Basi di dati

Appunti delle lezioni di
Paolo Alfano

Note Legali

Liberatoria, aggiornamenti, segnalazione errori:

Quest'opera viene pubblicata in formato elettronico senza alcuna garanzia di correttezza del suo contenuto. Il documento, nella sua interezza, è opera di Paolo Didier Alfano
e viene mantenuto e aggiornato dallo stesso, a cui possono essere inviate eventuali segnalazioni all'indirizzo *paul15193@hotmail.it*
Ultimo aggiornamento: 31 gennaio 2016

Indice

1 Sistemi

def. Database : un database è una raccolta di dati permanenti divisi in due parti:
metadati che definiscono lo schema dei dati
dati che rappresentano i fatti

nella costruzione di un database sono coinvolti un committente e un fornitore. Quest'ultimo deve mettere a punto il database e gestirlo tramite il DBMS(Data Base Management System). Solitamente i database vanno a sostenere i sistemi informativi delle organizzazioni.

def. Sistema informativo : il sistema informativo è l'insieme di procedure e risorse(umane o materiali) per la raccolta, l'archiviazione, l'elaborazione e lo scambio delle informazioni necessarie all'attività[1]

def. Sistema informatico : insieme di tecnologie a supporto delle attività di un'organizzazione

def. Sistema informativo automatizzato : parte del sistema informativo in cui le informazioni vengono manipolate per mezzo del sistema informatico

i sistemi informatici possono essere di due tipi:

1. Sistemi informatici operativi: le informazioni vengono organizzati in basi di dati e sono utilizzate per le classiche e ripetitive attività dell'azienda. In questo tipo di sistemi si elaborano le informazioni tramite *transazioni*, eseguendo operazioni semplici, con pochi dati necessariamente aggiornati

2. Sistemi informatici direzionali: i dati sono organizzati in *Data Warehouse* e vengono analizzati per supportare le decisioni. Le operazioni su questi sistemi informatici possono essere complesse, con grandi quantità di dati che non devono essere necessariamente aggiornati

	Operativi	Direzionali
Scopo	supporto operatività	supporto decisionale
Utenti	molti, esecutivi	pochi, analisti e dirigenti
Quantità di dati	bassa (decine)	alta (milioni)
Aggiornamenti	frequenti	rari
Ottimizzati per	transazioni	analisi dati

def. DBMS: un data base management system è un sistema centralizzato o distribuito che offre opportuni linguaggi per: definire lo schema di un database, sceglierne le strutture dati, memorizzare i dati rispettando i vincoli definiti nello schema, recuperare e modificare i dati

[1] Un semplice esempio di sistema informativo potrebbe essere quello creato da un comune per la gestione dei servizi demografici, o per quelli amministrativi e sanitari

un DBMS deve prevedere più modalità d'uso per soddisfare i diversi utenti che vi accedono: un'interfaccia grafica per accedere ai dati, un linguaggio di interrogazione per gli utenti non programmatori, un linguaggio di programmazione per chi sviluppa le applicazioni.
Allo stesso tempo il DBMS deve garantire il mantenimento delle proprietà dei dati, la loro protezione da usi non autorizzati e da malfunzionamenti hardware o software

def. Transazione: sequenza di azioni in lettura o scrittura in memoria avente le seguenti proprietà:

- atomicità, nel senso che le transazioni terminate prematuramente sono trattate come se non fossero esistite
- serializzabilità, più transazioni concorrenti hanno l'effetto di un'esecuzione seriale
- persistenza, le modifiche eseguite da un transazione effettuata correttamente sono permanenti

2 Progettazione di Basi di dati

28/09/2015

La progettazione di basi di dati inizia trascurando le operazioni che dovremo andare a fare. Quello che dobbiamo fare inizialmente consiste nello stabilire quali siano le informazioni che ci servono. Tali informazioni vengono fornite dal committente. Trascuriamo le operazioni da fare per due motivi:

1. i dati durano per più tempo delle operazioni. Infatti le operazioni sono volatili e flessibili(nel senso che possono essere aggiunte nel tempo)
2. abbiamo anche un motivo tecnologico, ovvero a inserire le operazioni e ad ottimizzarle ci penseremo tramite il DBMS

quindi inizialmente dobbiamo soltanto rappresentare le informazioni nel modo più fedele possibile.
Il meccanismo linguistico che usiamo nella progettazione delle basi di dati è inizialmente quello degli *oggetti* e delle *classi*. Questo meccanismo viene rappresentato graficamente tramite i diagrammi ER(Entità-Relazione).
Abbiamo alcune differenze tra la classe dei linguaggi ad oggetti e la classe che utilizzeremo nel seguito. Le classi del linguaggio ad oggetti sono *intensionali* ovvero permettono di generare nuovi oggetti mentre la classe che intendiamo noi è una collezione di elementi omogenei alla quale posso ad esempio chiedere chi vi appartenga.
La classe, nella base di dati verrà rappresentata come segue:

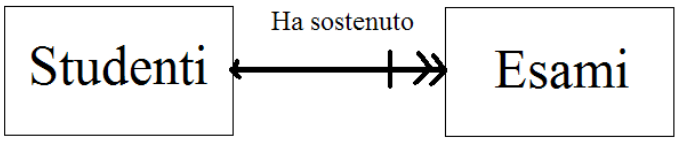


Figura 1: L'associazione tra studenti ed esami sostenuti

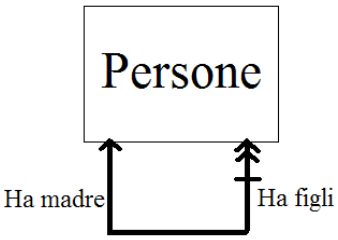


Figura 2: L'associazione ricorsiva sulla classe Persone

nome classe	
attr_1	tipo_1
attr_2	tipo_2
$\vdots$	$\vdots$
attr_n	tipo_n

è dunque evidente che ogni oggetto ha degli *attributi*.

def. Associazione : le associazioni sono fatti che correlano due o più entità

Un esempio di associazione potrebbe essere:
"X ha sostenuto l'esame Y"
che è la generalizzazione dell'istanza
"Mario ha sostenuto Basi di Dati"
Le associazioni vengono rappresentate come in figura 1.

Ogni associazione è esprimibile come un attributo. Ad esempio nella classe studenti potremmo inserire l'attributo *esami sostenuti*, ma come vedremo, questa non è una buona pratica programmativa.
Le associazioni possono anche essere ricorsive, come è possibile vedere dalla figura 2.

Possiamo inoltre collegare ad ogni associazione delle proprietà. Ad esempio potremmo dire che *"X prende in prestito il libro Y in data Z"*. La proprietà *data* non è né di Mario né del libro, è invece dell'associazione.
Bisogna però dire che di norma le associazioni non hanno proprietà, avviene

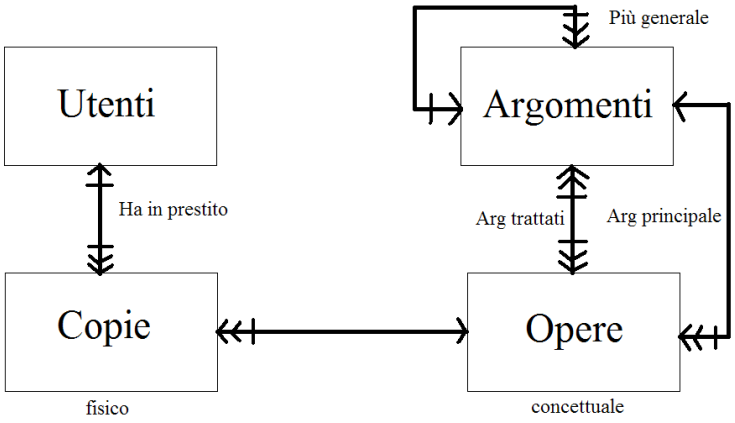


Figura 3: Lo schema di proggettazione della base di dati

solo in rari casi.
Vediamo adesso un esempio di progettazione di una base di dati.

Esempio 1 : supponiamo che il committente desideri gestire i libri di una biblioteca dividendoli per argomento. Supponiamo che le classi che andremo ad utilizzare(i rettangoli che dunque vedremo nel nostro schema) siano: libri, utenti e generi.

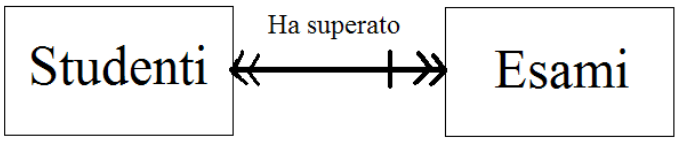
Lo schema, che viene riportato in figura 3, illustra diversi formalismi adottati nella rappresentazione delle associazioni. Anzitutto notiamo che quando il committente parla di un libro, commette un'ambiguità, perché di un libro ne possono esistere diverse copie e dunque è bene introdurre due classi, una per le copie(aspetto fisico) del libro e una per l'opera(aspetto concettuale). Consideriamo l'associazione tra gli Utenti e le copie. Visto che ogni utente può avere molti libri in prestito la cardinalità è superiore a 1, e dunque poniamo due segni nella freccia che va dagli utenti alle copie. Al contempo ogni utente può anche avere 0 libri in prestito. Questo fatto è evidenziato dalla linea perpendicolare alla freccia in direzione delle copie. Allo stesso tempo ogni copia può essere posseduta da un solo utente(cardinalità 1) dunque facciamo un solo segno della freccia che va dalle copie agli utenti. Ma allo stesso tempo ogni copia può anche essere posseduta da nessun utente. Anche in questo caso allora poniamo la linea perpendicolare alla freccia.
Consideriamo adesso l'associazione tra copie e opere. Poiché per ogni copia vi è associata una sola opera(ma almeno una sì!) allora poniamo un solo segno nella freccia che va verso le opere. Per ogni opera possono esistere più copie, quindi facciamo due segni dalle opere verso le copie. Allo stesso tempo se tutte le copie di un libro vanno perdute allora la cardinalità può essere 0. Allora possiamo porre la linea perpendicolare alla freccia

in direzione opere-copie.
In ogni caso consideriamo che la cardinalità non è banale. Infatti potrebbe anche essere un'opera è una raccolta di più opere, e allora avrei dovuto mettere due segni da sinistra verso destra. Un altro modo per rappresentare questo fatto poteva essere quello di creare un'associazione ricorsiva dalle opere alle opere.
Per quanto riguarda l'associazione argomenti-opere, ho che per ogni argomento posso avere molte opere (2 segni verso il basso) ma anche nessuno(barra orizzontale in basso). Inoltre per ogni opera posso avere più argomenti(2 segni verso l'alto) ma anche che un'opera non tratta alcun argomento(barra orizzontale in alto). Volendo potrei anche considerare *l'argomento principale* di un'opera. Per farlo aggiungo una seconda associazione tra gli argomenti e le opere. Allora ogni argomento può essere argomento principale di più opere (2 segni in basso) ma anche essere argomento principale di nessuna opera(segno perpendicolare in basso). Invece ogni opera può avere un solo argomento principale(1 segno in alto).
Un'alternativa alle due associazioni consisterebbe nel mettere una proprietà booleana che mi indica se l'argomento è principale o meno. Mettere due classi è invece sbagliato perché nella mente del committente gli argomenti non si dividono in principali e secondari, bensì ogni argomento può essere principale in un'opera e secondario in un'altra.
Le due associazioni tra le classi sono spesso legate da una relazione. Nel nostro esempio, l'argomento principale è **sempre** trattato. In questo caso la relazione è di unione. Se invece avessimo avuto due relazioni diverse, ovvero: argomento principale e argomenti secondari, allora un argomento principale non può mai essere anche argomento secondario. In questo caso abbiamo una relazione di esclusione.
Per determinare la specificità di un argomento inserisco una associazione ricorsiva sugli argomenti. Cercare di rappresentare questo fatto tramite gli attributi della classe argomenti è sbagliato.
In generale è sempre bene cercare di esprimere tramite classi e associazioni tutto quello che possiamo esprimere tramite esse.
Infine, se non siamo sicuri per quanto riguarda la cardinalità di una associazione è meglio mettere la cardinalità "molti" (due segni sulla freccia) in quanto essa include la cardinalità singola(segno singolo sulla freccia)

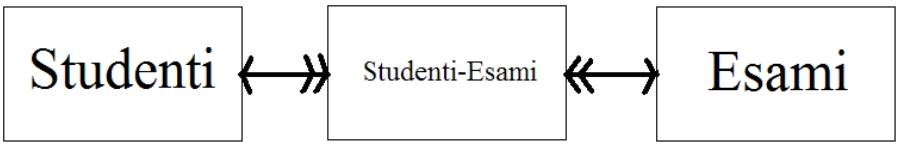
30/09/2015

Quando andiamo a modellare, ci sono diversi aspetti cui fare attenzione:

- L'ottimizzazione di un modello è rischiosa perché eliminando classi e/o relazioni, potremmo non riuscire a rappresentare correttamente la realtà.
- Le classi vengono istituite quando hanno almeno 2 elementi. Da ciò si evince che i nomi delle classi devono sempre essere plurali
- Solitamente un'associazione molti a molti può essere promossa a classe come in figura 4.



