DENTRO LA SCATOLA

Rubrica a cura di

Fabio A. Schreiber

Dopo aver affrontato negli scorsi anni due argomenti fondanti dell'Informatica – il modo di codificare l'informazione digitale e la concreta possibilità di risolvere problemi mediante gli elaboratori elettronici – con questa terza serie andiamo ad esplorare "*Come parlano i calcolatori*". La teoria dei linguaggi e la creazione di linguaggi di programmazione hanno accompagnato di pari passo l'evolversi delle architetture di calcolo e di gestione dei dati, permettendo lo sviluppo di applicazioni sempre più complesse, svincolando il programmatore dall'architettura dei sistemi e consentendogli quindi di concentrarsi sull'essenza del problema da risolvere.
Lo sviluppo dell'Informatica distribuita ha comportato la nascita, accanto ai linguaggi per l'interazione tra programmatore e calcolatore, anche di linguaggi per far parlare i calcolatori tra di loro – i protocolli di comunicazione. Inoltre, la necessità di garantire la sicurezza e la privatezza delle comunicazioni, ha spinto allo sviluppo di tecniche per "non farsi capire" da terzi, di quì l'applicazione diffusa della crittografia.
Di questo e di altro parleranno le monografie quest'anno, come sempre affidate alla penna (dovrei dire tastiera!) di autori che uniscono una grande autorevolezza scientifica e professionale ad una notevole capacità divulgativa.

Linguaggi Formali per Basi di Dati

Letizia Tanca

1. Introduzione

Come anticipato in un precedente articolo di questa rubrica[1], le basi di dati relazionali godono della particolarità di fornire, oltre che linguaggi di interrogazione adottati dai vari DBMS, anche linguaggi *formali*, quindi con una sintassi e una semantica ben definite che permettono di condurre ragionamenti formali e prove affidabili. I linguaggi formali più noti e usati sono l'*Algebra Relazionale* e il *Calcolo Relazionale* (in due versioni, dette *Calcolo delle tuple* e *Calcolo dei domini*), il primo ispirato a una notazione di tipo algebrico e il secondo basato sul calcolo dei predicati. I due linguaggi sono perfettamente equivalenti, nel senso che possono rappresentare esattamente lo stesso insieme di interrogazioni. Tale insieme è quello considerato "soddisfacente" per un linguaggio formale di interrogazione dalla comunità delle basi di dati; questo tipo di completezza espressiva viene detto "nel senso di Bancilhon-Paredaens".
Un altro linguaggio, con specificità un po' diverse, è stato introdotto e intensamente studiato negli anni '80 e '90; si tratta del linguaggio *Datalog*, basato sul paradigma della programmazione logica. Scopo principale della sua introduzione è il tentativo di estendere il potere espressivo dei linguaggi di interrogazione con la possibilità di esprimere interrogazioni ricorsive, e quindi di permettere anche ai linguaggi di interrogazione di esprimere computazioni; in buona sostanza, si tratta di un tentativo di superare il cosiddetto *impedence mismatch* o disaccoppiamento di impedenza, definito nell'articolo precedente.
Nei prossimi paragrafi descriveremo in modo non troppo formale questi linguaggi, e discuteremo la questione della ricorsione e del potere espressivo di Datalog. Dapprima esamineremo l'Algebra Relazionale, che verrà esposta con molti dettagli, anche perchè essa si trova alla base delle tecniche di ottimizzazione delle interrogazioni. Nella sezione successiva verranno discussi i linguaggi basati sulla logica: da una parte, il cal-

[1] Letizia Tanca: Linguaggi per Basi di Dati. *Mondo Digitale*, n. 18, Giugno 2006 p. 65-72.

colo relazionale, nella versione *Calcolo delle tuple*, e dall'altra il Datalog. L'articolo terminerà con una breve discussione sul potere espressivo di questi linguaggi, confrontato anche con quello di SQL, già introdotto nel precedente articolo.

