SQL e linguaggi di programmazione

Angelo Montanari

Dipartimento di Matematica e Informatica
Università di Udine

Introduzione - 1

Obiettivo: accesso a basi di dati SQL da programmi applicativi

Accesso alternativo a:

- accesso mediante **interfaccia interattiva** (ad esempio, il comando SQLPLUS attiva l'interfaccia interattiva in un RDBMS Oracle)
- accesso via **file di comandi** (attraverso l'interfaccia interattiva può essere creato ed eseguito un file di comandi digitando @ $\langle$ nome file $\rangle$)

L'interfaccia interattiva viene utilizzata per la creazione di schemi e per l'esecuzione di interrogazioni occasionali

Introduzione - 2

I programmi applicativi (o applicazioni di basi di dati) vengono usati come **transazioni standard** dagli utenti finali

Altro utilizzo delle tecniche di programmazione per basi di dati: accesso alla base di dati attraverso un programma applicativo che implementa un'**interfaccia web**

La maggior parte delle applicazioni di commercio elettronico sul web include alcuni **comandi** per l'accesso alla base di dati

Approcci alla programmazione per basi di dati: esistono diversi modi di includere interazioni con le basi di dati all'interno di programmi applicativi

Approcci alla programmazione per basi di dati

1. **incapsulamento di comandi** di basi di dati in un linguaggio di programmazione (general-purpose) di alto livello

2. progettazione di un **linguaggio del tutto nuovo** (viene progettato ex-novo un linguaggio di programmazione per basi di dati totalmente compatibile con il modello dei dati e il linguaggio di interrogazione), ad esempio PL/SQL, e suo uso per la stesura di **procedure memorizzate** e **trigger**

3. utilizzo di una **libreria di funzioni** di basi di dati

Incapsulamento di comandi

Il prefisso per l'**Embedded SQL** (SQL immerso) è usualmente la stringa EXEC SQL

Un **precompilatore** (o preprocessore) esamina il codice sorgente del programma per identificare le istruzioni per l'interazione con la base di dati ed estrarle per consentirne l'elaborazione con il DBMS

Tali istruzioni vengono sostituite nel programma con **chiamate di funzione** al codice generato dal DBMS

Procedure memorizzate e trigger - 1

Pur essendo concettualmente distinti, procedure memorizzate (stored procedure) e trigger vengono spesso definiti utilizzando linguaggi quali PL/SQL

Procedure memorizzate: programmi eseguiti dal DBMS su **esplicita richiesta** di applicazioni/utenti. In SQL92 sono definite come procedure con un nome, un insieme di parametri e un corpo costituito da un unico comando SQL, raggruppabili attraverso un meccanismo di moduli. Di fatto, per definirle vengono usati linguaggi più ricchi quali PL/SQL in Oracle e Transact/SQL in Sybase

Trigger: procedure memorizzate nella base di dati attivate **automaticamente** dal DBMS quando vengono eseguite determinate operazioni sulle tabelle

Procedure memorizzate e trigger - 2

Una volta definita, una procedura memorizzata può essere utilizzata **come se** facesse parte dei **comandi SQL predefiniti**.

Esempio.

AssegnaCittà (:Dip $char(20)$, :Città $char(20)$)
$UPDATE$ Dipartimento
SET Città $=$:Città
$WHERE$ Nome $=$:Dip

Al momento della chiamata, ai parametri viene associato un valore:

AssegnaCittà (:NomeDip, :NomeCittà)

dove NomeDip e NomeCittà sono due variabili del programma all'interno del quale la procedura è stata invocata

Libreria di funzioni

Viene messa a disposizione una **libreria di funzioni** per il linguaggio ospite per le chiamate a basi di dati.

Esempi. Funzioni per stabilire il collegamento con la base di dati; funzioni per l'esecuzione di un'interrogazione; funzioni per l'esecuzione di un aggiornamento

I comandi effettivi di interrogazione/aggiornamento, e ogni altra informazione necessaria, sono inclusi come parametri nelle chiamate di funzione (analogie con SQL dinamico). Questo approccio fornisce ciò che è comunemente noto come **API** (Application Programming Interface, o interfaccia per programmi applicativi) per accedere ad una base di dati dai programmi applicativi

Conflitto di impedenza - 1

Il primo e il terzo approccio sono più comuni (molte applicazioni sono scritte in linguaggi di programmazione di altro livello); il secondo approccio è appropriato per applicazioni che hanno una forte interazione con la base di dati.

Il primo e il terzo approccio presentano il problema del **conflitto di impedenza**: problemi che si verificano a causa delle differenze tra il modello della base di dati e il modello del linguaggio di programmazione

Diversità dei **tipi di dato** (dei linguaggi di programmazione e del modello dei dati della base di dati): per ogni linguaggio di programmazione ospite occorre definire un collegamento tra i tipi degli attributi e i tipi compatibili del linguaggio di programmazione (tale collegamento può variare da linguaggio a linguaggio)

Conflitto di impedenza - 2

Diversità delle **strutture dati**: i risultati della maggior parte delle interrogazioni sono insiemi o multinsiemi di tuple e ogni tupla è formata da una sequenza di valori di attributi, mentre i programmi di norma accedono a singoli valori di singole tuple.

