Basi di dati a oggetti

Paolo Atzeni

(con contributo iniziale di Luca Cabibbo)

26/05/2006

Tecnologia delle basi di dati relazionali

- I sistemi di gestione di basi di dati relazionali (RDBMS) hanno permesso la realizzazione efficace ed efficiente di applicazioni di tipo gestionale, caratterizzate da
  - persistenza, condivisione, affidabilità
  - dati a struttura semplice, con dati di tipo numerico/simbolico
  - transazioni concorrenti di breve durata (OLTP)
  - interrogazioni complesse, espresse mediante linguaggi dichiarativi e con accesso di tipo “associativo”
- La rapida evoluzione tecnologica (miglioramento di prestazioni, capacità, e costi dell’hardware) ha fatto emergere (fin dalla metà degli anni '80) nuove esigenze applicative — per le quali la tecnologia relazionale è inadeguata

Alcune aree applicative emergenti

- Progettazione assistita da calcolatore
  - CASE (Computer-Aided Software Engineering)
  - CAD (Computer-Aided Design)
  - CAM (Computer-Aided Manufacturing)
- Gestione di documenti
  - testi e automazione d'ufficio
  - dati ipertestuali
  - dati multimediali
- Altro
  - scienza e medicina
  - sistemi esperti, per la rappresentazione di conoscenza

Esempi di nuove applicazioni

- una base di dati con informazioni sui requisiti di adiacenza dei componenti di un chip e l'esigenza di funzioni di ottimizzazione
- un archivio di 40.000 fotografie, con didascalie, coordinate geografiche ed esigenze di interrogazioni complesse:
  - "trova le foto con un tramonto scattate a Roma o dintorni"
- archivio sinistri di una compagnia assicurativa (con foto, grafici, luogo) finalizzato alla ricerca delle frodi
- archivio del personale con informazioni sul curriculum, foto, residenza per una gestione integrata (riconoscimento, valutazione delle competenze, proposta di "car-pool")
- gestione di "codici parlanti" (ad esempio il codice fiscale)

Caratteristiche delle nuove aree applicative

- Oltre alle caratteristiche consuete di persistenza, condivisione e affidabilità, possiamo individuare
  - dati a struttura complessa
    - dati non-numerici — immagini, dati spaziali, sequenze temporali, ...
    - tipi pre-definiti e tipi definiti dall'utente (e riutilizzati)
    - relazioni esplicite ("semantiche") tra i dati (riferimenti), aggregazioni complesse
  - operazioni complesse
    - specifiche per i diversi tipi di dato — es. multimedia
    - associate anche ai tipi definiti dall'utente
    - transazioni di lunga durata

Basi di dati a oggetti

- Alcune delle precedenti caratteristiche suggeriscono l'utilità di introdurre, nel mondo delle basi di dati, idee proprie del paradigma orientato agli oggetti
- A partire dalla metà degli anni '80, sono stati realizzati numerosi sistemi di gestione di basi di dati a oggetti (ODBMS) — di tipo prototipale o commerciale

Tecnologia degli ODBMS

- La prima generazione di ODBMS è composta dai linguaggi di programmazione a oggetti *persistenti*, che realizzano solo alcune caratteristiche delle basi di dati, senza supporto per l'interrogazione, in modo incompatibile con gli RDBMS
- Gli ODBMS della seconda generazione realizzano un maggior numero di caratteristiche delle basi di dati, e generalmente forniscono un supporto all'interrogazione
- Due tecnologie di ODBMS
  - OODBMS (*Object-Oriented*): una tecnologia *rivoluzionaria* rispetto a quella degli RDBMS
  - ORDBMS (*Object-Relational*): una tecnologia *evoluzionaria* rispetto a quella degli RDBMS

