Modelli Avanzati e Architetture delle Basi di Dati

Fabio Strocco

26 febbraio 2010

Indice

Introduzione

Le **basi di dati**, o *database*, costituiscono un utile strumento per la memorizzazione e gestione di grosse quantità di dati.

Generalmente per la gestione di un sistema informativo sono necessari vari programmi che trattano i dati (si pensi alla gestione di un'azienda). Inizialmente ogni programma memorizzava i dati con un proprio standard e in propri file, cosicché si creava ridondanza nei vari file gestiti da un programma se questi trattavano alcune informazioni in comune.
In ogni caso sorge la necessità di standardizzare il formato dei dati memorizzati da ogni programma. In questo modo ogni programma esistente o nuova applicazione possono accedere ai dati nello stesso modo. Inoltre, sono necessari meccanismi avanzati per la gestione della concorrenza e la lettura efficiente dei dati.
Conviene dunque appoggiarsi ad un programma che si occupa di eseguire tutte queste operazioni. Queste applicazioni o insiemi di applicazioni prendono il nome di **DBMS**: *Database Management System*.

Questo documento suppone che il lettore abbia già una buona conoscenza informatica e del *modello relazionale* delle basi di dati e tratta principalmente il *modello orientato agli oggetti* e l'architettura dei database.
Maggiori informazioni su questo argomento possono essere trovate quì [1].

Diamo ora una definizione precisa di database.

Definizione 1. *Una* ***base di dati*** *è un insieme di dati* atomici*,* strutturati *e* persistenti*, raggruppati in* insiemi omogenei *in relazione tra loro organizzati con la* minima ridondanza *per essere utilizzati da applicazioni diverse in modo sicuro e controllato.*

I dati sono accessibili dalle applicazioni mediante uno schema logico. Ad esempio nel modello relazionale si definisce uno schema logico come un insieme di coppie (attributo, dominio) e quindi è possibile accedere ai dati tramite questo schema. Due concetti fondamentali delle basi di dati sono i seguenti

- **Indipendenza logica**: Le applicazioni che utilizzano i database non devono variare al variare dello *schema logico* se questo preserva le informazioni originarie. Questo è possibile grazie alle *viste*, che forniscono un'astrazione dello schema logico. Le viste tabelle virtuali composte tramite interrogazioni quindi basta cambiare le interrogazioni per fornire lo stesso schema anche se varia quello logico.

- **Indipendenza fisica**: Lo schema logico deve rimanere invariato se varia lo *schema fisico* ossia il modo di memorizzare i dati. Se non varia lo schema logico allora non variano neanche le applicazioni che accedono ai database siccome esse vi accedono tramite viste o schema logico.

I primi modelli di DBMS erano il modello gerarchico e il modello reticolare.
Il **modello reticolare** è basato su record e puntatori, mentre il **modello gerarchico** utilizza strutture ad albero per rappresentare i dati.
Da questi due modelli deriva il **modello relazionale**, basato invece sugli schemi relazionali.
Con l'avvento dei linguaggi di programmazione orientati agli oggetti, è nato il **modello relazionale ad oggetti** che è un'estensione del modello relazionale con gli oggetti.

Gli **RDBMS**, ossia i DBMS relazionali offrono una serie di funzionalità quali *gestione della concorrenza* (accessi multipli a database), *transazioni ACID*, *integrità dei dati* (tramite vincoli che permettono di mantenere i dati memorizzati in un database consistenti), *efficenza*, *interrogazioni* e *sicurezza* (ad esempio con la gestione dei permessi).

Gli **OODBMS**, ossia i DBMS orientati agli oggetti offrono tutte le funzionalità degli *RDBMS* più nuove funzionalità, quali *tipi di dati astratti*, *ereditarietà* (come nei linguaggi OO), *identità degli oggetti* (ogni oggetto ha un OID, ossia un identificatore).

Gli approcci orientati agli oggetti permettono di implementare strutture più vicine alla realtà avendo dunque una maggiore comprensibilità delle strutture.

Parte I

Modelli Avanzati dei DB

Capitolo 1

Database relazionali ad Oggetti

Prima di esaminare il modello relazionale ad oggetti esaminiamo il concetto di paradigma orientato ad oggetti.

1.1 Oggetti

Definiamo innanzitutto cos'è un oggetto.

Definizione 2. *Un* ***oggetto*** *è un entità software costituita da*

- ***stato****: un insieme di valori di attributi chiamati anche* variabili di istanza
- ***comportamento****: un insieme di* metodi *che restituiscono un valore al chiamante e modificano eventualmente lo stato dell'oggetto.*
- ***identità****: un valore che identifica univocamente un oggetto detto* OID*, cioè Object Identifier.*

Un oggetto è dunque un insieme di valori ed azioni che modificano lo stato di un oggetto.
E' dunque possibile inviare (da parte di un oggetto) un *messaggio* ad un oggetto, che comporta l'esecuzione di un metodo.
I metodi sono sostanzialmente alternative alle funzioni offerte dai linguaggi funzionali e imperativi.
Quando un oggetto invia un messaggio ad un altro oggetto (o eventualmente a se stesso) deve farlo riferendosi ad un metodo specifico che il ricevente deve eseguire. Un oggetto mette a disposizione alcuni metodi verso tutti gli oggetti, e questo costituisce l'*interfaccia* di un oggetto.

In molti linguaggi OO (Object Oriented) il tipo di dato svolge un ruolo fondamentale sul controllo che le operazioni vengano svolte correttamente.

Definizione 3. *Un **tipo di dato astratto** (ADT) è la descrizione astratta di tutto ciò che accomuna oggetti di una determinata categoria.*

Comunemente la nozione di tipo è fornita dalle classi. Una **classe** definisce stato e comportamento di oggetti di determinate categorie e genera istanze. Un'**istanza di una classe** è una particolare oggetto che ha esattamente gli attributi ed il comportamento descritti dalla classe.

Facciamo ora un esempio.

```
class Impiegato
    String nome;
    String cognome;
    Azienda azienda;

    void setAzienda(Azienda azienda) {
        self->azienda = azienda;
    }

class Azienda
    String nome;
    List impiegati;

    void aggiungiImpiegato(Impiegato imp) {
        impiegati->add(imp);
        imp->setAzienda(self);
    }
```

Questo codice java-like definisce due *classi*: Impiegato e Azienda.
Per impiegato lo stato è costituito dagli attributi *nome*, *cognome* e *azienda*.
Il comportamento è definito dall'unico metodo metodo *setAzienda*.
Ogni oggetto che rispetterà questa descrizione e generato dalla classe *Impiegato* sarà di tipo *Impiegato*.
In generale possiamo dunque dire che una classe contiene la definizione della struttura dell'oggetto, un costruttore (responsabile della creazione degli oggetti) e un distruttore (responsabile della loro distruzione).

Tutte queste considerazioni possono essere fatte anche per la classe *Azienda*.
Notare che l'istruzione $imp{-}\rangle setAzienda(self)$; invia il messaggio *setAzienda* all'oggetto *imp* di tipo *Impiegato* con parametro l'OID dell'oggetto stesso su

cui si invia il messaggio *aggiungiImpiegato*. L'OID è fornito dal parametro implicito self (*this* in Java e C++).

Si crea dunque una catena di invio di messaggi ed esecuzioni di metodi.

Notare che se in un sistema esistono due istanze di $Impiegato$ $(OID = 1,'Paolo','Rossi', OIDAzienda)$ e $(OID = 2,'Paolo','Rossi', OIDAzienda)$ queste, pure avendo lo stesso stato sono due istanze diverse in quanto hanno OID differenti.

La classe *Impiegato* contiene un campo che fa riferimento all'OID di un oggetto di tipo *Azienda*. Questo principio di contenimento di oggetti da parte di un oggetto si chiama **incapsulamento**.

Un altro principio importante del paradigma OO è l'**information hiding**. Secondo questo principio gli aspetti implementativi vanno separati da quelli esterni, ossia di interfacciamento verso l'esterno.
Nella programmazione OO l'information hiding consiste nel rendere pubblici i metodi ma private le loro implementazioni così come lo stato degli oggetti. Nelle basi di dati OO invece, il concetto è un po' diverso: oltre ai metodi anche lo stato (le variabili di istanza) è pubblico ma non sono disponibile verso l'esterno i tipi di dato utilizzati dallo stato.

Nelle basi di dati la classe ha un aspetto prevalentemente estensionale.
Un'**estensione** di una classe in un determinato istante è l'insieme di tutte le istanze di quella classe che non sono ancora state rimosse.
Nelle basi di dati la classe è dunque un *insieme di istanze* sul quale si possono eseguire determinate operazioni come aggiunta di una nuova istanza, rimozione ecc.

Nelle basi di dati esistono svariati modi per garantire la correttezza delle applicazioni e quindi l'integrità dei dati.
Il sistema di tipi fornisce un controllo sulla struttura degli oggetti di una determinata classe. I vincoli, come quelli dell'algebra relazionale (chiave, integrità referenziale, not null, ecc.) favoriscono un controllo aggiuntivo. Esiste anche un approccio basato su pre-condizioni e post-condizioni. Le *pre-condizioni* sono asserzioni che stabiliscono l'insieme dei valori che può assumere un input per un dato metodo mentre le *post-condizioni* stabiliscono l'insieme di valori che può assumere l'output. Gli *invarianti* sono invece asserzioni che devono essere sempre rispettate e possono mettere in relazione anche più oggetti.

Un applicazione ben progettata non deve mai violare gli invarianti, che vengono stabiliti in fase di progettazione.
In ogni caso, può succedere che per cause impreviste anche se un input soddisfa le pre-condizioni si abbia una situazione anomala per cui non si può soddisfare gli invarianti oppure le post-condizioni. In questi casi è necessario ricorrere alle **exception**, ossia eccezioni che terminano l'esecuzione del metodo e vanno sempre gestite dal chiamante del metodo.
Questa metodologia di sviluppo, che arricchisce il paradigma OO con invarianti, pre-condizioni e post-condizioni è detta *Design by contract*.

Un meccanismo molto potente fornito dal paradigma OO è l'**ereditarietà**. L'ereditarietà è un meccanismo mediante il quale una classe eredita la struttura ed il comportamento di un'altra classe (sopraclasse). Un'istanza di una classe è dunque anche istanza di tutte le eventuali sopraclassi. La classe che eredita le caratteristiche di un'altra classe è detta sottoclasse.
Esistono vari approcci all'ereditarietà

- **Strutturale**: una classe eredita tutti gli attributi della sopraclasse e può aggiungerne di nuovi oppure modificare il tipo di attributi esistenti purché i nuovi tipi siano sottotipi di quelli originali. Ad esempio Se una classe B è sottoclasse di A e A contiene l'attributo a di tipo K allora B può ridefinire a assegnandoli tipo L purché L sia una sottoclasse di K.

- **Comportamentale**: una classe eredita tutti i metodi di una sopraclasse e può aggiungerne di nuovi.

- **Overloading procedurale**: una classe eredita tutti i metodi di una sopraclasse, può aggiungerne di nuovi oppure ridefinire metodi esistenti. Questi metodi hanno la stessa signature[1] ma possono avere diverse implementazioni.

- **Framework procedurale**: una classe può contenere richiami a metodi non implementati dalla classe stesso, all'interno dell'implementazione di metodi implementati dalla classe, e le sottoclassi possono fornire l'implementazione mancante.
  Più precisamente, una classe A che contiene almeno un metodo $m1$ la cui implementazione richiama su se stessa almeno un metodo $a1$, con $self{-}\rangle a1(...)$, è una **classe astratta**. Una classe B sottoclasse di A è una sua **sottoclasse concreta** se contiene l'implementazione

[1]La signature di un metodo è la specifica che comprende il suo nome, tipo di risultato e tipi dei suoi parametri. In Java ad esempio la signature è del tipo $T\ m\ (T_1\ a_1, ..., T_n\ a_n)$

del metodo $a1$ e di tutti gli altri eventuali metodi non forniti dalla sopraclasse (*astratti*).

Notare che il numero di istanze di una classe è superiore o uguale a quelle delle sue sottoclassi.
Infatti, tutte le istanze delle sottoclassi di una classe C sono anche istanze di C e tutte le istanze di C ma non delle sottoclassi sono ancora ovviamente istanze di C.
Sembra dunque ragionevole introdurre la definizione di estensione profonda.
L'**estensione profonda** di una classe è l'unione dell'*estensione* di una sottoclasse e delle *estensioni profonde* delle sue sottoclassi.
L'estensione profonda di una classe è dunque l'insieme di istanze di tutte le sottoclassi (quindi a loro volta delle loro estensioni profonde) e delle istanze della classe in questione.

Oltre alla nozione di *sottoclasse*, che evidenzia l'aspetto implementativo dell'ereditarietà (possibilità riutilizzare l'implementazione dei metodi di una classe e le variabili di istanza) si affianca quella di *sottotipo* che evidenzia l'aspetto gerarchico dell'ereditarietà. Riprendendo la definizione di ADT precedentemente fornita, ossia un tipo di dato astratto è la descrizione delle caratteristiche che accomunano oggetti di una determinata categoria, possiamo concludere che un tipo è sottotipo di un altro descrive le stesse caratteristiche del sovratipo e ne aggiunge alcune.

Formalmente indichiamo la relazione di sottotipo con $T_1 \leq T_2$ che si legge T_1 è sottotipo di T_2.
Questa relazione è

- **riflessiva**: un tipo è sottotipo di se stesso
- **antisimmetrica**: se T_1 è sottotipo di T_2 allora T_2 non può essere sottotipo di T_1 con T_1 e T_2 tipi distinti, altrimenti si incorrerebbe in una definizione ciclica di sottotipo.
- **transitiva**: se T_1 è sottotipo di T_2 e T_2 è sottotipo di T_3 allora T_1 è sottotipo di T_3.
  Ad esempio se un rettangolo è un (*sottotipo*) quadrilatero e un quadrilatero è una figura geometrica sembra ragionevole concludere che un rettangolo è una figura geometrica.

Un importante principio riguardante i tipi è il **principio di sostituibilità**: Siano T_1 e T_2 due tipi tali che $T_1 \leq T_2$, allora è possibile utilizzare

un'istanza di T_1 ovunque si possa utilizzare un'istanza di T_2.
Riprendendo l'esempio dei rettangoli, tutte le considerazioni (o operazioni, nei linguaggi di programmazione) valide su una figura geometrica generica valgono anche per un rettangolo. Notare che NON è vero il contrario, così come il principio di sostituibilità non vale al contrario, ossia non è possibile sostituire un'istanza di T_1 con un'istanza di T_2.

