Azzolini Riccardo 2020-02-20

Object-oriented data model

1 Contesto storico

Il modello dei dati **object-oriented** è stato introdotto circa trent'anni fa, quando:

- il modello più utilizzato per le basi di dati era quello relazionale "puro", "piatto";
- i linguaggi orientati agli oggetti esistevano già da una ventina d'anni (Smalltalk, C++, ecc., ma non ancora Java): essi permettevano una rappresentazione molto più espressiva dei dati, attraverso concetti che il modello relazionale piatto non supporta direttamente, e, inoltre, consentivano di associare ai tipi di dati delle azioni (i metodi).

In questo contesto, è nata l'idea di partire da un linguaggio ad oggetti, ed estenderlo per ottenere un "vero e proprio" modello dei dati:

- aggiungendo un linguaggio d'interrogazione, simile a SQL (dato che esso è particolarmente facile da usare, soprattutto per le interrogazioni semplici);
- implementando la persistenza, una delle caratteristiche fondamentali dei DBMS (che sono, appunto, le implementazioni dei modelli dei dati); invece, al termine di un programma scritto in un "normale" linguaggio ad oggetti, tutti i dati che non sono stati salvati manualmente (ad esempio, in un file) "spariscono".

2 Caratteristiche del modello

In realtà, non esiste un "singolo" modello object-oriented: esistono varie implementazione (per i principali linguaggi OO), ciascuna di esse diversa dalle altre.

In generale, comunque, i "capisaldi" di questo tipo di modello sono:

- Classi dotate di uno **stato** (i dati che ne caratterizzano le istanze) e di un **comportamento** (le operazioni associate alle istanze, cioè i metodi).
- **Persistenza**: meccanismi che garantiscono l'esistenza dei dati indipendentemente dalle applicazioni che li usano. Però, a differenza dei DBMS relazionali, nei quali la persistenza è automatica e trasparente, i DBMS object-oriented richiedono che essa venga specificata in modo esplicito, tramite apposite parole chiave, all'interno della definizione dello schema.

3 Definizione della struttura dei dati

Per la definizione della struttura dei dati si usa un **ODL** (**Object Definition Language**), che svolge un ruolo analogo al DDL di un DBMS relazionale.

3.1 Tipi

L'ODL permette di definire tipi complessi, anche arbitrariamente nidificati: oltre ad attributi "piatti", un tipo può contenere dei **costruttori di tipo**, come ad esempio:

- **tuple**, paragonabili ai record/struct disponibili in vari linguaggi di programmazione;
- liste, insiemi, ecc., analoghi (ad esempio) ai tipi collezione di Java.

3.1.1 Esempio

Un esempio di definizione di un tipo (in una sintassi semplificata) è:

```
define type Department
  tuple( name:      string;
         number:    integer;
         head:      tuple( manager: Employee;
                           startDate: Date;
                         );
         locations: set(string);
       );
```

Questo tipo è formato da una tupla con vari attributi. In particolare, l'attributo `head` è a sua volta costituito da una tupla, e gli attributi di quest'ultima sono riferimenti a istanze di altri tipi complessi (`Employee` e `Date`). Si ha quindi una struttura gerarchica.

3.2 Classi

La definizione di un tipo specifica solo lo stato delle sue istanze, e non il comportamento. In particolare, se si vogliono effettuare operazioni di scrittura/aggiornamento, è necessario creare una classe, che, oltre allo stato, specifica anche le operazioni (metodi). La definizione di una classe comprende infatti le dichiarazioni di:

- tipo di stato;
- segnature delle operazioni (*senza implementazione*, come se fossero metodi astratti).

Le operazioni indicate nella classe devono poi essere implementate *nel linguaggio ospite* su cui si basa il DBMS (e non nel linguaggio d'interrogazione).

3.2.1 Esempio

Come esempio, vengono aggiunte delle operazioni al tipo `Department` definito in precedenza, trasformandolo in una classe (rappresentata ancora con una sintassi semplificata):

```
define class Department
  type tuple ( name:       string;
               ...
             );
  operations   createDept: Department;
               ...
```

L'unica operazione qui mostrata, `createDept`, non ha argomenti e restituisce un oggetto di tipo `Department`.

Un altro esempio è la definizione di una classe corrispondente a un insieme di dipartimenti:

```
define class SetOfDepartments
  type       set(Department);
  operations addDept(d: Department):    boolean;
             removeDept(d: Department): boolean;
```

Le due operazioni di questa classe (per l'aggiunta/rimozione di oggetti all'insieme) hanno ciascuna un argomento di tipo `Department`, e restituiscono valori booleani.

3.3 Persistenza

Di default, uno schema composto da tipi/classi è "volatile": quando si arresta il DBMS, tutti i dati vanno persi.

Per specificare la persistenza di determinate informazioni, è necessario creare degli **extent** ("estensioni", materializzazioni) per le classi corrispondenti, assegnando a essi dei nomi. Un extent è quindi un *named persistent object* (oggetto persistente con nome), al quale dovranno fare riferimento tutte le operazioni sui dati che si vuole siano persistenti.