Un modello dei dati a oggetti

- Una base di dati a oggetti è una collezione di oggetti
- Ciascun oggetto ha un identificatore, uno stato, e un comportamento
  - l'identificatore (OID) garantisce l'individuazione in modo univoco dell'oggetto, e permette di realizzare riferimenti tra oggetti
  - lo stato è l'insieme dei valori assunti dalle proprietà dell'oggetto — è in generale un valore a struttura complessa
  - il comportamento è descritto dall'insieme dei metodi che possono essere applicati all'oggetto

Modello di dati a oggetti e modello relazionale

- Due differenze fondamentali
  - struttura complessa:
    - nidificata
    - con identificatori gestiti dal sistema
  - comportamento

Tipi — parte statica

- Un *tipo* descrive le *proprietà* di un oggetto (la parte statica) e l'interfaccia dei suoi *metodi* (la parte dinamica)
- parte statica: i tipi vengono costruiti a partire da
  - un insieme di *tipi atomici* (numeri, stringhe, ...)
  - un insieme di *costruttori di tipo*, tra loro ortogonali
    - *record-of*(A_1:T_1, ..., A_n:T_n)
    - *set-of*(T), *bag-of*(T), *list-of*(T)
  - un *riferimento* ad altro tipo definito nello schema è considerato un tipo, utilizzato per rappresentare relazioni tra oggetti (associazioni)

Un tipo complesso (ingenuo)

```
automobile: record-of(
   targa: string, modello: string,
   costruttore: record-of(
        nome: string,
        presidente: string,
        stabilimenti: set-of(
               record-of(
                      nome: string, citta: string,
                      addetti: integer))),
   colore: string,
   prezzo: integer,
   partiMeccaniche: record-of(
        motore: string,
        ammortizzatore: string)
)
```

Valori complessi

- L’uso di tipi e valori complessi permette di associare ad un singolo oggetto una struttura qualunque
- Viceversa, nel modello relazionale alcuni concetti devono essere rappresentati tramite più relazioni
- Tuttavia, la rappresentazione proposta per **`automobile`** non è certo la migliore: sicuramente presenta ridondanze (per ogni automobile, ripetiamo il costruttore con tutte le informazioni!)

Modello nidificato con riferimenti "valore"

```
automobile: record-of(
   targa: string, modello: string,
   costruttore: string,
   colore: string,
   prezzo: integer,
   partiMeccaniche: record-of(
        motore: string,
        ammortizzatore: string)
)
costruttore: record-of(
        nome: string,
        presidente: string,
        stabilimenti: set-of(
                record-of(
                        nome: string, citta: string,
                        addetti: integer))
)
```