Gli approcci all'ereditarietà precedentemente mostrati sono conformi al principio di sostituibilità. Per mostrarlo suddividiamo una classe in due aspetti: strutturale (attributi) e comportamentale (metodi).
Consideriamo ora due classi A e B dove B è sottoclasse di A. Se un attributo può avere come tipo un insieme di elementi di un tipo t, $set(t)$, un tipo primitivo oppure una tupla, rappresentiamo proprio come tupla $[t_1\ A_1, ..., t_n\ A_n]$ l'insieme delle variabili di istanza di un oggetto dove $t_1, ..., t_n$ sono tipi. Otteniamo che, se $t'_1, ..., t'_n$ sono tali che $t'_i \leq t_i$ con $1 \leq i \leq n$ allora possiamo dire che

$$[t'_1\ A_1, ..., t'_n\ A_n, t_{n+1}\ A_{n+1}, ..., t_{n+k}\ A_{n+k}] \leq [t_1\ A_1, ..., t_n\ A_n]$$

Questo equivale all'approccio strutturale. Ogni operazione fatta sulla seconda tupla può essere fatta nello stesso modo anche sulla prima dato che i tipi degli attributi in comuni con la seconda sono sottotipi.
Analogamente possiamo dire che

$$set(t') \leq set(t)$$

Per quanto riguarda l'aspetto comportamentale dell'ereditarietà, questo è conforme al principio di sostituibilità se i metodi che vengono modificati nella sottoclasse hanno la stessa signature di quelli della sopraclasse.

Per quanto riguarda la design by contract, siano A e B due classi con B sottoclasse di A, e m un metodo definito in entrambe le classi relativamente con specifiche $pin_A, pout_A, inv_A$ e $pin_B, pout_B, inv_B$. Per preservare il principio di sostituibilità abbiamo che se per un dato input le specifiche di input pin_A sono soddisfatta anche quelle di B, pin_B devono esserlo. In questo modo si può utilizzare un'istanza di B al posto di una di A potendo richiamare m con gli stessi valori.

$$pin_A \rightarrow pin_B$$

Per quanto riguarda gli invarianti, se l'esecuzione di m con un dato input rispetta gli invarianti inv_B, quindi è 'corretta' per B deve esserla anche per

A. In questo modo si può sostituire un'istanza di A con una di B ottenendo che laddove l'esecuzione per m era corretta per A può non esserlo più per B, ma dove è corretta per B lo è anche per A.
Le stesse considerazioni valgono anche per i predicati di output. Dunque abbiamo

$$inv_B \rightarrow inv_A$$

$$pout_B \rightarrow pout_A$$

1.2 Oggetti semistrutturati

Un oggetto semi-strutturato è un oggetto che non è dotato di una particolare struttura rigida come quella dei dati dei DB relazionali (con vincoli, ecc.) ma le informazioni sono organizzate in strutture quali insiemi, tuple, array, ecc.
In questo paragrafo l'idea è quella di semplificare l'esposizione dei database ad oggetti mostrando solo la struttura principale dello stato degli oggetti, tralasciando le definizioni di classe, vincoli, ecc.

Gli oggetti vengono raggruppati in uno **spazio degli oggetti complessi** (semistrutturati). Gli oggetti che vengono memorizzati nelle basi dati vengono detti **persistenti**. Un oggetto non persistente è detto **transiente**.
Esistono principalmente tre modelli strutturali per gli oggetti: CODL, FAD e ibrido.

Modello CODL

Il modello **Complex Object Database Language** (*CODL*) consiste nell'organizzare gli oggetti basandosi puramente sui valori, quindi non è possibile referenziare altri oggetti tramite OID.
Formalmente abbiamo:
Sia $\mathcal{A}$ un insieme di nomi, $\mathcal{D}$ un insieme di *domini di base* (numeri, stringhe, ecc.) unito a $\{null\}$, un oggetto può essere:

- un valore $v \in \mathcal{D}$, ossia un *valore atomico*
- una tupla (*oggetto tupla*) $[a_1 : Q_1, ..., a_n : Q_n]$ dove $\forall i \ a_i \in \mathcal{A} \wedge Q_i$ *oggetto*
- un insieme (*oggetto set*) $\{Q_1, ..., Q_n\}$ dove $Q_1, ..., Q_n$ sono oggetti

Ad esempio, supponiamo di dover memorizzare i dati di alcune persone. Possiamo avere ad esempio il seguente spazio degli oggetti complessi

$[nome : Mario, cognome : Rossi, indirizzo : [via : Roma, numero : 1], figli : \{Luca, Paolo\}]$

$[nome : Paolo, cognome : Rossi, indirizzo : [via : Roma, numero : 1], figli : null]$

$[nome : Maria, cognome : Verdi, indirizzo : [via : Garibaldi, numero : 3, figli : \{Marco\}]]$

Gli oggetti contenuti nello spazio degli oggetti sono qui tre tuple. Ogni tupla ha come attributo nome un oggetto atomico (stringa), come cognome un altro oggetto atomico mentre l'indirizzo è costituito a sua volta da una tupla contenente due valori atomici. L'attributo figli è invece un insieme.

Il concetto di uguaglianza è dato dalle seguenti regole

- Due oggetti atomici sono uguali se è uguale il loro valore (es. $1 = 1, Mario = Mario$, *ecc.*)
- Due oggetti tupla $[a_1 : Q_1, ..., a_n : Q_n], [b_1 : R_1, ..., b_n : R_n]$ sono uguali se hanno gli stessi nomi di attributi e se dati due attributi con lo stesso nome a_i, b_j i relativi oggetti sono uguali, $Q_i = R_j$
- Due oggetti set (insiemi) A, B sono uguali se contengono gli stessi elementi, ossia $e \in A \leftrightarrow e \in B$.

Notare che una tavola relazionale, dunque un insieme di tuple (l'unico modo in cui vengono memorizzati i dati in un database relazionale) non è che un caso particolare di oggetto tupla che contiene solo valori atomici.

Modello FAD

Il modello **FAD** consiste nell'organizzare gli oggetti basandosi sui loro OID. Un *OID* è un valore che identifica univocamente un oggetto. Un oggetto potrebbe anche essere identificato con un nome, ma si rischierebbe l'ambiguità o l'omonimia (concetti uguali non potrebbero essere rappresentati con lo stesso nome), con un indirizzo (path come nei file system) ma questo richiederebbe la conoscenza dell'intero percorso di provenienza dell'oggetto che non è molto comodo oppure con un vincolo di chiave primaria, il che richiederebbe comunque di utilizzare join come nel modello relazionale per unire informazioni provenienti da più tuple.
Per ovviare a questi problemi nasce il concetto di OID. A differenza della chiave primaria di una tupla, l'OID è un valore unico in tutto il database e non solo nella relazione di appartenenza. Inoltre tale valore dovrebbe essere

invisibile all'utente.

Su queste considerazioni si basa il modello FAD. Formalmente
Dato un insieme di attributi $\mathcal{A}$, un dominio di base $\mathcal{D}$ contenente l'elemento $\{null\}$ ed un insieme di identificatori $\mathcal{ID}$ un oggetto è nella forma

$$(OID, stato)$$

dove lo stato è nella forma

$$(tipo, valore)$$

e il valore può essere

- un valore $v \in \mathcal{D}$, ossia un valore atomico se il tipo è atomico
- una tupla (oggetto tupla) $[a_1 : i_1, ..., a_n : i_n]$ dove $\forall j \ a_j \in A \wedge i_j \in \mathcal{ID}$ se il tipo è tupla
- un insieme (oggetto set) $\{i_1, ..., i_n\}$ dove $\forall j \ i_j \in \mathcal{ID}$ ossia un insieme di indirizzi se il tipo è set

Le tuple e i set contengono quindi indirizzi e non valori come nel modello CODL.

Riprendendo l'esempio di prima (nel modello CODL), i tre oggetti presenti nello spazio degli oggetti sono quì rappresentati come segue

$$(i_1, (tupla, [nome : i_4, cognome : i_5, indirizzo : i_6, figli : i_7]))$$

$$(i_4, (atomico, Mario))$$
$$(i_5, (atomico, Rossi))$$
$$(i_6, (tupla, [via : i_8, numero : i_9]))$$
$$(i_8, (atomico, Roma))$$
$$(i_9, (atomico, 1))$$
$$(i_7, (set, \{i_{10}, i_{11}\}))$$
$$(i_{12}, (atomico, Luca))$$
$$(i_{13}, (atomico, Paolo))$$

$$(i_{14}, (tupla, [nome : i_{12}, cognome : i_5, indirizzo : i_6, figli : i_{null}]))$$

$$(i_{15}, (tupla, [nome : i_{16}, cognome : i_{17}, indirizzo : i_{18}, figli : i_{21}))$$

$$(i_{16}, (atomico, Maria))$$

$$(i_{17}, (atomico, Verdi))$$

$$(i_{18}, (tupla, [via : i_{19}, numero : i_{20}]))$$

$$(i_{19}, (atomico, Garibaldi))$$

$$(i_{20}, (atomico, 3))$$

$$(i_{21}, (atomico, Paolo))$$

$$(i_{null}, (atomico, null))$$

Il concetto di uguaglianza è dato da una sola regola: due oggetti atomici $(i_1, s_1), (i_2, s_2)$ sono uguali se è uguale il loro indentificatore, cioè se $i_1 = i_2$. Notare che se $s_1 = s_2$ (s indica lo stato) ma $i_1 \neq i_2$ allora gli oggetti sono diversi. Uno spazio degli oggetti consistente è dunque un insieme di oggetti in cui ogni OID identifica uno ed un solo oggetto.

Notare che questo metodo permette di minimizzare la ridondanza potendo referenziare ogni dato duplicato, ma vengono quando un valore è immutabile o comunque non viene condiviso è più comodo inserirlo all'interno di un oggetto e non referenziarlo.
Per questo gli OODBMS scelgono una soluzione ibrida che uniscono il modello CODL con quello FAD.

Modello ibrido

Il modello **ibrido** unisce gli aspetti del modello CODL con quelli del modello FAD.
Formalmente
Sia $\mathcal{A}$ un insieme di nomi, $\mathcal{D}$ un insieme di valori atomici compreso $null$, $\mathcal{ID}$ un insieme di identificatori un oggetto è nella seguente forma

$$(OID, stato)$$

dove lo stato è nella seguente forma

$$(tipo, valore)$$

e il valore può essere

- un valore qualunque del modello CODL (quindi una tupla contenente valori, un insieme, ecc.) se il tipo è $valoreStrutturato$
- un identificatore se il tipo è Ref
- un insieme di identificatori se il tipo è $SetRef$
- una tupla $[a_1 : s_1, ..., a_n : s_n]$ dove $a_i \in \mathcal{A}$ per ogni i e s_i è un stato per ogni i se il tipo è tupla

Riprendendo il solito esempio, questo può essere ristrutturato come segue

$$(i_1, (tupla, [nome : (valoreStrutturato, Mario), cognome : (valoreStrutturato, Rossi)$$
$$, indirizzo : (Ref, i_2), figli : (Ref, i_3)]))$$
$$(i_2, (valoreStrutturato, [via : Roma, numero : 1]))$$
$$(i_3, (valoreStrutturato, \{Luca, Paolo\}))$$

$$(i_4, (tupla, [nome : (valoreStrutturato, Paolo), cognome : (valoreStrutturato, Rossi),$$
$$indirizzo : (Ref, i_2), figli : (valoreStrutturato, null)]))$$

$$(i_5, (tupla, [nome : (valoreStrutturato, Maria), cognome : (valoreStrutturato, Verdi),$$
$$indirizzo : (Ref, i_6), figli : (Ref, i_7)]))$$
$$(i_6, (valoreStrutturato, [via : Garibaldi, numero : 3]))$$
$$(i_7, (\{Marco\}))$$

Il concetto di uguaglianza di stati è dato dalle seguenti regole

- Due stati di tipo $valoreStrutturato$ sono uguali in base alle regole definite per il modello CODL.
- Due stati di tipo Ref sono uguali se hanno lo stesso valore (che indica un OID).
- Due stati di tipo $SetRef$ sono uguali in base al criterio di uguaglianza insiemistica (per gli indirizzi) dato nel modello FAD

- Due stati di tipo *tupla* sono uguali in base allo stesso criterio dato nel modello FAD ma considerando l'uguaglianza di stati al posto dell'uguaglianza di indirizzi (perché qui le tuple contengono stati e non indirizzi).

Sugli oggetti strutturati secondo questo modello è possibile eseguire delle operazioni di confronto, copia, insiemistiche e di merging.

- **Confronto**: esistono tre test, date due variabili x_1 e x_2 che contengono OID di due oggetti.

  1. **Equals**: $x_1 == x_2$ è vero se e solo se l'OID contenuto in x_1 è uguale all'OID contenuto in x_2.
     Ad esempio $(i_1, (...)) == (i_1, (...))$ ossia si tratta dello stesso oggetto.
  2. **Shallow equals**: $x_1 = x_2$ è vero se e solo se gli stati degli oggetti referenziati dalle due variabili sono uguali (si veda la definizione di uguaglianza tra stati precedentemente descritta).
     Ad esempio $(i_1, (valoreStrutturato, 1)) = (i_2, (valoreStrutturato, 1))$
     ma non è vero che $(i_1, (valoreStrutturato, 1)) == (i_2, (valoreStrutturato, 1))$ perché $i_1 \neq i_2$[2].
  3. **Deep equals**: $x_1??x_2$ è vero se e solo se dove gli stati degli oggetti sono di tipo tupla, Ref o SetRef, tutti gli oggetti identificati dagli OID (all'interno di tuple, set, ecc.) sono deep equals (notare che la definizione è ricorsiva) e e dove gli stati degli oggetti siano valoreStrutturato i valori siano uguali.
     Ad esempio $(i_1, (Ref, i_3))??(i_2, (Ref, i_4))$ con
     $(i_3, (valoreStrutturato, 10)), (i_4, (valoreStrutturato, 10))$ è vero in quanto gli oggetti identificati da i_3 e i_4 sono deep equals in quanto hanno lo steso valore (caso base della definizione ricorsiva).
     Non è vero però che i due oggetti (identificati da i_1 e i_2) sono shallow equals siccome non hanno lo stesso stato perché sono diversi gli indirizzi contenuti in essi. Nello stesso modo i due oggetti non sono neanche uguali perché hanno due OID diversi.

     Si nota facilmente che *Equals* implica *Shallow equals* la quale implica *Deep equals*.

[2]Si assume qui che i_1 e i_2 siano costanti e non valori memorizzati in variabili, così da assumere che essi siano sempre diversi.

- **Copy**: esistono tre tipi di copia, date due variabili x_1 e x_2 che contengono OID.

    1. **Assegnamento**: $x_1 := x_2$ copia l'OID contenuto in x_2 in x_1 e quindi rende vera l'asserzione $x_1 == x_2$
    2. **Shallow copy**: $x_1 :=_s x_2$ crea un nuovo oggetto (quindi con un nuovo OID) con lo stesso stato dell'oggetto identificato da x_2 e lo assegna a x_1 rendendo vero il test $x_1 = x_2$ ma falso il testo $x_1 == x_2$.
    3. **Deep copy**: $x_1 :=_d x_2$ crea un nuovo oggetto con uno stato equivalente in profondità ma in cui gli OID referenziati dal nuovo oggetto sono tutti nuovi, per cui è vero $x_1??x_2$ ma falso $x_1 == x_2$.