Occorre definire un collegamento tra la struttura dei dati dei risultati delle interrogazioni (una tabella) e le strutture dati dei linguaggi di programmazione

In particolare, occorre un meccanismo per scorrere le tuple del risultato di un'interrogazione che consenta di accedere ad una singola tupla alla volta e di estrarre da ogni tupla i valori dei singoli attributi (uso dei **cursori** o delle variabili di iterazione)

Procedure memorizzate - 1

Principali vantaggi:

1. **condivisione** tra applicazioni diverse **di codice di interesse generale** (semplificazione delle applicazioni, che risultano più facili da mantenere; semplificazione, e uniformità, delle modifiche delle applicazioni)

2. **semantica uniforme** di alcune operazioni sulla base di dati

3. **controllo** centralizzato di **vincoli di integrità** non esprimibili a livello di singole tabelle (si vedano i trigger)

Procedure memorizzate - 2

4. **ridurre il traffico di rete** dovuto ad applicazioni remote: il client (programma applicativo) non interagisce con il DBMS server inviando un intero comando per volta, ma tramite una chiamata a procedura remota, eseguita dal DBMS server, ricevendo al termine dell'esecuzione solo il risultato finale (analogia con la Remote Procedure Call, utilizzata nella programmazione di sistemi distribuiti)
5. garantire la **sicurezza dei dati**: a certi utenti si può consentire di utilizzare solo certe procedure per ottenere dei dati senza accedere direttamente alle tabelle (analogie col meccanismo delle viste)

Procedure memorizzate - 3

Le procedure memorizzate introducono dei **problemi** nella progettazione delle basi di dati:

- le metodologie disponibili non supportano lo sviluppo di tali procedure
- occorrono delle nuove figure professionali (amministratore delle procedure, che può essere diverso dall'amministratore dei dati)

I trigger

I **trigger**, previsti in SQL89, non previsti in SQL 92, sono stati reintrodotti in SQL99 (sono sempre stati presenti nei sistemi commerciali più sofisticati)

Essi specificano un'**azione** da eseguire automaticamente (qualora l'eventuale condizione di valutazione risulti soddisfatta) al verificarsi di un'operazione di modifica su una tabella (INSERT, DELETE, UPDATE) e di un'interrogazione (SELECT)

Stretto legame tra trigger e **basi di dati attive**, basate sul modello ECA (Event-Condition-Action), a loro volta strettamente legate ai linguaggi di programmazione logica (Prolog) e ai linguaggi a regole (OPS5)

La struttura di un trigger - 1

Esempio: comando di creazione di un trigger in PL/SQL

```
CREATE TRIGGER nomeDelTrigger
tipoDiTrigger                          # BEFORE o AFTER
(tipoDiOperazione {OR tipoDiOperazione})
[OF nomeAttributo] ON nomeTabella
[FOR EACH ROW]
[WHEN condizione]
programma                              # codice da eseguire
```

La struttura di un trigger - 2

tipoDiTrigger specifica quando il trigger deve essere attivato (BEFORE o AFTER)

tipoDiOperazione specifica il tipo di operazione (SELECT, INSERT, DELETE e UPDATE) che attiva il trigger e la tabella, più, eventualmente, l'attributo, su cui agisce (*nomeTabella* e *nomeAttributo*)

FOR EACH ROW stabilisce quante volte il trigger debba essere attivato (una sola volta oppure tante volte quante sono le righe della tabella interessate dall'operazione)

condizione definisce un'eventuale ulteriore condizione da soddisfare affinché il codice del trigger venga eseguito

Trigger passivi e attivi - 1

Trigger passivi: servono a sollevare un fallimento sotto certe condizioni

Trigger attivi: in corrispondenza di certi eventi, modificano lo stato della base di dati

Utilità dei **trigger passivi**:

1. definire **vincoli di integrità** non esprimibili nel modello dei dati utilizzato (ad esempio, vincoli dinamici/temporali)
2. **controlli** sulle operazioni ammissibili degli utenti basati sui valori dei parametri dei comandi SQL (ad esempio, inserire altri dati solo se il codice del dipartimento è quello dell'utente che esegue l'operazione)

Trigger passivi e attivi - 2

Utilità dei **trigger attivi**:

1. **definire regole di gestione** che consentano di garantire la corretta evoluzione del sistema
2. memorizzare eventi sulla base di dati per ragioni di **controllo** (auditing/logging). *Esempio:* registrare operazioni eseguite su tabelle riservate – di norma solo le operazioni che terminano correttamente (questo può essere visto come un limite)
3. **propagare** gli effetti di operazioni su determinate tabelle
4. mantenere **allineati** dati duplicati
5. mantenere certi **vincoli di integrità** modificando la base di dati in modo opportuno (analogamente a quanto viene fatto dalle politiche di reazione associate alle chiavi esterne)

Uno schema relazionale di esempio

Si considerino le seguenti relazioni:

Cliente(CodiceCliente, CognomeENome, Città, Sconto)
Agenti(CodiceAgente, CognomeENome, Zona, Supervisore, Commissione)
Ordini(NumOrdine, CodiceCliente, CodiceAgente, Prodotto, Data, Ammontare)

e la seguente vista:

```
CREATE VIEW OrdiniPerAgente(CodiceAgente, TotaleOrdini) AS
SELECT CodiceAgente, SUM(Ammontare)
FROM Ordini
GROUP BY CodiceAgente
```

Un esempio di trigger passivo

```
CREATE TRIGGER ControlloFido
BEFORE INSERT ON Ordini
DECLARE DaPagare NUMBER;
BEGIN
SELECT SUM(Ammontare) INTO DaPagare
FROM Ordini
WHERE CodiceCliente = :new.CodiceCliente;
IF DaPagare >= 10000000 - :new.Ammontare
THEN RAISE_APPLICATION_ERROR(-2061, 'fido superato');
END IF;
END;
```

Tale trigger controlla che non vengano accettati ordini da clienti con uno scoperto maggiore di 10000000 (nel trigger :new denota il valore della ennupla da inserire o il valore modificato, :old il valore precedente)

Un esempio di trigger attivo

```
CREATE TABLE Totali                          # vista materializzata
(CodiceAgente CHAR(3) primary key, TotaleOrdini INTEGER);
CREATE TRIGGER esempioTrigger1 # trigger per l'aggiornamento
AFTER INSERT ON Ordini
FOR EACH ROW
DECLARE esiste NUMBER;
BEGIN # si controlla se l'agente dell'ordine è già presente in Totali
SELECT COUNT(*) INTO esiste FROM Totali
WHERE CodiceAgente = :new.CodiceAgente;
IF esiste = 0 # l'agente non è presente; va inserita una nuova riga
THEN INSERT INTO Totali
VALUES (:new.CodiceAgente, :new.Ammontare);
ELSE UPDATE Totali
SET TotaleOrdini = TotaleOrdini + :new.Ammontare
WHERE CodiceAgente = :new.CodiceAgente; END;
```