Riferimenti

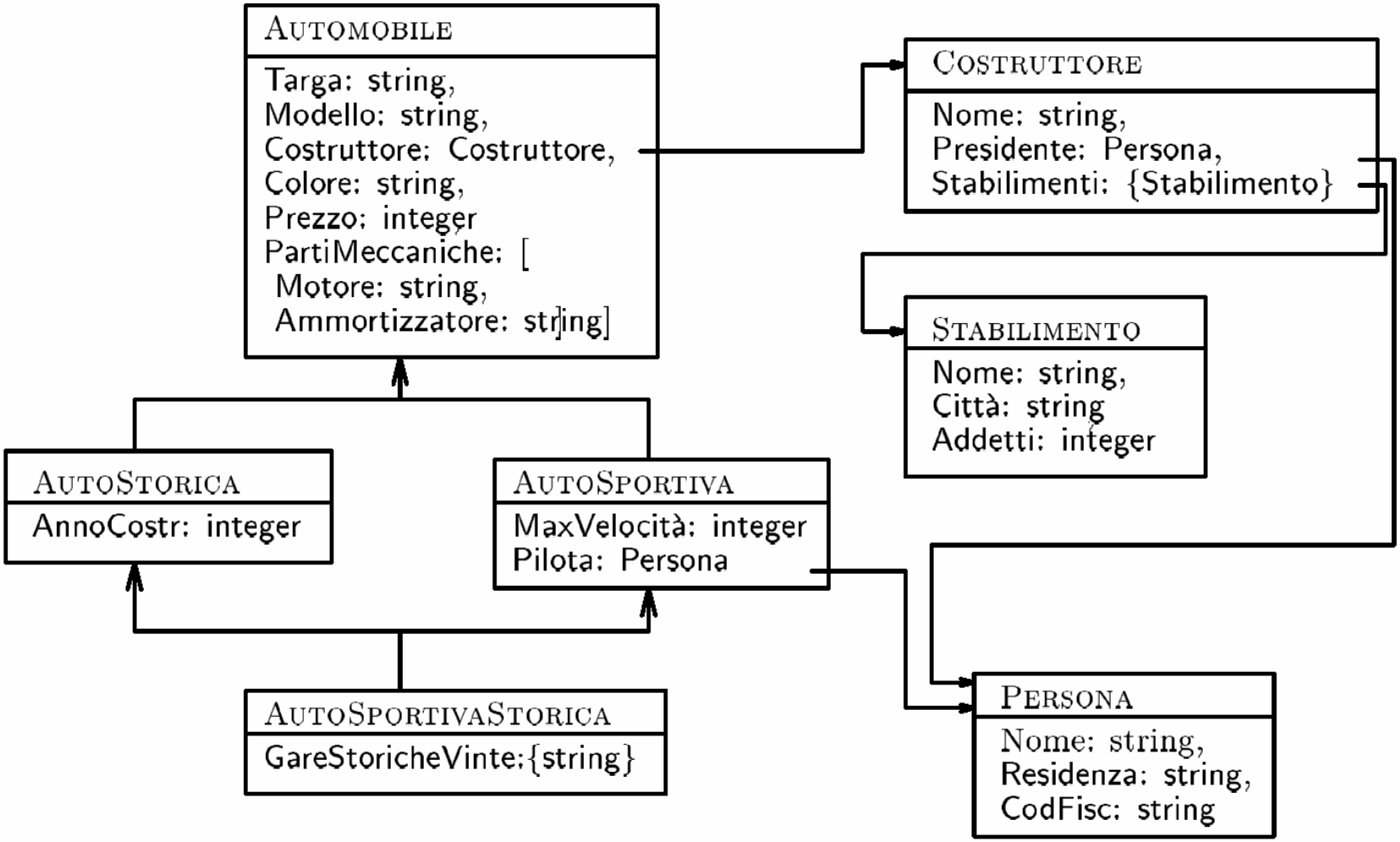
```
automobile: record-of(
        targa: string, modello: string,
        costruttore: *costruttore,
        colore: string, prezzo: integer,
        partiMeccaniche: record-of(
                motore: string,
                ammortizzatore: string))
costruttore: record-of(
        nome: string, presidente: string,
        stabilimenti:
                set-of(
                        record-of(
                                nome: string, citta: string,
                                addetti: integer)))
```

Valori e OID

- Un *oggetto* è una coppia (*OID, Valore*), dove *OID* (*object identifier*) è un valore atomico definito dal sistema e trasparente all'utente, e *Valore* è un valore complesso
- Il valore assunto da una proprietà di un oggetto può essere l'OID di un altro oggetto (realizzando così un riferimento)

Identità e uguaglianza

- Tra gli oggetti sono definite le seguenti relazioni
  - *identità* (O1=O2) — richiede che gli oggetti abbiano lo stesso identificatore
  - *uguaglianza superficiale* (O1==O2) — richiede che gli oggetti abbiano lo stesso stato, cioè stesso valore per proprietà omologhe
  - *uguaglianza profonda* (O1===O2) — richiede che le proprietà che si ottengono seguendo i riferimenti abbiano gli stessi valori (non richiede l'uguaglianza dello stato)
    - O1 = <OID1, [a, 10, OID3]>
    - O2 = <OID2, [a, 10, OID4]>
    - O3 = <OID3, [a,b]>
    - O4 = <OID4, [a,b]>