- **Merge**: dati due oggetti o_1 e o_2, shallow equals crea un oggetto equivalente ai due $o_3 = merge(o_1, o_2)$ e ad ogni variabile che referenzia o_1 o o_2 viene associato il nuovo valore o_3 e vengono rimossi gli oggetti o_1 e o_2.

- **Operazioni insiemistiche**: dati due insiemi di oggetti S_1 e S_2 le operazioni insiemistiche sono $S_1 \cup S_2$ (unione), $S_1 \cap S_2$ (intersezione), $S_1 - S_2$ (differenza).
  Queste operazioni possono essere effettuate in diversi modi in base al tipo di confronto tra oggetti. Ad esempio $S_1 \cup S_2$ significa che il nuovo insieme contiene gli oggetti di S_1 e S_2 ma se vi è un oggetto $o_1 \in S_1$ ed un oggetto $o_2 \in S_2$ tali che $o_1 = o_2$ il nuovo insieme dovrà contenere due oggetti distinti oppure no? Dipende da cosa si intende per uguaglianza (si scegli dunque un predicato di uguaglianza tra quelli descritti: equals, shallow, deep).

- **Selezione**: con selezione si intende l'operatore di selezione di tuple data una condizione

  $$Selection(S_1, ..., S_n, \lambda(x_1, ..., x_n, e)) = \{e(s_1, ..., s_n) | s_1 \in S_1 \wedge ... \wedge s_n \in S_n\}$$

  dove $S_1, ..., S_n$ sono insiemi di oggetti, per $\lambda(x_1, ..., x_n)$ si indica che si intende utilizzare le variabili $x_1, ..., x_n$ e e è un'espressione condizionale che indica la condizione che devono rispettare le tuple per appartenere all'insieme restituito ed il loro formato (proiezione).
  Ad esempio l'operazione di selezione $Selection(Persona, \lambda(x), x.nome =$

$Mario \rightarrow \{[nome : x.nome, cognome : x.cognome])\}$ restituisce un insieme di tuple nella forma indicata dall'espressione condizionale il cui nome coincide con Mario.

- **Nest**: si applica ad insiemi S di oggetti tupla $[a_1 : t_1, ..., a_i : t_i, ..., a_n : t_n]$.
  Sia $A = \{a_1, ..., a_n\}$,

  $$nest(S, A - \{a_i\}) = [a_1 : t_1, ..., a_i : \{t_i\}, ..., a_n : t_n]$$

  Ossia questa operazione crea un insieme di tuple dove tutti i valori di t_i presenti in tutte le tuple i cui valori degli attributi $A - \{a_i\}$ coincidono sono raggruppati in un'unica tupla. Si tratta dunque di un'operazione simile a group by.
  Ad esempio, $nest(\{[nome : Mario, cognome : Rossi], [nome : Luca, cognome : Rossi]\}, nome) = \{[nome : \{Mario, Luca\}, cognome : Rossi]\}$

- **UnNest**: è l'operazione inversa di nest. Dato un insieme di tuple $S = \{[a_1 : t_1, ..., a_i : \{t_i\}, ..., a_n : t_n]\}$

  $$UnNest(S, A_i) = \{[a_1 : s.a_1, ..., a_i : x, ..., a_n : s.a_n] | s \in S \wedge x \in s.a_i\}$$

Un altro aspetto fondamentale delle basi di dati è la gestione della *persistenza*, ossia come viene strutturata l'area che memorizza i dati (in questo caso oggetti) nel database.
Esistono due possibili approcci

- *estensioni persistenti delle classi*: la classe gestisce la persistenza della sua estensione (oggetti generati dalla classe).
  Questo si può realizzare rendendo persistente ogni oggetto creato, permettendo di specificare l'opzione persistenza nei suoi costruttori oppure mettendo a disposizione dei metodi specifici per gestire la persistenza degli oggetti.

- *persistenza per raggiungibilità*: si creano uno o più oggetti database, che sono tuple dove ogni attributo contiene un valore set dove sono presenti tutti gli oggetti persistenti. Un oggetto è dunque persistente se è raggiungibile dall'oggetto database.
  Tutti gli oggetti transienti invece sono contenuti da altri oggetti che non fanno parte degli oggetti database.

1.3 Relazioni annidate

Il *modello relazionale a oggetti* si basa su una proposta di estensione del modello relazionale dove le relazioni non sono in prima forma normale. Queste relazioni prendono il nome di **relazioni annidate**
Quello che viola la prima forma normale è l'esistenza di attributi composti.
Una **relazione** è dunque una regola nella seguente forma

$$R = (R_1, ..., R_n)$$

dove $R, R_1, ..., R_n$ sono nomi di attributi (si omette il dominio per semplicità).
Se, dato un insieme di relazioni ed un R_i questo non appare come nome di relazione allora si tratta di un **nome di ordine zero**, che significa che questo attributo ha un dominio primitivo (intero, stringa, ecc.). In caso contrario si parla di **nome di ordine maggiore di zero**, cioè di un attributo composto da altri attributi (cosa vietata nella prima forma normale).
In ogni caso, non sono ammessi richiami di nomi ricorsivi. Quindi ad esempio non è ammesso il seguente schema

$$R = (R_1, ..., R, ..., R_n)$$

Ad esempio, date le seguenti relazioni
Persona(Nome, Cognome, Indirizzo)
Abitazioni(Via, Numero)
Gli attributi Nome, Cognome, Via e Numero sono di ordine zero, mentre Indirizzo no in quanto è composto da Via e Numero.

Un'**istanza** di una relazione R è una coppia $\langle R, r \rangle$ dove r è il valore abbinato al nome R della relazione, ossia un insieme finito di tuple.
Quando vi è un attributo di ordine maggiore di zero, il suo valore r è dunque un insieme di attributi.
Ad esempio, siano date due istanze con valori r_1, r_2 della relazione Persona.

r_1

Nome	**Cognome**	**Abitazioni**	
		Via	**Numero**
Mario	Rossi	Roma	1
		Pitagora	30
Mario	Rossi	Massimo	10
Paolo	Rossi	Roma	1
Maria	Verdi	Garibaldi	3
		Verdi	6

r_2

Nome	**Cognome**	**Abitazioni**	
		Via	**Numero**
Enrico	Verdi	Garibaldi	30
		Roma	3
Mauro	Bruni	Vittorio	8

Si noti che queste istanze gli l'attributo abitazioni può avere più valori, in quanto il nome richiama una relazione esistente.

L'algebra relazionale per questo tipo di modello (modello relazionale esteso con le relazioni annidate) che è un'estensione dell'algebra relazionale classica deve dunque prevedere, oltre ad attributi con un singolo valore (come nel modello relazionale classico) anche insiemi di tuple.
L'algebra relazionale è qui estesa con due nuovi operatori citati nel precedente paragrafo: nest e unnest.
Siano $R1$, B, $R2$ tre regole nella seguente forma

$$R1 = (a_1, ..., a_n, b_1, ..., b_m)$$

$$B = (b_1, ..., b_m)$$

$$R2 = (a_1, ..., a_n, B)$$

e sia r_1 un'istanza di $R1$, r_2 un'istanza di $R2$, $A = \{a_1, ..., a_n\}$ e $t[a_1, ..., a_n]$ la proiezione di una tupla t per gli attributi $a_1, ..., a_n$

$$Nest(r_1; B) = \{t | \forall x \; x \in r_1 \quad t[a_1, ..., a_n] = x[a_1, ..., a_n] \rightarrow x[b_1, ..., b_m] \in t[B]]\}$$

Questo insieme è l'istanza $r2$.
Questo significa che l'operatore nest forma tuple in cui ogni tupla i cui valori di $a_1, ..., a_n$ coincidono nella relazione originaria sono raggruppati in una unica i cui valori $b_1, ..., b_m$ sono raggruppati in un insieme di tuple.
Possiamo dunque dire che, data una relazione in prima forma normale, l'operazione di Nest ne crea una annidata e dunque non più in forma normale.
L'operazione unnest è invece (così si desidera) l'operazione inversa di nest

$$UnNest(r_2; B) = \{t | \exists x \in r_2 \quad t[a_1, ..., a_n] = x[a_1, ..., a_n \rightarrow t[b_1, ..., b_n] \in x[B]]\}$$

L'operazione UnNest, dato un attributo composto di un'istanza di relazione, crea dunque un insieme di tuple formate da attributi semplici al posto di quello composto. Si possono effettuare consecutivamente più UnNest, uno

per ogni attributo composto fino ad ottenere un insieme di tuple in prima forma normale. Questa operazione è detta **flatten**.

Viene ora mostrato un esempio di Nest e uno di Unnest sulla relazione $r1$.

$$UnNest(r_1, Abitazioni)$$

Nome	**Cognome**	**Via**	**Numero**
Mario	Rossi	Roma	1
Mario	Rossi	Pitagora	30
Mario	Rossi	Massimo	10
Paolo	Rossi	Roma	1
Maria	Verdi	Garibaldi	3
Maria	Verdi	Verdi	6

$$Nest(UnNest(r_1, Abitazioni), Abitazioni)$$

Nome	**Cognome**	**Abitazioni**	
		Via	**Numero**
Mario	Rossi	Roma	1
		Pitagora	30
		Massimo	10
Paolo	Rossi	Roma	1
Maria	Verdi	Garibaldi	3
		Verdi	6

Notare che l'istanza appena prodotta non è uguale a r_1 in quanto per Mario Rossi c'è un unica tupla che raggruppa tutte le sue abitazioni mentre in r_1 ce n'erano due.
Concludiamo quindi che

$$Nest(UnNest(r_1, Abitazioni), Abitazioni) \neq r_1$$

In generale questo avviene proprio perché possono esistere tuple distinte con attributi semplici o composti $a_1, ..., a_n$ identici che quindi raggruppano più insiemi di tuple B. Nell'operazione di nest invece (rispetto a B) invece tutte le tuple dove i valori degli attributi $a_1, ..., a_n$ coincidono vengono raggruppate in un'unica tupla.
Affinché nest sia l'operazione inversa di unnest, occorre che le tuple si presentino in una forma particolare: la **PNF**.

Definizione 4. *Dato uno schema relazionale* $R = (a_1, ..., a_n, X_1, ..., X_m)$ *con* $a_1, ..., a_n$ *nomi di ordine zero e* $X_1, ..., X_m$ *nomi di ordine maggiore di zero, l'istanza* r *di* R *è in* ***Partition Normal Form*** *(*PNF*) se e solo se*

- $a_1, ..., a_n$ *è una superchiave*
- *per ogni* X_i, $\forall t \in r$, $t[X_i]$ *è in PNF.*

Da questo segue che $Nest(UnNest(r, B), B) = r$ dove B è un nome di ordine maggiore di zero di R e r è istanza di R e r è in PNF.
Possiamo estendere la definizione di alcuni operatori dell'algebra relazionale in modo da ottenere la chiusura rispetto alla forma PNF.
Siano r_1, r_2 due istanze di relazioni

$$r_1 \cup r_2 = Ricost(Flatten(r_1) \cup Flatten(r_2))$$

Dove *Ricost* è la ricostruzione dell'insieme ottenuto con l'unione classica del flatten delle due istanze, ossia l'annidamento consecutivo del risultato fino a riottenere un'istanza nella stessa forma di r_1 e r_2.
Questo significa che si riconducono in prima forma normale le relazioni r_1 e r_2, si effettua l'unione e si re-annidano così come lo erano originariamente. Ad esempio $UnNest(r_1, Abitazioni)$ nel caso dell'istanza r_1 mostrata nell'esempio equivale al flatten. Quindi $r_1 \cup r_2$ dove r_1 e r_2 sono le istanze dei due esempi sopra citati equivale a

$r_1 \cup r_2$

Nome	**Cognome**	**Abitazioni**	
		Via	**Numero**
Mario	Rossi	Roma	1
		Pitagora	30
		Massimo	10
Paolo	Rossi	Roma	1
Maria	Verdi	Garibaldi	3
		Verdi	6
Enrico	Verdi	Garibaldi	30
		Roma	3
Mauro	Bruni	Vittorio	8

che è in PNF.
Altre estensioni sono

$$r_1 - r_2 = Ricost(Flatten(r_1) - Flatten(r_2))$$

Il prodotto cartesiano $r_1 \times r_2$ è già chiuso rispetto alle relazioni in PNF perché si assume che gli schemi di r_1 e r_2 siano disgiunti e ovviamente r_1 e r_2 sono in PNF.
L'operatore di selezione $\sigma_c(r)$ è già chiuso rispetto alle relazioni in PNF (ma occorre aggiungere nelle condizioni c insiemi di tuple costanti, operazioni di confronto tra insiemi e operazione $\in$), mentre la proiezione π è chiusa rispetto alle relazioni in PNF.

Secondo il modello delle relazioni annidate, un'istanza di relazione è un insieme di tuple che possono contenere a loro volta insiemi di tuple. Abbiamo dunque che un'istanza è un *set* di *tuple* che possono contenere *set* di *tuple*.
Se rilasciamo questo vincolo sull'ordine di applicazione *set*, *tuple* si ottiene il modello generale ad oggetti complessi, discusso nel precedente paragrafo (ad esempio il modello ibrido).
Tale modello è riconducibile al modello annidato.
Infatti si hanno due casi

- $[a_1 : q_1, ..., a_i : [b_1 : r_1, ..., b_m : r_m], ..., a_n : q_n]$ questo è riconducibile a $[a_1 : q_1, ..., b_1 : r_1, ..., b_m : r_m, ..., a_n : q_n]$.
- $set(set(t))$ diventa invece $set([x : set(t)])$

Capitolo 2

OODBMS e standard

Ora che è stato introdotto il **modello relazionale a oggetti** possiamo descrivere le sua applicazioni.
Ciò che distingue il modello relazionale ad oggetti da quello relazionale è sostanzialmente l'inserimento di OID che identificano oggetti, mentre nel modello relazionale il concetto chiave è che tutto è un valore ed alcuni valori (chiavi) identificano le tuple.

Un **OODBMS** (*Object Oriented DBMS*) è un DBMS orientato agli oggetti. Tipicamente questi DBMS sono quasi completamente retrocompatibili con i RDBMS. Il linguaggio utilizzato è sempre SQL ma è stato introdotto un nuovo standard.
Se **SQL2** era lo standard utilizzato per i DBMS relazionali, una sua estensione **SQL99**, introdotta nel 1999, è ora utilizzata per gli OODBMS.

