Appunti sui Database Attivi

Basato sulle slide del Prof. Danilo Montesi

17 maggio 2025

Indice

1 Dai Database Passivi ai Database Attivi

L'idea di base dei database attivi è quella di rendere il database stesso più "intelligente" e "reattivo", capace cioè di eseguire automaticamente delle azioni in risposta a determinati eventi, senza che sia l'applicazione a dover gestire tutta questa logica.

1.1 Database Passivi (Tradizionali)

- Eseguono solo le operazioni esplicitamente richieste dall'utente o dall'applicazione.
- Un primo, rudimentale esempio di "reattività" nei database passivi sono le **strategie di reazione ai vincoli di integrità referenziale**.
  - Esempio SQL: `ON DELETE CASCADE`, `ON UPDATE SET NULL`, `ON DELETE SET DEFAULT`, `ON DELETE NO ACTION`.
  - Qui il database "reagisce" a un `DELETE` o `UPDATE` su una tabella primaria, eseguendo un'azione sulla tabella correlata.
- L'idea è di estendere questa capacità introducendo costrutti linguistici specifici (chiamati **regole attive**) per gestire una parte del comportamento procedurale che altrimenti sarebbe nell'applicazione.
- **Vantaggio:** Se questo comportamento è a livello di database, è "condiviso" tra tutte le applicazioni che accedono a quei dati, garantendo consistenza e promuovendo l'indipendenza dei dati.

1.2 Database Attivi

- Hanno un componente dedicato per gestire **regole attive** basate sul paradigma **ECA (Event-Condition-Action)**.
  - **Evento (Event):** Un cambiamento nel database (es. `INSERT`, `UPDATE`, `DELETE` su una tabella specifica).
  - **Condizione (Condition):** Una verifica (un predicato SQL) che deve essere vera affinché l'azione scatti. Se la condizione è omessa, si assume sempre vera.
  - **Azione (Action):** Una o più istruzioni SQL (o codice in un linguaggio procedurale specifico del DBMS, come PL/SQL per Oracle) da eseguire.
- Questi database hanno un **comportamento reattivo**: non si limitano a eseguire le transazioni dell'utente, ma *reagiscono* agli eventi eseguendo anche le regole definite.
- Nei DBMS commerciali (standard SQL3 e successivi), le regole attive sono implementate principalmente tramite i **trigger**.

2 Evoluzione dell'Architettura e Ruolo dei Database Attivi

Le slide mostrano un'evoluzione nel tempo di come la logica applicativa e la gestione dei dati sono state organizzate:

- **Anni '70 (No DBMS):** Le applicazioni accedevano direttamente ai file tramite il Sistema Operativo.
- **Anni '80 (Primi DBMS):** Le applicazioni interagivano con un DBMS, che gestiva "tabelle di dati".
- **Anni '90 (Comportamento Procedurale):** Esigenza di spostare parte del **comportamento procedurale condiviso** all'interno del DBMS.
  - **Stored Procedures:** Introdotte per condividere logica comune. Problemi: non standardizzate, impedance mismatch.
  - **Trigger (Database Attivi):** Introdotte regole specifiche (i **trigger**) per modellare il comportamento procedurale condiviso, gestito direttamente dal DBMS.

- **Anni 2000 (Applicazioni Web):** Architettura client-server a più livelli (Client JS, Web App Server Java/Node, Server con Active DBMS).
- **Anni 2010 (Mobile Apps):** Simile, con client mobile.

Concetto chiave dell'evoluzione: Tendenza a spostare la logica strettamente legata ai dati e condivisa all'interno del database stesso.

3 Trigger: Il Cuore dei Database Attivi

Un trigger è una procedura memorizzata nel database che viene eseguita automaticamente quando si verifica un determinato evento su una tabella specifica.