AUTOMOBILE
Targa: string,
Modello: string,
Costruttore: Costruttore,
Colore: string,
Prezzo: integer
PartiMeccaniche: [
Motore: string,
Ammortizzatore: string]
COSTRUTTORE
Nome: string,
Presidente: Persona,
Stabilimenti: {Stabilimento}
STABILIMENTO
Nome: string,
Città: string
Addetti: integer
AUTOSTORICA
AnnoCostr: integer
AUTOSPORTIVA
MaxVelocità: integer
Pilota: Persona
AUTOSPORTIVASTORICA
GareStoricheVinte:{string}
PERSONA
Nome: string,
Residenza: string,
CodFisc: string

Semantica dei riferimenti

- Il concetto di riferimento presenta analogie con quello di *puntatore* nei linguaggi di programmazione, e con quello di *chiave esterna* in un sistema relazionale. Tuttavia
  - i puntatori possono essere corrotti (dangling, appesi); i riferimenti a oggetti (in un buon ODBMS) vengono invalidati automaticamente in caso di cancellazione di un oggetto referenziato
  - le chiavi esterne sono visibili, in quanto realizzate tramite valori; gli identificatori d'oggetto non sono associati a valori visibili dall'utente
  - modificando gli attributi di una chiave esterna, è possibile perdere riferimenti; modificando il valore di un oggetto referenziato, il riferimento continua ad esistere

Metodi — parte dinamica

- Il paradigma OO deriva dal concetto di *tipo di dato astratto*
- Un *metodo* è una procedura utilizzata per incapsulare lo stato di un oggetto, ed è caratterizzata da una *interfaccia* (o segnatura) e una *implementazione*
  - l'interfaccia comprende tutte le informazioni che permettono di invocare un metodo (il tipo dei parametri)
  - l'implementazione contiene il codice del metodo
- Il *tipo* di un oggetto comprende, oltre alle proprietà, anche le interfacce dei metodi applicabili a oggetti di quel tipo

Metodi

```
automobile: record-of(
        targa: string, modello: string,
        costruttore: *costruttore,
        colore: string, prezzo: integer,
        partiMeccaniche: record-of(
                motore: string,
                ammortizzatore: string),
        public Init(                        // costruttore
                targa_par: string,
                modello_par: string, colore_par: string,
                prezzo_par: integer): automobile,
        public Prezzo(): integer,  // accessore
        public Aumento(            // trasformatore
                ammontare: integer))       // void
```

Classi — estensioni dei tipi

- Una classe è un contenitore di oggetti, che possono essere dinamicamente aggiunti o tolti alla classe (tramite costruttori e distruttori)
- Ad una classe è associato un solo tipo, ovvero gli oggetti in una classe sono tra loro omogenei (hanno le stesse proprietà e rispondono agli stessi metodi)
- Due alternative:
  - una classe raccoglie tutti gli oggetti di uno stesso tipo
  - più classi possono avere lo stesso tipo
- Una classe definisce anche l'implementazione dei metodi, che va specificata in qualche linguaggio di programmazione

Implementazione e invocazione di metodi

```
body Init(targa_par: string, modello_par: string,
   colore_par: string, prezzo_par: integer):
   automobile  in class automobile
co2{
   self->targa = targa_par;
   self->modello = modello_par;
   self->colore = colore_par;
   self->prezzo = prezzo_par;
   return(self);
}$

execute co2{         // in un programma
   o2 automobile X; // dichiarazione di variabile O2
   X = new(automobile);
   [X Init("Mi56T778", "Panda", "blu", 12M)]; }$
```

Disadattamento di impedenza

- In un RDBMS, esiste un "disadattamento di impedenza" tra i linguaggi con i quali vengono scritte le applicazioni (che manipolano variabili scalari) e l'SQL, che estrae insiemi di ennuple. Si usano allo scopo i *cursori*
- Si dice che gli ODBMS risolvono questo problema, in quanto gli oggetti persistenti possono essere manipolati direttamente tramite le istruzioni del linguaggio di programmazione (procedurale)
- Il PL di un ODBMS deve permettere l'accesso alle componenti di un valore complesso — ad esempio, i record con l'operatore dot ".", i riferimenti con l'operatore "->", le collezioni con opportuni iteratori