2.1 SQL99

Lo standard SQL99 aggiunge nuove funzionalità ai DBMS tra cui nuovi tipi di dato non strutturati detti *Large OBject* (LOB).
Questi oggetti sono trattati come stringhe di bit o di caratteri e il DBMS non assegna alcuna semantica a questi dati: è compito dell'utente e delle applicazioni manipolarli.
Dal nome (large object) è evidente che questi tipi vengano utilizzati per memorizzare grosse quantità di dati come contenuti multimediali (immagini, video, musica, ecc.).
I tipi di dati non strutturati forniti dall'SQL99 sono

- **BLOB**: Binary Large Object - è una stringa di bit. Si dichiarano come BLOB(n) dove n è la dimensione del dato

- **CLOB**: Character Large Object - è una stringa di caratteri. Si dichiarano come CLOB(n) dove n è la dimensione del dato

Per quanto riguarda la dimensione n, questa può essere espressa in byte (n, ad esempio 10), in kilobyte (nK, ad esempio $10k$), in megabyte (nM) o in gigabyte (nB).
Un *CLOB* o un *VARCHAR* può essere utilizzato con l'istruzione facoltativa **CHARACTER SET char_base** per indicare che l'insieme di caratteri da utilizzare è l'insieme chiamato char_base.
Notare che non si possono eseguire le comuni operazioni su questi tipi di dato: si possono solo eseguire test di uguaglianza ($=, \neq$).

SQL99 aggiunge poi alcune operazioni per manipolare i BLOB (utilizzate anche per manipolare le stringhe ed estese ai blob) tra cui

- *POSITION(str1 IN str2)* che restituisce la posizione della prima occorrenza di str1 in str2 (stringhe di bit), contando che il primo carattere della stringa è in posizione 1.
  Ad esempio $POSITION(X'110'INb) = 3$ se il blob è la seguente stringa di bit $X'00\mathbf{110}00001'$

- *str1 —— str2* concatena le due stringhe

- *SUBSTRING(str FROM pos [FOR n])* estrae n caratteri a partire dal carattere in posizione *pos* della stringa *str*

- *CHARLENGTH(str)* numero di caratteri (byte) presenti nella stringa. Notare che è anche disponibile la funzione *BITLENGTH(str)* che restituisce il numero di bit della stringa.

- *LIKE* come in SQL2, ma esteso ai BLOB

Un altro tipo di dato aggiunto è **BOOLEAN** che può assumere tre valori costanti: *true*, *false* e unknown così come è definito nella semantica di CODD[1].

Lo standard SQL99 permette anche di definire funzioni ausialiarie da utilizzare nelle query o comunque per restituire un valore.
La sintassi è la seguente

[1]Il valore unknown viene restituito per quei test in cui sono coinvolti valori NULL per cui non si può stabilire un valore di verità. Ad esempio il test $salario > 10$ è unknown se salario è NULL.

```
CREATE FUNCTION nome (par1 TIPO, ..., parn TIPO)
                    RETURNING tipo_di_ritorno
BEGIN
    ...
    RETURN valore
END
```

Questa dichiarazione crea una funzione chiamata 'nome' che restituisce valori di tipo 'tipo di ritorno'.
Ad esempio possiamo creare la seguente funzione

```
CREATE FUNCTION distanza(start INTEGER, end, INTEGER)
        RETURNING INTEGER
BEGIN
    RETURN end - start;
END
```

Ed utilizzarla nella seguente query

```
SELECT *
FROM PERCORSO AS P
WHERE distanza(P.inizio, P.fine) > 100
```

per selezionare tutti i percorsi di lunghezza maggiore di 100 (assunto che la tabella percorso contenga i punti inizio e fine di una tracciato).

Se il linguaggio offerto per implementare le funzioni non è abbastanza espressivo, occorre trattare in modo particolare dati multimediali o occorre accedere a primitive offerte offerte dal sistema operativo, è anche possibile implementare esternamente le funzioni, con la seguente sintassi

```
CREATE FUNCTION nome(...)
        RETURNING tipo;
EXTERNAL NAME "...";
LANGUAGE nome-linguaggio;
```

Oltre alle funzioni e ai tipi di dato primitivi aggiunti, SQL99 permette di inserire *tipi di dato definiti dall'utente*.
Esistono tre approcci per definire un tipo di dato.
Il primo metodo consiste nel creare tipi distinti, ossia tipi con nome diverso da quelli esistenti ma che sono equivalenti.
La sintassi è

```
CREATE TYPE tipo AS tipo-definito;
```

dove 'tipo-definito' è un tipo che è già stato definito dall'utente oppure un tipo primitivo.
Ecco un esempio di creazione di tipo e conseguente definizione di tabella.

```
CREATE TYPE TESTO AS VARCHAR2(1000);
CREATE TABLE ALBERGO
(
      NOME VARCHAR2(30),
      INDIRIZZO VARCHAR2(1000),
      DESCRIZIONE TESTO,
      NOTE TESTO
);
```

Notare che l'attributo INDIRIZZO e l'attributo DESCRIZIONE, pur rappresentando lo stesso tipo di dato non sono confrontabili.
L'idea che conduce a questa scelta è quella di avere un sistema di tipi forte, per cui si tenta di localizzare subito gli errori di tipo e non durante l'esecuzione dell'applicazione.
Di fatto la scelta di imporre il tipo TESTO per gli attributi DESCRIZIONE e NOTE va fatta perché si tratta di tipi di dato di natura diversa. La definizione di tipo aiuta anche a migliorare la leggibilità (ad esempio è più comprensibile avere un attributo debito di tipo MONEY piuttosto che un attributo di tipo DECIMAL).
Tuttavia, per confrontare i campi TESTO e INDIRIZZO (l'istruzione $ALBERGO.DESCRIZIONE = ALBERGO.INDIRIZZO$ è illegale) si può ricorrere ai costruttori. In generale, dato un tipo T definito con *CREATE TYPE*, il costruttore è
T(tipo o tipi di base)
e l'operazione
tipo-base(T) è detta invece distruttore.
L'istruzione *TESTO('stringa')* richiama dunque il costruttore di TESTO fornendo un valore alfanumerico di tipo TESTO.
Il confronto tra TESTO e INDIRIZZO è così possibile tramite il test $ALBERGO.DESCRIZIONE = TESTO(ALBERGO.INDIRIZZO)$.

Altri tipi di dato definibili sono i *tipi opachi*, ossia tipi non strutturati di cui si dichiara solo la dimensione occupata.
La sintassi è la seguente

```
CREATE TYPE nome
      (INTERNALLENGTH = size)
```

dove 'size' indica il numero di byte occupati.
Il programmatore avrà il compito di definire i metodi di memorizzazione nel database dei dati di questo tipo.

L'ultimo approccio, e quello più espressivo, è la costruzione di tipi *riga*. Questo consente di creare tipi che contengono all'interno più attributi (i quali a loro volta possono essere di tipo riga).
La sintassi è la seguente

```
CREATE TYPE nome AS
(
    attr1 TIPO1,
    ...
    attrn TIPOn
);
```

Questi tipi possono essere utilizzati come tipi di attributi nella definizione di una tabella, dove si possono dichiarare anche tipi riga anonimi.
Ecco un esempio

```
CREATE TYPE INDIRIZZO AS
(
    VIA VARCHAR(30),
    NUMERO INTEGER
);
CREATE TABLE PERSONA
(
    NOME VARCHAR(30),
    COGNOME VARCHAR(30),
    ABITAZIONE INDIRIZZO,
    OCCUPAZIONE ROW(TIPO VARCHAR(30),
                    STIPENDIO INTEGER)
);
```

L'attributo OCCUPAZIONE è stato tipato con un tipo riga anonimo.
Un tipo così costruito ha associato un costruttore implicito i cui parametri sono tutti gli attributi presenti nella dichiarazione di tipo.
Ecco dunque come inserire i dati nella tabella PERSONA

```
INSERT INTO PERSONA VALUES(
        'Mario','Rossi', INDIRIZZO('Roma', 1),
        ROW('Programmatore', 1500));
```

Dove *ROW* è il costruttore per i tipi di dato anonimi.
Data un'istanza p di PERSONA, per accedere alla via in cui abita, si può usa-

re la solita notazione con il punto, ossia *p.abitazione.via* (via è un attributo dell'attributo abitazione di tipo indirizzo).

Per essere più completi, occorre dire che questo tipo di dichiarazione permette di creare tipi di oggetti perché si possono inserire metodi.
Ad esempio, l'approccio usato da ORACLE (un OODBMS) è quello di definire prima l'interfaccia, ossia l'insieme di metodi e attributi (fanno parte dell'interfaccia nelle basi di dati) e poi l'implementazione dei metodi in una dichiarazione separata.
La sintassi per la definizione dei tipi è la seguente (in ORACLE)

```
CREATE TYPE nome AS OBJECT
(attr1 TIPO1, ..., attrn TIPOn
MEMBER PROCEDURE m1(par1 TIPOx1, ..., parn TIPOxn),
...
MEMBER FUNCTION mn RETURN TIPO);
```

dove PROCEDURE sta per funzione che non restituisce nessun valore (come i metodi void di Java).
L'implementazione dei metodi è invece fornita dalla seguente sintassi

```
CREATE TYPE BODY nome AS
MEMBER PROCEDURE m1(par1 TIPOx1, ..., parn TIPOxn) IS
BEGIN
      ...
END m1;
...
MEMBER FUNCTION mn RETURN TIPO IS
BEGIN
      ...
END mn;
END;
```

Dove all'interno del corpo dei metodi sono accessibili gli attributi.
Notare che, dato un attributo di tipo oggetto, è possibile richiamare una funzione f dichiarata con CREATE TYPE ovunque sia possibile richiamare un attributo, quindi ad esempio nelle query. Ad esempio, come è possibile ottenere il valore del nome di una persona con *persona.nome* è anche possibile richiamare un metodo *persona.getIndirizzo*(), supponendo che nome sia un attributo della tabella di cui persona è istanza, e che lo stesso valga per il metodo getIndirizzo.
Come per tutti i linguaggi object oriented, ORACLE mette a disposizione nel corpo dei metodi il parametro implicito SELF (this in java).

SQL99 offre la possibilità di creare tabelle come insiemi di oggetti tupla tramite la seguente istruzione

```
CREATE TABLE nome OF tipo
(vincoli);
```

Questa istruzione crea una tabella in cui ogni tupla è identificata da un OID e contiene gli attributi dichiarati nella dichiarazione di 'tipo', dove 'tipo' è un tipo riga. Inoltre, è possibile aggiungere vincoli su tali attributi come vincoli di chiave, not null ecc. Una tabella definita in questo modo è detta **tavola tipata**. Le tavole tipate hanno dunque lo stesso ruolo delle classi, dove le loro tuple (estensione della classe definita nel precedente capitolo) sono oggetti, ossia istanze della tavola tipata.
I vincoli vanno specificati nella forma *attributo WITH OPTIONS ....*

Per completare il modello ibrido mostrato nel capitolo 1, mancano i referenziamenti, ossia la possibilità di riferirsi ad oggetti tramite OID e mancano i set.
SQL99 offre la possibilità di avere attributi di tipo **REF(T)**, ossia riferimento ad una tupla di tipo T tramite OID.
Notare la differenza tra REF(T) ed il tipo T. Sia T un tipo riga, un attributo di tipo T contiene il valore di una tupla di tipo T, mentre un attributo di tipo T ne contiene l'indirizzo (OID).
Per specificare esattamente la tabella a cui un tipo REF si deve riferire (un attributo di tipo REF(T) può referenziare le tuple di tutte le tabelle OF T, cioè di tipo T) occorre inserire il vincolo *WITH OPTION SCOPE nome-tabella-tipo-T*.
Il vincolo *IS SYSTEM GENERATED* applicato a campi di tipo REF, indica invece che l'OID degli oggetti referenziati viene generato automaticamente dal sistema (è consigliato inserirlo sempre).
REF può essere usata anche come funzione dove, data una tupla oggetto t, **REF(t)** restituisce l'indirizzo della tupla oggetto.
Per inserire un valore di tipo REF in una tabella si può far ricorso a query, cioè se si ha una attributo A di tipo REF(T) con SCOPE impostato sulla tabella TAB, è possibile eseguire la seguente istruzione

```
INSERT INTO TABELLA (..., A, ...)
VALUES (..., (SELECT REF(A) FROM TAB WHERE COND), ...)
```

assumendo che venga restituita una sola tupla.

Un altro tipo di dato presente in SQL99 è l'array: TIPO ARRAY[n], che indica un array di elementi di tipo 'TIPO' di dimensione n.
Gli array possono essere di qualunque tipo eccetto array (non è possibile

dichiarare array di array). Pertanto sono ammessi solo array monodimensionali.
Sfortunatamente, SQL99 non tratta per ora il tipo SETOF, ossia gli insiemi, ma alcuni OODBMS li utilizzano comunque.
Il tipo SETOF(T) indica un insieme di oggetti di tipo T. Se T è un tipo riga, allora gli attributi di tipo T sono di ordine maggiore di zero (come mostrato nel capitolo 1 discutendo delle relazioni annidate).
Esiste poi l'operatore **IN** per verificare che un elemento appartenga ad un insieme (SETOF).

Un'altra opzione, fornita da SQL99 è la possibilità di creare tavole annidate: tavole che vengono utilizzate all'interno di altre tavole.

```
CREATE TYPE nome AS TABLE OF tipo-tuple-annidate;
```

Gli attributi di tipo 'nome' hanno come valore una tavola relazionale, ossia un insieme di tuple.
Con tutti questi costrutti, si rischia di avere confusione quando si utilizza un nome, se non si ha ben presente se questo è un nome di tipo, tavola o tavola annidata.
Oracole suggerisce di inserire i suffissi **_TY** per i tipi e **_NT** per le tavole annidate.

Con la sintassi descritta possiamo dunque strutturare gli oggetti secondo il modello ibrido e creare *relazioni annidate*.
L'operatore UnNest (vedi capitolo 1) può essere implementato in Oracle tramite il costrutto TABLE(t) dove t è una tavola annidata, che restituisce l'insieme di tuple della tavola trasformandolo in un valore di tipo tuple.
Dati gli schemi $A = (A_1, ..., A_n)$ e $B = (B_1, ..., B_m, A)$, l'operazione $UnNest(b, A)$, dove b è un'istanza di B può essere implementata in SQL nel seguente modo

```
SELECT b.B1, b.B2, ..., b.Bn, a.*
FROM B b, TABLE(b.A) a;
```

Questo permette di ottenere tuple nel formato $(A_1, ..., A_n, B_1, ..., B_m)$ grazie al fatto che si trasforma la tavola annidata A in un insieme di tuple che vengono fatte corrispondere con quelle di $B - A$ tramite join.

Anche l'operatore Nest può essere implementato con Orocale, utilizzando due nuovi costrutti: cast (converte un tipo in un altro) e multiset (converte

una tavola, cioè un valore di tipo tuple, in un multi-insieme[2] di tuple).