Un altro esempio di trigger attivo

```
CREATE TRIGGER esempioTrigger2 # trigger per la cancellazione
AFTER DELETE ON Agenti
FOR EACH ROW
BEGIN
DELETE FROM Totali
WHERE CodiceAgente = :old.CodiceAgente;
END;
```

Tale trigger cancella la riga della tabella *Totali* quando il corrispondente agente viene rimosso dalla tabella *Agenti*

Vantaggi e problemi dei trigger

Vantaggi dei trigger:

a. semplificare la stesura dei programmi applicativi (che non devono preoccuparsi dei controlli effettuati dai trigger)

b. introdurre dei controlli, definiti e amministrati centralmente, cui nessuno (utente interattivo e programma applicativo) può sottrarsi

Problemi coi trigger: i sistemi commerciali adottano una diversa semantica per l'attivazione dei trigger e prevedono modalità diverse di interazione fra i meccanismi di attivazione dei trigger e l'esecuzione della transazione che li attiva

Problemi: Granularità - 1

Granularità: se una modifica riguarda più tuple (come può accadere con UPDATE, INSERT, DELETE), il trigger può essere eseguito una sola volta (trigger di comando o istruzione) o tante volte quante sono le tuple interessate dall'esecuzione del comando (trigger di riga)

Oracle consente di scegliere una delle due alternative; Sybase supporta solo trigger di comando (si possono utilizzare le tabelle speciali **inserted** e **deleted** che contengono le righe inserite o cancellate dal comando)

Problemi: Granularità - 2

Un esempio di utilizzo della tabella **deleted**:

```
CREATE TRIGGER esempioTrigger3
AFTER DELETE ON Agenti
BEGIN
DELETE FROM totali
WHERE CodiceAgente IN
(SELECT deleted.CodiceAgente FROM deleted);
END;
```

Problemi: Risoluzione dei Conflitti

Se più trigger sono attivati dal medesimo comando/evento, come deciderne l'**ordine**?

In Oracle l'ordine di attivazione coincide con l'ordine di definizione

In alcuni sistemi è possibile specificare in modo esplicito l'ordine di attivazione

In altri sistemi (ad esempio, ILLUSTRA) l'ordine è stabilito dal sistema

Problemi: Trigger in cascata

L'esecuzione di più **trigger in cascata** pone problemi di coerenza e di terminazione: se l'esecuzione di un trigger $T1$ può attivare l'esecuzione di un trigger $T2$, possono presentarsi delle situazioni critiche (ad esempio, dei **cicli** di attivazioni).

In Sybase è possibile consentire o vietare le attivazioni in cascata; se permesse, si può decidere se consentire o meno attivazioni ricorsive

In Oracle non sono consentite attivazioni ricorsive

Problemi: Quando Eseguire i Trigger? - 1

La valutazione della condizione e l'eventuale esecuzione dell'azione di un trigger di regola vengono compiute immediatamente, come parte della transazione cui appartiene l'evento che attiva il trigger (**considerazione immediata**). Si distinguono due casi: **prima** e **dopo** l'esecuzione dell'evento scatenante il trigger

Altra possibilità è la **considerazione differita**: la valutazione della condizione del trigger e la sua eventuale attivazione vengono compiute al termine dell'esecuzione della transazione che contiene l'evento scatenante

Attenzione: possono esservi più trigger, attivati da eventi distinti, in attesa della valutazione della loro condizione e dell'eventuale esecuzione

Problemi: Quando Eseguire i Trigger? - 2

Utilità della considerazione differita: un *esempio*.

Supponiamo che esista un trigger T che elimina un dipartimento quando esso risulta privo di direttore e che venga eseguita una transazione che elimina un impiegato, direttore di un dipartimento (politica di reazione SET NULL) e, successivamente, assegna al dipartimento un nuovo direttore

Con la considerazione immediata di T, il dipartimento viene eliminato, con la considerazione differita il trigger produce l'effetto voluto (si noti che nel caso di una rimozione del direttore non seguita dalla nomina di un nuovo direttore, entrambe le esecuzioni provocano l'effetto voluto)

Nel caso in esame, disponendo della sola considerazione immediata, occorrerebbe invertire l'ordine delle operazioni: sostituzione del vecchio direttore col nuovo prima ed eliminazione del vecchio direttore poi

Problemi: Interazione con le transazioni - 1

Ulteriore possibilità (basi di dati attive): la valutazione della condizione del trigger e la sua eventuale esecuzione fanno parte di una transazione distinta, generata dalla transazione scatenante (**considerazione separata**)

Vantaggi. Se l'azione del trigger diventa parte della transazione scatenante, la sua esecuzione diventa parte della esecuzione **atomica** della transazione: se l'azione del trigger viene abortita, abortisce anche la transazione e viceversa.

Ciò è spesso, ma **non sempre**, ciò che si vuole.

Specificare la valutazione della condizione di un trigger e la sua eventuale esecuzione come elementi di una transazione separata risolve il problema

Problemi: Interazione con le transazioni - 2

Utilità della considerazione separata: un *esempio*.

