocco PL/SQL o una dichiarazione CALL (per particolari sull'utilizzo di CALL consultare il paragrafo successivo). Qualsiasi dichiarazione legale in un blocco PL/SQL è legale in un corpo di trigger, conformemente alle seguenti limitazioni:

- un trigger può non emettere alcuna dichiarazione di controllo di transazione, COMMIT, ROLLBACK, SAVEPOINT o SET TRANSACTION. Il compilatore PL/SQL ammetterà la creazione di un trigger che contenga una di queste dichiarazioni, ma si riceverà un errore quando il trigger viene attivato. Questo perché è attivato come parte dell'esecuzione della dichiarazione di attivazione e si trova nella stessa transazione della dichiarazione di attivazione. Quando la dichiarazione di attivazione viene confermata o annullata, anche il lavoro nel trigger viene confermato o annullato (è possibile creare un trigger che venga eseguito come transazione autonoma; in tal caso il lavoro nel trigger può essere confermato o annullato indipendentemente dalla dichiarazione di attivazione. Per ulteriori informazioni sulle transazioni anonime consultare il Capitolo 4);

- analogamente, qualsiasi procedura o funzione chiamata dal corpo del trigger non può emettere alcuna dichiarazione di controllo di transazione (a meno che anch'esse siano dichiarate come autonome);
- il corpo del trigger non può dichiarare alcuna variabile LONG o LONG RAW. Inoltre :new e :old non possono fare riferimento a una colonna LONG o LONG RAW nella tabella per cui è definito il trigger;
- il codice nel corpo di un trigger può fare riferimento a colonne LOB (Large Object) e utilizzarle, ma potrebbe non modificare i valori delle colonne. Questo vale anche per colonne oggetto.

Esistono anche limitazioni sulle tabelle a cui può accedere il corpo di un trigger. Secondo il tipo di trigger e le limitazioni sulle tabelle, le tabelle possono essere mutanti. Questa situazione è descritta in dettaglio nel paragrafo “Tabelle mutanti” più avanti nel capitolo.

Corpi di trigger

Prima di Oracle8i, i corpi di trigger dovevano essere blocchi PL/SQL. In Oracle8i e successivi, tuttavia, un corpo di trigger può essere costituito invece da una dichiarazione CALL. La procedura chiamata può essere un sottoprogramma memorizzato PL/SQL o un wrapper per una routine C o Java. Questo consente di creare trigger in cui il codice funzionale è scritto in Java. Per esempio, si supponga di voler registrare le connessioni e le sconnessioni al database, nella seguente tabella che si trova nello schema UserA:

```
-- Disponibile online come parte di of tables.sql
CREATE TABLE connect_audit (
  user_name VARCHAR2(30),
  operation VARCHAR2(30),
  timestamp DATE);
```

Per registrare connessioni e sconnessioni è possibile utilizzare il seguente package:

```
-- Disponibile online come LogPkg1.sql
CREATE OR REPLACE PACKAGE LogPkg AS
  PROCEDURE LogConnect(p_UserID IN VARCHAR2);
  PROCEDURE LogDisconnect(p_UserID IN VARCHAR2);
END LogPkg;
/

CREATE OR REPLACE PACKAGE BODY LogPkg AS
  PROCEDURE LogConnect(p_UserID IN VARCHAR2) IS
  BEGIN
    INSERT INTO connect_audit (user_name, operation, timestamp)
      VALUES (p_UserID, 'CONNECT', SYSDATE);
  END LogConnect;
```

```

PROCEDURE LogDisconnect(p_UserID IN VARCHAR2) IS
BEGIN
    INSERT INTO connect_audit (user_name, operation, timestamp)
        VALUES (p_UserID, 'DISCONNECT', SYSDATE);
END LogDisconnect;
END LogPkg;
/

```

LogPkg.LogConnect e LogPkg.LogDisconnect prendono come argomento un nome utente e inseriscono una riga in connect_audit. Infine è possibile chiamarle da trigger LOGON e LOGOFF, come di seguito:

```

-- Disponibile online come LogConnects.sql
CREATE OR REPLACE TRIGGER LogConnects
AFTER LOGON ON DATABASE
CALL LogPkg.LogConnect(SYS.LOGIN_USER)
/

CREATE OR REPLACE TRIGGER LogDisconnects
BEFORE LOGOFF ON DATABASE
CALL LogPkg.LogDisconnect(SYS.LOGIN_USER)
/

```

NOTA *Poiché LogConnects e LogDisconnects sono trigger di sistema sul database (e non su uno schema), per crearle è necessario avere il privilegio di sistema ADMINISTER DATABASE TRIGGER.*

Il corpo di trigger per LogConnects e LogDisconnects è semplicemente una dichiarazione CALL, che indica la procedura da eseguire. Come unico argomento viene passato l'utente corrente. Nell'esempio precedente, la destinazione di CALL è una procedura di package PL/SQL standard, ma potrebbe essere un wrapper per una routine esterna C o Java. Per esempio, si supponga di caricare la seguente classe Java nel database e provarla.

