Azzolini Riccardo 2020-03-05

Object-relational data model

1 Sviluppo e implementazione del modello

In seguito allo sviluppo del modello object-oriented "puro", che, come già detto, non ha mai avuto molto successo, si è invece scelta un'altra strada: estendere il modello relazionale con alcune caratteristiche e funzionalità object-oriented.

Il modello così ottenuto, detto **object-relational**, non è particolarmente elegante (a causa di alcune idiosincrasie tra i due modelli che cerca di integrare), ma, nonostante ciò, è stato:

- standardizzato, a partire da SQL3/SQL:1999;
- implementato praticamente da tutti i DBMS "relazionali" moderni (anche se, come sempre, con differenze di sintassi e/o comportamento tra le varie implementazioni).

2 Caratteristiche

La principale caratteristica di SQL:1999 è la possibilità di **definire tipi di dati complessi**, che:

- permettono di rappresentare dati con una *struttura nidificata* all'interno delle relazioni;
- possono essere organizzati in **gerarchie di tipi**, e, associando tali tipi a delle relazioni, anche queste ultime possono formare delle gerarchie (con super-relazioni che vengono specializzate in sotto-relazioni).

3 Vantaggi

Il modello object-relational preserva il principale vantaggio del modello object-oriented, cioè la possibilità di modellare dati arbitrariamente complessi / strutturati in un modo (abbastanza) diretto, senza, ad esempio, la necessità di eliminare le gerarchie.

Rispetto al modello OO, però, una volta definita la struttura del database, si può operare su di esso usando semplicemente il DDL di SQL, con la necessità di imparare solo poca sintassi aggiuntiva (invece che un intero linguaggio di interrogazione diverso).

L'uso di SQL significa anche che, nella maggior parte dei casi, si mantiene anche l'efficienza tipica dei DBMS relazionali (maggiore di quella dei DBMS object-oriented).

4 Row type

I tipi complessi più importanti sono i **row type** ("tipi riga"), così chiamati perché sono di fatto tuple,[1] come quelle che formano le righe di una tabella.

La sintassi per la definizione di un row type con nome[2] è volutamente simile a quella per la creazione di una tabella. Ad esempio:

```
CREATE TYPE Name AS (
  first_name VARCHAR(20),
  last_name VARCHAR(20)
);
```

Nota: La sintassi esatta varia in base al DBMS. Ad esempio, può essere:

- `CREATE ROW TYPE Name AS (...);`
- `CREATE TYPE Name AS OBJECT (...);`

È anche possibile specificare che il tipo creato non è ulteriormente raffinabile, cioè non ammette sottotipi, mediante la parola chiave `FINAL`:

```
CREATE TYPE Name AS (
  first_name VARCHAR(20),
  last_name VARCHAR(20)
)
FINAL;
```

Di default, invece, la definizione di sottotipi è ammessa. Volendo, ciò può comunque essere specificato esplicitamente con la sintassi `NOT FINAL`, come nel seguente esempio:

```
CREATE TYPE Address AS (
  street VARCHAR(50),
  city VARCHAR(30),
  zip_code NUMERIC
)
NOT FINAL;
```

I row type sono un esempio di **collection type**. Come concetto, questi ultimi sono analoghi a quelli di Java, ma SQL ne ha molti meno; oltre ai row type, i più usati sono *array* e *multiset*.

[1] Questi tipi si possono anche considerare analoghi agli struct del linguaggio C, o alle classi Java.
[2] È anche possibile definire row type senza nome; ciò verrà illustrato successivamente.

4.1 Uso dei row type nelle tabelle

Una volta definito, un row type può essere assegnato a un attributo di una tabella (così come un tipo predefinito). Ad esempio, dati i tipi `Name` e ~Address:

```
CREATE TABLE Person (
  name Name,
  address Address,
  birth DATE
);
```

Così facendo, si definisce una struttura nidificata dei dati: un'istanza della tabella persona è composta da tuple con tre attributi, ma gli attributi `name` e `address` contengono, a loro volta, dei "sotto-attributi". Il modello relazionale esteso con questa possibilità prende il nome di *modello relazionale nidificato*.