In pratica, si ha un disaccoppiamento tra la definizione dei tipi e la specifica della persistenza. Questa complicazione era, al tempo, necessaria: sarebbe stato troppo inefficiente

materializzare in modo automatico tutti i dati, poiché i tipi potrebbero essere arbitrariamente complessi. La decisione di cosa materializzare, che non è banale, è allora stata affidata alle capacità progettuali di chi crea lo schema. Ciò nonostante, le performance dei DBMS object-oriented sono sempre rimaste inferiori a quelle dei DBMS relazionali.[1]

3.3.1 Esempio

Sempre usando una sintassi semplificata, nell'esempio seguente viene definito un extent per la classe `SetOfDepartments`, al fine di rendere persistente l'insieme di tutti i dipartimenti. Il nome assegnato a tale extent è `AllDepartments`.

```
define class SetOfDepartments
  type       set(Department);
  operations addDept(d: Department):    boolean;
             removeDept(d: Department): boolean;
  extent     AllDepartments:            SetOfDepartments;
```

4 Interrogazioni

Il linguaggio di interrogazione standard, **OQL** (**Object Query Language**, definito dall'ente ODMG), cerca di rifarsi al modello dichiarativo di SQL, detto **SFW**, "`SELECT FROM WHERE`", che consiste nella specifica di:

- dove prendere i dati (`FROM`);
- le proprietà che i dati restituiti devono soddisfare (`WHERE`);
- un ulteriore raffinamento del risultato, mediante la clausola (`SELECT`).

A differenza di SQL, però, OQL deve consentire anche la navigazione di una struttura complessa di dati. A tale scopo, esso impiega le **path expression**, con la stessa sintassi (`oggetto.attributo`, detta *dot notation*) utilizzata tipicamente dai linguaggi ad oggetti.

4.1 Esempio

La seguente interrogazione restituisce i nomi dei capi di tutti i dipartimenti memorizzati nell'extent `AllDepartments`:

```
select d.head.name
from   d in AllDepartments;
```

[1] Grazie alla sua semplicità, il modello relazionale si presta a numerose ottimizzazioni: questo è proprio uno dei principali vantaggi di tale modello.

4.2 Risultati delle interrogazioni

Un'altra differenza rispetto a SQL è il tipo restituito dalle interrogazioni. Nel caso di SQL, infatti, tutte le operazioni sono eseguite sulle tabelle, perciò è naturale che anche le interrogazioni restituiscano tabelle, le quali possono allora essere facilmente riutilizzate per altre interrogazioni.

Invece, per un database object-oriented, di default il tipo restituito è un **multi-insieme**: un insieme non ordinato di valori che possono ripetersi. È anche possibile specificare tipi diversi, con determinate conseguenze sul risultato: ad esempio, se si specifica un insieme (set), vengono rimossi i duplicati.[2]

Da un lato, quindi, si ha una complessità aggiuntiva nel riuso dei risultati per altre interrogazioni, ma, dall'altro, la flessibilità dei tipi restituiti permette la **ristrutturazione delle risposte**: è possibile costruire un risultato che abbia una struttura arbitraria, indipendente dalla struttura dei dati presenti nel database.[3]

4.3 Raggiungibilità dei dati

Dato uno schema (classi con struttura arbitraria), del quale possono essere materializzate anche solo alcune parti, non è garantito che tutti i dati siano raggiungibili dalle query.[4]

Ad esempio, se

- si hanno due classi, A e B, legate da un'associazione;[5]
- le istanze di A sono materializzate;
- le istanze di B vengono calcolate a runtime, quando è necessario accedervi;

allora:

- una query che "parte" da A, e accede alle istanze di B associate, funziona correttamente;
- una query che, invece, "parte" dalle istanze di B, non materializzate, *non funziona.*

La scelta dei dati da rendere persistenti deve quindi tenendo anche conto delle interrogazioni che si effettueranno: da essa non dipendono solo le prestazioni del database, ma anche la correttezza delle interrogazioni. Di conseguenza, la fase di progettazione risulta più complessa rispetto a quanto lo sia per un database relazionale.

[2] Quale degli oggetti duplicati venga tenuto, e quali vengano invece rimossi, non è specificato dallo standard, ma dipende dall'implementazione.
[3] Questa possibilità è "portata all'estremo" con i modelli semi-strutturati, come ad esempio XML.
[4] Con SQL, invece, qualsiasi dato può sempre essere raggiunto: è sufficiente specificare la tabella (`FROM`), la colonna (`SELECT`) e la riga (`WHERE`) in cui si trova.
[5] Come nei database relazionali, le associazioni sono definite mediante appositi vincoli d'integrità.

5 Vantaggi e svantaggi

Il principale vantaggio del modello object-oriented è la rappresentazione (più o meno) naturale dei dati, nella complessità con cui si presentano: non è necessario "appiattire" le strutture annidate, come bisognerebbe invece fare in un database relazionale.

In compenso, questo modello dei dati presenta vari svantaggi:

- ogni DBMS realizza una variante diversa del modello, con un proprio linguaggio di interrogazione (lo standard OQL non è mai stato implementato nella sua interezza);
- la necessità di implementare le operazioni nel linguaggio ospite lega fortemente il database alla scelta del linguaggio di programmazione;
- bisogna specificare esplicitamente la persistenza, e, in base a essa, le interrogazioni possono non funzionare;
- storicamente, le prestazioni sono sempre state inferiori a quelle dei DBMS relazionali;
- un DBMS object-oriented non può interagire (facilmente) con un DBMS relazionale "classico".

6 Adozione e influenza

In pratica, a causa dei suoi svantaggi, il modello object-oriented non è mai stato ampiamente adottato. Esso, però, ha comunque avuto un ruolo importante, dal punto di vista della sua influenza sui modelli dei dati sviluppati in seguito (in particolare, quelli semi-strutturati e object-relational):

- ha introdotto la strutturazione arbitraria dei dati;
- ha sottolineato l'importanza delle path expression;
- ha introdotto la possibilità di ristrutturare le risposte delle interrogazioni;
- ha stimolato l'estensione del modello relazionale con caratteristiche object-oriented.