Prima di cercare di caricare il programma Java nel database, è necessario controllare di avere l'ambiente impostato correttamente. È necessario controllare che sia impostata la variabile di ambiente CLASSPATH. Questa operazione viene effettuata in modo leggermente diverso in Windows e Unix; sono indicate le sintassi per entrambi i sistemi. Se l'archivio Java classes12.zip si trova nel CLASSPATH, potrebbe non essere necessario impostarlo.

UNIX

```
# echo $CLASSPATH
```

Windows

```
C:> echo %CLASSPATH%
```

Se la variabile di ambiente CLASSPATH non contiene classes12.zip e un riferimento alla directory di lavoro presente, è necessario aggiungerli al classpath.

Gli archivi Java possono avere diversi tipi di estensioni, ma i più diffusi sono *.jar e *.zip. Quando li si inserisce nella variabile di ambiente CLASSPATH, è necessario trattarli come directory. Sarà necessario avere la directory di lavoro presente quando si esegue un comando loadjava sul database. Se esiste una variabile CLASSPATH, deve essere inserita prima del file classes12.zip.

UNIX

```
# export set CLASSPATH=$ORACLE_HOME/jdbc/lib/classes12.zip:.
```

Windows

```
C:> set CLASSPATH=%ORACLE_HOME%/jdbc/lib/classes12.zip:.
```

Copiare il file Logger.java che segue dal sito Web o dal tipo nella propria directory di lavoro e compilare il file. La sintassi è la stessa in Unix e Windows. Questo creerà un programma client Thick Java che deve essere eseguito dal server in cui si trova il database Oracle. Questa limitazione è dovuta alle dipendenze di libreria esterne all'implementazione Oracle JDBC.

```
javac Logger.java
```

Questo genererà un byte file Java, chiamato Logger.class. Verrà caricato nel database con l'utility loadjava e la sintassi di seguito. Questo caricherà il bytecode Java nel database per lo schema Example.

```
loadjava -r -f -o -user example/example Logger.class
```

// Disponibile online come Logger.java

```
import java.sql.*;
import oracle.jdbc.driver.*;

public class Logger {
    public static void LogConnect(String userID)
        throws SQLException {
        // Prende connessione JDBC predefinita
        Connection conn = new OracleDriver().defaultConnection();

        String insertString = "INSERT INTO connect_audit " +
            "(user_name, operation, timestamp) " +
            "VALUES (?, 'CONNECT', SYSDATE)";

        // Prepara ed esegue una dichiarazione che fa l'inserimento
        PreparedStatement insertStatement =
            conn.prepareStatement(insertString);
        insertStatement.setString(1, userID);
        insertStatement.execute();
    }

    public static void LogDisconnect(String userID)
        throws SQLException {
```

```

// Prende connessione JDBC predefinita
Connection conn = new OracleDriver().defaultConnection();

String insertString =
    "INSERT INTO connect_audit (user_name, operation, timestamp)" +
    " VALUES (?, 'DISCONNECT', SYSDATE)";

// Prepara ed esegue una dichiarazione che fa l'inserimento
PreparedStatement insertStatement =
    conn.prepareStatement(insertString);
insertStatement.setString(1, userID);
insertStatement.execute();
}
}

```

Si crea il package PL/SQL LogPkg come wrapper per la classe Java creata.

```

-- Disponibile online come LogPkg2.sql
CREATE OR REPLACE PACKAGE LogPkg AS
    PROCEDURE LogConnect(p_UserID IN VARCHAR2);
    PROCEDURE LogDisconnect(p_UserID IN VARCHAR2);
END LogPkg;
/

CREATE OR REPLACE PACKAGE BODY LogPkg AS
    PROCEDURE LogConnect(p_UserID IN VARCHAR2) IS
        LANGUAGE JAVA
        NAME 'Logger.LogConnect(java.lang.String)';

    PROCEDURE LogDisconnect(p_UserID IN VARCHAR2) IS
        LANGUAGE JAVA
        NAME 'Logger.LogDisconnect(java.lang.String)';
END LogPkg;
/