3.1 Definizione

- Definiti con istruzioni DDL (Data Definition Language), es. `CREATE TRIGGER`.
- Seguono il paradigma **ECA**:
  - **Evento:** Un'operazione di modifica dei dati (`INSERT`, `DELETE`, `UPDATE`).
  - **Condizione:** Un predicato SQL opzionale (clausola `WHEN`).
  - **Azione:** Una sequenza di istruzioni SQL o un blocco di codice procedurale.
- **Flusso intuitivo:** Attivazione (evento) $\rightarrow$ Verifica (condizione) $\rightarrow$ Esecuzione (azione).
- Ogni trigger è associato a una **tabella target**.

3.2 Granularità dei Trigger

- **Row-level (per tupla/riga):** Il trigger si attiva e la sua azione viene eseguita *per ogni singola riga* affetta dall'istruzione SQL.
- **Statement-level (per istruzione):** Il trigger si attiva e la sua azione viene eseguita *una sola volta per l'intera istruzione SQL*.

3.3 Modalità (Timing) dei Trigger

- `IMMEDIATE` **(Immediata):** L'azione del trigger viene eseguita immediatamente *prima* (`BEFORE`) o *dopo* (`AFTER`) l'evento. Modalità più comune.
- `DEFERRED` **(Differita):** L'azione del trigger viene posticipata e eseguita solo al momento del `COMMIT` della transazione.

3.4 Modello Computazionale e Problemi

Data una transazione utente $T^U = U_1; \ldots; U_n$. Se le regole P sono del tipo E, C $\rightarrow$ A. U_i^P è la sequenza di azioni indotte da U_i.

- **Semantica Immediata:** $T^I = U_1; U_1^P; U_2; U_2^P; \ldots; U_n; U_n^P$.
- **Semantica Differita:** $T^D = U_1; \ldots U_n; U_1^P; \ldots; U_n^P$.
- **Problemi Potenziali:**
  - **Terminazione:** L'esecuzione a cascata dei trigger deve terminare (evitare cicli infiniti).
  - **Confluenza:** Se più trigger possono essere attivati, il risultato finale è lo stesso indipendentemente dall'ordine?
  - **Equivalenza:** Diverse definizioni di regole portano allo stesso comportamento?

4 Sintassi dei Trigger (SQL:1999 Standard)

```
CREATE TRIGGER nomeTrigger
{ BEFORE | AFTER } -- Timing
{ INSERT | DELETE | UPDATE [OF nomeColonna [, nomeColonna]...] } -- Evento
ON nomeTabellaTarget -- Tabella target

[ REFERENCING -- Variabili per righe/tabelle vecchie e nuove
-- Per trigger STATEMENT-LEVEL:
[ OLD TABLE [AS] varTabellaVecchia ]
[ NEW TABLE [AS] varTabellaNuova ]
-- Per trigger ROW-LEVEL:
[ OLD [ROW] [AS] varTuplaVecchia ] -- Solitamente OLD
[ NEW [ROW] [AS] varTuplaNuova ]   -- Solitamente NEW
]

[ FOR EACH { ROW | STATEMENT } ] -- Granularità

[ WHEN (condizioneSQL) ] -- Condizione (opzionale)

SQLProceduralStatement; -- Azione
```

4.1 `BEFORE` vs `AFTER`

- `BEFORE`: Eseguito *prima* dell'operazione. Utile per validare/modificare dati in ingresso.
- `AFTER`: Eseguito *dopo* l'operazione. Utile per logging, aggiornare tabelle dipendenti.

4.2 Clausola `REFERENCING` (`OLD` e `NEW`)

Permette di accedere ai valori dei dati *prima* e *dopo* la modifica.