Altre caratteristiche del paradigma a oggetti

- Tra i tipi (e le classi) di una base di dati a oggetti, è possibile definire una gerarchia di ereditarietà, con le usuali relazioni di sottotipo, l'ereditarietà dei metodi, la possibilità di overloading, overriding, late binding, ereditarietà multipla, ...
- Tutto funziona come nei linguaggi di programmazione a oggetti
- Esiste tuttavia una importante differenza: gli oggetti di un programma sono oggetti di breve durata, gli oggetti di una base di dati sono oggetti di lunga durata, con conseguenze non banali...

Migrazione tra classi

- Nel corso della propria esistenza, un oggetto deve mantenere la propria identità, in modo che sia possibile riferirsi ad esso in modo univoco
- Tuttavia, è possibile che un oggetto *cambi tipo*:
  - una persona diviene uno studente, poi uno studente lavoratore, poi un lavoratore, poi una persona sposata, ...
- …

Persistenza

- Gli oggetti possono essere **temporanei** (come nei programmi tradizionali) o **persistenti**.
- La persistenza può essere specificata in vari modi (non sempre tutti disponibili in uno stesso sistema):
  - inserimento in una classe persistente
  - raggiungibilità da oggetti persistenti
  - "denominazione": si può definire un nome ("handle") per un oggetto

OQL — Object Query Language

- Linguaggio SQL-like per basi di dati a oggetti, inizialmente sviluppato per O2, adottato (con modifiche) da ODMG, basato sui seguenti principi
  - non è computazionalmente completo, ma può invocare metodi, e metodi possono includere interrogazioni
  - permette un accesso dichiarativo agli oggetti
  - basato sul modello ODMG
  - ha una sintassi astratta — ed almeno una sintassi concreta, *simile* a SQL
  - ha primitive di alto livello per le collezioni
  - non ha operatori di aggiornamento

Interrogazioni OQL

- OQL permette di interrogare *interattivamente* oggetti, partendo dai loro nomi
- Alternativamente, comandi OQL possono essere immersi in un linguaggio di programmazione — utilizzandoli come argomenti (di tipo String) di metodi che li eseguono e ne restituiscono il risultato
- Facciamo riferimento ad uno schema con tipi
  - **Impiegato** (nome, nascita:(citta, data), stipendio, subordinati, età(), assegna_subordinato(...)) — con estensione **Impiegati**
  - **Dipartimento** (nome_dip, sedi:{(nome_sede, citta_sede)}, direttore) — con estensione **Dipartimenti**
- Un impiegato è designato come **Presidente**

OQL per esempi (1)

- Seleziona il presidente

```
Presidente
```

- Seleziona i subordinati del presidente

```
Presidente.subordinati
```

- Seleziona gli stipendi degli impiegati di nome Pat

```
select distinct x.stipendio
from x in Impiegati
where x.nome = “Pat”
```

- Seleziona nome e età degli impiegati di nome Pat

```
select distinct struct(n: x.nome, e: x.eta())
from x in Impiegati
where x.nome = “Pat”
```

OQL per esempi (2)

- Partiziona gli impiegati in base allo stipendio

```
group x in Impiegati by (
        low: x.stipendio < 1000,
        mid: x.stipendio >= 1000 and
               x.stipendio < 5000),
        high: x.stipendio >= 5000)
```

- Questa espressione ha tipo

```
set<struct(low: boolean, mid: boolean,
                         high: boolean,
                         partition: set<Impiegato>)>
```

- Una versione estesa dell'operatore **group** permette il calcolo di funzioni aggregative