```

Prima di provare i wrapper, è necessario creare una copia della tabella connect_audit nello schema Example. Se questa operazione non viene effettuata, quando si cerca di provare i wrapper PL/SQL si ottengono un errore Java non intercettato e un errore ORA-00942.

```

-- Disponibile online come parte di of createConnectAudit.sql
CREATE TABLE connect_audit (
    user_name VARCHAR2(30),
    operation VARCHAR2(30),
    timestamp DATE);

```

È possibile creare un programma PL/SQL con blocco anonimo per provare la connessione e la sconnessione. Sono definite come segue:

— Disponibile online come testLogPkg.sql

```
DECLARE
  v_string VARCHAR2(80) := 'USERA';
BEGIN
  logpkg.logconnect(v_string);
END;
/
```

```
DECLARE
  v_string VARCHAR2(80) := 'USERA';
BEGIN
  logpkg.logdisconnect(v_string);
END;
/
```

Si possono vedere i risultati del test interrogando la tabella connect_audit. Analogamente, è possibile utilizzare gli stessi trigger per ottenere l'effetto desiderato. Per ulteriori informazioni sulle routine esterne consultare il Capitolo 12.

NOTA *I predicati di trigger quali INSERTING, UPDATING e DELETING, e gli identificatori di correlazione :old e :new (e :parent), possono essere utilizzati solo se il corpo del trigger è un blocco PL/SQL completo e non una dichiarazione CALL.*

Privilegi di trigger

Esistono cinque privilegi di sistema che si applicano ai trigger, descritti nella Tabella 10.8. Oltre a questi, il possessore di un trigger deve avere i privilegi di oggetto necessari sugli oggetti a cui fa riferimento il trigger. Poiché un trigger è un oggetto compilato, tali privilegi devono essere concessi direttamente e non attraverso un ruolo (i trigger sono definiti solo con diritti di definitore).

Trigger e dizionario di dati

Simili ai sottoprogrammi memorizzati, alcune viste di dizionario di dati contengono informazioni sui trigger e sul loro stato. Tali viste vengono aggiornate ogniqualvolta un trigger viene creato o eliminato.

Tabella 10.8 Privilegi di sistema relativi ai trigger.

PRIVILEGIO DI SISTEMA	DESCRIZIONE
CREATE TRIGGER	Consente al concessionario di creare un trigger nel proprio schema.
CREATE ANY TRIGGER	Consente al concessionario di creare trigger in qualsiasi schema ad eccezione di SYS. È sconsigliato creare trigger su tabelle di dizionari di dati.
ALTER ANY TRIGGER	Consente al concessionario di abilitare, disabilitare o compilare trigger di database in qualsiasi schema ad eccezione di SYS. Si noti che se il concessionario non ha CREATE ANY TRIGGER, non può modificare il codice del trigger.

(segue)

Tabella 10.8 Privilegi di sistema relativi ai trigger. *(continua)*

PRIVILEGIO DI SISTEMA	DESCRIZIONE
DROP ANY TRIGGER	Consente al concessionario di eliminare trigger di database in qualsiasi schema ad eccezione di SYS.
ADMINISTER DATABASE TRIGGER	Consente al concessionario di creare o modificare un trigger di sistema sul database (al contrario dello schema corrente). Il concessionario deve anche avere CREATE TRIGGER o CREATE ANY TRIGGER.

Viste di dizionari di dati

Quando un trigger viene creato, il suo codice sorgente è memorizzato nella vista di dizionario di dati `user_triggers`. Questa vista comprende il corpo del trigger, la clausola `WHEN`, la tabella attivante e il tipo di trigger. Per esempio, la seguente query formattata restituisce informazioni su `UpdateMajorStats` dopo l'esecuzione dello script `GenerateAuthorID.sql` nello schema `UserA`:

```
COL table_name FORMAT A10
COL triggering_event FORMAT A20
SELECT trigger_type, table_name, triggering_event
FROM user_triggers
WHERE trigger_name = 'GENERATEAUTHORID';
```

Nell'output della query si vedranno il tipo di trigger, il nome di tabella e l'evento attivante.

```
TRIGGER_TYPE    TABLE_NAME    TRIGGERING_EVENT
-----
BEFORE EACH ROW AUTHORS      INSERT OR UPDATE
```

La vista `user_triggers` contiene informazioni sui trigger posseduti dall'utente corrente. Esistono anche altre due viste: `all_triggers` contiene informazioni sui trigger accessibili all'utente corrente (ma che potrebbero essere posseduti da un utente diverso) e `dba_triggers` contiene informazioni su tutti i trigger nel database.

