7 - Sviluppo di applicazioni per Basi di Dati

Nell'utilizzare una base di dati, il dialogo diretto con l'interprete SQL è riservato a pochi utenti esperti. L'accesso di gran lunga più tipico a una base di dati avviene attraverso applicazioni integrate nel sistema informativo, le quali forniscono agli utenti un'interfaccia semplificata che favorisce l'interazione.
SQL supporta le applicazioni in due modi:

- Incrementando le funzionalità del DBMS per mezzo della definizione di **procedure** e **trigger** (visti nel capitolo 4 - SQL, più precisamente nelle caratteristiche evolute);
- Integrando comandi SQL con istruzioni di un linguaggio di programmazione (procedurale/oggetti) utilizzando SQL Embedded per i linguaggi datati, oppure il Cal Level Interface per linguaggi di programmazione più recenti

SQL Embedded

SQL Embedded prevede di introdurre direttamente nel programma sorgente scritto nel linguaggio di alto livello le istruzioni SQL, distinguendole dalle normali istruzioni tramite un opportuno separatore. Lo standard SQL prevede che il codice SQL sia preceduto dalla stringa exec sql e termini con il carattere ';'. Dal punto di vista dell'implementazione, è necessario far precedere la compilazione del linguaggio di alto livello all'esecuzione di un preprocessore che riconosca le istruzioni SQL e sostituisce a esse un insieme di chiamate ai servizi del DBMS, tramite una libreria specifica per ogni sistema.

Tutte le variabili usate per scambiare dati fra il programma e il DBMS sono dichiarate in un blocco di istruzioni compreso tra i due comandi:

$$\text{exec sql begin declare section}$$
$$\text{exec sql end declare section}$$

Ogni volta che si dichiara una variabile (x) in questo blocco nel linguaggio ospite, viene creata una variabile copia identica in SQL il cui nome è lo stesso ma preceduto dal carattere ' : '$(: x)$. Ogni modifica effettuata x viene effettuata anche se $: x$ e viceversa.

Per lavorare con un database da linguaggio ospite:

1. Si stabilisce una connessione con il sistema specificando il database su sui lavorare tramite il comando seguente:

   $$\text{CONNECTED TO} < IdUtente > \text{IDENTIFIED BY} < password > \text{USING} < Database >$$

2. Il preprocessore introduce implicitamente la dichiarazione di una struttura SQLCA (SQL communication area) per gestire la comunicazione tra programma e DBMS
3. Si può accedere al campo SQLCODE della struttura SQLCA, il cui valore è un intero che codifica l'effetto dell'ultima operazione SQL effettuata. Se il valore è uguale a zero indica

la corretta esecuzione del comando, se è diverso da zero indica che si è verificata una anomalia ed il comando non è andato a buon fine.

4. In caso di query scalary (unico risultato) il comando SELECT è esteso con la clausola

$$\mathrm{into} < Variabile > \{, < Variabile >\}$$

per assegnare a delle variabili del programma il valore degli attributi dell'unica tupla del risultato. Se il comando SELECT può assegnare ad una variabile il valore nullo (non previsto nel linguaggio ospite) la variabile va dichiarata come:

$$: Variabile\ \mathrm{INDICATOR} < \mathrm{IndVariabile} >$$

Se dopo l'esecuzione $IndVariabile$ ha valore minore di zero allora $Variabile$ ha valore significativo (quindi non null).

Un esempio di implementazione di SQL Embedded tramite C è questo:

```c
#include <stdlib.h>
#include <stdio.h>

main()
{
    exec sql begin declare section;
        char *NomeDip = "Manutenzione";
        char *CittaDip = "Pisa";
        int NumeroDip = 20;
    exec sql end declare section;

    exec sql connect to utente@librobd;

    if (sqlca.sqlcode != 0) {
        printf("Connessione al DB non riuscita\n");
    }
    else {
        exec sql insert into Dipartimento
            values(:NomeDip, :CittaDip, :NumeroDip);

        exec sql disconnect all;
    }
}
```

Un importante problema che caratterizza l'integrazione tra SQL e i normali linguaggi di programmazione è il cosiddetto **conflitto di impedenza**. I linguaggi di programmazione

accedono agli elementi di una tabella scandendone le righe una a una (tuple-oriented). Al contrario SQL è un linguaggio di tipo set-oriented, che opera su intere tabelle e restituisce come risultato di un'interrogazione un'intera tabella. Le soluzioni a questo problema si ottengono con l'utilizzo dei **cursori** e l'utilizzo di linguaggi con costruttori di tipo in grado di gestire una struttura del tipo "insieme di righe" (Call Level Interface).