(a) Senza classe aggiuntiva



(b) Con classe aggiuntiva

Figura 4: Conversione di relazione molti a molti in una classe

- Quando abbiamo una associazione 1 a molti avente una proprietà, probabilmente posso includerla in una delle due classi
- Un ciclo nel grafo potrebbe(ma non necessariamente)indicare una ridondanza di associazioni

2.1 Sottoclassi

Le sottoclassi sono uno strumento per andare a specializzare una superclasse. Ad esempio gli "studenti" sono una sottoclasse delle "persone". Solitamente le sottoclassi vengono introdotte quando vogliamo raccogliere una maggior quantità di informazioni, che spesso coincide con l'inserimento di nuovi attributi nella classe. Infatti se non aggiungiamo attributi solitamente non è necessario aggiungere una sottoclasse.
A livello pratico, quando due o più classi presentano gli stessi attributi o attributi molto simili, potremmo essere in una situazione in cui è necessario trasformarle in sottoclassi raccolte sotto un'unica superclasse. Le sottoclassi possono anche essere caratterizzate tramite i *vincoli*:

- due sottoclassi sono *sottoclassi scorrelate* se un elemento può appartenere ad entrambe le sottoclassi, ovvero se, prese due sottoclassi A e B
  $\exists x : (x \in A) \wedge (x \in B)$
- due sottoclassi sono *sottoclassi disgiunzione* nel caso in cui un elemento non possa appartenere ad entrambe le classi, ovvero se, prese due classi, A e B vale che $A \cap B = \emptyset$
- due sottoclassi sono *sottoclassi copertura* se ogni elemento della superclasse appartiene ad una delle sottoclassi, ovvero se presa una superclasse C e due sottoclassi A e B vale che $\forall c \in C \rightarrow c \in A \cup B$

Figura 5: Rispettivamente, la rappresentazione di sottoclassi disgiunzione e copertura

- due sottoclassi sono *sottoclassi partizione* se sono sia sottoclassi disgiunzione che sottoclassi copertura.
  Ad esempio se consideriamo la superclasse delle persone e le due sottoclassi *minorenne* e *maggiorenne*, abbiamo che le due sottoclassi sono copertura(perché coprono tutte le persone) e disgiunzione perché nessun elemento può appartenere ad entrambe le sottoclassi.

In questa gerarchia possiamo avere anche ereditarietà multipla, come mostrato dalla figura 6.

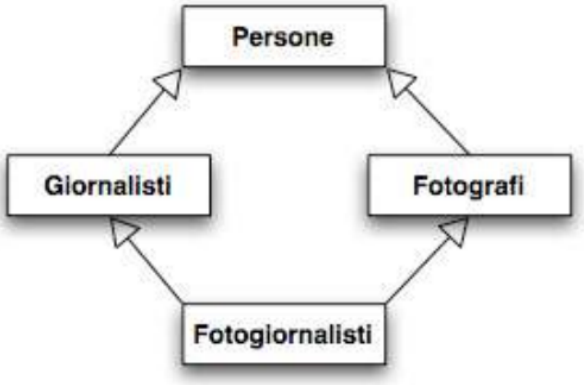


Figura 6: Ereditarietà multipla

3 Il modello relazionale

05/10/2015

Nel modello relazionale cominciamo a trasformare gli schemi ad oggetti in schemi logici basati su tabelle. Le tabelle che andiamo a costruire cercano di esprimere le classi e le relazioni che avevamo.
Nel modello relazionale abbiamo due entità fondamentali:

1. n-upla: è un riga della tabella
2. tabella/relazione: è l'insieme delle n-uple

Andiamo ad elencare alcuni dei termini che incontreremo tramite un esempio

Esempio 1 Supponiamo di avere le due seguenti tabelle

Studenti

Cognome	Matricola	Provincia	Anno nascita
Alfano	501234	PI	1994
Carosi	502112	PI	1994
Dominici	491212	MB	1994
Ruggeri	511011	LU	1995

Esami

Materia	Candidato	Data	Voto
SOL	501234	04/09/15	25
SOL	502112	09/09/15	23
SOL	501234	15/01/15	30

Ad ogni n-upla abbiamo associato un *tipo n-upla* che fornisce le informazioni sulla n-upla tramite un insieme di coppie della forma (nome attributo, tipo). Ad esempio il tipo n-upla della prima tabella è
$\{(cognome, stringa), (matricola, intero),$
$(Provincia, stringa), (Annonascita, intero)\}$
Abbiamo anche una *struttura di relazione* o *schema di relazione* che nelle due tabelle è costituito dalla prima riga con in testa il nome della tabella. Ad esempio per la seconda tabella la struttura di relazione è data da
$\{Esami, Materia, Candidato, Data, Voto\}$
Inoltre abbiamo che schema di relazione+istanza=relazione
Sottolineiamo inoltre che le n-uple sono insiemi, dunque l'ordine con cui vengono espresse non conta.

3.1 Superchiavi, chiavi..

def. Superchiave: una superchiave è un'insieme di attributi di una struttura di relazione tale per cui, se due n-uple coincidono su quell'insieme di attributi, allora coincidono anche sugli altri.

Riferendoci all'esempio di poco fa, il numero di Matricola potrebbe essere una superchiave poiché se due entry nella tabella coincidono sul numero di matricola, allora le due entry rappresentano lo stesso studente.
Questo non succedeva nello schema ad oggetti dove potevamo avere entità identiche ma distinte(le copie del libro). Supponiamo di avere una superchiave A sulla tabella ABCD. Matematicamente la definizione di superchiave ci dice due cose:[2]

1. $t_1[A] = t_2[A] \Rightarrow t_1[BCD] = t_2[BCD]$
2. $t_1[BCD] \neq t_2[BCD] \Rightarrow t_1[A] \neq t_2[A]$

[2]Ricordando che in logica $A \rightarrow B \Leftrightarrow \neg B \rightarrow \neg A$ e che $A \wedge (\neg B) \rightarrow \perp$ dove $\perp$ indica l'impossibilità

Adesso possiamo ricavare la definizione di *chiave*

def. Chiave: un insieme di campi è detto chiave se è una *superchiave minimale* ovvero quando è superchiave e rimuovendone un campo cessa di essere superchiave

Esempio 2: la coppia (matricola, cognome) è una superchiave in quanto identifica in modo univoco ogni studente. Il numero di matricola è però anche una chiave in quanto fa parte di una superchiave e poiché rimuovendo il cognome la distinzione resta univoca. Non si può dire altrettanto per il cognome che invece non da una rappresentazione univoca degli studenti.

def. Chiave primaria: la chiave scelta a priori dal progettista che identifica l'elemento nel tempo. La chiave primaria è una delle possibili chiavi, preferibilmente quella con meno attributi

Solitamente la chiave primaria è un attributo costante e obbligatorio per ogni entry della tabella. Possibilmente deve essere anche unico per rendere più semplice la gestione.
Solitamente il progettista introduce una sua variabile che identifichi in modo totalmente univoco gli elementi della tabella(verosimilmente un indice crescente).
Nella nostra notazione, la chiave primaria viene sottolineata all'interno della struttura di relazione.

def. Chiave esterna: una chiave è detta esterna quando l'attributo è chiave primaria per un'altra tabella

Nell'esempio di prima il campo candidato della seconda tabella è chiave esterna visto che è contenuto anche nella tabella studenti e ne è chiave primaria.
Notiamo che di solito le chiavi esterne servono per rappresentare le relazioni del linguaggio a oggetti. Riprendendo l'esempio di prima, l'inserimento del campo candidato nello schema di relazione serve per rappresentare la relazione "ha sostenuto" che intercorre tra la classe studenti e la classe esami. Dobbiamo però ricordarci che nel nuovo schema relazionale in primo luogo è una colonna della tabella, dunque un attributo.
Nello schema di relazione aggiungeremo in fondo al nome della chiave esterna il simbolo $*$ per segnalare che si tratta effettivamente di una chiave esterna.
A questo punto, è necessario evidenziare che è possibile passare da uno schema ad oggetti ad uno schema relazionale trasformando le associazioni uno a molti in una chiave esterna all'interno di una tabella. Questo procedimento è indicato in figura 7. Come possiamo notare, la relazione molti a molti è stata spezzata in due relazioni uno a molti. Inoltre notiamo che l'informazione contenuta nello schema relazionale è inferiore a quella contenuta nello schema ad oggetti. Ad esempio il fatto che ogni editore debba avere almeno un libro non è evidenziato dallo schema relazionale. Questo perché lo schema relazionale è sempre affiancato dalle tabelle che servono per disambiguare.
A questo punto potremmo chiederci come rappresentare il concetto di sottoclasse nello schema relazionale. Per farlo abbiamo tre metodi:

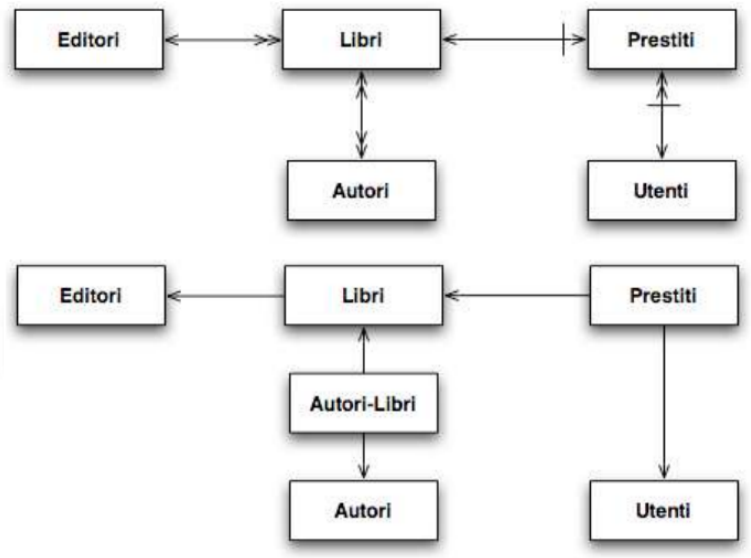


Figura 7: Conversione da modello oggetti a modello relazionale

1. Partizionamento verticale: ogni sottoclasse diventa una nuova tabella che contiene una chiave esterna che coincide con la chiave primaria della tabella associata alla superclasse
2. Campi opzionali: aggiunta di una serie di campi opzionali alla superclasse, di modo che possa rappresentare anche gli elementi della sottoclasse
3. Tabelle distinte: definisco due tabella distinte senza ridondanza di informazioni. Ad esempio definisco la tabella studenti contenente tutti gli studenti che però non potranno appartenere anche alla tabella persone

3.2 Attributi multivalore

07/10/2015

Gli attributi multivalore si presentano ogni volta che per ogni oggetto avente un attributo, questo può essere non unico. Un esempio potrebbe essere rappresentato dalla classe *dipendenti* avente attributo *numero di telefono*. Potrebbe esistere un dipendente che ha più di un telefono o, se parliamo di un telefono dell'ufficio potrebbe succedere che un telefono sia usato da più dipendenti. Se vogliamo eliminare gli attributi multivalore dal nostro schema, è necessario trasformare l'attributo multivalore in una classe. Ad esempio potremmo introdurre la classe *telefoni* che contiene la coppia (ID dipend. , numero) che ad ogni dipendente associa un entry per ogni telefono

Esercizio 1: L'Università vuole usare una base di dati per raccogliere informazioni sui dipendenti, sui corsi di laurea e sugli studenti iscritti. Degli

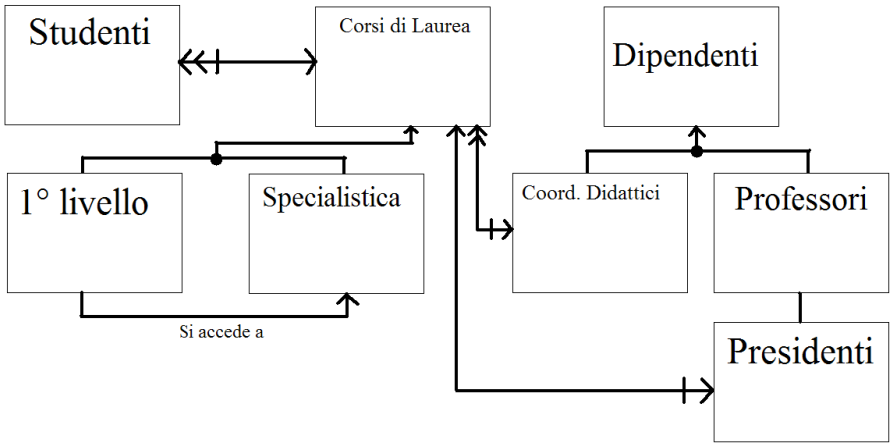


Figura 8: Schema a oggetti dell'esercizio 1

studenti interessano la matricola, il nome, l'anno di nascita e il corso di laurea. Dei corsi di laurea interessano il codice, il nome,il presidente, un dipendente professore ordinario o associato, e il dipendente con funzione di coordinatore didattico. Un docente può essere presidente di un solo corso di laurea. Un coordinatore didattico può essere assegnato a più corsi di laurea. Dei presidenti di corsi di laurea interessa l'anno di nomina. I corsi di laurea si dividono in corsi di primo livello ed in corsi di laurea specialistica. Per ciascuno corso di primo livello interessa conoscere quali sono i corsi di laurea specialistici in cui è possibile proseguire gli studi senza debito formativo e quali quelli in cui sia possibile proseguire con debito; in questo caso, interessa la quantità massima di CFU per tale debito. Dei dipendenti interessano il codice, il nome, la qualifica, i recapiti telefonici, e l'anno di assunzione.

Lo schema che rappresenta meglio questa realtà, è quello rappresentato in figura 8. Una scelta diversa che avremmo potuto fare riguarda i Corsi di Laurea. Potremmo inserire una relazione ricorsiva sui Corsi di Laurea per rappresentare l'accesso da un corso triennale ad uno di secondo livello. Questa scelta non è errata ma può dare luogo a qualche problema. Ad esempio potremmo avere accesso da corsi di laurea triennali a corsi di laurea triennali.
Una generalizzazione che potremmo fare è quella di racchiudere in una superclasse *persone* gli studenti e i professori visto che hanno diversi attributi in comune.

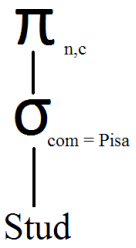


Figura 9: AST dell'esempio 2

3.3 Interrogazioni di basi di dati

14/10/2015

Le interrogazioni, ovvero le richieste che possono essere fatte alla base di dati, vengono espresse in due modi differenti:

1. Algebra relazionale: formata da un'insieme di operatori su relazioni(tabelle) che danno come risultato altre relazioni.
2. Calcolo relazionale: è un linguaggio dichiarativo logico che vedremo meglio più avanti

Questi due modi di fare un'interrogazione hanno lo stesso potere espressivo, nonostante i due approcci siano diversi. Iniziamo analizzando l'algebra relazionale. Nell'algebra relazionale abbiamo due operatori fondamentali:

- Proiezione: indicata con $\pi_{A,B}(r)$, viene letta come "proietto r su A,B". L'operazione di proiezione consiste nel rimuovere dalla tabella un'insieme di colonne che non ci servono
- Restrizione: $\sigma_{cond}(r)$, viene letta come "restringo r su cond". L'operazione di restrizione consiste nello scartare un'insieme di righe che non ci servono.