The Object-Oriented Database Manifesto
(Atkinson, Bancilhon, DeWitt, Dittrich, Maier, Zdonik)

- Una lista di funzionalità per la definizione (e la valutazione) di OODBMS.
- Include:
  - Funzionalità obbligatorie (the "golden rules")
  - Funzionalità opzionali
  - Scelte aperte

THE GOLDEN RULES

- Thou shalt support complex objects
- Thou shalt support object identity
- Thou shalt encapsulate thine objects
- Thou shalt support types or classes
- Thine classes or types shalt inherit from their ancestors
- Thou shalt not bind prematurely
- Thou shalt be computationally complete
- Thou shalt be extensible
- Thou shalt remember thy data
- Thou shalt manage very large databases
- Thou shalt accept concurrent users
- Thou shalt recover from hardware and software failures
- Thou shalt have a simple way of querying data

Funzionalità obbligatorie

- Oggetti complessi
- Identità di oggetto
- Incapsulamento
- Tipi e/o classi
- Gerarchie di classi o di tipi
- Overriding, overloading e late binding
- Completezza computazionale
- Estensibilità
- Persistenza
- Gestione della memoria secondaria
- Concorrenza
- Recovery
- Linguaggio o interfaccia di interrogazione

The Third Generation Database System Manifesto

(Stonebraker, Rowe, Lindsay, Gray, Carey, Brodie, Bernstein, Beech)

- Una risposta al manifesto OODMS
- "I DBMS della prossima generazione dovranno essere ottenuti come risultato dell'evoluzione dei DBMS esistenti (relazionali)"

I principi del contromanifesto

- I DBMS di terza generazione dovranno
  - essere una generalizzazione (compatibile) con i DBMS della seconda generazione
  - (oltre a fornire i servizi tradizionali di gestione dei dati) permettere la definizione di oggetti complessi e regole
  - essere aperti ad altri sottosistemi

Manifesto 3GDBMS: dettagli

[1.1] rich type system
[1.2] inheritance
[1.3] functions and encapsulation
[1.4] OID's only if there are no keys
[1.5] rules and triggers

[2.1] non procedural, high level access languages
[2.2] specification techniques for collections
[2.3] updatable views
[2.4] transparency of physical parameters

[3.1] multiple high level languages
[3.2] persistent x, for many x's
[3.3] SQL is a standard (even if you don't like it)
[3.4] queries and their results are the lowest levest of communication

Basi di dati “Object-Relational”

- Modello dei dati
- Linguaggio di interrogazione

- Facciamo riferimento a proposte presentate nella letteratura come "SQL-3", poi recepite solo in parte nel nuovo standard SQL:1999 (approvato nel dicembre 1999)
- Si tratta di una estensione “compatibile” di SQL-2 (cioè il codice SQL-2 è valido anche come SQL-3)

Modello dei dati di SQL-3

- È possibile definire tipi:
  - "distinct types," derivati dai tipi base, finalizzati alla tipizzazione forte
  - tipi strutturati con struttura anche complessa e con gerarchie
    - usati come "valore", cioè componenti di ennupla
    - usati come "oggetti", per definire tabelle con lo stesso schema

Esempio di "distinct type"

```
CREATE TYPE NumeroMaglia AS INTEGER FINAL
CREATE TYPE NumeroScarpa AS INTEGER FINAL

CREATE TABLE Calciatori (
   Nome                    CHARACTER VARYING (20),
   Squadra                 CHARACTER VARYING (20),
   Scarpini                NumeroScarpa,
   Maglia                  NumeroMaglia,
   ... )
```

Interrogazioni su "distinct type"

```
SELECT *
FROM Calciatori
WHERE Maglia > Scarpini

SELECT *
FROM Calciatori
WHERE CAST (Maglia AS INTEGER) >
        CAST (Scarpini AS INTEGER)
```