Si pensi ai trigger utilizzati per memorizzare operazioni eseguite sulla base di dati per ragioni di controllo: se una transazione legge un certo dato, tale fatto va registrato in un'opportuna tabella anche nel caso in cui la transazione fallisca e tutti i suoi eventuali effetti sulla base di dati vengano cancellati (rollback)

I trigger e la progettazione delle basi di dati

Come le procedure memorizzate, anche i trigger introducono dei **problemi** nella progettazione delle basi di dati:

- le metodologie disponibili non supportano lo sviluppo dei trigger
- occorrono delle nuove figure professionali (amministratore delle procedure memorizzate e dei trigger, che può essere diverso dall'amministratore dei dati)

I trigger e la realizzazione delle applicazioni - 1

In un ambiente in cui sono presenti dei trigger, la realizzazione delle applicazioni deve tener presenti alcuni **problemi** che possono insorgere:

- **complessità** (occorre tener conto sia dell'azione diretta dell'applicazione sulla base di dati, sia della sua azione indiretta tramite i trigger, specie se i trigger sono attivi e possono, a loro volta, provocare l'attivazione di altri trigger)
- **rigidità** (non è possibile non tener conto dei trigger; occorre definire delle strategie di azione tenendo presente l'effetto dell'eventuale attivazione di un trigger)

I trigger e la realizzazione delle applicazioni - 2

Come gestire il problema della **complessità**? Il comportamento di un'applicazione può diventare facilmente imprevedibile qualora non sia possibile controllare quali trigger, e quando, vengano attivati (dato che i trigger attivi modificano lo stato/contenuto della base di dati, il tempo di esecuzione ne influenza, in generale, la semantica)

Accorgimento: impedire l'attivazione mutuamente ricorsiva di più trigger

Come gestire il problema della **rigidità**? Introdurre delle regole di attivazione, disattivazione ed eliminazione dei trigger (basi di dati attive)

Interazione tra programma e base di dati - 1

Interazione tra programma e base di dati (modello client/server): un **programma client** gestisce la logica del programma applicativo ed effettua delle chiamate a uno o più **server di basi di dati** per l'accesso ai/aggiornamento dei dati

Tipica **sequenza di interazione**:

1. il programma stabilisce (o apre) una connessione al server della base di dati (specifica dell'indirizzo internet/URL della macchina ove si trova il server della base di dati, il nome di registrazione dell'utente e la relativa password);
2. stabilita la connessione, vengono eseguiti i comandi SQL desiderati (interrogazioni, aggiornamenti,..);
3. il programma termina (o chiude) la connessione.

Interazione tra programma e base di dati - 2

In alcuni ambienti di programmazione di basi di dati può essere attiva **una sola** connessione per volta, in altri possono essere attive contemporaneamente **più** connessioni.

Un *esempio* (Embedded SQL).

Il comando SQL per **stabilire** una connessione ha la forma:
CONNECT TO *nomeServer* AS *nomeConnessione*
AUTHORIZATION *nomeAccountUtenteEPassword*

In generale, possono essere stabilite più connessioni con più server di basi di dati, ma in un determinato momento solo una può essere attiva.
nomeConnessione può essere usato per **cambiare** la connessione attiva col comando SQL:
SET CONNECTION *nomeConnessione*

Il comando SQL per **terminare** una connessione ha la forma:
DISCONNECT *nomeConnessione*

Embedded SQL - 1

Come incapsulare comandi SQL in un linguaggio di programmazione general-purpose (**linguaggio ospite**)?

Ogni comando SQL è preceduto dalle parole chiave EXEC SQL per consentire ad un preprocessore (o precompilatore) di **separare** i comandi SQL incapsulati dal codice del linguaggio ospite. I comandi SQL terminano con ';' o con END-EXEC

Dal punto di vista dell'**implementazione**, occorre far precedere la compilazione del linguaggio di alto livello dall'esecuzione di un apposito preprocessore che individuerà i comandi SQL e li sostituirà con un opportuno insieme di chiamate a servizi del DBMS, utilizzando una libreria specifica per ogni sistema (il preprocessore predisporrà un opportuno insieme di strutture dati ausiliarie richieste per l'esecuzione dello specifico comando SQL)

Embedded SQL - 2

Affinché tale approccio sia concretamente applicabile, deve essere disponibile un preprocessore per ogni specifica **combinazione** di DBMS, piattaforma, linguaggio, compilatore utilizzata (ad esempio, per la combinazione Oracle, Solaris, C, gcc)

I sistemi commerciali più diffusi mettono a disposizione soluzioni di questo tipo per le configurazioni d'uso più frequente (ad esempio, il preprocessore ECPG di SQL Embedded per il linguaggio C e il DBMS Postgres)

Considereremo il caso del linguaggio C.
Le variabili del linguaggio C possono essere riferite in un comando SQL incapsulato facendole precedere da ':' (tali variabili vengono dette **variabili condivise**); ciò consente di distinguere le variabili del programma dai nomi di relazione e di attributo

Dichiarazione delle variabili - 1

Nel seguito proporremo alcuni esempi relativi all'usuale dominio aziendale (cf. lucidi modello relazionale), assumendo che la connessione appropriata sia già stata stabilita e che sia la connessione attiva

La seguente dichiarazione stabilisce una corrispondenza tra le variabili del programma e gli attributi delle relazioni presenti nella base di dati (i successivi segmenti di programmi C non conterranno dichiarazioni delle variabili)

Le variabili condivise sono dichiarate all'interno della **sezione dichiarativa**

Dichiarazione delle variabili - 2

```
int loop; # variabile C non utilizzata in ambiente SQL
EXEC SQL BEGIN DECLARE SECTION;
varchar nome_d [16], nome_batt [16], cognome [16], indirizzo [31];
# i vettori in C sono più lunghi di un carattere rispetto alle
# corrispondenti stringhe in SQL (carattere finale '\0' (null))
char ssn [10], data_n [11], sesso [2], iniz_int [2];
float stipendio, aumento;
int n_d, numero_d;
int SQLCODE;
char SQLSTATE [6];
EXEC SQL END DECLARE SECTION;
```

Le variabili SQLCODE e SQLSTATE vengono usate per la comunicazione di errori e condizioni di eccezione da base di dati a programma

Le variabili SQLCODE e SQLSTATE - 1

La **variabile SQLCODE** è una variabile intera. Terminata l'esecuzione di un comando SQL, il DBMS restituisce un valore in SQLCODE

- se è uguale a 0, il comando è stato eseguito con successo
- se è maggiore di 0 (SQLCODE = 100), significa che non sono disponibili altri dati (record) nel risultato dell'interrogazione
- se è minore di 0, si è verificato un errore

In alcuni sistemi (ad esempio, Oracle) SQLCODE è un campo di un record SQLCA (SQL Communication Area) e va riferito come SQLCA.SQLCODE. In tal caso, la definizione di SQLCA va inclusa nel programma C:

EXEC SQL include SQLCA;

Le variabili SQLCODE e SQLSTATE - 2

La **variabile SQLSTATE** è una variabile di tipo stringa di lunghezza 5.