In pratica, i row type consentono la rappresentazione diretta degli attributi composti, che non è quindi necessario eliminare nella fase di progettazione logica.

4.2 Typed table

I row type forniscono anche un modo alternativo per definire le tabelle:

1. creare un row type composto dagli attributi che la tabella dovrà avere:

   ```
   CREATE TYPE PersonType AS (
     name Name,
     address Address,
     birth DATE
   );
   ```

2. creare una tabella le cui righe sono istanze di questo tipo:

   ```
   CREATE TABLE Person OF PersonType;
   ```

Una tabella così definita è chiamata **typed table** ("tabella tipata"). Tale meccanismo ricalca il funzionamento del modello object-oriented, nel quale si creano prima un tipo, e poi le istanze di tale tipo.

4.3 Row type senza nome

Per creare lo schema di una tabella con attributi di tipi complessi, in alternativa a definire separatamente ciascun tipo, si possono specificare dei *row type senza nome* direttamente all'interno del comando `CREATE TABLE`. Ad esempio:

```
CREATE TABLE Person (
  name ROW (
    first_name VARCHAR(20),
    last_name VARCHAR(20)
  ),
  address ROW (
    street VARCHAR(50),
    city VARCHAR(30),
    zip_code NUMERIC
  ),
  birth DATE
);
```

Questa definizione è pressoché equivalente a quella basata sui row type con nome. La differenza principale è che, in questo caso, non è possibile riferirsi ai tipi degli attributi composti.

4.4 Interrogazioni

Nelle interrogazioni, è possibile accedere agli attributi di un tipo complesso attraverso le **path expression**, che in SQL (così come in Java) sono rappresentate mediante la *dot notation*. Ad esempio:

```
SELECT name.last_name, address.city
FROM Person;
```

5 Metodi

Un'altra estensione che il modello object-relational aggiunge a quello relazionale puro è la possibilità di associare dei **metodi** a un tipo.[3]

Per definire un metodo associato a un determinato tipo, è necessario specificare:

1. la segnatura del metodo, nel comando di creazione del tipo;
2. separatamente, l'implementazione del metodo, attraverso un apposito comando `CREATE INSTANCE METHOD`.

Ad esempio, il seguente codice associa al tipo `PersonType` un metodo `age_on_date`, che riceve un singolo parametro, di tipo `DATE`, e restituisce un valore di tipo `INTERVAL YEAR` (uno dei numerosi tipi temporali standard di SQL):

[3] Tale estensione è però una delle meno usate.

```
CREATE TYPE PersonType AS (
  name Name,
  address Address,
  birth DATE
)
METHOD age_on_date(on_date DATE)
  RETURNS INTERVAL YEAR;

CREATE INSTANCE METHOD age_on_date(on_date DATE)
  RETURNS INTERVAL YEAR
  FOR PersonType
  RETURN on_date - SELF.birth;
```

- Nel comando `CREATE INSTANCE METHOD` è necessario indicare, con la parola chiave `FOR`, il tipo al quale il metodo è associato: ciò è necessario in quanto tipi diversi potrebbero avere metodi con gli stessi nomi.
- Nel codice di implementazione del metodo, la parola chiave `SELF` si riferisce all'istanza del tipo (qui `PersonType`) sulla quale il metodo viene invocato (analogamente a `this` in Java).

Un esempio di invocazione di tale metodo in un'interrogazione è:

```
SELECT name.last_name, age_on_date(CURRENT_DATE)
FROM Person;
```

I metodi di SQL:1999 presentano alcune differenze rispetto a quelli dei database object-oriented:

- possono essere implementati direttamente in SQL, perciò il DBMS non ha per forza bisogno di appoggiarsi alla runtime di un linguaggio ospite per eseguirli;
- sono gestiti come entità appartenenti al database, alle quali è possibile assegnare permessi, ecc.