Eliminazione e disabilitazione di trigger

Come procedure e package, i trigger possono essere eliminati, con la sintassi indicata di seguito:

```
DROP TRIGGER nometrigger;
```

dove *nometrigger* è il nome del trigger da eliminare. Questo elimina definitivamente il trigger dal dizionario di dati. Come nei sottoprogrammi, può essere specificata la clausola `OR REPLACE` nella dichiarazione `CREATE` per il trigger. In tal caso il trigger viene prima eliminato, se esiste.

A differenza di procedure e package, tuttavia, un trigger può essere disabilitato senza essere eliminato. Quando un trigger viene disabilitato, esiste

ancora nel dizionario di dati, ma non viene mai attivato. Per disabilitare un trigger, utilizzare la dichiarazione ALTER TRIGGER:

```
ALTER TRIGGER nometrigger {DISABLE | ENABLE};
```

dove *nometrigger* è il nome del trigger. Per impostazione predefinita tutti i trigger sono abilitati al momento della loro creazione. ALTER TRIGGER può disabilitare e quindi abilitare nuovamente qualsiasi trigger. Per esempio, il seguente codice disabilita e quindi abilita nuovamente UpdateMajorStats:

```
ALTER TRIGGER GenerateAuthorID DISABLE;
```

```
ALTER TRIGGER GenerateAuthorID ENABLE;
```

Tutti i trigger per una particolare tabella possono essere abilitati o disabilitati utilizzando il comando ALTER TABLE, aggiungendo la clausola ENABLE ALL TRIGGERS o DISABLE ALL TRIGGERS. Per esempio:

```
ALTER TABLE authors ENABLE ALL TRIGGERS;
```

```
ALTER TABLE authors DISABLE ALL TRIGGERS;
```

La colonna status di user_triggers contiene ENABLED o DISABLED, a indicare lo stato corrente di un trigger. La disabilitazione di un trigger non lo elimina dal dizionario di dati. Per controllare lo stato è possibile utilizzare la seguente query:

```
SELECT nome_trigger, status
FROM user_triggers
WHERE trigger_name = 'trigger_name';
```

P-code di trigger

Quando un package o un sottoprogramma viene memorizzato nel dizionario di dati, oltre al codice sorgente per l'oggetto viene memorizzato il p-code compilato. Questo vale anche per i trigger, quindi i trigger possono essere chiamati senza ricompilazione, e vengono memorizzate le informazioni di dipendenza. Di conseguenza possono essere automaticamente invalidati allo stesso modo di package e sottoprogrammi. Quando un trigger viene invalidato, sarà ricompilato alla nuova attivazione.

10.3 Tabelle mutanti

Esistono limitazioni su tabelle e colonne a cui può accedere il corpo di un trigger. Per definire tali limitazioni, è necessario comprendere le tabelle mutanti e limitanti. Una *tabella mutante* è una tabella che viene modificata da una

dichiarazione DML. Per un trigger, questa è la tabella su cui viene definito il trigger. Anche le tabelle che possono avere necessità di essere aggiornate come risultato di una limitazione di integrità referenziale DELETE CASCADE sono mutanti. Una tabella limitante è una tabella che può avere necessità di essere letta da una limitazione di integrità referenziale. Per illustrare queste definizioni, si considerino le tabelle `students`, `classes` e `registered_students`. Le tabelle `students` e `classes` non hanno dipendenze, ma la tabella `registered_students` ha due dipendenze di chiave esterna. Una dipendenza è sulla chiave primaria della tabella `students`, l'altra sulla chiave primaria della tabella `classes`. Questo garantisce integrità referenziale a livello di database, ma porta un sovraccarico di elaborazione. È possibile eseguire tutti gli script di esempio utilizzando lo script `createObjects.sql`.