Il valore 00000 indica l'assenza di errori ed eccezioni.

Errori ed eccezioni sono codificati con valori diversi da 00000. Ad esempio, 02000 indica la 'fine dati'

Attualmente sia SQLCODE che SQLSTATE sono parte dello standard SQL. Si suppone, inoltre, che molti codici di errore e di eccezione restituiti in SQLSTATE siano standardizzati per tutti i fornitori e le piattaforme SQL, mentre i codici restituiti in SQLCODE sono definiti dallo specifico fornitore di DBMS. Ne segue che l'uso di SQLSTATE è da preferirsi all'uso di SQLCODE.

La programmazione in Embedded SQL

Distingueremo due casi:

- recupero di **singole tuple**
- recupero di **insiemi di tuple** mediante **cursori**

Un *esempio*.
Il seguente segmento di programma (ciclo) legge un numero di previdenza sociale di un impiegato e stampa alcune informazioni recuperate dalla corrispondente tupla della tabella IMPIEGATO

La **clausola INTO**, che compare nell'interrogazione SQL, specifica le variabili del programma cui vengono assegnati i valori degli attributi della tupla recuperata

Recupero di singole tuple (interrogazioni scalari)

```
# segmento di programma P1
loop = 1;
while (loop) {
prompt(″Inserire un numero di previdenza sociale: ″, ssn);
EXEC SQL
SELECT NOME_BATT, INIZ_INT, COGNOME, INDIRIZZO,
STIPENDIO
INTO :nome_batt, :iniz_int, :cognome, :indirizzo, :stipendio
FROM IMPIEGATO  WHERE SSN = :ssn;
if (SQLCODE == 0) printf(nome_batt, iniz_int, cognome, indirizzo,
stipendio)
else printf(″Non esiste il numero di previdenza sociale: ″, ssn);
prompt(″ nuovo numero di previdenza sociale (immettere 1 per sì, 0 per
no): ″, loop);
}
```

I cursori - 1

Un cursore può essere visto come un **puntatore** ad una singola tupla del risultato di un'interrogazione che restituisce più tuple

Il **cursore** viene **dichiarato** quando viene dichiarato il comando di interrogazione (tale dichiarazione non è necessaria nel caso di interrogazioni che restituiscono una singola tupla). Tale dichiarazione non esegue l'interrogazione. Il comando **OPEN** *nomeCursore* **esegue** l'interrogazione e restituisce il risultato, posizionando il cursore prima della prima riga del risultato; tale riga (fittizia) diventa la **riga corrente** per il cursore

Vengono, quindi, eseguiti i **comandi di FETCH** che spostano il cursore sulla riga successiva, rendendola la riga corrente e assegnando i valori dei suoi attributi alle corrispondenti variabili del programma (clausola INTO del comando FETCH)

I cursori - 2

Per stabilire quando la scansione del risultato è terminata, viene controllata la **variabile SQLCODE** (o **SQLSTATE**): se l'esecuzione del comando FETCH sposta il cursore oltre l'ultima riga, in SQLCODE viene restituito un valore positivo

Si possono aprire contemporanamente **più cursori**. Il comando CLOSE CURSOR dichiara chiusa l'elaborazione del risultato dell'interrogazione associata al cursore

Un *esempio*.
Il seguente segmento di programma utilizza un cursore per scandire l'insieme degli impiegati restituiti dall'interrogazione

Recupero di insiemi di tuple - 1

```
# segmento di programma P2
prompt(″Inserire il nome del dipartimento: ″, nome_d);
EXEC SQL
SELECT NUMERO_D  INTO :numero_d
FROM DIPARTIMENTO  WHERE NOME_D = :nome_d;
EXEC SQL DECLARE emp CURSOR FOR
SELECT SSN, NOME_BATT, INIZ_INT, COGNOME, STIPENDIO
FROM IMPIEGATO
WHERE N_D = :numero_d
FOR UPDATE OF STIPENDIO;
EXEC SQL OPEN emp;
EXEC SQL FETCH FROM emp INTO :ssn, :nome_batt, :iniz_int,
:cognome, :stipendio;
```

Recupero di insiemi di tuple - 2

```
while (SQLCODE == 0) {
printf(″Il nome dell'impiegato è: ″, nome_batt, iniz_int, cognome);
prompt(″ inserire l'importo dell'aumento: ″, aumento);
EXEC SQL
UPDATE IMPIEGATO
SET STIPENDIO = STIPENDIO + :aumento
WHERE CURRENT OF emp;
EXEC SQL FETCH FROM emp INTO :ssn, :nome_batt, :iniz_int,
:cognome, :stipendio;
}
EXEC SQL CLOSE emp
```

Cursori e aggiornamenti

Quando viene definito un cursore per eseguire una modifica, è necessario aggiungere la **clausola FOR UPDATE OF** nella dichiarazione del cursore ed elencare i nomi degli attributi che verranno aggiornati dal programma. Se devono essere cancellate alcune righe, vanno aggiunte le parole chiave **FOR UPDATE** senza specificare alcun attributo.
I comandi di modifica e cancellazione sono utilizzabili solo nel caso in cui il cursore consenta di accedere ad una riga effettiva di una tabella (non in presenza di join tra tabelle)