Vediamo subito un paio di esempi.

Esempio 1: se volessi ottenere tutti i nomi e i cognomi degli studenti prelevandoli da una tabella, utilizzerei $\pi_{n,c}(TAB)$

Esempio 2: vediamo come combinare insieme proiezione e restrizione. Se mi chiedessi, data una tabella, come ottenere i nomi e i cognomi di tutti gli studenti il cui comune di nascita è Pisa, dovrei chiedere:
$\pi_{n,c}(\sigma_{com=Pisa}(STUD))$
Notiamo che abbiamo utilizzato la notazione polacca inversa, ma solitamente possiamo andare ad esprimere un'interrogazione tramite un albero di sintassi astratta. Nel nostro caso l'albero è mostrato in figura

Notiamo che la proiezione restituisce una relazione che è un'insieme, allora eventuali ennuple uguali sono ridotte ad un solo elemento nel risultato. Non succede la stessa cosa se la proiezione opera su un *multiinsieme*

def. Multiinsieme: è una generalizzazione dell'insieme in cui possiamo avere degli elementi ripetuti ma il loro ordine non conta.

Ovviamente sugli insiemi possiamo richiedere tutta una serie di operazioni già note come la cardinalità o altro. Sui multiinsiemi, la proiezione non elimina i duplicati.
Notiamo che nell'algebra relazionale abbiamo le combinazioni booleane ma non facciamo uso dei quantificatori $\forall, \exists \ldots$. Possiamo dunque definire alcuni insiemi:

- $cond := exp$ theta $exp \mid cond$ and $cond \mid Not\ cond$
- $exp := attr|cost|exp\ op\ exp$
- $theta :=< \mid > \mid = \mid != \mid <= \mid >=$
- $op := +|-|*$

Abbiamo inoltre due operatori, l'operatore unione $\cup$ e l'operatore differenza $\setminus$. Questi operatori, per essere applicati, devono lavorare devono lavorare su operandi (cioè le tabelle) aventi stessi campi.

Notiamo che fino ad adesso abbiamo introdotto operatori che dopo che sono stati applicati "restringono" il numero degli elementi in gioco.
Il prodotto va nella direzione opposta:

Esempio 3: vediamo il prodotto tra due tabelle:

a	A
a1	A1

$\times$

b	B
b1	B1
b2	B2
b3	B3

$=$

a	A	b	B
a1	A1	b1	B1
a1	A1	b2	B2
a1	A1	b3	B3

Un proprietà interessante del prodotto è che può anche essere fatto tra tabelle non omogenee, ad esempio *studenti* $\times$ *esami*.
Inoltre se una delle due tabelle è vuota, il prodotto da una tabella vuota, mentre se una delle due tabelle è composta da una sola riga (come nell'esempio) il prodotto restituirà una tabella con tante righe quante sono le righe della tabella con più di una riga.
Come usiamo nella pratica tutti gli elementi che abbiamo spiegato finora?

Esempio 4: supponiamo di avere due tabelle:
Studenti(Matr, Nome, Cogn, Città, Sesso, Anno Nasc)
Esami(Cand*, Materia, Data, Voto)
Supponiamo di voler creare un'altra tabella:
Esami con nome(Cand, Matr, Nome, Cogn, Data, Voto)
Per creare una tabella del genere potremmo fare prima una proiezione $\pi_{Nome,Matr}$ sulla tabella degli studenti, seguita da un prodotto con la tabella degli esami, seguita da una restrizione $\sigma_{Cand=Matr}$ Lo schema è illustrato in figura 10.

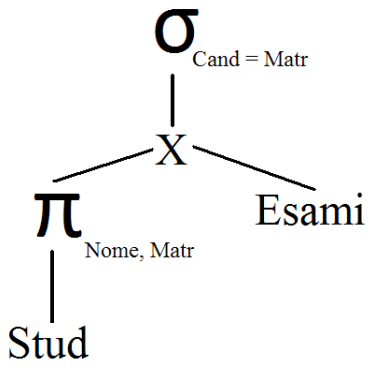


Figura 10: AST dell'esempio 4

Il prodotto deve essere usato attentamente perché se usato a caso può portare facilmente ad un eccesso di informazione. Per questo, il prodotto è solitamente seguito dall'operatore di restrizione σ.
Fare un prodotto, seguito da una restrizione è talmente comune che è diventato un operatore unico, la *giunzione*. Viene solitamente indicata con il simbolo $\bowtie_{A=B}$ dove A=B indica su quali righe devo fare la restrizione.

Esempio 5: introduciamo il simbolo di giunzione, e notiamo che avevamo lasciato due colonne ridondanti(Matr, Cand). Lo schema complessivo è mostrato in figura 11. Notiamo che il primo π è stato eliminato perché ne abbiamo inserito un altro dopo la giunzione, quindi era divenuto inutile.

Osserviamo che alcuni operatori vogliono operandi omogenei(unione $\cup$, differenza $\setminus$), altri operatori invece non li chiedono necessariamente(prodotto $\times$). Per poter utilizzare in contemporanea gli uni e gli altri introduciamo l'operatore di *ridenominazione* e di *giunzione naturale*

- Ridenominazione: indicato con $\rho_{A \to B}$ prende l'attributo A e cambia il suo nome in B.

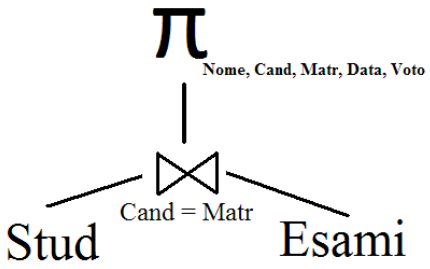
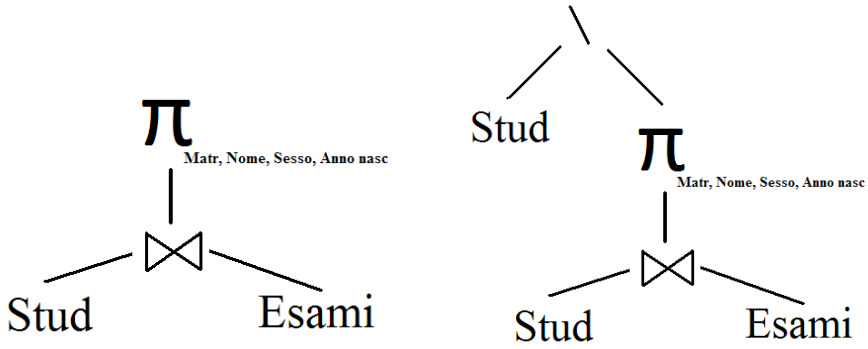


Figura 11: AST dell'esempio 5



(a) Esempio 6, prima interrogazione (b) Esempio 6, seconda interrogazione

Figura 12: Schema degli AST dell'esempio 6

- Giunzione naturale: indicata con $R \bowtie S$ restituisce una nuova tabella i cui attributi sono $attr_A \cup attr_B$ le cui ennuple sono formate concatenando quelle di R e S con valori uguali per gli attributi in comune.

Esempio 6: riprendiamo le due tabelle viste nei due esempi precedenti. Proviamo a risolvere qualche interrogazione.

- vogliamo ottenere la tabella contenente gli attributi Nome, Matr, Anno nasc, Sesso di tutti gli studenti che hanno fatto almeno un'esame. Per rispondere a questa interrogazione dobbiamo prima effettuare una giunzione naturale, poi una proiezione. La giunzione elimina tutti gli studenti che non hanno dato esami, questo perché implicitamente effettua una moltiplicazione tra lo studente e gli esami che ha dato. Se non ha dato neanche un'esame sto moltiplicando per una tabella vuota, dunque nella tabella finale non risulterà neanche un entry per quello studente. Lo schema complessivo è mostrato in figura 12 a.
- vogliamo ottenere una tabella contenente il nome e la matricola di tutti quelli che non hanno superato neanche un'esame. Lo schema è molto simile a quello di prima con una aggiunta. Aggiungere la differenza dopo la proiezione consente di sottrarre all'insieme degli studenti, tutti quelli che hanno fatto almeno un'esame.

19/10/2015

Introduciamo ora un nuovo operatore. Viene chiamato operatore di *raggruppamento*(o operatore di group by), e viene solitamente indicato con la lettera γ. Vediamo come opera attraverso un esempio

Esempio 7 : supponendo di avere le solite due tabelle contenenti gli studenti e gli esami, vogliamo ottenere una tabella che contenga per ogni studente

quanti esami ha fatto il voto minimo che preso, il voto massimo e la media dei suoi voti. Per farlo usiamo l'operatore group by
$\{candidato\}\gamma\{count(*), min(voto), max(voto), avg(voto)\}^{(esami)}$
Supponiamo che la tabella fosse:

Stud	Esami	Voto
1	AA	20
2	BB	20
1	CC	24
2	AA	30

Allora l'operatore di raggruppamento opera in due fasi:

Stud	Esami	Voto
1	AA	20
1	CC	24
1	BB	20
2	AA	30

$\rightarrow$

Stud	Count	Min	Max	Avg
1	2	20	24	22
2	2	20	30	25

Nella prima fase group by ordina e separa gli elementi della tabella in base al primo gruppo di parametri(in questo caso solo candidato), successivamente "raggruppa" tutte le righe di ogni candidato in una sola, ottenendo nuove informazioni(come la media), magari perdendone altre, come i voti che ha preso che non siano né il massimo né il minimo.

Notiamo che aver inserito count(∗) conta tutte le righe indistintamente, anche se vi sono dei duplicati. Se ad esempio avessimo inserito count(distinct.esame), e uno studente ha dato dieci volte lo stesso esame, allora otteniamo il valore uno come risultato.
In ogni caso l'uso del distinct deve essere pensato perché ad esempio invocarlo sui voti mi elimina i duplicati dei voti, che invece sono legittimi.

Esempio 8 : disegniamo l'albero di sintassi astratta che mi indica quali studenti hanno dato almeno 10 esami e che hanno una media maggiore di 27. La soluzione è mostrata in figura 13.

Notiamo che se avessimo voluto aggiungere la condizione che gli esami devono essere solo del primo anno, ci sarebbe bastato aggiungere un $\sigma_{Annoesame=1}$ subito sopra gli esami.

Una cosa che è importante comprendere è che tramite il raggruppamento, in qualche modo comprimo le informazioni perdendone alcune, dunque è importante il momento in cui lo applico perché potrebbero esserci delle operazioni che posso effettuare soltanto prima o soltanto dopo.
Un esempio di cattivo uso dell'operatore di raggruppamento è il seguente:
$\{voto\}\gamma\{avg(voto)\}$
Infatti raggruppo tutti gli esami dividendoli in base al voto. A quel punto calcolo la media su ogni gruppo ma è evidente che tutti gli elementi dello stesso gruppo avranno lo stesso voto, dando luogo a una richiesta poco significativa.
In modo leggermente più formale, in presenza di un raggruppamento
$\{a_1, a_2, \ldots\}\gamma\{b_1, b_2, \ldots\}$

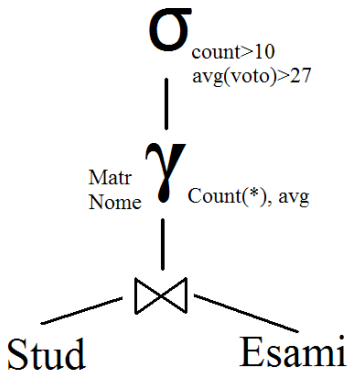


Figura 13: AST dell'esempio 8

chiameremo $a_1, a_2, \ldots$ *dimensione* mentre chiameremo *misure* $b_1, b_2, \ldots$
Notiamo allora che può avere senso inserire il voto nella dimensione, magari perché voglio vedere per ogni voto quali sono gli esami che lo hanno dato più spesso. Quello che è sbagliato è inserire il voto sia nella dimensione che nelle misure.

3.4 Trasformazioni algebriche

20/10/2015

Le trasformazioni algebriche mettono in relazione i diversi operatori visti finora consentendo di fare ottimizzazioni. Sono basate su regole di equivalenza fra espressioni algebriche e le più importanti sono:
$\pi_A(\pi_{A,B}(R)) \equiv \pi_A(R)$
$\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c1 \wedge c2}(R)$
$\sigma_{c1 \wedge c2}(R \times S) \equiv \sigma_{c1}(R) \times \sigma_{c2}(S)$
$R \times S \equiv S \times R$
$R \times (S \times T) \equiv (R \times S) \times T$
$\sigma_c(x\gamma F^{(R)}) \equiv x\gamma F^{\sigma_c(R)}$
Come dicevamo, queste trasformazioni consentono di ottimizzare, magari spostando una proiezione o una restrizione prima di un prodotto, rendendolo meno pesante.

4 SQL

Il linguaggio SQL(Structured Query Language), sviluppato nel 1973 è un linguaggio universale per la gestione di sistemi relazionali. Da una base generale, si sono sviluppati diversi *dialetti* di SQL. Noi faremo uso di SQL per creare delle

query. La differenza più importante che abbiamo con quanto visto finora, è che possiamo introdurre i quantificatori. Vediamo allora un po' di sintassi.

4.1 Sintassi

I comandi di base sono:

- SELECT, seguito da un'insieme di campi che mi indicano i campi del risultato della query.
- FROM, si trova solitamente subito dopo una SELECT. Mi indica in quali tabelle dovrò effettuare la query. Se abbiamo più di una tabella dopo il FROM, allora ne viene fatto il prodotto cartesiano.
- WHERE, stabilisce un'insieme di condizioni che gli elementi devono rispettare per essere restituiti. Se le condizioni sono più d'una, possono essere concatenate con un AND
- ORDER BY, il risultato della SELECT viene ordinato secondo il valore di certi attributi, in ordine decrescente se opportunamente specificato tramite l'opzione aggiuntiva DESC
- GROUP BY, crea diverse partizioni di un insieme di elementi andando a raggruppare più istanze in una sola. Dunque si comporta in modo analogo alla γ che avevamo visto nel modello relazionale.