2. L'Algebra Relazionale

L'Algebra Relazionale è un linguaggio funzionale, nel senso che esprime il calcolo effettuato da un programma come una funzione matematica. Su questo linguaggio si fonda il formalismo interno adoperato dai Sistemi di Gestione di Basi di Dati per rappresentare le interrogazioni, e per compiere i primi passi di ottimizzazione. È quindi molto importante comprenderne i fondamenti, perché essa fornisce allo studioso, da una parte, gli strumenti per esprimere una interrogazione in modo efficiente, e, dall'altra, la possibilità di un confronto rigoroso con gli altri formalismi di rappresentazione. Per questo motivo, ci dilungheremo su questo più che sugli altri linguaggi formali, per i quali lavoreremo invece basandoci su esempi.
L'Algebra Relazionale è basata su cinque operatori fondamentali, che si applicano a relazioni e, come previsto per un linguaggio relazionale, restituiscono relazioni. Mediante l'uso di questi cinque operatori si compongono espressioni di complessità crescente.
I cinque operatori fondamentali dell'algebra relazionale sono:

1. la *selezione*, rappresentata dal simbolo σ,
2. la *proiezione*, rappresentata dal simbolo Π,
3. il *prodotto cartesiano*, rappresentato dal simbolo $\times$,
4. l'*unione* insiemistica, rappresentata dal simbolo $\cup$, e
5. la *differenza* insiemistica, rappresentata dal simbolo $-$.

Si possono poi ottenere altri operatori definendoli a partire da questi cinque.
I primi due operatori sono unari, nel senso che prendono in input una sola relazione, mentre gli altri tre sono binari, cioè si applicano a due relazioni. Ovviamente al risultato di una operazione si può applicare ancora un'altra operazione, perciò un'operazione binaria può diventare n-aria semplicemente iterando la sua applicazione. Espressioni complesse possono essere composte applicando operatori a formule ottenute usando altri operatori.

Consideriamo per prima la selezione. Riprendiamo, come esempio, la piccola base di dati dell'articolo precedente:

CORSI (CodiceCorso, NomeCorso,Docente,Semestre,N_Crediti)
STUDENTI (MatricolaStudente, Nome, Cognome, Indirizzo, CorsoDiLaurea,Anno)
ESAMI (MatricolaStudente,CodiceCorso, Voto)

e immaginiamo di voler estrarre tutti gli studenti del quarto anno di corso. Occorrerà utilizzare la tabella STUDENTI, ed estrarne tutte le tuple che soddisfano la proprietà che l'anno di corso sia uguale a 4.
Questa interrogazione si esprime in SQL come segue:

```
SELECT MatricolaStudente, Nome, Cognome, Indirizzo, CorsoDiLaurea,Anno
FROM STUDENTI
WHERE ANNO = 4
```

Per fare ciò si può utilizzare una selezione nel seguente modo:

$$\sigma_{Anno=4}\ \text{STUDENTI}$$

Il significato di questa espressione è per l'appunto *estrarre tutte le tuple della relazione STUDENTI che soddisfino la proprietà indicata a pedice del simbolo di selezione*. Si noti quindi che il predicato (proprietà) che si trova a pedice del simbolo di selezione esprime la clausola WHERE del linguaggio SQL.
In generale, il risultato di una selezione è una tabella che ha lo stesso schema e un numero di tuple (cardinalità) minore o uguale di quella originaria.
Consideriamo ora la proiezione. Immaginiamo di voler estrarre tutti i nomi e cognomi degli studenti. Occorrerà utilizzare la tabella STUDENTI, ed estrarne le colonne **Nome** e **Cognome**. Questa interrogazione si esprime in SQL come segue:

```
SELECT Nome, Cognome
FROM STUDENTI
```