La clausola **FOR UPDATE OF** non va inserita se i risultati dell'interrogazione non vanno usati in operazioni di aggiornamento

Nel comando UPDATE (o DELETE) incapsulato, la condizione **WHERE CURRENT OF** *nomeCursore* specifica che la riga corrente a cui fa riferimento il cursore è la riga da aggiornare (o cancellare)

Dichiarazione dei cursori - 1

La forma generale della **dichiarazione di un cursore** è la seguente (abbiamo omesso alcune opzioni minori che fanno riferimento a proprietà legate alle transazioni):

DECLARE *nomeCursore* [SCROLL] CURSOR
FOR *corpoInterrogazione*
[ORDER BY *specificaOrdinamento*]
[FOR READ ONLY | FOR UPDATE [OF *elencoAttributi*]]

dove

FOR READ ONLY: solo recupero dati

FOR UPDATE [OF]: le tuple devono essere cancellate (modificate)

Dichiarazione dei cursori - 2

SCROLL (opzionale) consente di posizionare il cursore in modo diverso rispetto a quello base usato per la scansione sequenziale a partire dalla prima riga del risultato

In questo caso, è possibile associare al comando FETCH un **orientamento dell'estrazione** che può assumere i valori: NEXT (default), PRIOR, FIRST, LAST, ABSOLUTE i, RELATIVE i

Le prime quattro opzioni hanno un significato ovvio. In ABSOLUTE i, il valore i è un intero che specifica una posizione assoluta del cursore all'interno della sequenza di righe risultato; in RELATIVE i, il valore i specifica una posizione in modo relativo, facendo riferimento alla posizione corrente del cursore

Dichiarazione dei cursori - 3

In presenza della clausola SCROLL nella dichiarazione del cursore, il **comando FETCH** può assumere la seguente forma:

FETCH [[orientamentoEstrazione]
FROM] nomeCursore
INTO elencoVariabili

La clausola ORDER BY ordina le tuple del risultato in modo che il comando FETCH le estragga in uno specifico ordine (viene specificata come la clausola ORDER BY delle interrogazioni SQL)

SQL statico e dinamico

SQL statico: il comando SQL è parte del codice sorgente del programma (opzione sin qui presa in considerazione)

SQL dinamico: il comando SQL viene incorporato (ed eseguito) dinamicamente durante l'esecuzione del programma (occorre un supporto speciale da parte del sistema).

Esempio: un programma che accetta un'interrogazione SQL digitata dall'utente, la esegue e visualizza il risultato ottenuto (analogia con le interfacce interattive messe a disposizione dai DBMS)

Altro esempio: un'interfaccia amichevole che genera dinamicamente interrogazioni per l'utente sulla base di operazioni di puntamento e selezione (point-and-click) attraverso un'interfaccia grafica

SQL dinamico: un esempio

```
# segmento di programma P3


EXEC SQL BEGIN DECLARE SECTION ;
varchar sqlupdatestring [256];
EXEC SQL END DECLARE SECTION;
...
prompt("Inserire il comando di update: ", sqlupdatestring);
EXEC SQL PREPARE sqlcommand FROM :sqlupdatestring;
EXEC SQL EXECUTE sqlcommand
```

Il programma P3 legge una stringa inserita dall'utente (un comando di UDPDATE), la prepara (associandola alla variabile SQL *sqlcommand*) e la esegue

SQL dinamico

In SQL dinamico non è **possibile** alcun **controllo** della sintassi dei comandi SQL (né sono possibili altri tipi di controllo) in fase di compilazione, in quanto i comandi vengono resi disponibili solo in fase di esecuzione

Aggiornamenti vs. interrogazioni: l'esecuzione degli **aggiornamenti** in SQL dinamico è relativamente semplice (non restituiscono un risultato), quella delle **interrogazioni** è molto più complicata (non si conosce il numero e il tipo di attributi restituiti dall'interrogazione)

Qualora non sia disponibile a priori alcuna informazione sulla struttura attesa del comando, può essere necessaria una struttura dati complessa in grado di memorizzare un numero variabile di attributi di tipo diverso

Esecuzione immediata e differita

Se un comando deve essere eseguito più volte (più **comandi EXECUTE**), il **comando PREPARE** consente di 'prepararlo' una sola volta.

La preparazione consiste nel controllo della sintassi del comando, più altri controlli di sistema, e nella creazione del codice necessario per eseguire il comando.

I comandi PREPARE ed EXECUTE possono essere integrati nell'unico **comando IMMEDIATE EXECUTE**:

EXEC SQL IMMEDIATE EXECUTE :sqlupdatestring;

Esecuzione differita - 1

Il comando SQL argomento di una PREPARE può contenere dei parametri di ingresso, rappresentati dal carattere col punto interrogativo.

Ad esempio:

PREPARE sqlcommand FROM ″SELECT CITTÀ FROM DIPARTIMENTO WHERE NOME = ?″

Un comando preparato può essere rimosso col comando DEALLOCATE PREPARE.