Per completare il quadro, occorre definire l'ereditarietà nello standard SQL99.
L'ereditarietà è possibile tra tipi e tabelle inserendo dopo la loro dichiarazione **UNDER** nome-tabella o tipo.
Segue un esempio

```
CREATE TABLE Persona ...
CREATE TABLE PersonaConOcchiali UNDER Persona;
```

L'ereditarietà consiste qui nell'ereditare tutte le caratteristiche dei sovratipi, anche gli eventuali vincoli.
Notare che l'ereditarietà multipla in SQL99 non è supportata.
La semantica dell'ereditarietà è estensionale in profondità (vedi capitolo 1); ciò significa che ogni tupla di PersonaConOcchiali è ANCHE una tupla di Persona.
Quindi la query

```
SELECT *
FROM Persona;
```

restituisce anche tutte le tuple di persona con occhiali.
Se si vogliono ottenere le sole tuple di persona occorrerà inserire la clausola **ONLY**

```
SELECT *
FROM ONLY(Persona);
```

SQL99 prevede anche le query ricorsive con la seguente sintassi

```
WITH RECURSIVE R(A1, ..., An)
SELECT ...
FROM ..., R
 ...
```

Dove R è una relazione le cui tuple vengono considerate nella clausola FROM man mano che queste vengono elencate.
Questo significa che ogni tupla selezionata nella query viene a sua volta esaminata e testata nella clausola WHERE.
Un esempio di applicazione di questa ricorsione è la generazione delle chiusure transitive.

[2]SQL lavora con multi-insiemi a differenza dell'algebra relazionale poiché, non essendo obbligatoria la presenza di una chiave primaria, in una tavola o in una risposta SQL possono esserci tuple uguali

Si ricorda che data una relazione $R(A, B)$ ed un'istanza di R, la sua chiusura transitiva è un'istanza che comprende tutte le tuple dell'istanza originaria, e tutte le tuple $[a_i, b_j]$ tali che esiste una tupla $[a_i, b_i]$ e una tupla $[a_j = b_i, b_j]$ nella chiusura transitiva.
Un esempio è costituito dalla relazione *discendenza* che è la chiusura transitiva delle relazione *famiglia*: la relazione famiglia ha istanze con tuple di tipo $[padre, figlio]$, mentre quella discendenza ha tuple di tipo $[antenato, persona]$, ossia tutte le tuple $[padre, figlio]$ perché il padre è un antenato del figlio ma, esistendo le tuple di tipo $[nonno, padre]$ (il nonno è il padre del padre) esistono anche le tuple $[nonno, figlio]$ (il nonno è un antenato del figlio). Un'eventuale query che effettua tale chiusura transitiva a partire dalla relazione famiglia, una volta elencate le tuple di tipo $[bisnonno, nonno]$ e $[nonno, figlio]$ deve riconsiderarle (leggere ricorsivamente il risultato della stessa query) per aggiungere la tupla $[bisnonno, figlio]$ facente parte della chiusura (il bisnonno è un antenato del figlio).
La query è dunque la seguente

```
WITH RECURSIVE ANTENATI(Antenato, Discendente)
SELECT Padre, Figlio
FROM FAMIGLIA
UNION
SELECT f.Padre, f.Discendente
FROM FAMIGLIA f, ANTENATI a
WHERE f.Figlio = a.Antenato
```

questa query si può naturalmente generalizzare per ogni relazione $R(A, B)$.

Un'altra utile possibilità che offre SQL99 è quella di poter scegliere il livello di isolabilità di una transazione ACID, ossia quanto debba essere garantita la serializzabilità di una transazione. Un insieme di transazioni eseguite in modo completamente seriale sono meno efficienti dello stesso insieme eseguito in modo parallelo e, se non si è interessati ad un livello totale di serializzabilità si possono scegliere modalità meno restrittive per migliorare l'efficenza. Esistono quattro livelli di isolabilità delle transazioni

1. **Read uncommitted**(LV1): vengono ammesse le letture soprche, ossia se una transazione T1 aggiorna un dato A, la transazione T2 legge tale dato e successivamente T1 esegue un rollback, allora T2 ha letto un dato che non esiste in quanto il rollback di T1 comporta la riscrittura del vecchio dato presente in A.
   Tuttavia, se si è interessati ad esempio solo al conteggio delle tuple presenti in una tabella questo livello di isolabilità è adeguato per cui

lo si può scegliere avendo una maggiore efficienze nell'esecuzione delle query in parallelo con altri utenti.

2. **Read comitted**(LV2): vengono ammesse letture non ripetibili, ossia se una transazione T1 legge un dato A, una transazione T2 può successivamente scrivere sullo stesso dato A prima che T1 abbia effettuato il commit, quindi successive letture di dati A si riferiscono ad un dato diverso. Tuttavia, se T1 legge solo una volta i dati questo livello di isolabilità è adeguato.

3. **Repeatable read**(LV3): vengono ammesse letture fantasma, ossia se una transazione T1 legge un insieme di dati A e T2 inserisce un nuovo dato nell'insieme A, allora la successiva lettura di A comprende un dato in più.

4. **Serializzable**(LV4): garantisce un livello completo di serializzabilità, a discapito naturalmente dell'efficienza.

Notare che SQL99 fornisce anche la possibilità di impostare SAVEPOINT e di effettuare ROLLBACK ad un determinato SAVEPOINT.

L'ultima considerazione da fare è sulle **basi di dati attive**, ossia basi di dati che contengono azioni svolte in automatico al verificarsi di determinati eventi.

Queste operazioni sono svolte grazie ai **trigger**, operazioni che vengono svolte al verificarsi di determinati eventi.
La sintassi è la seguente

```
CREATE TRIGGER nome
{BEFORE | AFTER} evento ON nome-tabella
[REFERENCING {NEW | OLD} {ROW | TABLE} nome-elemento]
[FOREACH {ROW | STATEMENT}]
[WHEN condizione]
azione
```

Le parentesi quadre indicano che le istruzioni non sono obbligatorie.

Un 'evento' può essere INSERT, UPDATE[OF lista-colonne] o DELETE. BEFORE indica che il trigger viene attivato prima di un evento (ad esempio prima dell'inserimento di una riga nella tabella in esame, ossia nome-tabella) e dunque i valori della tabella in questione fanno riferimento allo stato **prima** dell'esecuzione dell'operazione. Al contrario con AFTER il trigger viene effettuato dopo lo svolgimento dell'operazione.

La clausola REFERENCING dichiara il nome della riga o tabella aggiornata, inserita o rimossa (in base all'evento scelto), per poter riferirsi ad essa tramite quel nome.
NEW o OLD indicano se ci si vuole riferire allo stato della riga prima o dopo l'eseguirsi dell'operazione che ha scaturito l'evento.
La clausola FOREACH indica se l'evento si deve attivare, ad ogni evento, una volta per ogni riga contenuta nella tabella in esame oppure una volta sola.

Per riassumere, abbiamo una **base di dati attiva** se un sono presenti i trigger.
Notare che quando si ricorre alla notazione **PL/SQL** si intende il linguaggio SQL unito a PL (Procedural Language) ossia il linguaggio che permette di dichiarare function e procedure (precedentemente descritte).
Molte delle estensioni comprese in SQL99, quindi principalmente la possibilità di creare e gestire oggetti, fanno invece parte dello standard **SQL/PSM** (PSM è l'acronimo di *Persistent Storage Module*).

2.2 Case study: una base di dati OO con Oracle

Questo paragrafo mostra un esempio di progettazione di base di dati Object Oriented.
Il caso di studio si riferisce al progetto di un database universitario, dove si vogliono memorizzare i dati relativi agli studenti, ai corsi e agli esami (dei relativi corsi) che essi sostengono.
Si vuole inoltre distinguere tra studenti e tesisti.

Progettiamo lo schema logico:
Corso(Nome, Crediti)
Esame(Corso, Voto)
Studente(Matricola, Nome, Cognome, Numeri_Telefono(0,n), Esami)
Tesista(Matricola, Nome, Cognome, Numeri_Telefono(0,n), Esami, Titolo_Tesi)

Scegliamo di implementare la tabella Corso come una normale relazione, come se usassimo il modello relazionale.
La tabella Studente contiene invece un array di numeri di telefono e un insieme di Esami. Siccome Oracle ha la struttura array ma non ha ancora la struttura SETOF, implementiamo Esami come tavola annidata all'interno di Studente.

Infine, sembra ragionevole stabilire che la tabella Tesista debba ereditare tutti gli attributi e metodi di Studente ed aggiungere un attributo Titolo_Tesi.

Cominciamo definendo i tipi di dato.
Creiamo il tipo corso.

```
CREATE OR REPLACE TYPE Corso_TY AS OBJECT
(
    Nome VARCHAR2(30),
    Crediti INTEGER,

    MEMBER PROCEDURE Create_Corso(
            Nome IN VARCHAR2,
            Crediti IN INTEGER),
    MEMBER PROCEDURE Delete_Corso
);
```

Definiamo in secondo luogo il tipo esame e la tabella annidata esami.

```
CREATE OR REPLACE TYPE Esame_TY AS OBJECT
(
    Materia REF Corso_TY,
    Voto INTEGER
);

CREATE OR REPLACE TYPE Esame_NT AS TABLE OF Esame_TY;
```

Ogni tupla della tavola annidata contiene un riferimento ad una tupla oggetto di un corso. Possiamo utilizzare questa alternativa ai vincoli di integrità referenziale. Quì l'attributo Materia contiene l'OID di una tupla oggetto di un corso.
Questo approccio presenta alcuni vantaggi rispetto ai VIR (vincoli di integrità referenziale). Innanzitutto la dichiarazione è semplificata. Infatti, utilizzare un VIR può comportare l'aggiunta di un insieme di attributi. Ad esempio se la tabella Corso avesse come chiave primaria Nome e Codice corso, allora nella tabella Esame, l'attributo Materia non avrebbe potuto essere unico: sarebbe stato necessario introdurre ben due attributi, Nome corso e Codice corso. Inoltre, quando la chiave di una tupla referenziata viene modificata, questo ha ripercussioni costose sulla tabella che referenzia le tuple di quel tipo. Infatti in ogni tupla che referenzia una tupla che modifica la sua chiave vanno opportunamente modificati tutti gli attributi che referenziano la tupla. Inoltre le query su relazioni con VIR, necessitano di equijoin, mentre in questo caso, per accedere ad esempio al nome di un corso di un esame basta

richiamare $Esame.Materia.Nome$, Evitando di effettuare un join tra Esame e Materia con $REF(Esame.Materia) = REF(Corso)$.
Tuttavia, questo approccio fornisce alcuni svantaggi. Infatti l'aggiornamento degli attributi referenziati è più complicato da gestire e inoltre le cancellazioni di tuple referenziate non modificano gli attributi di tipo REF generando così riferimenti pendenti.

Creiamo ora i tipi studente e tesista.

```
CREATE OR REPLACE TYPE NUMERI AS VARRAY(30) OF INTEGER;

CREATE OR REPLACE TYPE Studente_TY AS OBJECT
(
    Matricola INTEGER,
    Nome VARCHAR2(30),
    Cognome VARCHAR2(30),
    Numeri_Tel NUMERI,
    Esami Esame_NT,

    MEMBER PROCEDURE Delete_Studente,
    /*Aggiunge un nuovo esame sostenuto dallo studente*/
    MEMBER PROCEDURE Insert_Esame(titolo IN VARCHAR2,
                                  voto IN INTEGER),
    MEMBER FUNCTION Media RETURN NUMBER
)
NOT FINAL
;

CREATE TYPE Tesista_TY UNDER Studente_TY
(
    Titolo_Tesi VARCHAR2(60)
);
```

Notare la dichiarazione del tipo NUMERI: si tratta di un array (VARRAY) di 30 interi.

Creiamo ora le rispettive tabelle

```
CREATE TABLE Corso OF Corso_TY
(
    Nome PRIMARY KEY,
    Crediti NOT NULL
);
```

```
CREATE TABLE Studente OF Studente_TY
(
      Matricola PRIMARY KEY,
      Nome NOT NULL,
      Cognome NOT NULL
) NESTED TABLE Esami STORE AS Esami_NTab;

CREATE TABLE Tesista OF Tesista_TY
(
      UNIQUE (Titolo_Tesi)
) NESTED TABLE Esami STORE AS Esami_Tesisti_NTab;
```

Le tabelle Esami_NTab e Esami_Tesisti_NTab non sono accessibili dall'utente. Si tratta delle tavole annidate di Studente e Tesista, che vengono memorizzate in tabelle distinte.

Trattiamo ora l'implementazione del tipo Studente_TY (omettiamo l'implementazione degli altri tipi, che è analoga).

```
CREATE OR REPLACE TYPE BODY Studente_TY AS
    MEMBER PROCEDURE Delete_Studente IS
    BEGIN
        DELETE FROM Studente s
                    WHERE s.Matricola = self.Matricola;
    END Delete_Studente;

    MEMBER PROCEDURE Insert_Esame(titolo IN VARCHAR2,
                                  voto IN INTEGER) IS
        materia REF Corso_TY;

    BEGIN
        SELECT REF(c) INTO materia
        FROM Corso c
        WHERE c.Nome = titolo;

        INSERT INTO
                TABLE (
                        SELECT s.Esami
                        FROM Studente s
                        WHERE s.Matricola =
                                self.Matricola
                      )
                VALUES(materia, voto);
    END Insert_Esame;

    MEMBER FUNCTION Media RETURN NUMBER IS
    media NUMBER;
    BEGIN
        SELECT AVG(e.Voto) INTO media
        FROM TABLE(
                    SELECT ss.Esami
                    FROM Studente ss
                    WHERE ss.Matricola =
                                self.Matricola) e;
        RETURN media;
    END Media;
END;
```

Inseriamo ora, a titolo esemplificativo tre tuple di corsi nella rispettiva tabella.

```
INSERT INTO Corso VALUES ('Algoritmi', 9);
INSERT INTO Corso VALUES ('Basi di Dati', 9);
INSERT INTO Corso VALUES ('Fondamenti dell''informatica', 6);
```

Questo tipo di inserimento è analogo a quello dell'algebra relazionale.
Inseriamo ora una tupla di studente con due esami sostenuti.

```
INSERT INTO Studente
VALUES
(30002, 'Mario', 'Rossi', NUMERO(303030, 30410231, 134068),
 Esame_NT(Esame_TY(
                    (SELECT REF(c)
                     FROM Corso c WHERE
                      c.Nome = 'Algoritmi'), 24),
          Esame_TY(
                    (SELECT REF(c)
                     FROM Corso c WHERE
                      c.Nome = 'Basi di Dati'), 28)));
```

Questa operazione è più laboriosa.
Il costruttore NUMERO(...) crea un array di tre interi. Il costruttore Esame_NT crea una tavola annidata. I suoi parametri sono tuple (se ne può inserire un numero arbitrario).
Una tupla si forma a sua volta con il costruttore Esame_TY. All'interno dobbiamo inserire l'OID di un corso (a sua volta dobbiamo quindi eseguire una query) e il numero di crediti.

I tesisti si inseriscono in modo analogo

```
INSERT INTO Tesista
VALUES
(30002, 'Paolo', 'Bianchi', NUMERO(3303123),
 Esame_NT(Esame_TY(
                    (SELECT REF(c)
                     FROM Corso c WHERE
                      c.Nome = 'Algoritmi'), 24),
          Esame_TY(
                    (SELECT REF(c)
                     FROM Corso c WHERE
                      c.Nome = 'Basi di Dati'), 28)),
         'Un approccio alternativo alle Basi di Dati');
```

Si aggiunge semplicemente un campo in più: il titolo della tesi.