La prima cosa da notare è che tramite la sintassi di SQL possiamo ricostruire quasi tutti gli elementi già visti nel modello relazionale

Esempio 1 : vediamo come ricostruire facilmente la proiezione sul seguente caso: vogliamo ottenere la tabella contenente nome, cognome e provincia di nascita degli studenti contenuti nella solita tabella studenti

```
SELECT Nome, Cognome, Provincia
FROM Studenti
```

Esempio 2 : vediamo come ricostruire la restrizione con il seguente esempio: vogliamo ottenere tutte le informazioni riguardanti i soli studenti di Pisa.

```
SELECT *
FROM Studenti
WHERE Studenti.Provincia = Pisa
```

Inserire il simbolo di asterisco nella SELECT indica di tenere tutti i campi nel risultato

Esempio 3 : vediamo come ricostruire il prodotto con il seguente caso: voglio ottenere tutti gli esami dati da un certo studente

```
SELECT *
FROM Studenti, Esami
```

Esempio 4 : vediamo come ricostruire la giunzione con il seguente caso: vogliamo la lista degli studenti che hanno superato l'esame di Basi di Dati con voto uguale a 30

```
SELECT *
FROM Studenti, Esami
WHERE Esami.Voto = 30 AND Studente.Matricola = Esami.Matricola
    AND Esami.Materia = BD
```

Esempio 5 : vediamo un utilizzo della clausola ORDER BY per effettuare ordinamenti. Supponendo di voler ottenere gli studenti ordinati per nome la query sarà

```
SELECT *
FROM Studenti
ORDER BY Nome
```

Esempio 6 : vediamo come ricostruire le funzioni di aggregazione in SQL con il seguente esempio: supponiamo di voler ottenere l'anno di nascita massimo, minimo e medio degli studenti

```
SELECT max(AnnoNascita), min(AnnoNascita), avg(AnnoNascita)
FROM Studenti
```

Esempio 7 : ricostruiamo l'operatore di raggruppamento con il seguente esempio: vogliamo ottenere il nome e il voto medio di ogni studente. La query in questo caso è

```
SELECT s.Nome, avg(e.Voto)
FROM Studenti s, Esami e
WHERE s.Matricola = e.Matricola
GROUP BY s.Matricola, s.Nome
```

Notiamo che gli attributi espressi e non aggregati nella SELECT(nel nostro caso solo s.Nome) devono essere citati nella GROUP BY. Inoltre gli attributi aggregati(come avg(e.Voto)) devono essere selezionati tra quelli NON presenti nella GROUP BY

In SQL esiste una distinzione fra le espressioni *analitiche* e *sintetiche*.
Le prime sono quelle che ottengo senza effettuare vere e proprie manipolazioni sui dati, gli esempi da 1 a 5 visti sopra calcolano espressioni analitiche. Le espressioni sintetiche sono tutte quelle che necessitano di un qualche calcolo, come visto nell'esempio 6.

21/10/2015

Notiamo che fino ad adesso abbiamo parlato di SELECT ma in realtà abbiamo visto solo delle sottoselect. Infatti la sintassi della SELECT si basa sulle sottoselect. Con questo si intende dire che attraverso gli opportuni operatori di unione, intersezione e altri, possiamo combinare diverse sottoselect. Esistono diversi operatori per quanto riguarda l'unione. La UNION elimina i duplicati, la UNION ALL invece no.

4.2 Il valore NULL

Il valore NULL è un valore speciale che può essere assunto da uno o più campi di un'ennupla. Normalmente lo si associa a campi indeterminati o sconosciuti. Il valore NULL non può però trovarsi in un campo che è chiave per una data tabella(può invece esserlo se quel campo è chiave esterna).
L'intuizione che sta dietro al valore NULL è che un campo uguale a NULL potrebbe contenere qualunque valore del dominio.
L'introduzione del NULL porta diversi problemi logici a cui dobbiamo porre rimedio. Per farlo, utilizzeremo una logica a tre valori {TRUE, FALSE, NULL} a cui facciamo seguire alcune leggi logiche.
La logica che usiamo può essere *stretta* o *non stretta*. La logica si dice stretta se, nel caso in cui sia coinvolto un valore NULL, il risultato restituito sarà sempre NULL. La logica non stretta fa invece qualche eccezione.
Ad esempio il confronto tra due valori di cui uno sia NULL non è pensabile. In questo caso la logica è stretta, dunque restituiamo sempre NULL. Anche nel caso di espressioni aritmetiche usiamo una logica stretta. Con questo si intende che ad esempio, NULL sommato a un qualunque valore darà sempre NULL, lo stesso possiamo dire della moltiplicazione e così via.
La logica non stretta la applichiamo nei casi in cui il risultato di un'espressione è determinato indipendentemente dal valore NULL. Ad esempio:

FALSE $\wedge$ NULL $\Rightarrow$ FALSE
TRUE $\vee$ NULL $\Rightarrow$ TRUE

in questi casi infatti, qualunque valore sul dominio sia assunto da NULL, il risultato dell'espressione logica è noto. Non si può dire altrettanto per gli esempi che seguono in cui il valore restituito è infatti NULL

FALSE $\vee$ NULL $\Rightarrow$ NULL
TRUE $\wedge$ NULL $\Rightarrow$ NULL

Vediamo come utilizzare NULL in un esempio

Esempio 1 : vogliamo trovare tutti gli studenti di cui non ho il numero di telefono.

```
SELECT *
FROM Studenti s
```

```
WHERE s.Telefono IS NULL
```

Attenzione che scrivere WHERE s.Telefono = NULL è gravemente sbagliato. Per fare controlli di questo tipo esiste l'operatore speciale IS NULL che abbiamo utilizzato sopra

Diamo ora qualche altro dettaglio sul valore NULL.
Quando viene effettuata una GROUP BY, tutti coloro che hanno valore NULL finiscono in uno stesso gruppo. La motivazione è pratica, infatti se ad esempio volessi ottenere una divisione che mi indica quanti studenti sono nati in ogni comune, allora otterrò un campo per ogni comune più uno con un valore che mi indica tutti quelli di cui non conosco il comune, piuttosto che una entry per ognuno di essi.
Nel caso vi sia da calcolare espressioni sintetiche contenenti un valore uguale a NULL(ad esempio il calcolo di una media che comprende un voto uguale a NULL), tali valori vengono ignorati.

4.3 Quantificazione

Le quantificazioni sono di due tipi: *esistenziali* e *universali*.
Notiamo che fra queste due quantificazione esiste un rapporto ben chiaro. L'una è la negata dell'altra.

Esempio 1 : Non tutti i voti sono ≤ 24 = Almeno un voto >24 (esistenziale)
Non esiste voto diverso da 30 = Tutti i voti sono uguali a 30 (universale)

Le due quantificazioni possono essere espresse in vari modi. Il modo che useremo di più per esprimere la quantificazione esistenziale è quello mostrato nell'esempio che segue

Esempio 2 : vogliamo sapere gli studenti con almeno un voto maggiore di 27

```
SELECT s.Nome
FROM Studenti s
WHERE EXISTS (SELECT *
        FROM Esami e
        WHERE e.Matricola = s.Matricola AND e.Voto > 27)
```

Per risolvere questa query abbiamo fatto uso dell'operatore EXISTS con l'ovvio significato. Un altro modo per farlo è usare l'operatore ANY

Esempio 3 :utilizzando l'operatore ANY, la query dell'esempio 2 diviene

```
SELECT s.Nome
FROM Studenti s
WHERE 27 <ANY (SELECT e.Voto
                FROM Esami e
                WHERE e.Matricola = s.Matricola)
```

Quando parliamo di quantificazione universale invece, dobbiamo stare attenti all'errore più comune che viene illustrato nell'esempio che segue

Esempio 4 : vogliamo tutti gli studenti che hanno preso sempre 30. La query che potremmo pensare è

```
SELECT s.Nome
FROM Studenti s, Esami e
WHERE e.Matricola = s.Matricola AND e.Voto = 30
```

Il problema è che questa query sta invece esprimendo una quantificazione esistenziale! Basandoci sulla logica del primo ordine, la formula che vogliamo esprimere è:
$\{Studenti : \forall e \in Esami : e.Voto = 30\}$
La versione corretta è della query è

```
SELECT s.Nome
FROM Studenti s
WHERE FORALL Esami e WHERE e.Matricola=s.Matricola: e.Voto=30
```

Il problema è che in SQL non esiste il quantificatore universale FORALL. Dunque dobbiamo ricavarlo tramite le leggi di De Morgan come segue:
$\forall x \in E : P(x) \Leftrightarrow \neg \exists x \in E : \neg P(x)$

Esempio 5 : riprendendo l'esempio 4, in definitiva, trasformiamo la query

```
SELECT s.Nome
FROM Studenti s
WHERE FORALL Esami e WHERE e.Matricola=s.Matricola: e.Voto=30
```

nella query che segue

```
SELECT s.Nome
FROM Studenti s
WHERE NOT EXISTS (SELECT *
                  FROM Esami e
                  WHERE e.Matricola=s.Matricola AND e.Voto<>30)
```

Esiste una regola (quasi) generale per stabilire se una query è esistenziale o universale. Solitamente le richieste dentro agli EXISTS sono universali, quelle fuori invece sono esistenziali.
In realtà molte delle interrogazioni universali semplici possono essere risolte tramite la GROUP BY, vediamo invece un caso in cui è necessario utilizzare i quantificatori universali introdotti.

Esempio 6 : supponendo di avere una tabella di persone identificate tramite un id, voglio ottenere la lista di persone che hanno almeno un amico con

meno di 20 anni. La formula logica è
$\{y : \forall xAy.\ x.eta < 20\}$
La query che otteniamo è

```
SELECT P1.nome
FROM persone P1
WHERE NOT EXISTS (SELECT *
                  FROM amico_di A, persone P2
                  WHERE A.ID1=P1.id AND A.id2=P2.id AND
                  AND NOT(P2.età<20))
```

26/10/2015

Torniamo nuovamente sulla quantificazione universale con un esempio

Esempio 7 : vogliamo trovare
$\{stud$ s | $\exists$ esame e: e.matr = s.matr $\wedge$ e.voto>27 $\}$
potremmo anche usare la quantificazione universale
$\{stud$ s | $\forall$ esame e: e.matr = s.matr $\wedge$ e.voto>27 $\}$
ma questa richiesta è assurda perché sto chiedendo che tutti gli esami della tabella esami siano fatti dallo studente s.
Possiamo porre rimedio a questa situazione ponendo un $\Rightarrow$ al posto del $\wedge$. Dobbiamo comunque stare attenti perché se uno studente non ha dato esami, per la regola dell'implicazione questa query su di lui restituirà sempre TRUE, che è sbagliato.

4.4 Dualità di De Morgan

In logica diciamo che AND e OR sono duali nel senso che l'uno può essere espresso con l'altro. Questo concetto può anche essere esteso ai quantificatori.
A $\wedge$ B $\equiv \neg(\neg$A $\vee \neg B)$
$\neg(\forall x.P(x)) \equiv \exists x.\neg P(x)$
$\neg(\exists x.P(x)) \equiv \forall x.\neg P(x)$
Componendo queste regole otteniamo
$\neg(\forall x \in A.P(x)) \equiv \neg(\forall x \in A \Rightarrow P(x)) \equiv \neg(\forall x.\neg x \in A \vee P(x)) \equiv \exists x.x \in A \vee \neg P(x) \equiv \exists x \in A.\neg P(x)$
In questo modo abbiamo ottenuto il risultato che avevamo intuitivamente introdotto la scorsa lezione, ovvero che $\forall$ diventa $\exists$
Infine una formula ovvia ma che viene sbagliata spesso è la seguente:
$\neg(A \wedge B) \equiv \neg A \vee \neg B$

4.5 SQL per inserzione

Per inserire qualcosa abbiamo
INSERT INTO tabella $[A_1, \ldots, A_n]$
(VALUES$(V_1, \ldots, V_n)$ | AS select)
Abbiamo diverse forme di inserimento, per esempio quella appena vista inserisce riga per riga, un'altra prevede l'inserimento di un'intera tabella.

Per eliminare qualcosa dalla tabella usiamo:
DELETE FROM tabella
WHERE condizione

Con SQL possiamo anche creare tabelle:

Esempio 1 : Creiamo la tabella degli impiegati

```
CREATE TABLE Impiegati(
        Codice CHAR(8) NOT NULL,
        Nome CHAR(20),
        AnnoNascita INTEGER CHECK (AnnoNascita < 2000),
        Qualifica CHAR(20) DEFAULT "Impiegato",
        Supervisore CHAR(8),
        PRIMARY KEY pk_impiegato (Codice),
        FOREIGN KEY fk_Impiegati (Supervisore)
        REFERENCES Impiegati )
```

I vincoli più importanti sono PRIMARY KEY e FOREIGN KEY che definiscono la chiave primaria e quella esterna

Possiamo anche calcolare delle tabelle temporanee per effettuare operazioni particolari con la CREATE VIEW

Esempio 2 : Vediamo una CREATE VIEW

```
CREATE VIEW Supervisori AS
        SELECT Codice, Nome, Qual., Stip.
        FROM Impiegati
        WHERE Supervisore IS NULL
```

quando viene invocata *supervisori* viene lanciata "al volo" la query specificata

La differenza è che le CREATE creano effettivamente una tabella mentre la CREATE VIEW crea una tabella all'interno di una query che una volta conclusa rimuove la tabella. Le tabelle fatte con una CREATE VIEW, anche dette *tabelle vista* si interrogano come le altre ma non si modificano.
Notiamo infine che le CREATE VIEW aumentano effettivamente il nostro potere espressivo.
Esiste anche un comando ALTER per modificare una tabella esistente. Posso aggiungere colonne(operazione poco pericolosa) o rimuoverle. Rimuovere le colonne può essere pericoloso perché se quella colonna è chiave esterna per qualcuno la base di dati smette di funzionare.
Come gestire allora l'eliminazione di un elemento a in una tabella A che è chiave esterna per un'altra tabella B? Abbiamo tre alternative

1. NO ACTION: l'eliminazione viene impedita
2. CASCADE: elimino tutti gli elementi della tabella B di cui a era chiave esterna

3. SET NULL: setto tutti gli elementi della tabella B di cui a era chiave a NULL

Esempio 3 : viene eliminato uno studente s dalla tabella degli studenti S. La matricola dello studente s era però chiave esterna per la tabella esami E. I tre modi di reagire sono la NO ACTION che impedisce l'eliminazione di s, oppure la CASCADE che elimina tutti gli esami sostenuti da s contenuti nella tabella E oppure la SET NULL che setta tutti gli esami sostenuti da s nella tabella E a NULL

09/11/2015

Abbiamo visto come con SQL si possano fare interrogazioni, creazioni di tabelle e altre cose complesse. Adesso ci chiediamo come usarlo nei programmi. Abbiamo tre approcci possibili

1. Linguaggio integrato: viene utilizzato un linguaggio scritto appositamente per poter utilizzare anche il codice scritto in SQL. Un linguaggio di questo tipo è PL/SQL, che prende la sintassi del Pascal come base aggiungendovi anche le funzioni di SQL. In questo linguaggio gli errori di tipo vengono segnalati a compile time.