Cursori

Un cursore è una variabile speciale che permette ad un programma di accedere alle righe di una tabella una alla volta; il cursore viene definito su una generica interrogazione, con la seguente sintassi:

$$\text{DECLARE } NomeCursore \text{ [SCROLL] CURSOR FOR } SelectSQL$$
$$\text{[FOR}\langle\text{READ ONLY | UPDATE[OF}Attributo \text{ \{ , } Attributo \text{ \}] } \rangle \text{]}$$

Al momento della sua dichiarazione, il cursore si riferisce ad una struttura vuota. L'avvaloramento del cursore avviene mediante il comando $\text{OPEN } Nomecursore$ e determina l'esecuzione dell'interrogazione associata al cursore, il suo risultato poi diventa accessibile tramite il comando FETCH:

$$\text{FETCH [} Posizione \text{ FROM] } NomeCursore \text{ INTO } ListaDiFetch$$

Il comando copia il contenuto di una riga dal cursore nelle variabili del linguaggio ospite enumerate in $ListaDiFetch$. In particolare, $ListaDiFetch$ contiene una variabile per ogni elemento della target list dell'interrogazione, con una corrispondenza tra colonne della tabelle e variabili del linguaggio ospite dettata dalla posizione della variabile nella lista; ciascuna variabile dalla lista di fetch deve avere un tipo compatibile con i domini degli elementi della target list dell'interrogazione SQL.

Il cursore è una variabile speciale dotata di un proprio stato, infatti il parametro $Posizione$ permette di specificare quale riga deve essere oggetto dell'operazione di fetch; il parametro può assumere i valori:

- NEXT sposta il cursore alla riga successiva alla corrente
- PRIOR alla precedente alla corrente
- FIRST alla prima riga del risultato
- LAST all'ultima riga del risultato
- $\text{ABSOLUTE} < EspressioneIntera >$, ammesso che espressione intera sia uguale a i, posiziona il cursore alla i-esima posizione a partire dalla prima riga del risultato
- $\text{RELATIVE} < EspressioneIntera >$, ammesso che espressione intera sia uguale a i, posiziona il cursore alla i-esima posizione a partire dalla riga corrente in cui si trova il cursore
  Di default il cursore va usa il comando NEXT

I comandi di UPDATE e DELETE permettono di apportare modifiche alla base di dati tramite l'uso di cursori tramite seguente sintassi:

$$\begin{aligned}&\text{UPDATE } NomeTabella\\&\quad\text{SET } Attributo = \langle\ Espressione \mid \text{NULL} \mid \text{DEFAULT}\ \rangle\\&\quad\{\ , Attributo = \langle\ Espressione \mid \text{NULL} \mid \text{DEFAULT}\ \rangle\ \}\\&\quad\text{WHERE CURRENT OF } NomeCursore\end{aligned}$$

$$\text{DELETE FROM } NomeTabella \text{ WHERE CURRENT OF } NomeCursore$$

Infine, esiste il comando CLOSE che comunica al sistema che il risultato dell'interrogazione non serve più, chiudendo il cursore e rilasciando l'area di buffer occupata da tale, si definisce come $\text{CLOSE } NomeCursore$

Il vantaggio di utilizzare linguaggi che ospitano SQL consiste nella facilità con cui un programmatore può accedere ad un database utilizzando linguaggi già conosciuti. Lo svantaggio consiste nel curare la conversazione dei dati fra i tipi del linguaggio host e quelli relazionali (**conflitto di impedenza**).
Quanto visto fin'ora è definibile come SQL statico, perché le interrogazione effettuate hanno una struttura predefinita e ciò che varia è solamente il valore dei parametri usati in ingresso. È però possibile che l'applicazione richieda di effettuare interrogazioni non note a priori ma create a run time, sulla base dell'evoluzione delle informazioni contenute del database stesso o sulla base dell'interazione dell'utente con il database.