Ora formiamo una query che mostra tutti gli studenti non tesisti e la loro media (aritmetica).

```
SELECT DISTINCT s.Matricola, s.Nome, s.Media()
FROM Studente s;
```

Formiamo ora la query che mostra tutti gli esami (nome e voto) dati dallo studente con matricola 10.

```
SELECT e.Materia.Nome, e.Voto
FROM TABLE(
                SELECT s.Esami
                FROM Studente s
                WHERE s.Matricola = 10
               ) e;
```

2.3 Vincoli di integrità nello standard SQL99

SQL99 aggiunge nuovi vincoli di integrità e nuove opzioni.
Si ricorda **vincoli di integrità** sono predicati che stabiliscono quali operazioni sono ammesse. I VI (*vincoli di integrità*) servono a mantenere la base di dati consistente (ad esempio ad evitare che un impiegato abbia stipendi negativi, età negativi, informazioni mancanti, ecc.).

Esistono due possibili forme di consistenza: a livello di *statement* e a livello di *transazione*.
Quando un DBMS è in modalità autocommit, una transazione ACID è costituita da un solo statement (insert, update, query, ecc.). Questo significa che ad esempio ogni operazione di inserimento viene effettuata in maniera atomica e viene aggiornato lo stato del database.
Se il DBMS non è in modalità autocommit è compito dell'utente terminare la transazione con l'istruzione SQL *COMMIT*.
La **consistenza a livello di statement** consiste nel portare la base di dati da uno stato consistente prima dell'esecuzione di ogni statement ad uno stato consistente dopo la sua esecuzione. Questo significa che, indipendentemente dalla lunghezza della transazione, dopo l'esecuzione di uno statement tutti i VI che seguono questo criterio devono essere rispettati.
La **consistenza a livello di transazione** consiste invece nel portare la base di dati da uno stato consistente prima dell'esecuzione di una transazione ad

uno stato consistente dopo la sua esecuzione, ma con possibili stati inconsistenti al suo interno.
Questo significa che all'interno di una transazione si possono eseguire alcuni statement che ne violano l'integrità, ma la BD deve essere comunque consistente dopo il commit.
Da questo segue che la consistenza a livello di statement IMPLICA la consistenza a livello di transazione.
Vedremo fra poco che SQL99 permette di specificare il criterio di consistenza.

Vediamo innanzitutto come classificare un vincolo. Abbiamo diversi criteri.
Un vincolo può essere classificato in base alla sua **granularità**: *di colonna* (coinvolge una colonna di una tabella), *di tabella* (coinvolge più colonne di una tabella) e *globali* (coinvolge più tabelle).
Possiamo classificare i vincoli anche in base all'aspetto temporale: *di stato* (vincoli a livello di statement o di transazione), *di transizione di stato* (coinvolgono il cambiamento di uno stato, ossia mettono in relazione uno stato precedente con quello attuale) oppure *di sequenza di stati* (richiedono che la sequenza di stati abbia particolari proprietà, ad esempio che non si possa riassumere un impiegato che è stato licenziato).

Esistono due modi per specificare i sollevate: **dichiarativo** (vincoli dichiarati, come NOT NULL, PRIMARY KEY, UNIQUE, ecc.) e **procedurale** (ossia tramite l'utilizzo dei trigger).

I vincoli dichiarativi possono essere espressi in due modalità: **IMMEDIATE**, che sono vincoli a livello di statement, ossia verificati immediatamente dopo l'esecuzione di uno statement, e **DEFERRED**, cioè vincoli a livello di transazione.
Inoltre è possibile modificare questa modalità all'interno di una transazione utilizzando ***SET CONSTRAINTS*** *nome1, ..., nomen modalità*. In ogni caso un vincolo può essere **DEFERRABLE** (può essere reso DEFERRED) oppure **NOT DEFERRABLE** (non può essere reso DEFERRED).
esempio

CONSTRAINT Nome_NNULL **NOT NULL**(Nome)
INITIALLY IMMEDIATE, **NOT DEFERRABLE**

2.3.1 Vincoli dichiarativi

Vediamo ora la sintassi e semantica dei vincoli dichiarativi.
In generale un vincolo dichiarativo o a livello di riga ha la seguente sintassi

```
CONSTRAINT nome specifica-vincolo [modalita]
```

le espressioni tra quadre sono facoltative.
Fanno eccezione il vincolo globale ASSERTION la cui sintassi è la seguente

```
CREATE ASSERTION nome (
    CHECK(query con condizione)
) [modalita]
```

e il vincolo DOMAIN che crea un nuovo tipo di dato primitivo

```
CREATE DOMAIN nome tipo-base
DEFAULT valore-iniziale
(VALUE IN (lista valori)) [modalita]
```

Seguono due esempi

```
CREATE ASSERTION controlloStipendio (
CHECK (
        NOT EXISTS (SELECT *
                    FROM Persona p, Ufficio u
                    WHERE p.Stipendio >
                    u.massimo_stipendio)
    )
) INITIALLY IMMEDIATE, DEFERRABLE;

CREATE DOMAIN Categoria INTEGER
DEFALUT 0
(CHECK(VALUE IN (0, 1, 2, 3)));
```

Descriviamo ora i vincoli dichiarativi.

CHECK

Il vincolo **CHECK** è inserito all'interno della definizione di tabella.
Si tratta di un vincolo a livello di tabella o di colonna.

```
CREATE TABLE R
(
    ...
    CHECK (condizione)
    ...
);
```

Possiamo ad esempio avere il vincolo *CHECK(Stipendio ≥ 0)*.
Il vincolo è utile per imporre i valori assunti da alcuni attributi anche in relazione ad altri attributi.
Un'altro impiego utile è quello di esprimere dipendenze funzionali.
La clausola CHECK ha comunque delle limitazioni.
Prima di tutto, anche se una condizione può essere espressa tramite una query (come EXISTS(SELECT ...)) non si può fare riferimento ad altre tabelle diverse da quella su cui è inserito il vincolo ed in ogni caso tutti le variabili fanno riferimento ai soli attributi della tabella.
Inoltre, non si possono utilizzare funzioni dipendenti dal tempo (come CURRENT_DATE), l'operatore UNION, le funzioni MIN, MAX, ecc. e le funzioni o procedure implementate all'esterno dei DBMS.

NOT NULL

E' un vincolo di colonna. Specifica che una colonna non può assumere il valore speciale NULL.
Ad esempio

```
Nome VARCHAR2(30) NOT NULL
```

UNIQUE

Il vincolo **UNIQUE** esprime un vincolo di chiave secondaria.

```
\begin{lstlisting}
        CREATE TABLE R
        (
            ...
            UNIQUE (lista colonne)
            ...
        );
```

Il vincolo UNIQUE permette alle colonne coinvolte di assumere valori NULL.
La definizione di chiave secondaria secondo SQL99 è dunque la seguente:
Dato un valore o una sequenza di valori v, sia *NullIn(v)* una funzione che restituisce VERO se v contiene valori NULL, FALSO altrimenti e $R(A)$ una relazione con schema A, una chiave K è definita nel seguente modo

$$\forall t_1, t_2 \in R \quad (\not NullIn(t_1) \wedge \not NullIn(t_2)) \rightarrow (t_1[K] = t_2[K] \rightarrow t_1[A] = t_2[K])$$

PRIMARY KEY

Il vincolo **PRIMARY KEY** esprime un vincolo di chiave primaria, la cui sintassi è analoga a quella di UNIQUE.
A differenza di UNIQUE, questo vincolo non ammette valori nulli, avendo dunque la seguente definizione di chiave primaria:
Dato un valore o una sequenza di valori v, sia $R(A)$ una relazione con schema A, una chiave K è definita nel seguente modo

$$\forall t \in R \quad \not{N}ullIn(t) \wedge \forall t_1, t_2 \in R(t_1[K] = t_2[K] \rightarrow t_1[A] = t_2[K])$$

FOREIGN KEY

Il vincolo **FOREIGN KEY** esprime vincoli di integrità referenziale.
La sintassi è la seguente

```
FOREIGN KEY (lista colonne)
REFERENCES tabella(lista colonne)
[MATCH {SIMPLE | PARTIAL | FULL}]
[ON UPDATE azione]
[ON DELETE azione]
```

La lista delle colonne referenziate deve coincidere con la PRIMARY KEY della tabella referenziata (anche se il nome può essere diverso).
La voce 'azione' è a scelta tra 4 opzioni possibili (più SET DEFAULT) sotto determinate condizioni: ***ON DELETE*** *azione*, ***ON UPDATE azione***.
La prima specifica l'azione che deve avvenire quando si rimuove una tupla della *tavola referenziata*, mentre la seconda specifica l'azione da eseguire quando aggiorna una tupla della *tavola referenziata*.
Le azioni possibili, data una tabella A le cui tuple referenziano tuple della tabella B, sono

- **CASCADE**: applicato a ON DELETE, alla rimozione di una tupla di B (tabella referenziata) rimuove tutte le tuple in A (tabella referenziante) che referenziano quella tupla. Applicato a ON UPDATE, quando si aggiorna la chiave primaria di una tupla di B, si aggiornano i valori di tutte le tuple di A che referenziano quella tupla.

- **SET NULL**: quando un tupla di B viene rimossa o aggiornata tutte le tuple di A che referenziano quella tupla vengono modificate in modo tale che i valori coinvolti nel vincolo di chiave siano NULL. Questo naturalmente è solo possibile se tutte le colonne soggette al vincolo non sono NOT NULL.

- **RESTRICT**: impedisce di aggiornare e/o rimuovere tuple nella tabella B.

- **NO ACTION**: analogo a RESTRICT, il quale però effettua questo controllo prima dell'esecuzione di un'istruzione UPDATE/DELETE, mentre NO ACTION la effettua dopo.
  In questo modo, un trigger in seguito ad un'azione UPDATE o RESTRICT, può controllare la violazione prima dell'esecuzione dell'operazione UPDATE o DELETE (CREATE TRIGGER nome BEFORE ...) e modificare lo stato della base di dati in modo tale che, dopo l'esecuzione dello statement UPDATE o DELETE (e quindi del trigger) - momento in cui viene controllato il vincolo - il database sia consistente e non vi sia dunque nessun errore.

- **SET DEFAULT**: utilizza l'azione di default.

L'opzione **MATCH** specifica alcune informazioni su come impostare il vincolo di integrità referenziale.
In particolare abbiamo tre modalità, date le relazioni $A(..., r_1, ..., r_n, ...)$ e $B(..., k_1, ..., k_n, ...))$ con $r_1, ..., r_n$ un insieme di attributi che referenziano l'insieme di attributi $k_1, ..., k_n$ che formano la chiave primaria di B.

- **SIMPLE**: se almeno un valore di $r_1, ..., r_n$ è NULL allora il vincolo è rispettato, altrimenti $r_1, ..., r_n$ deve riferirsi alla tupla. Formalmente

$$\forall t_1 \in A \bigwedge_i t_1[r_i] \neq NULL \rightarrow (\exists t_2 \in B \quad \bigwedge_i t_1[r_i] = t_2[k_i])$$

- **PARTIAL**: se almeno un valore di $r_1, ..., r_n$ non è NULL, tutti i valori non NULL devono corrispondere con i relativi valori della chiave primaria di una qualche tupla di B. Formalmente

$$\forall t_1 \in A \bigvee_i t_1[r_i] \neq NULL \rightarrow \exists t_2 \in B \quad \bigwedge_i (t_1[r_i] = NULL \vee t_1[r_i] = t_2[k_i])$$

- **FULL**: se tutti i valori sono NULL allora il vincolo è rispettato, se nessun valore è NULL allora si controlla che il vincolo venga rispettato nello stesso modo dei primi due, ma se non tutti i valori sono NULL e non tutti sono diversi da NULL allora il vincolo non è rispettato. Formalmente

$$\forall t_1 \in A \bigvee_i t_1[r_i] \neq NULL \rightarrow \exists t_2 \in B \quad \bigwedge_i (t_1[r_i] \neq NULL \wedge t_1[k_i] = t_2[k_i])$$

Notare che se nessun valore è null oppure se $n = 1$ le tre modalità sono equivalenti.
Notare inoltre che FULL implica PARTIAL che a sua volta implica SIMPLE.

2.3.2 Vincoli procedurali

Come già detto, i vincoli procedurali sono implementati tramite i **trigger**, discussi anche nei paragrafi precedenti.
Occorre precisare due cose in merito ai trigger.
Innanzitutto i trigger vengono attivati al primo richiamo, ossia se vi sono inconsistenze nella base di dati al momento della creazione di un trigger (inconsistenze dettate dal trigger stesso), queste sono verificate solo alla prima attivazione del trigger.
L'altra precisazione riguarda il fatto che l'ereditarietà nei trigger non è specificata in SQL99, nel senso che non è specificato se il trigger si attiva su tutte e sole le azioni effettuate nella tabella specificata o anche sulle azioni specificate nelle tabelle che sono sottotipi di questa.