In algebra relazionale, si usa la proiezione nel seguente modo:

$$\Pi_{Nome,Cognome}\ \text{STUDENTI}$$

Il significato di questa espressione è per l'appunto *estrarre le colonne* (*attributi*) *della relazione STUDENTI indicate a pedice del simbolo di proiezione*. Si noti quindi che il pedice del simbolo di proiezione esprime la clausola SELECT del linguaggio SQL. Purtroppo, come capita spesso nell'area delle basi di dati, accade che alcuni termini abbiano significato diverso in diversi contesti, e così succede con la parola SELECT che ha due significati diversi (ortogonali!) nei due linguaggi.
In generale, il risultato di una proiezione è una tabella che ha un numero di attributi inferiore, ed eventualmente cardinalità minore o uguale di quella originaria. Questo secondo punto va spiegato brevemente: si immagini, nel nostro esempio, che tra gli studenti elencati in tabella esistano alcuni omonimi. In tal caso, le relative tuple, nella tabella risultante dall'applicazione della proiezione, saranno indistinguibili. La semantica del modello relazionale prevede che le tabelle siano degli insiemi, e perciò privi di duplicati. Eventuali duplicati vanno quindi rimossi, e il numero di tuple della relazione risultante sarà, in generale, minore o uguale di quello della relazione originaria. Si noti che, nel caso tra gli attributi della proiezione compaia la chiave della relazione, evidentemente non vi saranno duplicati e la proiezione non abbasserà la cardinalità originaria.
Applicando il prodotto cartesiano a due tabelle, otteniamo una tabella che ha per tuple tutte le coppie possibili di tuple della prima e della seconda tabella. In generale, il risultato del prodotto cartesiano è una tabella che ha per schema l'unione dei due schemi, e per cardinalità il prodotto delle due cardinalità, delle tabelle in ingresso.
Il prodotto cartesiano, pur essendo un operatore fondamentale, di per sè non riveste grande interesse; infatti, giustapporre righe di tabelle in tutti i modi possibili non appare molto utile. L'interesse di questo operatore consiste, però, nell'operatore derivato che se ne ottiene *associando il prodotto cartesiano con la selezione*.
Supponiamo di voler estrarre tutti i dati degli studenti con i loro voti, e consideriamo la corrispondente interrogazione SQL:

```
SELECT STUDENTI.MatricolaStudente, Nome, Cognome, Indirizzo, CorsoDiLaurea, Anno, MatricolaStudente,CodiceCorso,Voto
FROM STUDENTI,ESAMI
WHERE STUDENTI.MatricolaStudente= ESAMI.MatricolaStudente
```

Abbiamo preso tutte le colonne delle due tabelle STUDENTI ed ESAMI, e abbiamo imposto la condizione che la matricola dello studente fosse uguale nelle tuple delle due tabelle. In realtà, ciò corrisponde ad accoppiare in tutti i modi possibili (prodotto cartesiano) le tuple delle due tabelle, e poi a prendere solo quelle che hanno la matricola uguale.
Per fare ciò in Algebra Relazionale, si possono utilizzare il prodotto cartesiano e la selezione nel seguente modo:

$$\sigma_{\substack{\text{STUDENTI.MatricolaStudente=}\\ \text{ESAMI.MatricolaStudente}}} (\text{STUDENTI} \times \text{ESAMI})$$

Esiste però un altro modo, più compatto, di esprimere la stessa interrogazione, ed è adoperando l'operatore *join*($\bowtie$), definito appunto come composizione di selezione e prodotto cartesiano:

$$\sigma_{\substack{\text{STUDENTI.MatricolaStudente=}\\ \text{ESAMI.MatricolaStudente}}} (\text{STUDENTI} \bowtie \text{ESAMI})$$

Il significato di questa espressione è esattamente lo stesso dell'espressione precedente, ma la notazione risulta più compatta.
Cerchiamo ora di mettere insieme gli operatori visti finora per comporre un'interrogazione più complessa.
L'interrogazione "*Trovare nomi e cognomi degli studenti che hanno ottenuto almeno il voto 28 all'esame di fondamenti di informatica*", che in SQL risulta:

```
SELECT Nome, Cognome
FROM STUDENTI,ESAMI,CORSI
WHERE  STUDENTI.Matricola=ESAMI.Matricola
AND ESAMI.CodiceCorso=CORSI.CodiceCorso AND
CORSI.NomeCorso="Fondamenti di Informatica"
and ESAMI.Voto>=28.
```

si esprime in algebra relazionale come segue:

$$\prod_{\text{Nome, Cognome}} [\sigma_{\substack{\text{CORSI.NomeCorso=}\\ \text{“Fondamenti di Informatica”}\\ \text{AND ESAMI.Voto}\geq 28}} (\text{STUDENTI} \bowtie \text{ESAMI} \bowtie \text{CORSI})]$$