SQL dinamico

Nel caso di SQL statico, i comandi SQL sono noti a tempo di compilazione e vengono gestiti dal preprocessore, venendo ottimizzati solo una volta, e non ogni volta che il comando deve essere eseguito. Questo comporta grossi vantaggi in termini di prestazioni. L'SQL dinamico non può avvalersi della fase di preprocessamento, non essendo noti a priori i comandi da ottimizzare, quindi la costruisce a tempo di esecuzione. Per questo motivo SQL dinamico mette a disposizione due modalità di interazione:

1. Esecuzione immediata: si esegue immediatamente l'interrogazione/istruzione specificata direttamente o in un parametro di tipo stringa

   $$\text{execute immediate: } < VariabileConIstruzioneSQL >$$

   Questo è possibile solo quando si hanno una o più istruzioni conosciute a priori nella sua interezza, ma non si sa quando o quali verranno eseguite (perché magari dipende dalla scelta dell'utente). Come si può vedere il comando non richiede parametri né in ingresso né in uscita (variabile di tipo stringa).
2. La seconda modalità permette di eseguire istruzioni dipendenti non solo dalla dinamicità del sistema, ma prevede anche dei parametri in ingresso o in uscita. Tale modalità si compone di due fasi:
   - **Fase di preparazione**: il comando prepare analizza l'istruzione SQL e la traduce nel linguaggio interno al DBMS. Inoltre il comando prepare associa alla traduzione dell'istruzione un nome, che può essere usato dagli altri comandi:

     $$\text{PREPARE } NomeComando \text{ FROM } IstruzioneSQL$$

L'istruzione SQL può contenere dei parametri in ingresso, rappresentati dal carattere di punto interrogativo, per esempio:

$$\begin{array}{l}\text{PREPARE } :comando \\ \text{FROM SELECT } Città \text{ FROM } Dipartimento \text{ WHERE } Nome = \ ?\end{array}$$

In questo modo alla variabile comando del programma corrisponde la traduzione dell'istruzione, con un parametro di ingresso che rappresenta il nome del dipartimento che deve essere selezionato dall'interrogazione.

- **Fase di esecuzione**: per eseguire un comando che è stato preelaborato da una istruzione prepare si una l'istruzione execute, che ha la seguente sintassi:

$$\text{EXECUTE } NomeComando \ [\ \text{INTO } ListaTarget\ ]\ [\ \text{USING } ListaParametri\ ]$$

La lista dei target contiene l'elenco dei parametri in cui deve essere scritto il risultato dell'esecuzione del comando, la lista dei parametri specifica i valori che devono essere assunti dai parametri variabili. Un esempio vi è:

Se l'istruzione SQL restituisce più di un risultato (un insieme di tuple) è necessario usare i cursori. L'uso dei cursori con SQL dinamico è molto simile all'uso dei cursori in SQL statico. Le uniche due differenze consistono nel fatto che si associa al cursore l'identificativo dell'interrogazione invece che l'interrogazione stessa, e che i comandi d'uso del cursore ammettono le clausole into e using che permettono la specifica degli eventuali parametri di ingresso e uscita.

$$\begin{array}{l}\text{PREPARE } :comando \text{ FROM } :istruzioneSQL \\ \text{DECLARE } Cursore \text{ CURSOR FOR } :comando \\ \text{OPEN } Cursore \text{ USING } :nome1\end{array}$$

Quando un'istruzione SQL che era stata preparata non serve più, è possibile rilasciare la memoria occupata dalla traduzione dell'istruzione utilizzando il comando:

$$\text{DEALLOCATE PREPARE } NomeComando$$

Call Level Interface (CLI)

I meccanismi descritti finora ricadono nella famiglia delle soluzioni SQL Embedded, in cui si prevede di disporre di un preprocessore, utilizzato a tempo di compilazione o di esecuzione. È possibile anche interfacciarsi direttamente con un DBMS, da un programma usando apposite primitiva (librerie per realizzare il dialogo con il database). Quello offerto dal CLI è uno strumento meglio integrato con il linguaggio di programmazione. Lo svantaggio consiste nel dover gestire esplicitamente aspetti che in SQL Embedded vengono risolti automaticamente dal preprocessore.
Diversi sistemi offrono proprie CLI sulla base del sistema operativo e del linguaggio di programmazione. Il modo generale d'uso di questi strumenti è il seguente:

- Si utilizza un servizio CLI per creare una connessione con il DBMS
- Si invia sulla connessione un comando SQL che rappresenta la richiesta

- Si riceve come risposta del comando una struttura relazionale in un opportuno formato
- Al termine della sessione di lavoro:
  - Si chiude la connessione
  - Si rilasciano le strutture dati usate per la gestione del dialogo

Object Relational Mapping (ORM)

Le soluzioni viste fino a questo momento sono basate sul trasferimento di dati dagli oggetti/variabili temporanei dell'applicazione alla fonte persistente e viceversa. I sistemi di mappatura relazionale (ORM), sviluppati in ambito Object Oriented, offrono e gestiscono automaticamente la corrispondenza tra oggetti temporanei e tuple.