2. Linguaggio e API: abbiamo un linguaggio che utilizza delle librerie linkate per poter comprendere le richieste effettuate in SQL. Tutto questo viene fatto nei casi in cui non si abbia a disposizione un precompilatore. Il linguaggio con API è diverso dal linguaggio integrato perché con le API il controllo di tipo solitamente non viene fatto a compile time perché le richieste in SQL sono interpretate come stringhe e dunque non controllabili

11/11/2015

3. Linguaggio che ospita SQL: il codice in SQL viene inserito all'interno di porzioni di codice di un diverso linguaggio, permettendo ai due linguaggi di cooperare.
   Uno dei problemi di maggior rilievo in questo caso è legato alla compatibilità dei tipi a basso e ad alto livello tra i due linguaggi. Ad esempio, ad alto livello, SQL prevede l'esistenza degli insiemi che invece non si trovano in tutti i linguaggi. A basso livello invece potremmo avere problemi di dimensione dei dati perché magari SQL supporta interi fino ad una certa cifra mentre Java no, o viceversa.
   Problemi di questo tipo ce ne sono molti e per ovviare ad essi è stato introdotto il concetto di *cursore*. Il cursore in termini moderni è un oggetto, ovvero un'area di memoria a cui associo dei metodi con cui posso crearlo, modificarlo e consentire un trasferimento dai campi delle tabelle SQL alle opportune variabili del linguaggio ospite. Il cursore è talmente comune che viene usato anche in linguaggi integrati in alternativa ai meccanismi classici.

5 Teoria delle BD relazionali

Con la teoria delle basi di dati relazionali possiamo riprogettare BD mal strutturate piuttosto che gettarle via e ricominciare da capo.
Per stabilire se una BD è pensata male utilizzeremo il concetto di *forma normale*. Quando una tabella non sarà in forma normale, utilizzeremo opportuni *metodi di normalizzazione*

Esempio 1 : Consideriamo la seguente BD.

N Inv	Stanza	Resp	Oggetto	Produttore	Descrizione
1012	256	Ghelli	Mac Mini	Apple	PC
1015	312	Albano	Dell XPS	Dell	Notebook
1034	256	Ghelli	Dell XPS	Dell	Notebook
1112	288	Leoni	Mac Mini 2	Apple	PC

La tabella appena vista è mal pensata perché ogni volta che un professore ha più di un computer, abbiamo una ridondanza di informazioni sul professore. Ulteriori "dritte" per migliorare una base di dati possono esserci date dal committente

Esempio 2 : consideriamo la tabella data dai seguenti campi
StudentiEdEsami(Matricola, Nome, Provincia,AnnoNascita, Materia,Voto)
Anche in questo caso abbiamo ridondanza di informazione perché ogni studente appare più volte per ogni esame con tutte le sue informazioni associate.

Nei due esempi appena visti il problema certamente non sta nello spreco di spazio, piuttosto nel fatto che le ridondanze possono portare alla formazione di inconsistenze nella base di dati, oppure alla formazione di anomalie nell'inserimento(inserisco due esami fatti da uno stesso studente ma per sbaglio inserisco dati differenti sullo studente) o nell'eliminazione.
La cosa migliore per ovviare a questi problemi sarebbe quella di pensare la tabella degli studenti separata da quella degli esami, inserendo in entrambe solo e soltanto il campo Matricola per legarle.

5.1 Dipendenze funzionali

Le dipendenze funzionali sono un linguaggio formale che viene utilizzato per modellare le proprietà dei fatti che ci interessano.
Il primo concetto che introduciamo è quello di *dipendenza funzionale*

Def. Dipendenza funzionale : data una tabella R(T) e X, Y $\subseteq$ T, una dipendenza funzionale è un vincolo su R del tipo X $\rightarrow$ Y, dove X determina funzionalmente Y o Y è determinato da X, se per ogni istanza valida di R un valore di X determina in modo univoco un valore di Y. In termini matematici:

$\forall$ r istanza valida di R
$\forall$ t1, t2 $\in$ r. se t1[X] = t2[X] allora t1[Y] = t2[Y]

Da qui in avanti, il termine alla sinistra di $\rightarrow$ verrà detto *determinante* mentre il termine a destra verrà chiamato *determinato*.
Informalmente, tramite le dipendenze funzionali posso dire se, dati un'insieme di attributi, ne posso determinare univocamente un altro

Esempio 1 : consideriamo la BD
DotazioniLibri(CodiceLibro, NomeNegozio, IndNegozio, Titolo, Quantità)
alcune dipendenze funzionali sono
CodiceLibro $\rightarrow$ Titolo
NomeNegozio $\rightarrow$ IndNegozio
CodiceLibro, NomeNegozio $\rightarrow$ IndNegozio, Titolo, Quantità

Con questo si intende dire che dato il CodiceLibro, posso determinare in modo univoco il Titolo del libro. Oppure che dato il CodiceLibro unito al NomeNegozio posso ottenere IndNegozio, Titolo, Quantità.
Questa ultima DF presenta dati ridondanti nella parte destra infatti come evidenziato dalla prima delle tre DF il titolo era già univocamente determinato dal codice e l'indirizzo del negozio dal Nome del negozio.

Notiamo che esistono due asimmetrie in quanto visto nell'esempio di cui sopra. La prima è che la parte destra può essere spezzata a nostro piacimento come segue:

CodiceLibro, NomeNegozio $\rightarrow$ IndNegozio, Titolo, Quantità

può diventare

CodiceLibro, NomeNegozio $\rightarrow$ IndNegozio
CodiceLibro, NomeNegozio $\rightarrow$ Titolo
CodiceLibro, NomeNegozio $\rightarrow$ Quantità

e viceversa.
Inoltre notiamo che tanti più campi aggiungo nella parte destra della DF quanto più ottengo informazioni in uscita, mentre inserendo più campi nella parte sinistra, si riduce la quantità di informazione che ottengo.

Un altro problema che si presenta spesso è quello di interpretare le dipendenze funzionali in base a quello che viene detto dal committente. Vediamolo tramite un esempio.

Esempio 2 : supponendo di avere la BD determinanta dalla n-upla
Orari(CodAula, NomeAula, Piano, Posti, Materia, CDL, Docente, Giorno, OraInizio, OraFine)
che riguarda le aule di un polo universitario. Supponiamo che il committente formuli le seguenti condizioni

a) "non esistono due docenti che insegnano la stessa materia in uno stesso corso di Laurea"
La DF che ricaviamo in modo intuitivo è
CDL, Materia → Docente

b) "non posso avere due docenti diversi in una stessa aula nello stesso giorno alla stessa ora"
In questo caso la DF è
Giorno, Aula, Ora → Docente

c) "tutte le lezioni durano 2 ore"
In questo caso otteniamo due diverse DF
OraInizio → OraFine
OraFine → OraInizio
Anche se dobbiamo notare che queste DF sono più deboli di quella da me richiesta visto che queste due DF interpretano anche tante altre richieste come quella che le lezioni durino sempre 3 ore.

d) "se ho 2 aule con stesso codice e diverso numero di posti, allora le 2 aule sono su piani diversi"
questa DF non è intuitiva come le altre perché implicitamente intende dire che il codice dell'aula non è chiave nella base di dati. Una DF che potremmo pensare potrebbe essere:
CodAula, Posti → Piano
ma non è così infatti basti pensare a due aule, una avente codice 1 e 100 posti al primo piano e l'altra avente sempre codice 1 e 100 posti ma al secondo piano. Risulta evidente che a questo punto tramite il codice dell'aula e i posti non riesco ad individuare in modo univoco il piano a cui mi trovo. Quella corretta è:
Piano, CodAula → Posti

In realtà esiste un metodo più generale e semplice per stabilire quali siano le DF chieste dal committente. Prendiamo a titolo d'esempio:
$A_= \wedge B_\neq \wedge C_= \rightarrow D_= \wedge E_= \wedge F_\neq$
posso spostare a destra e a sinistra della freccia della DF i vari membri cambiandone il "segno". Lo scopo è spostare tutti i membri affinché restino solo uguaglianze. Allora riprendendo quanto detto poco fa otteniamo
$A_= \wedge C_= \wedge F_= \rightarrow D_= \wedge E_= \wedge B_=$
Applichiamolo al caso d) dell'esempio 2

Esempio 3 : la richiesta era
"se ho 2 aule con stesso codice e diverso numero di posti, allora le 2 aule sono su piani diversi"
che può essere scritta come:
$\text{CodAula}_= \wedge \text{Posti}_\neq \rightarrow \text{Piano}_\neq$
mentre io la vorrei formata da soli membri con il simbolo =
Allora sposto a destra della DF il membro Posti e a sinistra della DF il membro Piano cambiando il pedice ad entrambi. In tal modo ottengo:

CodAula$_=$∧ Piano$_=$ → Posti$_=$
che è esattamente la DF vista nell'esempio 2.

16/11/2015

5.2 Terminologia

def. DF complete : una dipendenza funzionale $X \to Y$ è completa quando, $\forall\ W \subseteq X$ non può succedere che $W \to Y$

Con questo si intende dire che l'insieme di attributi X che creano la dipendenza su Y, sono tutti e soli gli attributi necessari per generare la dipendenza, dunque se prendo un sottoinsieme di tali attributi, non generano la dipendenza.

def. Superchiave : se X è superchiave, allora determina ogni altro attributo della relazione $X \to T$

def. Chiave : Se X è una chiave, allora $X \to T$ è una DF completa

Spesso succede che da un'insieme di DF se ne possano derivare altre

def. Dipendenze Implicate : sia F un insieme di DF sullo schema R, diremo che F implica logicamente $X \to Y$, se ogni istanza r di R che soddisfa F soddisfa anche $X \to Y$. La notazione che usiamo per rappresentare questo fatto è $F \models X \to Y$

Esempio 1 : consideriamo F=$\{A \to B, B \to C\}$. Possiamo concludere che $A \to C$? Sì! Infatti è abbastanza facile dimostrare che considerate due righe qualunque di una tabella, se vale l'ipotesi vale anche la tesi.

Più in generale abbiamo che la dipendenza implicata deve valere su ogni riga secondo questa formula:

$$\forall r_0(r_0 \models F \Rightarrow r_0 \models X \to Y)$$

dove r_0 rappresenta la generica riga.
Le dipendenze funzionali sono *banali* quando il determinato è un sottoinsieme del determinante[3].
Come capire allora quando una dipendenza è implicata da un'insieme di altre dipendenze o meno? Se vogliamo dimostrare che un'insieme di dipendenze F non implica un'ulteriore DF, basta presentare un controesempio.
Per dimostrare invece che $\forall r_0(r_0 \models F \Rightarrow r_0 \models X \to Y)$ devo costruire un sistema di regole di inferenza basato sugli "*assiomi di Armstrong*"[4]. Li presentiamo di seguito:

[3] Ad esempio, presi A e B insiemi di attributi con $A \to B$, abbiamo che tutti gli attributi di A sono contenuti in B, ovvero $A \subseteq B$. Un esempio potrebbe essere $AB \to B$

[4] Usiamo le virgolette perché in realtà non sono veri assiomi, ma soltanto un'insieme di regole di inferenza

1. Se $Y \subseteq X$, allora $X \to Y$ (Riflessività R)
2. Se $X \to Y$, $Z \subseteq T$, allora $XZ \to YZ$ (Arricchimento A)
3. Se $X \to Y$, $Y \to Z$, allora $X \to Z$ (Transitività T)

diamo adesso la definizione di *derivabile*

def. Derivabile : Sia F un insieme di DF, diremo che $X \to Y$ sia derivabile da F (e lo indicheremo con $F \vdash X \to Y$), $\Leftrightarrow X \to Y$ può essere inferito da F usando gli assiomi di Armstrong.

Tramite la nozione di derivabilità si ricavano i due seguenti risultati intuitivi:
$\{X \to Y, X \to Z\} \vdash X \to YZ$ (unione U)
$Z \subseteq Y\{X \to Y\} \vdash X \to Z$ (decomposizione D)
Dall'unione U e dalla decomposizione D si ricava inoltre che se $Y = A_1 A_2 \dots A_n$ allora $X \to Y \Leftrightarrow \{X \to A_1, X \to A_2, \dots, X \to A_n\}$

Notiamo che l'implicazione $\models$ e la derivazione $\vdash$ sono diverse ma legate da una proprietà:
$\forall f \; F \vdash f \Rightarrow F \models f$ (correttezza)
In realtà un sistema diventa molto più interessante se vale anche la *completezza* ovvero se a partire dagli assiomi riesco a derivare nuovi elementi, ovvero se vale che
$\forall f \; F \models f \Rightarrow F \vdash f$ (completezza)
In generale per aggiungere nuove DF devo prendere quelle che ho e derivarne di nuove utilizzando gli assiomi di Armstrong. Questo procedimento è finito perché il numero dei simboli è finito.