Tipi strutturati

- permettono di definire tipi da utilizzare per i singoli attributi (anche a struttura complessa)
- possono avere funzioni associate (qui si ha la “estensibilità” del sistema di tipi), definite in SQL-3 o in linguaggi esterni

```
create type CarPartType(
     Engine char(10),
     Power integer,
     Cylinders integer,
     greater than GreaterPower)

create function GreaterPower(:p1 CarPartType, p2: CarPartType)
     returns boolean;
     returns ((:p1.Power > :p2.Power) or
               ((:p1.Power = :p2.Power and
                (:p1.Cylinders > :p2. Cylinders)))
```

Esempio

```
CREATE TYPE film AS (
   titolo           CHARACTER VARYING (100),
   descrizione      CHARACTER VARYING (500),
   minuti           INTEGER )
NOT FINAL
METHOD durata ( )
   RETURNS INTERVAL HOUR(2) TO MINUTE
…
CREATE INSTANCE METHOD  durata ( )
   RETURNS INTERVAL HOUR(2) TO MINUTE
FOR film
RETURN CAST(CAST SELF.minuti AS INTERVAL
   MINUTE(4)) AS INTERVAL HOUR(2) TO MINUTE )
```

Esempio

```
CREATE TABLE movie_table (
   stock_number          CHARACTER(8),
   movie_info            film,
   rental_quantity       INTEGER,
   rental_cost           DECIMAL(5,2)      )

SELECT m.movie_info.durata()
FROM   movie_table AS m
WHERE  m.movie_info.title() = 'Star Wars'
```

Metodi "osservatori" e "modificatori"

```
SELECT m.movie.runs
FROM movie_table
WHERE title = ‘Star Wars’
```

```
SELECT movie.runs()
FROM movie_table
WHERE title = ‘Star Wars’
```

```
UPDATE movie_table
SET movie.runs = 113
WHERE title = ‘Star Wars’
```

```
UPDATE movie_table
SET movie =
   movie.runs(113)
WHERE title = ‘Star Wars’
```

Tipi strutturati e tabelle tipate

```
create type PersType
        Name varchar (30) not null,
        Residence varchar(30),
        SSN char(16) primary key)

create table Professor of type PersType
create table Student of type PersType
```

- le ennuple sono gli oggetti
- le relazioni le classi

- gli identificatori possono essere manipolabili
- si possono usare riferimenti e/o incorporare oggetti

Tipi e tabelle tipate

```
create type FactoryType(
        Name varchar (25),
        City varchar (7),
        NoOfEmployees(7))
        ref is system generated

create type ManufacturerType(
        Name varchar (25),
        President ref(PersType))
        ref is system generated

create type CarPartType(
        Engine char(10),
        SchockAbsorber char(5))
        ref is system generated

create type AutoType(
        RegistrationNumber char(10) primary key,
        Model varchar (30),
        Maker ref(ManufacturerType),
        MechanicalParts ref(CarPartType))
        ref is system generated
```

```
create table Automobile of AutoType

create table Manufacturer of ManufacturerType
  (REF IS ManufID system generated)
   President with option scope Professor

create row type VintageCarType(
  Manufactureyear integer)
  under AutoType

create table VintageCar of VintageCarType
         under Automobile
```

– **alternativamente (ma con tipo “non riutilizzabile”):**

```
create table VintageCar(
  Manufactureyear integer)
  under Automobile
```

Interrogazioni in SQL-3

- Le interrogazioni SQL-2 sono ammesse in SQL-3
- Inoltre:
  - si possono “seguire” i riferimenti
  - si possono citare gli OID (se visibili)
  - si può accedere alle strutture interne
  - si puo’ nidificare (nest) e denidificare (unnest)

Interrogazioni in SQL-3

```
select President -> Name
from Manufacturer
where Name = ‘Fiat’


select Name
from Manufacturer, Industrial
where Manufacturer.Name = ‘Fiat’
  and Manufacturer.President = Industrial.ManufId

select Maker -> President -> Name
from Automobile
where MechanicalParts->Engine = ‘XV154’
```