Gestione delle transazioni

Come detto nel capitolo 1 - Introduzione, una transazione è un'operazione che il DBMS esegue garantendone:

- **Atomicità**: una transazione non può essere incompleta, se una transazione fallisce i suoi effetti vengono annullati ed il database torna alla fase di consistenza precedente.
- **Seralizzabilità**: garantita con il meccanismo del bloccaggio dei dati (record and table locking). È una gestione ottenuta mediante tecniche di programmazione concorrente in cui prima di leggere/modificare un dato, una transazione deve bloccare questo dato il lettura/scrittura.

Idealmente per assicurare che accessi concorrenti non provochino stati di inconsistenza nel database, ogni transazione dovrebbe essere effettuata in serie. Questo comporterebbe una estrema lentezza del DB in caso di accessi concorrenti. Con la tecnica del record and table locking, invece, si riesce a serializzare le letture e scritture su un dato, bloccando solo il dato interessato dall'operazione. Si interviene quindi con la programmazione delle transazioni, ovvero bloccare, a livello di codice, il dato di interesse per il tempo strettamente necessario all'esecuzione della transazione. Questo permette di prevedere due condizioni:

- Quando il programma rileva un'anomalia, può essere opportuno disfare solo una parte delle operazioni fatte
- Quando un programma richiede un tempo lungo per terminare le proprie operazioni, può essere opportuno spezzare il programma in più transizioni.

I DBMS relazionali permettono di spezzare un programma in più transazioni usando le operazioni di COMMIT e $\mathrm{ROLLBACK}$. Una transazione è considerata iniziata dal sistema quando un programma esegue un'operazione su una tabella e termina quando:

- Viene eseguito il comando $\mathrm{EXEC\ SQL\ COMMIT\ WORK}$, cioè la transazione termina esplicitamente, rilasciando i blocchi sui dati usati, rendendoli disponibili ad altre transazioni (es. attesa di un input utente).

- Viene eseguito un comando$\mathrm{EXEC\ SQL\ ROLLBACK\ WORK}$ (abort transaction), che comporta la terminazione prematura della transazione, quindi il disfacimento di tutte le modifiche apportate ed il rilascio dei blocchi sui dati usati
- Il programma termina naturalmente senza errori
- Il programma termina con fallimento e provoca la terminazione prematura della transazione

La suddivisione di un programma in più transazioni, dunque l'esecuzione concorrente di varie transazioni, può causare alcuni problemi di correttezza. La presenza di queste anomalie comporta alcune complicazioni di programmazione. Un esempio di anomalia potrebbe essere data da una sequenza di transizioni del tipo:

1. Si avvia la transizione che informa l'utente in merito alla quantità ed al prezzo di un determinato prodotto e si termina la transizione.
2. Si avvia la transizione che permette la modifica del prezzo di quel prodotto e si termina la transizione
3. Si avvia la transizione che permette all'utente di ordinare il prodotto al prezzo che gli è stato riferito dalla prima transizione e si termina la transizione.

È assolutamente necessario che, prima della terza transizione, nel codice vada introdotto un controllo che permetta all'utente di procedere con l'ordine solo se il prezzo (o la quantità) del prodotto è rimasto invariato nel frattempo. È facile immaginare le dimensioni del problema per sistemi molto complessi.

Ripetizione delle transazioni

Nella programmazione di transazioni è importante prevedere la ripetizione della transazione quando questa viene interrotta dal DBMS per sbloccare una condizione di stallo (deadlock). Lo stallo è una situazione tipica dei sistemi in cui si utilizzano dei meccanismi di blocco delle risorse e si verifica quando due o più transazioni sono bloccate in attesa l'una dei dati dell'altra. Se il DBMS riconosce una situazione di stallo, interrompe una delle transazioni, segnalando al programma $\mathrm{sqlcode} = \mathrm{deadabort}$. Il programma può dunque decidere di far ripartire la transazione.
Per ottimizzare il sistema, si può decidere di non lasciare al DBMS il compito di individuare eventuali stalli, ma prevederli da codice.
Esempio: Un aggiornamento del supervisore di tutti gli agenti di una certa zona, può coinvolgere molte tuple. Si programmare il codice in maniera tale che, in caso di interruzione per uno stallo, prova ripetutamente l'operazione, fino ad un massimo di quattro volte.