Un altra cosa importante è l'ordine di attivazione dei vari vincoli, perché i trigger possono interferire con altri vincoli. Deve essere dunque noto l'ordine di attivazione dei vari vincoli (con modalità IMMEDIATE).
Un corretto ordine, data l'esecuzione di uno statement SQL, è il seguente

- Trigger BEFORE
- Esecuzione Statement SQL
- RESTRICT
- CASCADE, UPDATE, SET NULL
- NO ACTION
- altri vincoli
- Trigger AFTER

Ma quando utilizzare i trigger al posto dei vincoli dichiarativi?
In generale i trigger possono esprimere qualunque tipo di vincolo. Tuttavia è conveniente utilizzarli per i vincoli generali a livello di tabella (a causa delle limitazioni di CHECK), per i vincoli globali (quindi che coinvolgono più tabelle, qualora ASSERTION non sia abbastanza espressivo) e soprattutto per i vincoli di transizione di stato (mettendo a disposizione lo stato della tabella prima e dopo l'esecuzione di uno statement).
Notare che i trigger sono particolarmente utili anche se il DBMS non mette a disposizione vincoli UPDATE e CASCADE.
Ad esempio, sia $A(..., R, ...)$ una tabella dove R referenzia le tuple di $B(..., K, ...)$ tramite la chiave K.
Possiamo ottenere l'effetto ON DELETE CASCADE nel seguente modo

```
CREATE OR REPLACE TRIGGER Delete_Cascade
AFTER DELETE ON B
REFERENCING OLD tupla_b
FOR EACH STATEMENT
BEGIN
    DELETE FROM A tupla_a
    WHERE tupla_a.R = tupla_b.K;
END
```

Un altro impiego utile dei trigger sono i vincoli generali di dipendenza tra tabelle.

$$\forall t_1, ..., t_n \quad \bigwedge_{i=1}^{n} t_i \in R_i \wedge P(t_1, ..., t_n) \rightarrow \exists t_{n+1}, ..., t_m \quad \bigwedge_{i=n+1}^{n+m} t_i \in R_i \wedge Q(t_{n+1}, ..., t_{n+m})$$

Questo vincolo è la generalizzazione del vincolo di integrità referenziale. La formula esprime il vincolo che, date n tuple appartenenti a relazioni e che rendono vero il predicato P, allora esiste un insieme di tuple appartenenti a m relazioni che soddisfano il predicato Q.
L'implementazione generalizzata, con i trigger necessita prima di ristrutturare la formula. In altre parola il vincolo contiene nel body (BEGIN END) la segnalazione di errore e nella condizione WHEN le premesse perché si verifichi l'errore.
Semplifichiamo la formula cosicché sia adatta ad essere applicata ad un trigger

$$\forall t \in R \quad P(t) \rightarrow \exists t_1, ..., t_n \quad \bigwedge_{i} t_i \in R_i \ \wedge P(t_1, ..., t_n)$$

Tale formula si può semplificare eliminando il quantificatore universale (che non esiste in SQL) ottenendo

$$\neg\exists t \neg(t \in R \wedge P(t)) \rightarrow \exists t_1, ..., t_n \quad \bigwedge_{i} t_i \in R_i \ \wedge P(t_1, ..., t_n)$$

Ancora l'implicazione può essere eliminata ricordando che $\neg(A \rightarrow B) \equiv A \wedge \neg B$ abbiamo dunque

$$\neg\exists t \quad t \in R \wedge P(t) \wedge \neg\exists t_1, ..., t_n \quad \bigwedge_{i} t_i \in R_i \ \wedge P(t_1, ..., t_i)$$

Creiamo dunque il trigger generalizzato attivato su qualunque azione effettuata sulla tabella R.

```
CREATE TRIGGER Trigger_Generalizzato
AFTER INSERT OR UPDATE OR DELETE ON R
...
WHEN (P(t) AND
        NOT EXISTS(
            SELECT *
        FROM R1 r1, ..., Rn rn
        WHERE Q(r1, ..., rn);
        ))
BEGIN
    ...
END
```

2.4 Semantica di Codd dei valori NULL

Quando in una query o in qualsiasi statement SQL viene fatto riferimento a colonne di tabelle con valori NULL esiste una semantica particolare.
La semantica di Codd definisce come interpretare queste espressioni.
Cominciamo ad interpretare il significato del valore speciale NULL assegnato alla colonna di una tupla.
Esistono tre interpretazioni

- *Dato esistente ma non registrato.* Ad esempio se la si ha una tabella Persona(CF, Nome, SecondoNome, ..., DataNascita, Coniuge, ...), e la colonna DataNascita di una tupla è posta a NULL, significa che la data di nascita non è registrata - ma esiste perché ogni persona può avere una data di nascita.

- *Dato inesistente.* Ad esempio la colonna SecondoNome può essere NULL in una tupla perché la persona non ha il secondo nome.

- *Assenza di informazioni.* Ad esempio la colonna Coniuge è NULL perché non è noto se questo esiste.

La semantica di Codd introduce tre valori di verità: *true*, *false*, *unknown* (valore sconosciuto, cioè non è possibile stabilire se è vero o falso).
Sia $P(t)$ una formula proposizionale applicata su una tupla t, abbiamo le operazioni di confronto θ, che possono essere $\leq, <, =, \neq, >, \geq$ e sono nella forma $t[A] = t[B]$ o $t[A] = costante$ con A e B attributi o insiemi di attributi.
Vi sono inoltre gli operatori OR, AND e NOT.

Questi sono solitamente gli insiemi di operazioni ammissibili in una condizione SQL (a meno delle espressioni aritmetiche e degli operatori EXISTS e NOT EXISTS).
Valutiamo ora i valori di verità di tali operazioni con la semantica a tre valori di Codd.
$t[A]$ θ *costante* $= UNKNOWN$ se $t[A]$ è NULL (θ è una qualsiasi delle operazioni di confronto).
Analogamente $t[A]$ θ $t[B] = UNKNOWN$ se $t[A]$ o $t[B]$ è NULL.
Questo avviene perché i valori NULL non sono confrontabili con valori numerici, stringhe o altro, per cui non è noto il valore di verità del confronto (cioè è UNKNOWN).
Ora valutiamo gli operatori booleani. I valori di verità sono gli stessi della logica tradizionale quando unknown non è coinvolto, altrimenti abbiamo
U AND T = U, U AND F = F, U AND U = U, U OR T = T, U OR U = U, U OR F = U, NOT U = U.
Come per i valori true/false si possono assegnare i numeri 1 e 0, in questo caso si può assegnare 0 a false, 1 a unknown e 2 a true avendo le seguenti definizioni formali di operatori, dati due valori x_1 e x_2 che contengono numeri da 0 a 2 che rappresentano tali valori di verità

$$x_1 \; AND \; x_2 = min\{x_1, x_2\}$$

$$x_1 \; OR \; x_2 = max\{x_1, x_2\}$$

$$NOT(x_1) = 2 - x_1$$

Definiamo ora il collasso $C[E]$ di un'espressione E con tre valori di verità. Abbiamo che $C[E] = T$ se $E = TRUE$, $C[E] = FALSE$ altrimenti, ossia se E vale $UNKNOWN$ o $FALSE$.

Sia dunque la selezione definita come segue

$$\sigma_P(R) = \{t | t \in R \wedge C[P(t)]\}$$

dove P è il predicato appena definito e C il collasso.
Siccome il collasso di unknown è FALSE, le tuple t tali che $P(t)$ è unknown vengono scartate.
Di fatto in una query SQL quando si hanno tuple di questo genere, queste vengono scartate.
Notare che eseguire il collasso delle espressioni atomiche (confronti uguale, maggiore, ecc.) e calcolare di conseguenza le espressioni composte dagli operatori AND, OR, NOT con solo due valori di verità non è la stessa cosa.
Ad esempio, data l'espressione $NOT(A = 3)$, se A è $NULL$ il confronto

restituirà $UNKNOWN$. Se si effettua il collasso di $UNKNOWN$ si ottiene $FALSE$ avendo $NOT(FALSE) = TRUE$, mentre il risultato atteso era $UNKNOWN$ perché $NOT(UNKNOWN) = UNKNOWN$.

I valori NULL coinvolgono molti aspetti di SQL.
Nelle espressioni aritmetiche, se appaiono uno o più valori NULL il risultato è NULL.
Nella clausola WHERE delle query, questi sono trattati come appena mostrato, ossia le tuple il cui test restituisce valore unknown vengono scartate.
Al contrario il vincoli CHECK, anch'esso composto da un'espressione booleana accetta i valori unknown senza segnalare dunque la limitazione del vincolo: i NULL hanno dunque la semantica opposta quì.
La funzione COUNT conteggia le colonne con valori NULL esattamente come quelle con valori.
Le funzioni MAX, AVG, MIN, ecc. ignorano invece tali valori.
Nella clausola GROUP BY i valori NULL sono raggruppati in un'unica partizione.
Con l'operatore DISTINCT viene fornito, se esistono valori NULL in una colonna, un solo valore NULL.
Questo significa che due valori NULL vengono visti come valori duplicati.

Vediamo nel dettaglio cosa accade con le operazioni IN, NOT IN, ANY e ALL quando vi sono valori NULL.

- E_1 **IN** $(E_2, ..., E_n)$ dove E_1 è un valore e $E_2, ..., E_n$ sono i valori restituiti dalla query annidata.
  Abbiamo che questa operazione equivale a

  $$E_1 = E_2 \; OR \; ... \; E_1 = E_n \; OR \; FALSE$$

  $OR \; FALSE$ è stato aggiunto qualora $n = 1$, ossia l'insieme di valori $E_2, ..., E_n$ è vuoto.
  Da questo segue che, se per qualche i abbiamo che E_i è $NULL$ ma E_1 appartiene all'insieme allora il confronto ha esito TRUE. Se invece tutti i valori dell'insieme sono NULL allora il risultato finale è $UNKNOWN \; OR \; FALSE = UNKNOWN$.
  La condizione risulta $UNKNOWN$ anche se E_1 è NULL.

- E_1 **NOT IN** $(E_2, ..., E_n)$, equivale a

  $$E_1 \neq E_2 \; AND \; ... \; E_1 \neq E_n \; AND \; TRUE$$

Da questo segue che se per qualche i abbiamo che E_i è $NULL$ e E_1 è diverso da tutti gli E_j non nulli, il valore è comunque UNKNOWN. In altre parola non si può stabilire se un elemento non appartiene ad un insieme se esiste un elemento nullo. La condizione risulta $UNKNOWN$ anche se E_1 è NULL.

- $E_1\ \theta\ ANY(E_2, ..., E_n)$ con θ un operatore di confronto (uguale, maggiore, ecc.)
  Questo equivale a

  $$E_1\ \theta\ E_2\ OR\ ...\ E_1\ \theta\ E_n\ OR\ FALSE$$

  e le considerazioni sono simili al primo caso (con IN).

- $E_1\ \theta\ ALL(E_2, ..., E_n)$ equivale a

  $$E_1\ \theta E_2\ AND\ ...\ E_1 \theta E_n ANDTRUE$$

  e le considerazioni sono simili a NOT IN.

2.5 ODMG

ODMG (*Object Data Management Group*) è un consorzio che si occupa di stabilire standard per le basi di dati Object Oriented.
L'obiettivo di questo consorzio è ottenere tramite uno standard la portabilità delle applicazioni, ossia fare in modo che i linguaggi OO operino con gli oggetti dei database OO in modo trasparente, trattandoli cioè come oggetti veri e propri del linguaggio (senza linguaggi intermediari per recuperare oggetti dal DB, ecc.).

Lo standard OMG3, la terza versione dello standard rilasciato da ODMG, è suddiviso in diverse parti.

- **Object Model**: modello degli oggetti ispirato dal modello di OMG (consorzio che effettua standard OO).
  Si tratta di un modello costituito da moduli che contengono interfacce e classi.

- **Linguaggi di specificazione**: linguaggi che specificano la struttura dati e la loro manipolazione.
  Tra questi vi sono **OQL** (*Object Query Language*) per le interrogazioni ai DBMS ad oggetti, **ODL** (*Object Definition Language*) per definire

la struttura degli oggetti persistenti secondo l'Object Model (è indipendente dai linguaggi di programmazione) e **OIF** (*Object Interchange Format*) che non verrà trattato in questo documento.

- vincoli **Binding**: prevede una serie di librerie ed eventualmente linguaggi per manipolare gli oggetti (principalmente ODL e OQL) del DBMS in modo trasparente, ossia in modo tale che si abbia la percezione di utilizzare un unico linguaggio.
  E' disponibile per i linguaggi *C++*, *Smalltalk* e *Java*.

In ogni caso lo standard stabilisce dei concetti tramite l'Object Model, ODL, OQL ecc. ben definiti ma indipendenti dal linguaggio, che vengono poi applicati in base ai vari linguaggi (C++ binding, Java binding, ecc.) generalmente tramite librerie che permettono di implementare i vari concetti ODL nel linguaggio utilizzato tramite librerie, le quali permettono anche di effettuare query in OQL.

L'Object Model prevede che vi siano due costrutti: *Interface* e *Class*. Il costrutto Interface non può istanziare oggetti, mentre Class si. In ogni caso sia Interface che Class contengono definizioni di **aspetti statici** (letterali e relazioni) e **aspetti dinamici** (funzioni e eccezioni eventualmente sollevate).
I *letterali* sono attributi che contengono un valore, mentre le *relazioni* sono concettualmente come relazioni tra classi o tra entity nel modello ER cioè mettono in relazione uno a molti, molti a uno, uno a uno o molti a molti valori.
Ad esempio in Java questo viene realizzato utilizzando gli omonimi costrutti interface (dove gli attributi sono definiti con i metodi get e set) e class.
Ad ogni oggetto si possono associare uno o più nomi.
Tra le caratteristiche principali abbiamo che gli oggetti persistenti e quelli transienti hanno lo stesso ciclo di vita e si possono applicare le stesse operazioni su di essi. Questa è una differenza sostanziale rispetto alle basi di dati relazionali in cui i dati persistenti sono gestiti solo tramite SQL.
L'object model stabilisce anche che si specifichino delle chiavi e si attribuisca un nome alle estensioni delle classi.
L'ereditarietà tra interfacce è multipla, mentre tra classi è singola.

I vari binding dovranno fornire un preprocessore che, dati i file in ODL crei lo schema rispettivo nella base di dati e richiami il compilatore, che prende in input la parte di dichiarazione delle classi di ODL e l'applicativo generando il programma che comunica con il DBMS.

2.5.1 ODL

L'Object Definition Language è un linguaggio astratto che serve a specificare le modalità in cui vengono definiti gli oggetti persistenti secondo l'Object model di ODMG.
Notare che le classi definite in ODL contengono solo le specifiche esterne (ossia aspetto strutturale e comportamentale) ma non contengono l'implementazione, la quale è demandata ai linguaggi utilizzati dai vari binding (Java, C++, ecc.).

La sintassi per la dichiarazione delle interfacce è la seguente

```
interface nome [: lista interfacce]
attributi
relazioni
funzioni
eccezioni
```

la lista interfacce definisce le interfacce estese dall'interfaccia 'nome' (è supportata l'ereditarietà multipla tra interfacce).
Gli attributi sono dichiarati nella forma
attribute tipo nome
Le relazioni sono invece dichiarate nel modo seguente *relationship tipo nome [inverse nome]*
Il vincolo inverse permette di specificare relazioni inverse, ossia ogni oggetto a con una relazione che si riferisce ad un particolare oggetto e vincolo inverse, si vincola l'oggetto referenziato ad avere l'attributo specificato nel vincolo referenziante a sua volta l'oggetto che referenzia a.

Le funzioni costituiscono la parte dinamica degli oggetti, ossia il loro comportamento.
Queste vengono dichiarate nel seguente modo
tipo nome ([IN — OUT — INOUT] tipo parametro, ...) raises (nome eccezione)
La lista delle eccezioni definisce i nomi delle eccezioni sollevate dai metodi.
I loro nomi vengono definiti nel nome seguente
exception nome

Per quanto riguarda i letterali, ODMG stabilisce le seguenti categorie

- **atomici**: tipi per valori primitivi, quali int, unsigned int, float, double, octet, ecc. e il valore *nil* (assenza di valore, come il NULL per SQL ed è trattato con la stessa semantica)

- **collezione**: strutture dati con tipo parametrico. Esse sono **set**⟨t⟩, **bag**⟨t⟩, **list**⟨t⟩ e **array**⟨t⟩, dove t è un qualsiasi tipo di dato.

- **strutturali**: tipi con una determinata struttura, come date, timestamp, ecc.

Sono inoltre messe a disposizione enumerazioni (**enum**) e struct come in C (**struct**).
Per quanto riguarda i tipi oggetto, ODMG stabilisce le seguenti categorie

- **atomici**: tipi definiti dall'utente con *class* o *interface*.