Esempio 2 : supponiamo che l'insieme dei nostri attributi sia
$R = \{A, B, C, D\}$
e che l'insieme F delle DF sia:
$F = \{A \to B, BC \to D\}$
Possiamo dimostrare che AC è superchiave? Ovvero che $AC \to ABCD$?

$A \to B$	(ipotesi 1)
$AC \to BC$	(arricchimento)
$BC \to D$	(ipotesi 2)
$BC \to BCD$	(arricchimento)
$AC \to BCD$	(transitività)
$AC \to ABCD$	(arricchimento)

Abbiamo un algoritmo per automatizzare il procedimento appena visto? Tra poco sì, intanto vediamo due definizioni per avvicinarci

def. Chiusura di F : Dato un insieme F di dipendenze funzionali, la chiusura di F, denotata con F^+, è:
$F^+ = \{X \to Y \mid F \vdash X \to Y\}$

Che dimensione ha F^+? Esponenziale nel numero di attributi, infatti per ogni dipendenza posso sempre inferire tutte le sue dipendenze banali, ovvero tutti i sottoinsiemi che sono nell'ordine di 2^n.

def. Chiusura di X su F : Dato $R < T, F >$, dove T è l'insieme degli attributi di una tabella e F un'insieme di DF, e $X \subseteq T$, la chiusura di X rispetto ad F, denotata con X_F^+, (o X^+, se F è chiaro dal contesto) è $X_F^+ = \{A_i \in T \mid F \vdash X \to A_i\}$

Notiamo che la dimensione di X^+ è lineare nel numero di attributi.
Potremmo ora chiederci: come controllo se una DF appartiene a F^+?(ovvero posso ricavarla tramite gli assiomi di Armstrong a partire da F)
Unendo le definizioni appena date, possiamo ricavare un teorema:

$$F \vdash X \to Y \Leftrightarrow Y \subseteq X_F^+$$

questo teorema è molto utile perché mi permette di passare da un'insieme F e un'insieme X all'insieme X_F^+(dimensione lineare) senza passare per F^+(dimensione esponenziale).
Per determinare chi sia X_F^+ usiamo l'algoritmo di chiusura lenta presentato tramite un esempio

Esempio 3 : Sia $F = \{DB \to E, B \to C, A \to B\}$
Determinare $(AD)^+$
Considero $X^+ = AD$ scorro F e vedo se esiste una DF il cui determinante sia incluso in AD. In effetti c'è: $A \to B$. Allora posso aggiungere il determinato di $A \to B$ ad X^+
Dunque $X^+ = AD$ da cui ricavo $X^+ = ADB$
Ripeto il ragionamento e adesso vedo che il determinante di $DB \to E$ è sottoinsieme di X^+ dunque ricavo
$X^+ = ADBE$
Ripeto il ragionamento fino a quando non mi è più possibile aggiungere elementi ad X^+

Questo algoritmo generalizzato è detto *algoritmo di chiusura lenta*[5], lo presentiamo di seguito

$X^+ = X$
while (X^+ cambia)
    for ($W \to V$ in F with $W \subseteq X^+$ && $V \not\subseteq X^+$)
        $X^+ = X^+ \cup V$

Per arrivare alla definizione di *forma normale* ci manca ancora qualche pezzo.

def. Chiave candidata : Dato lo schema $R < T, F >$, diremo che $W \subseteq T$ è una chiave candidata di R se

[5]Così chiamato perché ha costo $O(n^3)$ e ne esiste una versione più veloce che noi non vedremo che ha costo $O(n^2)$

$W \to T \in F^+$ (W superchiave)
$\forall V \subseteq W, V \to T \notin F^+$ (se $V \subseteq W$, V non superchiave)

def. Attributo primo : attributo che appartiene ad almeno una chiave

Avevamo visto che in tempo polinomiale possiamo determinare se un attributo sia superchiave. In tempo polinomiale possiamo anche determinare se un attributo sia chiave? Sì! Mi basta prendere la superchiave e, rimuovendo un attributo alla volta verificare che resti superchiave.
Posso anche controllare se un certo attributo sia primo[6] in tempo polinomiale? No! Purtroppo è un problema NP-completo!

18/11/2015

Possiamo trovare un algoritmo che mi dia tutte le chiavi a partire da un insieme di attributi e DF? Sì, perché anche se il costo è teoricamente esponenziale, a livello pratico i costi si abbassano notevolmente.
Abbiamo uno schema ben preciso:
consideriamo l'insieme di tutte le DF, allora

1. Tutti gli attributi che stanno soltanto a sinistra in tutte le DF dovranno essere per forza incluse nella chiave
2. Tutti gli attributi che stanno soltanto a destra in tutte le DF dovranno essere per forza escluse dalla chiave
3. Tutti gli attributi che stanno sia a destra che a sinistra in almeno una DF non sappiamo se faranno parte o meno della chiave(vedremo poi come stabilirlo)
4. Tutti gli attributi che non compaiono in alcuna delle DF faranno per forza parte della chiave

Esempio 4 : Sia $F = \{A \to B, AB \to D, D \to E, C \to E\}$
Compiliamo una tabella di dove compaiono gli attributi

Sinistra	Destra-Sinistra	Destra
A	B	E
C	D	

notiamo che F non appare mai. Allora A, F e C faranno sicuramente parte della chiave(compaiono solo a sinistra oppure non sono presenti). L'attributo E non farà mai parte della chiave(compare solo a destra). Di B e di D non lo sappiamo, dunque dovremmo testarli.
Rappresentiamo graficamente questo fatto con la notazione

$$AFC^{BD}$$

Notiamo che avremo tanti termini all'esponente quanti sono gli attributi da testare. A questo punto allora dovremo testare al più 2^n casi, dove n

[6] Notiamo che questa richiesta è equivalente a quella di determinare tutte le chiavi

è il numero di attributi all'esponente. Studiamo i vari casi cercando di capire se abbiamo o meno una chiave.
In questo caso essendo 2 gli attributi all'esponente (ovvero B e D) abbiamo $2^2 = 4$ casi possibili:

$$\{\ \} \qquad \{B\} \qquad \{D\} \qquad \{BD\}$$

Partiamo dal primo, ovvero dall'insieme vuoto. Questo vuol dire che dobbiamo provare a vedere se AFC è già di per sé una chiave.
$AFC^+ = AFC = AFCB = AFCBD = AFCBDE$
dunque AFC è già chiave

Esempio 5 : Proviamo ora ad aggiungere la DF $E \to A$ dunque F diventa
$F = \{A \to B,\ AB \to D,\ D \to E,\ C \to E,\ E \to A\}$
La tabella è

Sinistra	Destra-Sinistra	Destra
C	B	
	D	
	A	
	E	

Dunque secondo la rappresentazione grafica ora abbiamo CF^{ABDE}
Considero
$CF^+ = CF = CFE = CFEA = CFEAB = CFEABD$
Anche stavolta è chiave

Esempio 6 : proviamo ora a togliere da F la DF $C \to E$, dunque F diventa

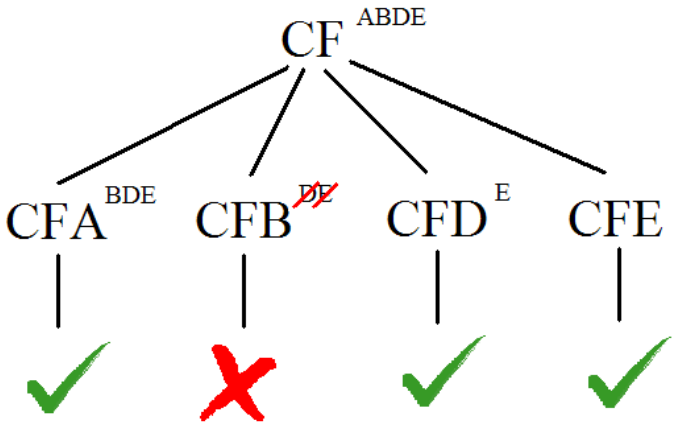


Figura 14: Schema per casi dell'esempio 6

$F = \{A \to B,\ AB \to D,\ D \to E,\ E \to A\}$
La tabella è

Sinistra	Destra-Sinistra	Destra
	B	
	D	
	A	
	E	

Dunque secondo la rappresentazione grafica ora abbiamo CF^{ABDE}
Proviamo a vedere se CF è chiave come già visto:
$CF^+ = CF$
Dunque da CF non posso ricavare gli altri attributi, dunque non è chiave.
Procediamo a "far scendere" dall'esponente gli attributi e consideriamo i seguenti 4 casi (uno per ogni attributo all'esponente)

1. CFA^{BDE}
   Vediamo se CFA è chiave: $CFA^+ = CFA = CFAB = CFABD = CFABDE$
   Dunque CFA è chiave, allora non mi serve di continuare ad esplorare l'albero in quella direzione perché so che, aggiungendo altri attributi otterrò per forza una chiave
2. CFB^{DE}
   Vediamo se CFB è chiave: $CFB^+ = CFB$
   Dunque CFB non è chiave, più avanti esploreremo il suo sottoalbero se necessario
3. CFD^{E}
   Vediamo se CFD è chiave: $CFD^+ = CFD = CFDE = \underline{CF}DE\underline{A}$
   Allora CFD è chiave. Infatti notiamo che $\underline{CF}DE\underline{A}$ include la chiave CFA che abbiamo visto poco fa, dunque sarà sicuramente chiave. Inoltre possiamo togliere la lettera D da tutti gli esponenti non risolti(ad esempio CFB^{DE}) visto che la D unita a CF darà sicuramente una chiave, dunque non è un caso che vale la pena testare.
4. CFE
   Vediamo se CFE è chiave: $CFE^+ = \underline{CF}E\underline{A}$
   Come prima otteniamo CFA che è chiave, dunque anche CFE è chiave. Inoltre possiamo togliere da tutti gli esponenti non risolti il simbolo E.

Adesso dovremmo proseguire ad esplorare il sottoalbero di CFB^{DE} ma poiché abbiamo rimosso la D al punto 3 e la E al punto 4, allora resterebbe da verificare solo il caso $CFB^{\{\}}$ che però abbiamo già controllato. Dunque la nostra ricerca termina qui, abbiamo determinato tutte le chiavi. Dunque tutti gli attributi sono primi tranne la B dalla quale non sono riuscito a ricavare una chiave

Proviamo invece a trovare una chiave qualsiasi a partire dai soli attributi

Esempio 7 : tutti gli attributi sono ABCDEF

Li elimino uno alla volta e vedo se ottengo comunque una chiave.
~~A~~BCDEF: togliendo la A resta chiave
~~AB~~CDEF: togliendo anche la B resta chiave
~~ABC~~DEF: se tolgo anche la C vedo che non ottengo più tutti gli attributi tramite le DF, quindi C deve necessariamente esserci
~~AB~~CDEF: ripristino la C che era indispensabile e provo ad eliminare la D. Ho ancora una chiave
~~ABC~~~~DE~~F: elimino la E ma non ottengo una chiave, quindi era indispensabile
~~AB~~CDE~~F~~: elimino la F ma non ottengo una chiave, quindi anche la F era indispensabile.
Dunque CEF è una chiave. Notiamo che la chiave che ottengo dipende dall'ordine con cui ho scorso gli attributi. Ricordiamo che con questo metodo che ha costo lineare ho certamente trovato una chiave, ma non per questo le ho trovate tutte

5.3 Coperture

def. Copertura : Due insiemi di DF, F e G, sullo schema R sono equivalenti $F \equiv G \Leftrightarrow F^+ = G^+$
Se $F \equiv G$, allora F è una copertura di G (e G una copertura di F).

def. Attributo estraneo : Data una $X \to Y \in F$, si dice che X contiene un attributo estraneo $A_i \Leftrightarrow (X - \{A_i\}) \to Y \in F^+$, cioè
$F \vdash (X - \{A_i\}) \to Y$

def. Dipendenza ridondante : $X \to Y$ è una dipendenza ridondante $\Leftrightarrow$ $(F - \{X \to Y\})^+ = F^+$, cioè
$F - \{X \to Y\} \vdash X \to Y$

Tramite queste prime tre definizioni ricaviamo un'altra definizione:

def. Copertura canonica : F è detta una copertura canonica se e solo se

1. il determinato di ogni DF in F è un attributo
2. non esistono attributi estranei
3. nessuna dipendenza in F è ridondante

Esiste anche un teorema che mi garantisce che dato un'insieme di DF ne esiste certamente una forma canonica.
Notiamo che questa è una definizione in un certo senso operativa poiché mi dice che, se vale la proprietà 1 allora devo prima rimuovere gli attributi estranei per poi togliere le dipendenze ridondanti. Rimuovere prima le dipendenze ridondanti è sconsigliato perché può capitare che tale rimozione introduca nuovi attributi estranei che devono essere eliminati. A quel punto dovrei nuovamente eliminare eventuali dipendenze ridondanti.

def. Decomposizione : Dato uno schema R(T)
$\rho = \{R_1(T1), \ldots, R_k(Tk)\}$
è una decomposizione di R $\Leftrightarrow \cup T_i = T$

Esempio 1 : supponendo di avere una tabella Studenti-Esami(Matr,Nome,Materia) una decomposizione possibile è: {Studenti(Matr, Nome), Esami(Matr, Materia)}
Notiamo che nel momento in cui decomponiamo una tabella, creiamo una ridondanza, infatti il campo matricola è presente in entrambe le tabelle. Ma questo fatto è necessario perché l'attributo ripetuto è quello che stabilisce il legame fra le righe delle due tabelle

Due proprietà che ci interessa che una decomposizione mantenga sono:

- Conservazione dei dati
- Conservazione delle dipendenze

def. Conservazione dei dati : sia $\rho = \{R_1(T_1), \ldots, R_k(T_k)\}$ è una decomposizione di R(T) che preserva i dati se e solo se per ogni istanza valida r di R:
$r = (\pi_{T1} r) \bowtie (\pi_{T2} r) \bowtie \ldots \bowtie (\pi_{TK} r)$

Notiamo che per ora ho una definizione per conservare i dati che è corretta ma che non è utilizzabile perché dovrei verificarla per ogni istanza valida della tabella, ovvero per infinite righe.
Inoltre, in questo caso, "conservare i dati" non vuol dire solo evitare che vengano perduti, ma anche evitare che se ne aggiungano di nuovi.