Livelli di isolamento

Rinunciando alla proprietà di serializzabilità e quindi di isolamento delle transazioni si possono aumentare le prestazioni dell'applicazione. In altre parole, se si limita il bloccaggio di una risorsa, allora più applicazioni possono accedere alla stessa risorsa. Vanno, ovviamente, prese in considerazione le eventualità di inconsistenze.

Nella proposta dell'SQL-92, con il comando SET TRANSACTIONsi può scegliere uno dei seguenti livelli di isolamento per consentire gradi di concorrenza decrescenti:

```
SET TRANSACTION ISOLATION LEVEL [READ UNCOMMITTED | READ
COMMITTED | REPEATABLE READ | SERIALIZABLE]
```

Analizziamo i vari livelli di isolamento per grado di concorrenza decrescente:

1. **Read uncommitted (degree of isolation 0):** consente transazioni che fanno solo operazioni di lettura (quelle di modifica sono proibite) che vengono eseguite dal sistema senza bloccare in lettura i dati. Si rende il sistema molto più veloce, ma può accadere che una transazione legga dati modificati da un'altra transazione non ancora terminata (dati sporchi) oppure abortita in seguito, motivo per cui questo livello di isolamento può applicarsi esclusivamente su porzioni di DB utilizzate sempre e solo in lettura.
2. **Read committed (degree of isolation 1):** a differenza del livello precedente in cui sui dati in lettura non vi era un bloccaggio, questo livello prevede che i dati in lettura siano bloccati esclusivamente per il tempo di lettura e subito rilasciati, mentre i dati in scrittura siano rilasciati alla terminazione della transazione. Questo comporta letture non ripetibili, ovvero letture successive sugli stessi dati possono dare risultati diversi perché i dati sono stati modificati da altre transazioni terminate nell'intervallo tra la prima e la seconda lettura.
3. **Repeatable read (degree of isolation 2)**: prevede che i blocchi in lettura e scrittura non sia applicati sull'intera tabella, ma siano assegnati solo su sottoinsiemi di tuple e vengano rilasciati alla terminazione della transazione. Questa soluzione evita il problema delle letture non ripetibili, ma non quello delle letture fantasma.
4. **Serializable (degree of isolation 4)**: le transazioni vengono serializzate in maniera sicura.

Controllo degli accessi

In SQL è possibile specificare chi (utente) può utilizzare la base di dati e quale tipo di autorizzazione fornire ad uno specifico utente (lettura, scrittura, etc). Oggetto dei privilegi (diritti di accesso) sono di solito le tabelle, ma anche altri tipi di risorse, quali singoli attributi, viste o domini. Un utente predefinito _SYSTEM, amministratore della base di dati, ha tutti i privilegi. Il creatore di una risorsa ha tutti i privilegi su di essa. Un privilegio è caratterizzato da:

- La risorsa cui si riferisce
- L'utente che concede il privilegio
- L'utente che riceve il privilegio
- L'azione che viene permessa
  - INSERT: permette di inserire nuovi oggetti (ennuple)
  - UPDATE: permette di modificare il contenuto
  - DELETE: permette di eliminare oggetti

    - SELECT: permette di leggere le risorse
    - REFERENCES: permette la definizione di vincoli di integrità referenziale verso la risorsa
    - USAGE: permette l'utilizzo di una risorsa in una definizione (per esempio un dominio)
- La trasmissibilità del privilegio

I privilegi possono essere concessi con il comando:

$$\text{GRANT} \langle\ \textit{Privileges} \mid \text{ALL PRIVILEGES}\ \rangle\ \text{ON}\ \textit{Resource}\ \text{TO}\ \textit{Users}\ [\ \text{WITH GRANT OPTIONS}\ ]$$

GRANT OPTIONS specifica se il privilegio può essere trasmesso ad altri utenti.
Chiaramente i privilegi possono essere anche revocati con il comando:

$$\text{REVOKE}\ \textit{Privileges}\ \text{ON}\ \textit{Resource}\ \text{FROM}\ \textit{Users}\ [\ \text{RESTRICT} \mid \text{CASCADE}\ ]$$

La gestione delle autorizzazioni deve nascondere gli elementi cui un utente non può accedere, senza sospetti.
Per autorizzare un utente a vedere solo alcune ennuple di una relazione si utilizzano le viste.
Infatti dopo aver definito una vista con una condizione di selezione, si attribuiscono le autorizzazioni sulla vista, anziché sulla relazione base.