Come si vede, una interrogazione è quindi formulata mediante una espressione algebrica i cui operandi sono i nomi delle relazioni cui vengono applicati operatori vari. Nell'esempio, alle tre relazioni viene applicato il cosiddetto join

naturale, che utilizza come default il predicato che impone l'uguaglianza degli attributi con lo stesso nome. In pratica, si tratta delle proprietà espresse dalla prima riga della clausola WHERE dell'interrogazione SQL. Applicando poi la selezione si prendono solo le tuple che soddisfano il predicato *CORSI.NomeCorso="Fondamenti di Informatica"* $\wedge$ *ESAMI.Voto>=28 (si ricorda che* $\wedge$ *rappresenta l'operatore "AND" logico*). La proiezione poi opera anch'essa una riduzione, prendendo solo gli attributi Nome e Cognome degli studenti.
Come si diceva all'inizio di questa sezione, l'algebra relazionale ben si presta a ragionamenti sull'ottimizzazione delle interrogazioni; si osservi, infatti, la seguente formulazione, semanticamente equivalente, della stessa interrogazione:

$$\prod_{\text{Nome,Cognome}} [(\text{STUDENTI} \rhd\!\lhd \sigma_{\text{ESAMI.Voto}\geq 28} \text{ESAMI} \rhd\!\lhd \sigma_{\text{CORSI.NomeCorso}=\text{"Fondamenti di Informatica"}} \text{CORSI})]$$

Risulta abbastanza intuitivo che le due operazioni di join, in questa nuova formulazione, verranno applicate a relazioni che in generale saranno molto più piccole delle precedenti, e quindi risulteranno meno costose. Ragionamenti analoghi, basati su simili *trasformazioni di equivalenza*, permettono di adoperare agevolmente l'algebra relazionale come formato interno al DBMS per l'ottimizzazione delle interrogazioni.
Concludiamo con la presentazione degli operatori insiemistici, l'unione e la differenza, dai quali, come succede anche nella teoria degli insiemi classica, si può facilmente definire anche l'operatore di intersezione (_). Supponiamo che la nostra base di dati contenga una tabella ulteriore, relativa agli studenti di dottorato:

STUDENTI_DOTTORATO (MatricolaStudente, Nome, Cognome, Indirizzo, CorsoDiDottorato,Anno)

e di voler conoscere l'elenco di tutti gli studenti dell'università. Questo risultato si ottiene applicando l'operatore di unione:

$$\text{STUDENTI} \cup \text{STUDENTI\_DOTTORATO}.$$

Analogamente, applicando l'operatore di differenza potremmo pensare di ricavare l'elenco degli studenti del corso di laurea che non hanno proseguito col dottorato.

$$\text{STUDENTI} - \text{STUDENTI\_DOTTORATO}.$$

Si noti che gli operatori insiemistici sono applicabili solo a coppie di tabelle che hanno schema compatibile, cioè i cui attributi, presi ordinatamente a due a due, hanno lo stesso tipo.
In SQL gli operatori insiemistici si esprimono mediante analoghi costrutti, come ad esempio la UNION:

```
SELECT MatricolaStudente, Nome, Cognome, Indirizzo, CorsoDiLaurea,Anno
FROM STUDENTI
UNION
SELECT MatricolaStudente, Nome, Cognome, Indirizzo, CorsoDiDottorato,Anno
FROM STUDENTI_DOTTORATO
```