- **Per trigger `ROW-LEVEL`:**
  - `OLD`: Pseudo-riga con valori *prima* della modifica (per `UPDATE`, `DELETE`). Accesso: `OLD.nomeColonna`.
  - `NEW`: Pseudo-riga con valori *dopo* la modifica (per `INSERT`) o proposti (per `UPDATE`). Accesso: `NEW.nomeColonna`.
- **Per trigger `STATEMENT-LEVEL`:**
  - `OLD TABLE`: Tabella temporanea con righe *prima* della modifica.
  - `NEW TABLE`: Tabella temporanea con righe *dopo* la modifica.
- **Disponibilità:**
  - `INSERT`: Solo `NEW` / `NEW TABLE`.
  - `DELETE`: Solo `OLD` / `OLD TABLE`.
  - `UPDATE`: Sia `OLD` / `OLD TABLE` che `NEW` / `NEW TABLE`.

5 Trigger in Oracle

5.1 Sintassi

```
CREATE [OR REPLACE] TRIGGER nomeTrigger
{ BEFORE | AFTER }
evento1 [OR evento2 OR evento3 ...] -- Es. INSERT OR UPDATE OF col1
ON nomeTabella
[ REFERENCING OLD AS nomeVarVecchia NEW AS nomeVarNuova ] -- Default :OLD, :NEW
[ FOR EACH ROW ] -- Se omesso, è STATEMENT level
[ WHEN (condizione) ]
DECLARE
-- variabili locali PL/SQL
```

```
BEGIN
-- corpo del trigger (logica PL/SQL)
-- accesso con :OLD.colonna e :NEW.colonna per FOR EACH ROW
EXCEPTION
-- gestione errori
END;
```

5.2 Semantica Oracle

- Modalità Immediata (`BEFORE`, `AFTER`).
- Ordine di Esecuzione:
  1. `BEFORE STATEMENT` triggers.
  2. Per ogni riga affetta:
     (a) `BEFORE ROW` triggers.
     (b) Operazione DML + controllo vincoli.
     (c) `AFTER ROW` triggers.
  3. `AFTER STATEMENT` triggers.
- Errore: Rollback dell'intera istruzione/transazione.
- Priorità: Basata su timestamp di creazione (non garantita).
- Cascata: Massimo 32 trigger.

5.3 Esempio Oracle (Riordino Prodotti)

Trigger `Reorder` su tabella `Warehouse`.

- **Evento:** `AFTER UPDATE OF QtyAvbl ON Warehouse`.
- **Granularità:** `FOR EACH ROW`.
- **Condizione:** `WHEN (NEW.QtyAvbl < NEW.QtyLimit)`.
- **Azione (PL/SQL):**

```
DECLARE
X NUMBER;
BEGIN
-- Controlla se esiste già un ordine pendente per questa parte
SELECT COUNT(*) INTO X
FROM PendingOrders
WHERE Part = :NEW.Part; -- :NEW si riferisce alla riga aggiornata

IF X = 0 THEN -- Se non ci sono ordini pendenti
-- Inserisce un nuovo ordine pendente
INSERT INTO PendingOrders (Part_ID, QuantityToReorder, OrderDate)
VALUES (:NEW.Part, :NEW.QtyReord, SYSDATE);
END IF;
END;
```

6 Trigger in DB2

6.1 Sintassi

```
CREATE TRIGGER nomeTrigger
{ BEFORE | AFTER } evento -- evento è INSERT, UPDATE, DELETE
ON nomeTabella
REFERENCING { OLD AS varTuplaVecchia | NEW AS varTuplaNuova |
```

```
       OLD_TABLE AS varTabellaVecchia | NEW_TABLE AS varTabellaNuova } ...
FOR EACH { ROW | STATEMENT }
[ WHEN (predicatoSQL) ]
SQLProceduralStatement; -- Può essere un blocco BEGIN ATOMIC ... END
```

6.2 Semantica DB2

- Modalità Immediata.
- I trigger `BEFORE` di norma *non possono modificare il database* (eccetto assegnare valori a `NEW` in `BEFORE INSERT/UPDATE ROW`), quindi non possono attivare altri trigger.
- Errore: Rollback.
- Priorità: Determinata dal sistema (timestamp).
- Cascata: Massimo 16 trigger.

6.3 Esempio DB2 (Controllo Riduzione Stipendio)

Trigger `checkWage` su tabella `Employee`.