-- Disponibile online come parte di `of createStudents.sql`

```
CREATE TABLE students (
  id                NUMBER(5)    NOT NULL,
  current_credits   NUMBER(2),
  major             VARCHAR2(20),
  last_name         VARCHAR2(20) NOT NULL,
  first_name        VARCHAR2(20) NOT NULL,
  middle_initial    VARCHAR2(1)  NOT NULL,
  CONSTRAINT students_pk PRIMARY KEY (id));
```

-- Disponibile online come parte di `of createClasses.sql`

```
CREATE TABLE classes (
  department        CHAR(3)      NOT NULL,
  course            NUMBER(3)    NOT NULL,
  current_students  NUMBER(3)    NOT NULL,
  num_credits       NUMBER(1)    NOT NULL,
  name              VARCHAR2(30) NOT NULL,
  CONSTRAINT classes_pk PRIMARY KEY (department,course));
```

-- Disponibile online come parte di `of createRegisteredStudents.sql`

```
CREATE TABLE registered_students (
  student_id        NUMBER(5)    NOT NULL,
  department         CHAR(3)      NOT NULL,
  course            NUMBER(3)    NOT NULL,
  grade             CHAR(1),
  CONSTRAINT rs_grade CHECK (grade IN ('A', 'B', 'C', 'D', 'F')),
  CONSTRAINT rs_student_id FOREIGN KEY (student_id) REFERENCES students (id),
  CONSTRAINT rs_department_course FOREIGN KEY (department, course) REFERENCES classes (department, course));
```

`registered_students` ha due limitazioni di integrità referenziale dichiarative. Come tale, sia `students` sia `classes` sono tabelle limitanti per `registered_students`. A causa delle limitazioni, `classes` e `students` possono avere necessità di essere modificate e/o interrogate dalla dichiarazione DML.

Inoltre `registered_students` stessa è mutante durante l'esecuzione di una dichiarazione DML su di essa.

Le dichiarazioni SQL in un corpo di trigger non possono:

- leggere o modificare alcuna tabella mutante della dichiarazione di attivazione. Questo comprende la tabella attivante stessa;
- leggere da o modificare le colonne di chiave primaria, unica o esterna di una tabella limitante della tabella attivante. Tuttavia possono modificare le altre colonne.

Tali limitazioni valgono per tutti i trigger a livello di riga; valgono per trigger di dichiarazione solo quando il trigger di dichiarazione sarebbe attivato come risultato di un'operazione DELETE CASCADE.

NOTA *Se una dichiarazione INSERT interessa solo una riga, i trigger before e after riga per tale riga non trattano la tabella attivante come mutante. Questo è l'unico caso in cui un trigger a livello di riga può leggere o modificare la tabella attivante. Dichiarazioni quali INSERT INTO table SELECT ... trattano sempre la tabella attivante come mutante, anche se la sottoquery restituisce una sola riga.*

Come esempio, si consideri il trigger `CascadeRSInserts`, mostrato di seguito. Anche se modifica `students` e `classes`, è legale perché le colonne in `students` e `classes` modificate non sono colonne chiave. Nel prossimo paragrafo si esaminerà un trigger illegale.

```
-- Disponibile online come cascadeRSInsert.sql
CREATE OR REPLACE TRIGGER CascadeRSInserts
/* Mantiene le tabelle registered_students, students e classes
   sincronizzate quando viene fatto un INSERT in registered_students. */
BEFORE INSERT ON registered_students
FOR EACH ROW
DECLARE
  v_Credits classes.num_credits%TYPE;
BEGIN
  -- Determina il numero di crediti per questa classe.
  SELECT num_credits
  INTO v_Credits
  FROM classes
  WHERE department = :new.department
  AND course = :new.course;

  -- Modifica i crediti correnti per questo studente.
  UPDATE students
  SET current_credits = current_credits + v_Credits
  WHERE ID = :new.student_id;

  -- Aggiunge uno al numero di studenti nella classe.
  UPDATE classes
```

```

    SET current_students = current_students + 1
    WHERE department = :new.department
    AND course = :new.course;
END CascadeRSInserts;
/