3. Il Calcolo Relazionale e il Datalog

Nel Calcolo Relazionale le interrogazioni si esprimono specificando l'insieme che soddisfa una certa proprietà logica. Esso si presenta in due varianti, la cui differenza riguarda il tipo di variabile che viene usata. Nel *Calcolo delle Tuple*, la proprietà viene espressa in termini di tuple, nel *Calcolo dei Domini*, le variabili hanno come insieme di definizione non la tupla intera, ma i suoi elementi.
Per motivi di brevità noi presenteremo il Calcolo delle Tuple, che esprime le interrogazioni nella seguente forma standard:

$$\{t \mid p\,(t)\}$$

Il significato di questa espressione è l'insieme di tutte le tuple *t* per le quali è vera la proprietà *p*. *p* (*t*) è una formula costruita tramite *atomi*, e cioè formule logiche elementari come $t_1 \in$ STUDENTI oppure $t_3[NomeCorso] =$ "*Fondamenti di Informatica*". Vediamo subito un esempio provando a esprimere qualcuna delle interrogazioni elementari già viste in Algebra Relazionale. Consideriamo l'interrogazione:

$$\Pi_{\text{Nome,Cognome}}\, \sigma_{\text{Anno}=4}\, \text{STUDENTI}$$

ottenuta componendo le prime due interrogazioni di esempio, che richiede il nome e cognome degli studenti del quarto anno. Questa si

esprimerà, in calcolo delle tuple, nel modo seguente:

{*t* | ∃ *t*1∈ STUDENTI : (*t*[Nome] = *t*1[Nome] ∧ *t*[Cognome] = *t*1[Cognome] ∧ *t*1[Anno]=4)}.

Il risultato è quindi specificato definendo le tuple *t* costituite da un attributo Nome e un attributo Cognome, il cui valore è uguale a quello assunto dalle tuple *t*1 della relazione STUDENTI che verificano la proprietà che l'anno di corso sia il quarto. Come si può vedere, la specifica dell'interrogazione in calcolo delle tuple ricalca formalmente la sua formulazione in linguaggio naturale, ed è totalmente dichiarativa.
Per un esempio più complicato, vediamo come si esprime la solita interrogazione sugli studenti che hanno preso almeno 28 all'esame di Fondamenti di Informatica:

{*t* | ∃ *t*1∈ STUDENTI, ∃ *t*2∈ ESAMI, ∃ *t*3∈ CORSI (*t*[Nome] = *t*1[Nome] ∧ *t*[Cognome] = *t*1[Cognome] ∧ *t*1[MatricolaStudente] = *t*2[MatricolaStudente] ∧ *t*2[CodiceCorso] = *t*3[CodiceCorso] ∧ *t*2[Voto] >= 28 ∧ *t*3[NomeCorso] = "Fondamenti di Informatica")}.

Si noti come la condizione di join, cioè l'uguaglianza delle matricole e dei codici dei corsi, sia espressa in questo linguaggio.
L'integrazione della programmazione logica con le basi di dati è stata di grande interesse intorno agli anni '90, e aveva il grande obiettivo di permettere l'espressione di interrogazioni e computazioni all'interno dello stesso paradigma di programmazione. Il fondamento di questo tentativo giace nel fatto che la forma relazionale delle tabelle ha delle forti analogie sia strutturali che semantiche con la forma dei predicati logici. Per esempio, dire che la tupla *<AG213, Giovanni, Rossi, Via Pacini 25 MILANO, Ingegneria Informatica>* appartiene alla tabella **STUDENTI** è equivalente ad asserire la verità del seguente predicato con costanti:

STUDENTI (AG213, Giovanni, Rossi, Via Pacini 25 MILANO, Ingegneria Informatica)

Un'interrogazione Datalog si esprime mediante una o più regole logiche. Una regola è composta da un lato sinistro (detto anche *conseguente*) e da un lato destro (*antecedente*). Il lato destro può contenere uno o più predicati, che esprimono condizioni che devono essere soddisfatte per rendere soddisfatto anche il lato sinistro. Le relazioni sono quindi rappresentate da predicati logici, e gli attributi da variabili con lo stesso nome. I predicati possono contenere come argomenti sia delle costanti sia delle variabili: le costanti servono per vincolare l'attributo della corrispondente posizione ad assumere il valore di quella costante, mentre le variabili servono per associare tra loro predicati che le contengono.
Facciamo subito un esempio riformulando in Datalog qualcuna delle interrogazioni precedenti.
L'interrogazione sui nomi e cognomi degli studenti del quarto anno di corso si esprimerebbe in Datalog mediante la seguente regola logica:

StudentiAnziani(Nome, Cognome):-
STUDENTI (_, Nome, Cognome, _, _,4).