- **Evento:** `AFTER UPDATE OF Wage ON Employee`.
- **Granularità:** `FOR EACH ROW`.
- **Condizione:** `WHEN (NEW.Wage < OLD.Wage * 0.97)`.
- **Azione:**

```
-- La sintassi specifica può variare leggermente in DB2 SQL PL
-- Questo è un esempio concettuale basato sulle slide
BEGIN
-- Se la riduzione è maggiore del 3%, la limita al 3%
-- L'azione qui presuppone che il trigger AFTER possa modificare la stessa riga
-- anche se più tipicamente si impedirebbe l'azione in un BEFORE trigger
-- o si farebbe l'update in modo più controllato.
-- La slide suggerisce un update, quindi lo riporto così:
UPDATE Employee
SET Wage = OLD.Wage * 0.97
WHERE EmpCode = NEW.EmpCode; -- O l'identificativo di riga corrente
END
```

Nota sull'esempio DB2: L'azione di un trigger `AFTER` che modifica la stessa riga che ha scatenato il trigger può portare a ricorsione se non gestita con attenzione. Spesso, per questo tipo di logica, si preferirebbe un trigger `BEFORE` per modificare `NEW.Wage` o per sollevare un errore se la condizione non è rispettata.

7 Estensioni dei Trigger (Non Sempre Disponibili)

- Eventi Temporali (periodici) o definiti dall'utente.
- Combinazioni Booleane di Eventi.
- Clausola `INSTEAD OF`: Esegue l'azione del trigger al posto dell'operazione originale (utile per viste non aggiornabili).
- Esecuzione "Detached" (transazione autonoma).
- Definizione di Priorità esplicita.
- Gruppi di Regole (attivabili/disattivabili).
- Regole su Query (`SELECT`).

8 Proprietà delle Regole Attive

- **Terminazione (essenziale):** L'esecuzione deve finire.
- **Confluenza:** Il risultato finale è indipendente dall'ordine di esecuzione di trigger concorrenti?
- **Determinismo delle Osservazioni:** L'utente osserva sempre lo stesso comportamento?

9 Applicazioni dei Trigger

9.1 Funzionalità Interne al DBMS

- **Gestione dei Vincoli di Integrità Complessi:** Oltre ai vincoli standard.
- **Replicazione dei Dati:** Catturare modifiche e replicarle.
- **Gestione delle Viste:**
  - **Viste Materializzate:** Propagare modifiche dalle tabelle base alla vista materializzata.
  - **Viste Virtuali (con `INSTEAD OF`):** Rendere aggiornabili viste complesse.

9.2 Funzionalità Applicative (Logica di Business nel DB)

- **Descrizione del Comportamento del Database:** Incapsulare logica di business direttamente nel DB per consistenza.
  - Esempi: Mantenere `last_modified_date`, inviare notifiche, audit, calcolare valori derivati, impedire operazioni basate su condizioni complesse.
- **Confronto Logica in Applicazione vs. Logica in Trigger:**
  - **Logica in Applicazione (es. Node.js con Prisma/Mongoose):**
    - *Pro:* Più facile da testare, linguaggio dell'applicazione, flessibilità.
    - *Contro:* Se il DB è accessibile esternamente, la logica può essere bypassata.
  - **Logica in Trigger DB:**
    - *Pro:* Consistenza garantita, logica vicina ai dati.
    - *Contro:* Minore visibilità/debug per lo sviluppatore applicativo, dipendenza dal linguaggio procedurale del DB, test più complessi.

10 Conclusione

I database attivi, attraverso i trigger, offrono un meccanismo potente per automatizzare reazioni a eventi sui dati, centralizzare la logica di business e garantire la consistenza. Tuttavia, il loro uso richiede un'attenta progettazione per evitare complessità, problemi di performance e difficoltà di manutenzione. È fondamentale bilanciare cosa implementare a livello di database tramite trigger e cosa lasciare alla logica applicativa.