- **collezione**: tipi collezione sui quali si possono eseguire operazioni. Questi tipi contengono attributi letterali collezione (set, list, ecc.) e ne implementano la versione a oggetti.
  Essi sono **Set**⟨t⟩, **Bag**⟨t⟩, **List**⟨t⟩ e **Array**⟨t⟩ e **Dictionary**⟨k, v⟩, dove t è un qualsiasi tipo di dato e k, v sono due tipi di dato qualsiasi che rappresentano chiave e valore (dictionary è una mapping chiave valore).
  Strutturali: tipi analoghi ai letterali strutturali ma sono la versione a oggetti (Date, TimeStamp, ecc.)

Gli attributi possono essere di tipo letterale o oggetto, devono essere ridefiniti in una sottoclasse e, se sono di tipo oggetto, sono monodirezionali. Le relazioni devono necessariamente essere di tipo oggetto o collezione di oggetti e possono essere bidirezionali.

La sintassi per la definizione delle classi è la seguente.

```
class nome [extends classe [: lista interfacce]]
(extent nome, keys chiavi)
attributi
relazioni
funzioni
eccezioni
```

La parola chiave **extent** (da non confondere con extend) specifica il nome dell'estensione della classe, ossia dell'insieme degli oggetti istanza di quella classe.
La semantica estensionale è in profondità. La parola chiave **keys** specifica gli attributi che costituiscono le chiavi candidate.

ODMG specifica una serie di classi e interfacce che il sistema deve mettere a disposizione.

L'implementazione di tali classi e interfacce è a cura del produttore del DBMS.
Le interfacce principali sono **ObjectFactory** e **Object** che vengono ereditate ed implementate implicitamente da ogni classe.
Object Factory contiene il metodo *new* per la creazione e persistenza degli oggetti.
La classe Object fornisce una serie di metodi come *copy* (effettua una shallow copy), *delete* per rimuovere l'oggetto (operazione che non viene fatta in cascata e rende l'OID dell'oggetto comunque non riutilizzabile).
Ogni oggetto contiene poi delle funzioni (definiti in Object) per i lock dell'oggetto (lo vedremo tra poco).
L'interfaccia **Collection** contiene le funzioni comuni per tutte le collezioni oggetto (Set, List, ecc.)
Gli elementi delle collezioni si possono scorrere con gli iteratori. Ogni tipo di collezione ne fornisce uno.
L'interfaccia **Iterator** viene utilizzata per definire iteratori monodirezionali.
L'interfaccia **BidirectionalIterator** contiene anche funzioni per scorrere gli elementi di una collezione nel verso opposto e sono dunque bidirezionali.
Un iteratore può essere anche creato come *stabile*, cioè non permette che accessi concorrenti (inserimenti o cancellazioni) alla stessa collezione modifichino la lettura dei dati di un processi.

ODMG stabilisce anche specifiche sulla gestione dei lock di un oggetto.
Esistono tre tipi di lock: *read* (lock in lettura), *upgrade* (lock in modifica), *write* (lock in scrittura).
Esistono tre analoghi protocolli: *read* che permette di detenere un lock in lettura ma può diventare anche write o upgrade, *upgrade* che permette di detenere un lock in upgrade che può diventare anche write e *write* che permette solo di detenere lock write.
ODMG specifica anche un'interfaccia **Transaction** e la relativa **TransactionFactory**.
Metodi particolari dell'interfaccia Transaction sono *join* e *leave*.
Questi metodi sono stati introdotti perché un thread può attivare una sola transazione alla volta.
Per rilasciare una transazione attiva si utilizza la funzione leave, per fare entrare un'altra transazione (nello stesso thread) precedentemente sospesa con leave si utilizza join.
Notare che la funzione leave non chiude la transazione e tanto mento comporta il rilascio dei lock, ma la sospende.

Infine abbiamo l'interfaccia **Database** che consente consente di effettuare

le operazioni sul Database.
Questa interfaccia offre tra le altre funzionalità, la possibilità di associare ad una particolare istanza un nome o di rimuovere tale associazione.
Sempre la classe Database, permette dunque di ottenere un'istanza dato un nome.

Riprendiamo ora l'esempio di una base di dati universitaria trattata nel paragrafo 2.2 e dichiariamo la struttura in ODL.
ODL permette di raccogliere tutte le classi, interfacce e dichiarazioni di eccezioni all'interno di un contenitore detto **Module**.
Nel nostro esempio abbiamo

```
Module universita {
        ...
}
```

Segue il codice delle varie classi all'interno del modulo università.

```
class Corso (extent corsi,
keys nome)
    attribute string nome;
    attribute unsigned int crediti;

class Esame {
    attribute Corso c;
    attribute unsigned int voto;
    relationship Studente studente
                inverse Studente::esami;
}

class Studente (extent studenti
keys matricola)
    exception EsameRipetuto {};
    //Attributi
    attribute unsigned int matricola;
    attribute String nome;
    attribute String cognome;
    relationship set<Esame> esami
                inverse Esame::studente;

    //Funzioni
    void aggiungi_esame(in Esame e) raise EsameRipetuto;
    Esame get_esame(in string nome);
```

```
class Tesista (extent tesisti)
    attribute unsigned int matricola;
    attribute string nome;
    attribute string cognome;
    attribute string titolo_tesi;
```

2.5.2 OQL

Il linguaggio **OQL** (*Object Query Language*) è un linguaggio funzionale utilizzato per effettuare le query in ODMG.
Si tratta di un linguaggio molto simile a SQL, tanto che si spera che prima o poi i due linguaggi convergano.

OQL soddisfa quattro criteri fondamentali, che tutti i linguaggi di interrogazione devono soddisfare.

1. deve essere **dichiarativo**, quindi in questo contesto deve permettere di definire le caratteristiche che devono avere i dati da estrarre senza specificare come questi vengono estratti.
   Di fatto, come in SQL, OQL specifica solo le caratteristiche che devono avere i dati piuttosto che permettere di implementare un determinato criterio di ricerca (operazione effettuata dal DBMS).

2. deve essere sufficientemente **espressivo**. Di fatto ci si accorgerà di quanto OQL sia espressivo.
   In ogni caso questo linguaggio offre tutte le funzionalità offerte dalle query SQL.

3. deve possedere meccanismi per la **trasformazione dei dati**. Di fatto OQL permette di richiamare funzioni (definite con ODL).

4. deve essere **composizionale**. Di fatto una query è formata da più sotto query le quali vengono eseguite prima di eseguire quella principale.

ODMG prevede che ogni linguaggio di programmazione che aderisce a questo standard preveda delle funzioni per effettuare query.
L'unica limitazione è che non si possono fare query ricorsive e nemmeno chiusure transitive.

La sintassi generale per le interrogazioni OQL (a meno di operatori di aggregazione) è la seguente

```
select [distinct] q
from
q1 in x1,
...
qn in xn
[where p]
```

Si nota come la sintassi sia davvero simile a SQL.
$q1, ..., qn$ sono query. Il nome di un'estensione (definito con *extent* in ODL) è una query stessa, che restituisce tutti gli oggetti (qui non si parla di tuple bensì di oggetti) dell'estensione di una classe.
Data una variabile, anche $x.a$ è una query. $x1, ..., xn$ sono nomi di variabili dove ogni xi appartiene alle tuple fornite dalla query qi. L'iterazione avviene quindi per ogni combinazione di oggetti restituita dalle n query (join).
q è una query che solitamente indica il formato della query principale.
p è un predicato, ossia una formula logica, analoga a quelle presenti nella clausola where SQL.
La dichiarazione $xi\ IN\ qi$ può essere sostituita con $qi\ AS\ xi$.
q può fare riferimento a tutte le variabili $x1, ..., xn$, $q1$ non può fare riferimento a nessuna variabile, $q2$ può riferirsi solo a $x1$, ecc.
Il predicato p può invece utilizzare tutte le variabili.
Questo, oltre a contenere operatori di confronto e booleani, può contenere anche i quantificatori universali ed esistenziali.
Possiamo dunque avere un predicato nella forma

$$exists\ x\ in\ q\ :\ c$$

dove x è un nome nuovo di variabile, q è una query e c un'espressione condizionale che può fare uso della variabile x.
La semantica di questo predicato, dove q la consideriamo come un insieme di elementi è

$$\exists x \in q \quad c(x)$$

Esiste anche la versione universale del quantificatore.

$$forall\ x\ in\ q\ :\ c$$

Analizziamo innanzitutto la struttura di q.

$$q = struct(a_1 : e_1, ..., a_k : e_k)$$

dove ogni a_i è il nome di un attributo della struct (concettualmente identica a OQL e quindi a C/C++) ed ogni e_i è il relativo valore, dove e_i è un'espressione o una query che fa uso delle variabili $x_1, ..., x_n$.

Il risultato della query è un insieme di struct, dunque la query è tipata

$$set\langle struct(a_1 : e_1, ..., a_k : e_k)\rangle$$

se **select** è accompagnato dalla clausola **distinct**

$$bag\langle struct(a_1 : e_1, ..., a_k : e_k)\rangle$$

altrimenti, ossia un multiinsieme che contiene le risposte.
In generale se T è il tipo di q allora la risposta è $set\langle T\rangle$ se è presente distinct, $bag\langle T\rangle$ se non è presente.
Tuttavia esiste un caso particolare in cui una query può avere la seguente forma

```
element(select e
...
)
```

Questa query deve restituire un solo valore.
q può comunque assumere anche altre forme, ad esempio può essere composta da una sola variabile.

Notare che il predicato p può fare riferimento a funzioni, ad esempio $oggeto.cancellaElemento()$ che alterano lo stato della base di dati con effetti collaterali.
Infine, è possibile effettuare cast di un'espressione e trasformandola in un tipo T con l'istruzione $(T)e$.

Riprendiamo l'esempio della base di dati universitaria.
Elenchiamo tutti gli studenti.

```
studenti
```

Come detto prima, questa è una query. Infatti 'studenti' è il nome dell'*extent* della classe Studente e quindi elenca tutti gli studenti (tesisti inclusi).
La query può essere riformulata come segue

```
select struct (studente : s)
from s in studenti
```

La query ha tipo $set\langle struct(studente : Studente)\rangle$.

Selezioniamo ora tutti gli studenti, che hanno sostenuto l'esame di 'Basi di Dati' e, per ogni studente l'insieme degli esami dove hanno preso 30.

```
select struct (matricola : s.matricola,
               nome : s.nome,
               esami : select e
                       from e in s.esami
                       where e.voto = 30)
from s in studenti
where exists e in s.esami :
      e.corso.nome = 'Basi di Dati'
```

Questa query itera l'estensione studenti (quindi l'insieme delle istanze di Studente) esaminando i soli studenti per cui esiste un esame da loro sostenuto (s.esami) il cui nome è 'Basi di Dati' (clausola where).
Inoltre nella clausola select si elencano gli esami sostenuti dallo studente con voto 30.
La query ha tipo $bag\langle struct(matricola : unsignedint, nome : string, esami : bag\langle Esame\rangle)\rangle$.

Notare che è anche possibile definire una query come funzione.

```
define query nome (par1, ..., parn)
as select ...
```

dove $par1, ..., parn$ sono variabili che possono essere utilizzate all'interno della query.
Una volta definita la query, è possibile richiamarla in tutte le altre query.

Completiamo ora la sintassi delle query OQL aggiungendo gli operatori di aggregazione.

```
select [distinct] *
from x1 in q1
     ...
     xn in qn
where p1(x1, ..., xn)
[
group by a1 : e1, ..., ak : ek
[having p2]
]
```

dove $p1$ e $p2$ sono predicati, $a1, ..., an$ sono nomi di variabile e $e1, ..., en$ sono espressioni.
L'operazione di aggregazione è del tutto analoga a quella SQL.

- Prodotto cartesiano $Q = q1 \times ... \times qn$
- Selezione delle ennuple $(x1, ..., xn) \in Q$ tali che $p1(x1, ..., xn)$ è vero.
- Partizionamento delle ennuple, dove $P_{e1,...,e_k} = \{(x1, ..., xn) | e1(x1, ..., xn) = e1 \wedge ... \wedge ek(x1, ..., xk) = ek\}$, ossia dati i valori delle espressioni $e_1, ..., e_k$ una partizione contiene tutti gli oggetti le cui espressioni hanno quel valore.
- Si mantengono le sole partizioni che rendono vere la clausola $p2$.

La query è tipata

$$set\langle struct(a1 : type(e1), ..., ak : type(ek), partition : bag\langle struct(type(x1), ..., type(xn))\rangle)\rangle$$

dove $type(e)$ è il tipo di un'espressione o di una variabile.

Raggruppiamo ad esempio gli esami dati dagli studenti il cui nome è 'Marco' e la media è maggiore di 25, per studente.

```
select *
from e in esami
where e.studente.nome = 'Marco'
group by studente : e.studente.matricola
having avg(select es.voto
           from es in studente.esami) > 25
```

La query ha tipo

$$set\langle struct(studente : unsigned\ int, bag\langle e : Esame\rangle)\rangle$$

OQL fornisce anche strumenti per la creazione degli oggetti.
Un oggetti di tipo classe T si crea con il costruttore $T(...)$ i cui parametri sono i valori iniziali degli attributi e relazioni.
Se un attributo o una relazione è di tipo set o oggetto si possono effettuare delle query al posto di annidare un costruttore.
Ad esempio, possiamo creare un nuovo esame nel seguente modo

```
Esame(Corso("Basi di Dati", 9), 30)
```

Notare che è stato omesso l'attributo studente (su cui vige il vincolo inverse).
In tal caso, tali valori vengono inizializzati a NULL.

Possiamo creare un esame anche nel seguente modo

```
Esame(Corso("Analisi", 12), 30,
    element
    (select s
    from s in studenti
    where s.matricola = 30000)
)
```

Possiamo anche creare un nuovo studente, che ha un set di esami, inizializzandolo con 3 esami: 'Basi di Dati', 'Analisi', e 'Logica'.

```
Studente(30000, "Mario", "Rossi",
    select c
    from c in corsi
    where c.nome = "Basi di dati" or
          c.nome = "Analisi" or
          c.nome = "Logica"
)
```

Un ultima precisazione riguarda l'accesso a campi con valore *nil*.
Tale accesso restituisce valore UNDEFINED.
La funzione is_defined(valore) restituisce true se il valore è DEFINED, false altrimenti.
I valori UNDEFINED vengono inseriti negli insiemi che contengono la risposta delle query, a meno che essi non vengano confrontati (come in SQL).

Facciamo un ultimo esempio. Selezioniamo i soli tesisti con nome non nullo.

```
select *
from (Tesista)t in tesisti
where is_defined(t.nome)
order by t.matricola
```

Notare che il cast a Tesista garantisce che si elenchino i soli tesisti e non gli studenti non tesisti.
Notare inoltre che, come in SQL esiste la clausola order by, che in questo caso ordina i tesisti per numero di matricola.

Bibliografia

[1] Fabio Strocco. Basi di dati. 2008.