Come si vede, per imporre che l'anno di corso sia 4 abbiamo inserito il valore 4 nella posizione che compete all'anno di corso del predicato STUDENTI. Inoltre, le variabili Nome e Cognome, che pure appaiono nel predicato STUDENTI, vengono "trasferite" al predicato StudentiAnziani, che in pratica raccoglie il risultato della interrogazione. I trattini bassi sono invece dei segnaposto, che indicano posizioni di variabili di cui non si fa uso nella specifica regola.
Le condizioni di uguaglianza tra attributi sono espresse dall'uso della stessa variabile nei vari predicati nel lato destro; consideriamo la solita interrogazione sugli studenti di fondamenti di informatica:

Studentebravo(Nome, Cognome):- STUDENTI (MatricolaStudente, Nome, Cognome, _, _,_),
CORSI (CodiceCorso, NomeCorso,_,_,_),
ESAMI(MatricolaStudente,CodiceCorso,Voto),
Voto>=28, NomeCorso="Fondamenti di Informatica".

Qui, la variabile MatricolaStudente si trova sia nel predicato STUDENTI che nel predicato CORSI, realizzando il join, e lo stesso vale per la variabile CodiceCorso e per i predicati CORSI ed ESAMI.

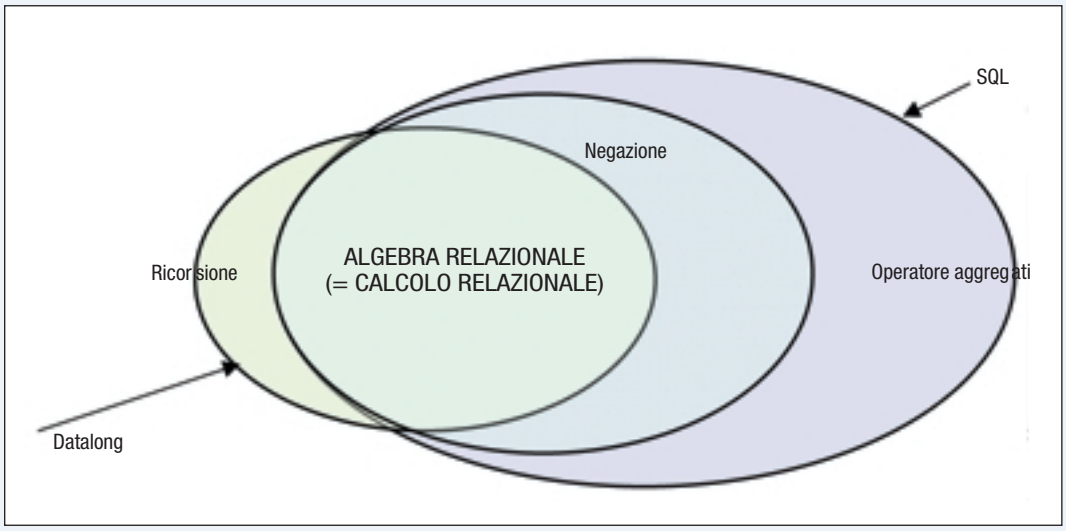

FIGURA 1
Poteri espressivi dei vari linguaggi

Mediante un tale linguaggio, si riescono ad esprimere computazioni più complesse che in algebra relazionale, come per esempio la computazione ricorsiva espressa dalla seguente coppia di regole:

Antenato(X,Y) :- Genitore(X,Y).
Antenato(X,Y) :- Antenato(X,Z),Genitore(Z,Y).