Ad esempio:

DEALLOCATE PREPARE sqlcommand

Esecuzione differita - 2

Per eseguire un comando elaborato con una PREPARE si usa il comando EXECUTE che ha il seguente formato generale:

EXECUTE nomeComando [INTO listaTarget] [USING listaParametri]

dove *listaTarget* contiene l'elenco dei parametri ove scrivere il risultato dell'esecuzione del comando e *listaParametri* specifica i valori che devono essere assunti dai parametri variabili della lista

Ad esempio:

EXECUTE sqlcommand INTO :città USING :dipartimento

Se alla variabile *dipartimento* viene assegnato quale valore la stringa *Produzione*, si ottiene l'interrogazione

SELECT CITTÀ FROM DIPARTIMENTO
WHERE NOME = 'Produzione'

Incapsulamento di comandi SQL in Java (cenni)

Il linguaggio SQL può essere incapsulato anche in un linguaggio di programmazione orientato agli oggetti, in particolare in Java

SQLJ è uno standard adottato da diversi produttori per incapsulare SQL in Java

Storicamente, SQLJ è stato sviluppato dopo JDBC, usato per accedere a basi di dati SQL da Java usando chiamate di funzione

Funzionamento di SQLJ: un traduttore SQLJ converte le istruzioni SQL in chiamate Java che vengono eseguite attraverso l'interfaccia JDBC (occorre, quindi, aver installato un driver JDBC per poter utilizzare SQLJ)

Utilizzo di una libreria di funzioni

Utilizzo di una libreria di funzioni: programmazione per basi di dati con chiamate di funzione

Rispetto all'Embedded SQL statico, l'uso delle chiamate di funzione è un approccio dinamico (alternativo all'SQL dinamico)

Per accedere alla base di dati viene usata una **libreria di funzioni**, nota col nome di API (Application Programming Interface)

È un approccio **più flessibile** (**non** è necessario alcun **preprocessore**), ma la verifica della sintassi e altri controlli sui comandi SQL possono essere eseguiti solo al **momento dell'esecuzione**.

Inoltre, l'uso delle API richiede alle volte una **programmazione più complessa** (tipo e numero degli attributi del risultato possono non essere noti anticipo)

SQL/CLI (Call Level Interface)

SQL/CLI è un'interfaccia di chiamate di funzione che perfeziona la tecnica precedente nota come **ODBC** (Open Data Base Connectivity) e fa parte dello standard SQL

Principale **vantaggio** di tali interfacce: semplificare l'accesso a più basi di dati da uno stesso programma applicativo anche se memorizzate in DBMS diversi

Prima di usare le chiamate di funzione in SQL/CLI è necessario installare sul server della base di dati le librerie appropriate (si ottengono dal fornitore del sistema DBMS utilizzato)

Nel seguito mostreremo come usare SQL/CLI in un programma C

Uso di SQL/CLI in un programma C

Le istruzione SQL vengono create dinamicamente e passate come parametri stringa nelle chiamate di funzione

Al momento dell'esecuzione (i comandi alla base di dati vengono elaborati in tale momento), si tiene traccia delle informazioni relative alle **interazioni** del programma ospite con la base di dati in apposite strutture dati

Le informazioni vengono mantenute in **quattro tipi di record**, rappresentati come strutture C: record d'ambiente, record di connessione, record di istruzione e record di descrizione

Ogni record è accessibile al programma attraverso una variabile C di tipo puntatore denominata **handle** al record

Tipi di record

- **record d'ambiente**: tiene traccia di una o più connessioni a basi di dati; viene usato per impostare le informazioni relative all'ambiente di esecuzione del programma
- **record di connessione**: tiene traccia delle informazioni relative ad una particolare connessione ad una base di dati
- **record di istruzione**: tiene traccia delle informazioni necessarie per l'istruzione SQL da eseguire
- **record di descrizione**: tiene traccia delle informazioni sulle tuple o sui parametri (ad esempio, numero di attributi in una tupla e loro tipi, numero di parametri in una chiamata di funzione e loro tipi)

Handle al record

Lo handle viene restituito quando viene creato il record. Per creare un record e restituire il suo handle, si usa la seguente funzione SQL/CLI: **SQLAllocHandle(tipoHandle, handle1, handle2)**

dove

tipoHandle specifica il tipo di record creato e può assumere i valori SQL_HANDLE_ENV (record d'ambiente), SQL_HANDLE_DBC (record di connessione), SQL_HANDLE_STMT (record di istruzione) e SQL_HANDLE_DESC (record di descrizione)

handle1 è il contenitore all'interno del quale è stato creato il nuovo handle (ad esempio, per un record di connessione, l'ambiente in cui viene creata la connessione; per un record di istruzione, la connessione)

handle2 è il puntatore (handle) al record appena creato di tipo *tipoHandle*

Un esempio d'uso di SQL/CLI - 1

Il seguente programma legge il numero di previdenza sociale di un impiegato e stampa il suo cognome e il suo stipendio

```
# include sqlcli.h;
void printSal() {
SQLHSTMT stmt1; SQLHDBC con1; SQLHENV env1;
SQLRETURN ret1, ret2, ret3, ret4;
ret1 = SQLAllocHandle(SQL_HANDLE_ENV,
SQL_NULL_HANDLE, &env1);
if (!ret1) ret2 = SQLAllocHandle(SQL_HANDLE_DBC, env1,
&con1) else exit;
if (!ret2) ret3 = SQLConnect(con1, ″dbs″, SQL_NTS, ″js″,
SQL_NTS, ″xyz″, SQL_NTS) else exit;
```

Un esempio d'uso di SQL/CLI - 2

```
if (!ret3) ret4 = SQLAllocHandle(SQL_HANDLE_STMT, con1,
&stmt1) else exit;
SQLPrepare(stmt1, "select COGNOME, STIPENDIO from
IMPIEGATO where SSN = ?", SQL_NTS);
prompt("Inserire un numero di previdenza sociale:", ssn);
SQLBindParameter(stmt1, 1, SQL_CHAR, &ssn, 9, &fetchlen1);
ret1 = SQLExecute(stmt1);
if (!ret1) {
SQLBindCol(stmt1, 1, SQL_CHAR, &cognome, 15, &fetchlen1);
SQLBindCol(stmt1, 2, SQL_FLOAT, &stipendio, 4, &fetchlen2);
ret2 = SQLFetch(stmt1);
if (!ret2) printf(ssn, cognome, stipendio) else printf("Non esiste il
numero di previdenza sociale:", ssn);
} }
```

Le fasi tipiche

Quando si scrive un programma C che contiene chiamate a basi di dati attraverso SQL/CLI bisogna eseguire le seguenti **fasi tipiche**