```

Esempio di tabella mutante

Si supponga di voler limitare a cinque il numero di studenti in ciascuna specializzazione. Si potrebbe fare con un trigger a livello di riga before INSERT o UPDATE su students, indicato di seguito:

```

-- Disponibile online come parte di of limitMajors.sql
CREATE OR REPLACE TRIGGER LimitMajors
/* Limita a 5 il numero di studenti in ciascuna specializzazione.
   Se tale limite viene superato, viene sollevato un errore con
   raise_application_error. */
BEFORE INSERT OR UPDATE OF major ON students
FOR EACH ROW
DECLARE
    v_MaxStudents CONSTANT NUMBER := 5;
    v_CurrentStudents NUMBER;
BEGIN
    -- Determina il numero corrente di studenti in questa
    -- specializzazione.
    SELECT COUNT(*)
    INTO v_CurrentStudents
    FROM students
    WHERE major = :new.major;

    -- Se non c'è spazio, solleva un errore.
    IF v_CurrentStudents + 1 > v_MaxStudents THEN
        RAISE_APPLICATION_ERROR(-20000,
            'Too many students in major ' || :new.major);
    END IF;
END LimitMajors;
/

```

A un primo sguardo, questo trigger sembra ottenere il risultato desiderato. Tuttavia, se si cerca di aggiornare students, attiverà il trigger LimitMajor. Sarà necessario popolare le tabelle con dati prima di provare la dichiarazione di aggiornamento. Questo può essere fatto eseguendo insertAcademicRecords.sql, oppure eseguendo nuovamente createObjects.sql.

```

-- Disponibile online come parte di of limitMajors.sql
UPDATE students
SET major = 'History'
WHERE id = 1;

```

Il trigger LimitMajor solleva la seguente eccezione:

```
UPDATE students
*
ERROR at line 1:
ORA-04091: table USERA.STUDENTS is mutating, trigger/function may not
see it
ORA-06512: at "USERA.LIMITMAJORS", line 7
ORA-04088: error during execution of trigger 'USERA.LIMITMAJORS'
```

L'errore ORA-4091 si verifica perché LimitMajors interroga la propria tabella attivante, che è mutante. ORA-4091 viene sollevato quando il trigger viene attivato, non quando viene creato.

Soluzione per l'errore di tabella mutante

Students è mutante solo per un trigger a livello di riga. Questo significa che non è possibile interrogarla in un trigger a livello di riga, ma è possibile farlo in un trigger a livello di dichiarazione. Tuttavia non è possibile semplicemente rendere LimitMajors un trigger a livello di dichiarazione, poiché nel corpo del trigger è necessario utilizzare il valore di :new.major. La soluzione è la creazione di due trigger: uno a livello di riga e uno a livello di dichiarazione. Nel trigger a livello di riga si registra il valore di :new.major, ma non si interroga students; la query viene effettuata nel trigger a livello di dichiarazione che utilizza il valore registrato nel trigger di riga.

Come si registra tale valore? Un modo è l'utilizzo di una tabella PL/SQL in un package. In questo modo è possibile salvare più valori per ciascun aggiornamento. Inoltre ogni sessione prende la propria istanziazione di variabili in package, quindi non ci si deve preoccupare di aggiornamenti contemporanei da parte di sessioni diverse. Questa soluzione è implementata con il package student_data e i trigger RLimitMajors e SLimitMajors:

```
-- Disponibile online come parte di of createNonMutating.sql
CREATE OR REPLACE PACKAGE StudentData AS
  TYPE t_Majors IS TABLE OF students.major%TYPE
    INDEX BY BINARY_INTEGER;
  TYPE t_IDs IS TABLE OF students.ID%TYPE
    INDEX BY BINARY_INTEGER;

  v_StudentMajors t_Majors;
  v_StudentIDs    t_IDs;
  v_NumEntries    BINARY_INTEGER := 0;
END StudentData;
/

CREATE OR REPLACE TRIGGER RLimitMajors
  BEFORE INSERT OR UPDATE OF major ON students
  FOR EACH ROW
```

```

BEGIN
  /* Registra i nuovi dati in StudentData. Non si fanno modifiche a
     students, per evitare l'errore ORA-4091. */
  StudentData.v_NumEntries := StudentData.v_NumEntries + 1;
  StudentData.v_StudentMajors(StudentData.v_NumEntries) :=
    :new.major;
  StudentData.v_StudentIDs(StudentData.v_NumEntries) := :new.id;
END RLimitMajors;
/

CREATE OR REPLACE TRIGGER SLimitMajors
  AFTER INSERT OR UPDATE OF major ON students
DECLARE
  v_MaxStudents      CONSTANT NUMBER := 2;
  v_CurrentStudents  NUMBER;
  v_StudentID        students.ID%TYPE;
  v_Major             students.major%TYPE;
BEGIN
  /* Scorre in ogni studente inserito o aggiornato e verifica
     di essere ancora nel limite. */
  FOR v_LoopIndex IN 1..StudentData.v_NumEntries LOOP
    v_StudentID := StudentData.v_StudentIDs(v_LoopIndex);
    v_Major := StudentData.v_StudentMajors(v_LoopIndex);

    -- Determina il numero corrente di studenti in questa specializzazione.
    SELECT COUNT(*)
      INTO v_CurrentStudents
      FROM students
      WHERE major = v_Major;

    -- Se non c'è spazio, solleva un errore.
    IF v_CurrentStudents > v_MaxStudents THEN
      RAISE_APPLICATION_ERROR(-20000,
        'Too many students for major ' || v_Major ||
        ' because of student ' || v_StudentID);
    END IF;
  END LOOP;

  -- Reimposta il contatore, quindi la prossima esecuzione userà nuovi dati.
  StudentData.v_NumEntries := 0;
END SLimitMajors;
/