Partendo da una relazione di base GENITORE, che contiene tutte le coppie genitore-figlio, queste due regole specificano l'insieme di tutte le coppie antenato-discendente. La prima regola costituisce il caso base: "*se sei mio genitore sei anche un mio antenato*", la seconda regola specifica la computazione ricorsiva: "*se sei un antenato di un mio genitore allora sei anche mio antenato*".
Si noti che un programma Datalog ha come risultato l'*insieme delle tuple che soddisfano le condizioni logiche espresse dalle sue regole*, perciò è anch'esso *orientato agli insiemi*. Il linguaggio Datalog, con le sue varie estensioni, può costituire quindi un'alternativa all'accoppiata SQL/linguaggio di programmazione, in quanto permetterebbe di esprimere uniformemente sia le interrogazioni che le computazioni generiche, e perciò di superare il disaccoppiamento di impedenza. D'altra parte, questa soluzione, molto brillante a livello teorico, comporta una serie di problemi non banali, che non ne hanno permesso l'applicazione pratica nei DBMS commerciali. Vi sono infatti, relativamente ai programmi logici ricorsivi, sia problemi semantici relativi all'uso della negazione e dei costrutti di aggregazione, sia problemi di ottimizzazione.

4. Conclusioni e considerazioni sul potere espressivo dei linguaggi per basi di dati

La figura 1 ben riassume le relazioni tra i poteri espressivi dei vari linguaggi trattati in questo articolo. Come si può vedere, l'Algebra Relazionale (equivalente al Calcolo Relazionale) è pienamente contenuta nell'SQL. Infatti, questo ultimo linguaggio offre, dal punto di vista della manipolazione delle tabelle, esattamente lo stesso potere espressivo dell'Algebra e del Calcolo, ma in più offre alcuni costrutti per svolgere semplici computazioni aggregate, come la media, la somma, il minimo e massimo, e la possibilità di contare quante sono le tuple che soddisfano una certa proprietà.
D'altro canto, il Datalog estende l'Algebra Relazionale mediante il costrutto di ricorsione, non presente nelle versioni comuni di SQL[2], ma non permette di esprimere la negazione (corrispondente, in Algebra, alla differenza) se non a prezzo di alcuni artifici semantici di cui non interessa parlare in questo contesto.
In conclusione, i linguaggi formali per basi di dati forniscono strumenti matematici per trattare con grande rigore questioni di ottimizzazio-

[2] Gli standard più recenti di SQL offrono qualche forma di ricorsione, ma di fatto tale possibilità non viene introdotta nella maggior parte dei sistemi commerciali.

ne, di potere espressivo e di complessità, difficilmente affrontabili su linguaggi come l'SQL, più verbosi e meno adatti al trattamento mediante strumenti matematici.

Bibliografia

Anche questa volta, piuttosto che elencare le migliaia di riferimenti ai concetti e linguaggi introdotti, indichiamo soltanto alcuni testi classici, da aggiungere a quelli proposti nell'articolo introduttivo, che peraltro già contengono molte delle nozioni viste in questo articolo.

[1] Abiteboul Serge, Hull Richard, Vianu Victor: *Foundations of Databases*. Addison-Wesley 1995.

[2] Ceri Stefano, Gottlob Georg, Tanca Letizia: *Logic Programming and Databases*. Springer 1990.

[3] Ullman Jeffrey D.: *Principles of Database and Knowledge-Base Systems*, Vol. I e II. Computer Science Press 1989.

Letizia Tanca è Professore Ordinario di Basi di Dati presso il Politecnico di Milano, e presidente del Consiglio di Corso di Studi di Ingegneria Informatica del Politecnico di Milano, campus Milano Leonardo. È autrice di diverse pubblicazioni internazionali sulle basi di dati e sulla teoria delle basi di dati, e del libro "Logic Programming and Databases", scritto con S. Ceri e G. Gottlob. Ha partecipato a vari progetti nazionali e internazionali. I suoi interessi di ricerca riguardano tutta la teoria delle basi di dati, in particolare le basi di dati deduttive, attive e orientate agli oggetti, i linguaggi a grafi per basi di dati, la rappresentazione e l'interrogazione di informazione semistrutturata, le basi di dati per piccoli dispositivi.
E-mail: tanca@elet.polimi.it