Esempio 2 : supponiamo di avere la seguente tabella:

A	B	C
a1	b	c1
a2	b	c2

Allora la seguente decomposizione non conserva i dati:

$\pi_{T1} =$

A	B
a1	b
a2	b

$\pi_{T2} =$

B	C
b	c1
b	c2

Infatti provando ad eseguire $\pi_{T1} \bowtie \pi_{T2}$ ottengo delle righe che originariamente non erano presenti nello schema.

Una particolare classe delle decomposizioni sono le *decomposizioni binarie*. Infatti se effettuiamo una scomposizione in due parti e notiamo che gli attributi in comune alle due tabelle sono chiave per almeno una delle due tabelle, allora la conservazione dei dati è garantita.

def. Conservazione dei dati (caso binario): Sia R<T, F> uno schema di relazione, la decomposizione
$\rho = \{R_1(T_1), R_2(T_2)\}$
preserva i dati se e solo se
$T_1 \cap T_2 \rightarrow T_1 \in F^+$ oppure $T_1 \cap T_2 \rightarrow T_2 \in F^+$

Esempio 3 : la scomposizione vista nell'esempio 1 conserva i dati perché la matricola anche se non è chiave nella tabella degli esami(ogni studente può aver dato più di un esame) è certamente chiave nella tabella degli studenti

Veniamo adesso alla conservazione delle dipendenze:
quando viene fatta una decomposizione(o proiezione) cosa succede alle DF? Supponiamo di avere una tabella T avente F insieme di DF e di aver effettuato una scomposizione binaria
$U(A_{U_1}, \ldots, A_{U_M})$
$V(A_{V_1}, \ldots, A_{V_N})$
Allora ognuna di queste tabelle avrà un suo insieme di DF che sarà un sottoinsieme di F.
$DF_U = (A_{U_1}, \ldots, A_{U_M})^+$
$DF_V = (A_{V_1}, \ldots, A_{V_M})^+$
Diremo allora che una proiezione preserva le DF se dopo aver scomposto la tabella originaria abbiamo che l'unione delle DF delle due sottotabelle è uguale all'insieme delle DF di partenza, ovvero che
$DF_U \cup DF_V$

23/11/2015

Possiamo adesso generalizzare quanto detto sulle decomposizioni binarie

teo. :Sia $\rho = \{R_i < T_i, F_i >\}$ una decomposizione di R<T, F> che preservi le dipendenze e tale che un T_j sia una superchiave per R. Allora ρ preserva i dati.

con questo importante teorema intendiamo dire che se nelle tabelle della decomposizione abbiamo un attributo che è chiave per almeno una delle tabelle allora la conservazione dei dati è garantita.

Esempio 4 : consideriamo la tabella per gestire un elenco telefonico
Telefoni(Prefisso, Numero, Località, Abbonato, Via)
Le cui dipendenze funzionali sono:
$\{PN \to LAV, L \to P\}$
Tramite il prefisso e il numero ricaviamo tutti gli altri dati di un cliente, inoltre ad ogni località è associato univocamente un prefisso.
Consideriamo una scomposizione della tabella:
$\rho = \{\text{Tel}< N, L, A, V, F1 >, \text{Pref}< L, P, F2 >\}$
con $F1 = \{LN \to AV\}$ e $F2 = \{L \to P\}$
Questa decomposizione preserva i dati ma non le dipendenze infatti dalle due tabelle non riesco più a ricavare la DF $PN \to LAV$
Infatti potrei avere due persone nella tabella Tel aventi lo stesso numero N, ma distinguibili a causa della località L. Non vengono violate dipendenze ma andando a fare la giunzione fra Tel e Pref ottengo una tabella con persone diverse aventi lo stesso numero.

5.4 Forme Normali

Nel corso degli anni sono state date diverse definizioni di forme normali, noi ne vedremo un paio. La *forma normale* è una certificazione riguardante la qualità di una base di dati, ovvero garantisce che una base di dati rispetti certe proprietà.

5.4.1 Forma Normale di Boyce Codd (FNBC)

L'idea che sta alla base della forma normale di Boyce Codd è di dividere in tre gruppi le dipendenze funzionali.

1. Quelle banali, che non consideriamo ma che esistono
2. Quelle che hanno a sinistra una superchiave
3. Tutte le altre, ovvero quelle che rappresentano ridondanza

Nelle FNBC le dipendenze devono essere solo di tipo 1 o di tipo 2. Prendiamo l'esempio 4 visto poco fa. In quel caso avevamo due dipendenze:
$PN \to LAV$, che è di tipo 2 in quanto tramite P N determino tutti gli altri attributi della tabella.
$L \to P$ invece è di tipo 3 in quanto non ha alla sua sinistra una chiave, infatti serve per evitare di mettere accanto ad ogni elemento della tabella Telefoni il prefisso
Vediamo ora la definizione

def. FNBC : prendiamo una tabella R con attributi T e l'insieme F delle sue DF. R<T, F> è in BCNF $\Leftrightarrow \forall\ X \to A \in F^+$, con $A \not\subseteq X$ (non banale), X è una superchiave.

che intuitivamente dice quanto avevamo detto sopra. Se una DF non è banale, allora deve essere superchiave. Quando viene richiesta la definizione di FNBC attenzione a non scordarsi le DF banali.
Questa definizione non è però utilizzabile, quindi ci avvaliamo di un teorema che ci semplifica la vita

teo. : R<T, F> è in BCNF $\Leftrightarrow \forall\ X \to A \in$ F non banale, X è una superchiave.

la definizione e il teorema sembrano simili ma la cruciale differenza è che la definizione va ad esplorare F^+ mentre il teorema soltanto F che è molto più contenuto in dimensione.

Esempio 1 : consideriamo la tabella Docenti(<u>CodiceFiscale</u>, Nome, Dipartimento, Indirizzo).
Questa non è in FNBC perché il Dipartimento da una DF sull'indirizzo, ovvero $D \to I$ è una DF di ridondanza e non di superchiave

Esempio 2 : Impiegati(<u>Codice</u>, Qualifica, <u>NomeFiglio</u>) Anche questa non è in FNBC perché l'unica DF è $C \to Q$ perché C non è chiave mentre CN invece è chiave.

Come trasformare una tabella per farla diventare in FNBC? Intuitivamente considero la colonna che mi da ridondanza e la metto in un'altra tabella mantenendo legata la nuova tabella alla vecchia tramite un attributo chiave.
Questo algoritmo è detto *algoritmo di analisi*

Esempio 3 : supponiamo di avere la solita tabella
StudentiEdEsami(Matricola, Nome, Provincia,AnnoNascita, Materia,Voto)
L'attributo chiave in questo caso è la Matricola. Allora creo una nuova tabella T_1 contenente tutti gli attributi che ottengo a partire dalla Matricola, ovvero si vengono a formare due tabelle
$T_1(M^+)$
$T_2(A_i - M^+, M)$
ovvero ho creato la tabella con tutti gli attributi definibili tramite le chiave e una seconda tabella contenente tutti i restanti attributi a cui ho aggiunto la chiave per mantenere il legame tra le due tabelle.
Adesso è possibile iterare il ragionamento su tutti gli attributi restanti. Questo ragionamento ha un costo esponenziale

Esempio 4 : consideriamo la tabella
R(ABCDE)
avente il seguente insieme di DF
$\{AB \to C, B \to E, D \to C\}$
Prendiamo la prima DF e a partire dal determinante studiamo la sua chiusura.
$AB^+ = ABCE$
Risulta evidente che questa tabella non è in FNBC visto che AB determinante di una delle DF non è superchiave. Dividiamo la tabella R in due sottotabelle
R1(AB^+) = R1(ABCE)
R2($A_i - AB^+$, AB) = R2(ABD)
Considero allora tutti i sottoinsiemi degli attributi e vedo quali altri attributi ricavo. Comincio prendendoli uno a uno:
$A^+ \equiv \backslash \qquad B^+ \equiv E \qquad C^+ \equiv \backslash \qquad E^+ \equiv \backslash$
Proseguo prendendoli due a due:
$AB^+ \equiv CE \qquad AC^+ \equiv \backslash \qquad AE^+ \equiv \backslash \qquad BC^+ \equiv E \qquad BE^+ \equiv \backslash$
$CE^+ \equiv \backslash$
Proseguo prendendoli tre a tre:
$ABC^+ \equiv E \qquad ABE^+ \equiv \backslash \qquad ACE^+ \equiv \backslash \qquad BCE^+ \equiv \backslash$
Nel complesso l'insieme di DF che ho ricvato su R1 è $\{B \to E, AB \to C\}$

Per R2 invece ho:
$A^+ \equiv \backslash \qquad B^+ \equiv E \qquad D^+ \equiv C$
$AB^+ \equiv \backslash \qquad AD^+ \equiv \backslash$
Potremmo proseguire ma poiché mi interessa trovare o A o B o D a destra delle DF ma noto che non esistono dipendenze cha abbiano o A o B o D a destra dunque per R2 l'insieme di dipendenze utili che ho trovato è {}

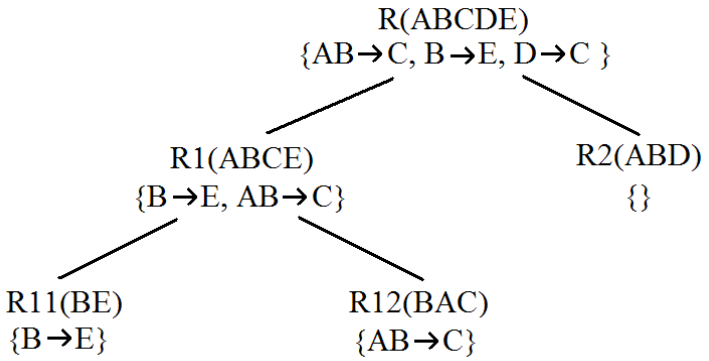


Figura 15: Schema per casi dell'esempio 4

Itero il ragionamento e considero R1(AB^+) = R1(ABCE) con DF $\{B \to E, AB \to C\}$
Considero la prima delle DF da cui ottengo
$B^+ = E$
Dunque divido la tabella in due parti
R11(BE)
R12(BAC)
Applichiamo adesso la costruzione delle dipendenze approssimata. Ovvero noto che dagli attributi di R11 ricavo sicuramente la prima delle due DF che avevo per R1, ovvero $B \to E$ e che dagli attributi di R12 ricavo l'altra dipendenza che avevo per R1 ovvero $AB \to C$.
Un teorema mi dice che avendo esaurito l'insieme di tutte le DF di R1, allora non devo più iterare. Dunque ho finito perché per ogni tabella ho trovato le DF che sono chiave per la tabella. Il tutto è stato riassunto nella figura 15. Notiamo infine, che le DF "scendono" lungo lo schema. In particolare o vanno a destra, o vanno a sinistra, o spariscono. Il punto è che le DF non vanno sia a destra che a sinistra.

5.4.2 Terza Forma Normale (3FN)

Questa è la seconda e ultima forma normale che vedremo

def. Terza Forma Normale : sia T una tabella e F il suo insieme di DF, allora R<T, F> è in 3FN se $\forall\ X \to A \in F^+$, con A $\notin$ X, X è una superchiave o A è primo.

notiamo che la definizione di 3FN è più permissiva in quanto abbiamo "allentato" il vincolo andando ad includere anche gli attributi primi[7]. Come nella FNBC, anche qui abbiamo un teorema che sposta il problema da F^+ ad F

[7] Questo implica che se uno schema è in FNBC allora è anche in 3FN, ma ovviamente non implica il viceversa

teo. : R<T, F> è in 3FN se per ogni $X \to A \in F$ non banale, allora X è una superchiave oppure A è primo.

Quindi anche in questo caso mi è sufficiente trovare una copertura per cui valga il teorema per stabilire che uno schema sia in 3FN.

Esempio 1 : Lo schema
Docenti(CodiceFiscale, Nome, Dipartimento, Indirizzo)
non è in 3FN(dunque neanche in FNBC) perché nella DF
$D \to I$
D non è superchiave e A non è attributo primo

Esempio 2 : Lo schema
Telefoni(Prefisso, Numero, Località, Abbonato, Via)
è in 3FN ma non è in FNBC in quanto la dipendenza $L \to P$ per Boyce Codd era portatrice di ridondanza, però P è attributo primo.

Esempio 3 : in realtà notiamo che la 3FN a volte è anche troppo permissiva.
Prendiamo lo schema
Esami(Matricola, Telefono, Materia, Voto)
Matricola Materia $\to$ Voto
Matricola $\to$ Telefono
Telefono $\to$ Matricola
Chiavi: Matricola Materia, Telefono Materia

notiamo che questo schema è già in 3FN quando in realtà andrebbe scomposto

Come in FNBC esiste un algoritmo, detto *algoritmo di sintesi* che mi permette di portare gli schemi in 3FN mantenendo i dati e le dipendenze. Talvolta con questo algoritmo otteniamo uno schema che è anche in FNBC, ma non sempre. I passi da seguire nell'algoritmo di sintesi sono i seguenti

1. Si partiziona F in gruppi tali che ogni gruppo ha lo stesso determinante.
2. Si definisce uno schema di relazione per ogni gruppo, con attributi gli attributi che appaiono nelle DF del gruppo, e chiavi i determinanti.
3. Si eliminano schemi contenuti in altri.
4. Se la decomposizione non contiene uno schema i cui attributi sono una superchiave di R, si aggiunge lo schema con attributi W, con W una chiave di R.