```

NOTA *Prima di eseguire lo script indicato in precedenza, eliminare il trigger `LimitMajors` errato.*

Ora è possibile provare questa serie di trigger aggiornando `students` fino ad avere troppi specializzandi in storia. Questo si può fare utilizzando lo script `testNonMutating.sql` o scrivendo la seguente dichiarazione di aggiornamento:

```
-- Disponibile online come parte di of testNonMutating.sql
UPDATE students
SET major = 'History'
WHERE id IN (1,2,3);
```

Nel trigger `SLimitMajors` il limite di specializzazioni è impostato a due. La dichiarazione di aggiornamento cerca di inserire nel sistema tre specializzandi in storia. Fallisce con il seguente messaggio di errore:

```
UPDATE students
*
ERROR at line 1:
ORA-20000: Too many students for major History because of student 2
ORA-06512: at "USERA.SLIMITMAJORS", line 21
ORA-04088: error during execution of trigger 'USERA.SLIMITMAJORS'
```

Questo è il comportamento desiderato. Questa tecnica può essere applicata alle occorrenze di ORA-4091 quando un trigger a livello di riga legge da o modifica una tabella mutante. Invece di effettuare l'elaborazione illegale nel trigger a livello di riga, si deferisce l'elaborazione a un trigger a livello di dichiarazione `after`, dove è legale. Le tabelle PL/SQL in package sono utilizzate per memorizzare le righe che sono state modificate.

Esistono diverse cose da notare rispetto a questa tecnica:

- le tabelle PL/SQL sono contenute in un package, in modo che saranno visibili ai trigger a livello di riga e a livello di dichiarazione. L'unico modo per garantire che le variabili siano globali è inserirle in un package;
- viene utilizzata una variabile di contatore, `StudentData.v_NumEntries`. Questa è inizializzata su zero quando viene creato il package. Viene incrementata dal trigger a livello di riga. Il trigger a livello di dichiarazione fa riferimento ad essa e quindi la reimposta a zero dopo l'elaborazione. Questo è necessario affinché la prossima dichiarazione UPDATE emessa da questa sessione abbia il valore corretto;
- il controllo in `SLimitMajors` del numero massimo di studenti doveva essere leggermente modificato. Poiché ora è un trigger di dichiarazione `after`, `v_CurrentStudents` conterrà il numero di studenti nella specializzazione dopo l'inserimento o l'aggiornamento, non prima. Quindi il controllo per `v_CurrentStudents + 1`, effettuato in `LimitMajors`, è sostituito da `v_CurrentStudents`;
- invece di tabelle PL/SQL si sarebbe potuta utilizzare una tabella di database. Non si consiglia questa tecnica, perché sessioni contemporanee che emettano un UPDATE interferirebbero tra di loro (in Oracle8i e successivi, tuttavia, si può utilizzare una tabella temporanea). Le tabelle PL/SQL in package sono uniche tra sessioni, il che evita il problema.

10.4 **Sommario**

Come si è visto, i trigger sono un'aggiunta importante a PL/SQL e Oracle. Possono essere utilizzati per rispettare limitazioni sui dati molto più complesse delle normali limitazioni di integrità referenziali, oltre a implementare il comportamento corretto per viste complesse. Le funzioni di attributo di eventi possono essere utilizzate per i trigger di sistema, per determinare tutti i tipi di informazioni sull'evento attivante e sulla situazione che lo ha causato. Nel prossimo capitolo si comincerà la discussione sui package incorporati con la comunicazione tra sessioni.