Per prima cosa, la **libreria di funzioni** che comprende SQL/CLI viene inclusa nel programma C attraverso il comando:

include sqlcli.h;

Poi si dichiarano le **variabili** dei tipi *SQLHSTMT*, *SQLHDBC*, *SQLHENV* e *SQLHDESC* rispettivamente per istruzioni, connessioni, ambienti e descrizioni necessarie al programma. Nell'esempio,

SQLHSTMT stmt1; SQLHDBC con1; SQLHENV env1;

Impostazione del record d'ambiente

Inoltre, si dichiarano le variabili di tipo *SQLRETURN* necessarie per contenere i **codici di ritorno** delle chiamate di funzione SQL/CLI (un codice di ritorno uguale a 0 indica esecuzione con successo della chiamata)

SQLRETURN ret1, ret2, ret3, ret4;

Il **record d'ambiente** viene impostato usando *SQLAllocHandle*. Poiché il record d'ambiente non è contenuto in alcun altro record, quando si crea un ambiente il parametro *handle1* è lo handle vuoto *SQL_NULL_HANDLE* (puntatore nullo). Lo handle (puntatore) al record d'ambiente appena creato viene restituito nella variabile *env1*

ret1 = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env1);

La gestione della connessione

Il **record di connessione** viene impostato usando *SQLAllocHandle*. Nell'esempio il record di connessione creato ha lo handle *con1* ed è contenuto nell'ambiente *env1*

if (!ret1) ret2 = SQLAllocHandle(SQL_HANDLE_DBC, env1, &con1) else exit;

In *con1* viene **stabilita una connessione** ad una particolare base di dati usando la funzione *SQLConnect* di SQL/CLI

if (!ret2) ret3 = SQLConnect(con1, ″dbs″, SQL_NTS, ″js″, SQL_NTS, ″xyz″, SQL_NTS) else exit;

dove *dbs* è il nome del server della base di dati cui ci si connette, *js* è il nome dell'account per la connessione, *xyz* è la password per la connessione e la parola chiave SQL_NTS funge da separatore

Preparazione dell'istruzione - 1

Viene impostato un **record di istruzione** usando *SQLAllocHandle*. Tale record ha lo handle *stmt1* e usa la connessione *con1*

if (!ret3) ret4 = SQLAllocHandle(SQL_HANDLE_STMT, con1, &stmt1) else exit;

L'**istruzione** viene **preparata** usando la funzione *SQLPrepare* di SQL/CLI

SQLPrepare(stmt1, ″select COGNOME, STIPENDIO from IMPIEGATO where SSN = ?″, SQL_NTS);

che assegna la stringa che definisce l'istruzione SQL allo handle di istruzione *stmt1*

? è un **parametro**, ossia un valore da determinare al momento dell'esecuzione del programma, di solito collegandolo ad una variabile del programma C

Preparazione dell'istruzione - 2

In generale vi possono essere più parametri, che vengono distinti sulla base dell'ordine di apparizione dei corrispondenti ? nell'istruzione

L'ultimo parametro di *SQLPrepare* dovrebbe specificare la lunghezza della stringa che definisce l'istruzione SQL espressa in byte; se si immette la parola chiave *SQL_NTS*, ciò indica che la stringa che specifica l'istruzione SQL è una **stringa terminata con *null*** e SQL dovrà calcolare automaticamente la lunghezza della stringa (ciò si applica anche agli altri parametri di tipo stringa nelle chiamate di funzione)

Collegamento parametri/variabili

Prima di eseguire l'interrogazione, i parametri vanno **collegati** alle variabili di programma usando la funzione *SQLBindParameter* di SQL/CLI

prompt(″Inserire un numero di previdenza sociale:″, ssn);
SQLBindParameter(stmt1, 1, SQL_CHAR, &ssn, 9, &fetchlen1);

dove *stmt1* specifica che si tratta di un parametro dell'istruzione *stmt1*, *1* specifica la posizione del parametro, *SQL_CHAR* definisce il tipo del parametro, *ssn* è la variabile che viene collegata al parametro e *9* è la dimensione del parametro

Se l'istruzione SQL contiene n parametri, vanno eseguite n chiamate della funzione *SQLBindParameter*, ciascuna con una diversa posizione del parametro $(1, 2, \ldots, n)$

Esecuzione dell'istruzione - 1

Si può, quindi, **eseguire** l'istruzione SQL riferita dallo handle *stmt1* usando la funzione *SQLExecute* di SQL/CLI:

ret1 = SQLExecute(stmt1);

L'esecuzione dell'istruzione **non** comprende l'assegnamento dei risultati ad una o più variabili del programma C

Per stabilire dove restituire il risultato di un'interrogazione, una tecnica comune è quella delle **colonne di collegamento**: ogni colonna del risultato è collegata ad una variabile del programma C usando la funzione *SQLBindCol* di SQL/CLI:

SQLBindCol(stmt1, 1, SQL_CHAR, &cognome, 15, &fetchlen1);
SQLBindCol(stmt1, 2, SQL_FLOAT, &stipendio, 4, &fetchlen2);

Esecuzione dell'istruzione - 2

Per **copiare** i valori delle colonne nelle variabili del programma C viene usata la funzione *SQLFetch* di SQL/CLI:

ret2 = SQLFetch(stmt1);

Tale funzione è simile al comando FETCH dell'Embedded SQL.

Se il risultato di un'interrogazione è un insieme di tuple, ogni chiamata di *SQLFetch* raggiunge la tupla successiva e copia i suoi valori di colonna nelle variabili del programma collegate. *SQLFetch* restituisce un codice di eccezione (diverso da 0) se non vi sono più tuple

Al posto delle colonne collegate, è possibile utilizzare una tecnica alternativa che usa funzioni SQL/CLI diverse (SQLGetCol e SQLGetData) per recuperare le colonne del risultato dell'interrogazione senza doverle prima collegare; tali funzioni vengono applicate dopo il comando SQLFetch