Vediamo come opera tramite un esempio

Esempio 3 : sia R(ABCDE) e F=$\{AB \to C, AB \to E, C \to A, AC \to D\}$
Raggruppiamo sul determinante e scriviamo accanto la tabella con tutti gli attributi raggruppati:

$AB \to CE$ $\quad$ $R1(ABCE)$
$C \to A$ $\quad$ $R2(AC)$
$AC \to D$ $\quad$ $R3(ACD)$

Adesso possiamo eliminare le tabelle incluse in altre tabelle
$R1(ABCE)$
~~$R2(AC)$~~
$R3(ACD)$

Adesso è sufficiente chiedermi: tra tutte le tabelle rimaste, ne esiste una i cui attributi sono superchiave? Sì, ABCE è superchiave dunque ho trovato una scomposizione in tabelle che è in 3FN.

Esempio 4 : supponiamo di introdurre un nuovo attributo F che non compare in alcuna DF nell'esempio 3. Allora è impossibile che gli attributi delle tabelle R1 e R3 lo ricavino visto che non è in alcuna DF. Dunque l'insieme degli attributi delle due tabelle non sono chiave. In tal caso avrei dovuto trovare una chiave con l'algoritmo che avevamo già visto. Ad esempio avremmo potuto ottenere
~~$ABCDEF$~~

25/11/2015

Esempio 4 : Consideriamo $R(ABCDEF)$ con $F = \{ABC \to E,\ C \to F,\ B \to D,\ C \to D,\ BF \to A\}$
Dividiamo sul determinante e scriviamo accanto la tabella con tutti gli attributi raggruppati:

$ABC \to E$ $\quad$ $R1(ABCE)$
$C \to FD$ $\quad$ $R2(CDF)$
$B \to D$ $\quad$ $R3(BD)$
$BF \to A$ $\quad$ $R4(BFA)$

Se consideriamo gli attributi della tabella R1 notiamo che formano una superchiave. Dunque questa scomposizione è in 3FN.
Notiamo che non è in FNBC, infatti il determinante della DF $E \to B$, contenuta in F=$\{ABC \to E,\ E \to B\}$ di R1(ABCE), non è superchiave.

6 Algoritmi e strutture dati per DBMS

Un DBMS è solitamente implementato a due livelli diversi

1. Macchina Fisica: implementa le transazioni e le strutture di accesso. In particolare il gestore di affidabilità garantisce l'atomicità delle operazio-

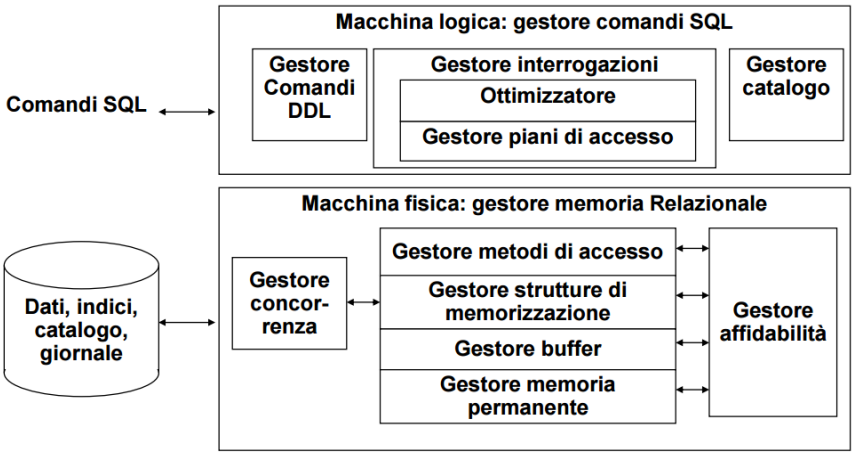


Figura 16: Struttura del DBMS

ni come l'avevamo definita nella prima parte del corso(ad esempio se un'operazione fallisce in una delle sue fasi se ne elimina per intero l'effetto)

2. Macchina Logica: prende quanto definito dalla macchina fisica e vi si interfaccia tramite SQL.

I DBMS operano diversamente anche in base al supporto di memorizzazione che usano, noi ci occuperemo di DBMS su dischi meccanici. Poiché il tempo di accesso al disco è molto maggiore del tempo che impieghiamo per elaborare le informazioni, quando dovremo valutare l'efficienza faremo riferimento al numero di pagine lette/scritte su disco.
Inoltre non potremo fare assunzioni sulla sequenzialità delle informazioni, ovvero tutte le informazioni che ci servono potrebbero essere sparpagliate nel disco in diverse posizioni.
A livello logico le informazioni sono memorizzate nei record, caratterizzati da una coppia (PID pagina, offset), in tal modo la lettura e la modifica dei record è semplice. La memorizzazione dei record può essere fatta in diversi modi

1. Organizzazione seriale: memorizzo i dati nell'ordine in cui li ricevo, dunque senza ordinarli.
   L'operazione di inserimento è efficiente, pari a O(1) ma le operazioni di ricerca e cancellazione sono più costose O(n)

2. Organizzazione sequenziale: c'è un ordinamento su uno o più attributi.
   L'inserimento è più costoso, O(n), ma la ricerca è nettamente più veloce, O(log(n)). Inoltre se viene fatta una query con richieste di ORDER BY, l'ordinamento è già pronto, altrimenti mi costerebbe O(n·log(n)). Una caratteristica importante nell'organizzazione sequenziale è che le pagine

in memoria devono essere riempite intorno al 80% per consentire l'inserimento di nuovi elementi

30/11/2015

3. Organizzazione con tabelle hash: scelgo quanto voglio che una pagina sia piena tramite il fattore di caricamento. Chiaramente quando uso una tabella hash devo anche pensare alla gestione delle collisioni e all'overflow. Le collisioni vengono gestite come di consueto mentre per l'overflow si aggiunge una pagina dedicata.
   Effettuare ricerche costa O(2) e l'inserimento O(1).
   Le tabelle hash sono un ottimo metodo statico di gestione. Ovvero funzionano bene se la dimensione del file resta costante o quasi.
   Una soluzione possibile è di impedire l'accesso alla tabella quando vengono fatte diverse aggiunte e/o cancellazioni.
   Il più grande difetto di questa organizzazione è che non posso fare ricerche su intervalli, ad esempio chiedere di stampare tutte le persone con cognome tra la "d" e la "i")

4. Organizzazione con B^+ alberi: sono una generalizzazione degli alberi binari, ogni nodo e foglia può contenere fino a quattro elementi. L'albero è perfettamente bilanciato e gli elementi sono disposti nelle foglie in ordine crescente. Dunque la ricerca costa $O(\log_4(n))$, ma visto che solitamente l'albero non è mai più alto di 3 o 4 livelli diremo che la ricerca costa O(3). L'inserimento costa O(4).
   Il B^+ albero è invece un ottimo metodo dinamico, ovvero capace di gestire bene numerosi inserimenti e cancellazioni.

Esempio 1 : consideriamo il B^+ albero in figura 17. Le foglie contengono valori puri mentre i nodi interni contengono il massimo contenuto nel sottoalbero da loro puntato.
Inoltre ogni elemento dell'albero possiede uno o più riferimenti agli elementi dell'albero dello stesso livello(segnati in arancione)

A volte i diversi approcci possono essere mescolati

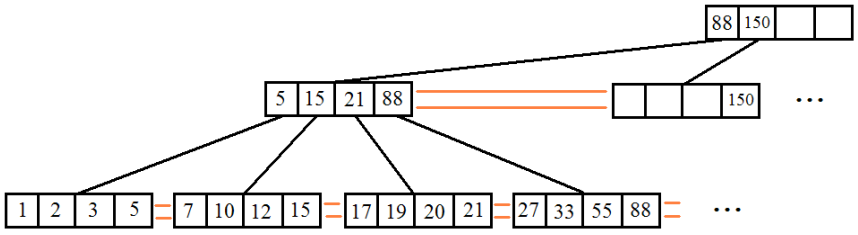


Figura 17: B^+ albero dell'esempio 1

Esempio 2 : supponiamo di avere una tabella degli studenti organizzata serialmente, ovvero senza un ordine preciso. Creo un'altra tabella contenente i campi RID(record ID) e la matricola che è organizzata tramite tabella hash.
A questo punto per cercare le informazioni relative ad uno studente mi è sufficiente cercare nella tabella hash il numero di matricola, in tal modo ottengo in O(1) il RID associato che posso cercare in O(1) ottenendo tutte le altre informazioni. Dunque costa in tutto O(2)

Nell'esempio 2 abbiamo usato una tabella ausiliaria contenente solamente il RID e il numero di matricola. Tale tabella viene detta *indice* perché tramite essa trovo il RID dello studente da cui ottengo tutte le informazioni.
In generale è una tecnica diffusa stabilire quali siano gli attributi più importanti per una certa tabella e farne un indice. Solitamente esiste un indice per ogni attributo che è chiave primaria e uno per ogni attributo che è chiave esterna. Solitamente gli indici sono organizzati in B^+ alberi.

6.1 Ricostruzione degli operatori relazionali

Nella prima parte del corso avevamo visto diversi operatori importanti. Proiezione, restrizione, raggruppamento e giunzione. Cerchiamo di ricostruirli:

- Proiezione: si legge dalla tabella R e si scrive una tabella T che contiene solo gli attributi della SELECT. Si ordina T su tutti gli attributi e infine si eliminano i duplicati

- Restrizione: può essere fatta usando un indice a costo O($I + D$) dove I è il costo di accesso all'indice e D è il costo di accesso ai dati.
  In alternativa può essere fatta senza indice pagando O(#pagine(R)) dove R è la tabella su cui fare restrizione

Di norma avere l'organizzazione con indice conviene ma non è sempre vero

Esempio 1 : supponiamo di avere una tabella in cui il 90% degli studenti è nato nella provincia di Pisa. Supponiamo anche di avere l'indice di tale tabella. Se adesso mi venisse chiesta una ricerca di tutti gli studenti nati nella provincia di Pisa, la ricerca tramite indice mi obbliga ad accedere più e più volte alle stesse pagine contenenti diversi studenti nati a Pisa, cosa che non sarebbe successa con una ricerca seriale

Proseguiamo costruendo gli ultimi due operatori

- Raggruppamento: Si ordinano i dati sugli attributi del GROUP BY, poi si visitano i dati e si calcolano le funzioni di aggregazione per ogni gruppo.

- Giunzione: viene fatta eseguendo un algoritmo preciso, detto *nested loops*, che effettua il prodotto cartesiano. L'algoritmo è fatto come segue:

  foreach record r in R do

```
foreach record s in S do
    if (r_i = s_j) then aggiungi <r, s> al risultato
```

dove R e S sono le tabelle su cui fare giunzione. Notiamo che senza dubbio esploriamo tutto il prodotto cartesiano in quanto per ogni elemento di R vado a controllarlo con ogni elemento di S. Dunque questa operazione ha costo quadratico. L'unico vantaggio che ho scrivendo l'algoritmo nested loops è che aggiungo alla tabella risultato solo le cose che mi interessano quindi in termini di spazio non sarà necessariamente quadratico

1/12/2015

Un altro metodo per implementare la giunzione consiste nell'utilizzare un altro algoritmo, detto *index nested loop* che può essere utilizzato soltanto quando la condizione della giunzione è un'uguaglianza semplice.
L'algoritmo è

```
foreach r in R do
    foreach s in get-throughindex(ISj,=r.i)
        aggiungi <r,s> al risultato
```

Intuitivamente l'algoritmo va a prendere tutti gli elementi r della tabella esterna R che rispettano la condizione di uguaglianza e li inserisce nel risultato. Essendo la tabella interna S indicizzata, l'accesso ad essa è circa costante, dunque il costo diventa:
O(#r · k)
ovvero il numero di righe r che rispettano la condizione moltiplicato il tempo di accesso alla tabella interna, ovvero una costante k
Notiamo inoltre che il costo di Index Nested Loop dipende dalla dimensione della tabella esterna R e non dalla dimensione della tabella interna indicizzata.
Un altro modo consiste nell'utilizzare il SortMerge che, date due tabelle ordinate, consiste nel tenere un puntatore su ogni tabella e far scorrere i puntatori.
Il suo algoritmo è il seguente

```
r = first(R);
s = first(S);
while r in R and s in S do
    if (r.i = s.j)
        avanza r ed s fino a che r.i ed s.j non cambiano entrambe,
        aggiungendo ciascun <r,s> al risultato
    else
        if (r.i < s.j) avanza r dentro R
        else if r.i > s.j avanza s dentro S
```

Il costo di SortMerge è $2n \cdot log(n)$

6.2 Trasformare alberi logici in alberi fisici

Possiamo ora introdurre gli alberi fisici, simili sotto certi aspetti agli alberi logici. In comune con questi ultimi hanno che rappresentano una query. La differenza principale è che negli alberi fisici i nodi possono anche essere dei veri e propri algoritmi, come l'Index Nested Loop.
Vediamo gli elementi che possiamo inserire nei nostri alberi

- TableScan(R): recupera la tabella R dalla memoria
- IndexScan(R, IdX): scansiona la tabella R in base ad un attributo IdX
- SortScan(R, $\{A_i\}$): semanticamente identico a IndexScan

I primi tre operatori appena visti vengono utilizzati per le foglie del nostro albero. Infatti è tramite essi che recuperiamo i dati che verranno successivamente modificati. Solitamente viene usato TableScan, IndexScan e SortScan sono meno utilizzati.

- Project(O, A_i): effettua la proiezione della tabella O sull'attributo i
- Distinct(O): elimina i duplicati dei record. Notiamo che la Distinct esige di ricevere i record già ordinati. Dunque gli elementi su cui effettuare la distinct andranno prima ordinati
- Filter(O, ψ): esegue la restrizione sulla tabella O ponendo la condizione ψ
- IndexFilter(R, IdX, ψ): esegue la restrizione con condizione ψ utilizzando l'indice IdX dei record di R. Poiché opera direttamente sui dati è un altro operatore che deve essere inserito tra le foglie

Esempio 1 : Supponiamo di avere la seguente query SQL

```
SELECT nome
FROM studenti S
WHERE provincia=Pisa
```

Ricaviamo abbastanza facilmente l'albero logico e da esso ricaviamo l'albero fisico, che è mostrato in figura 18

Vediamo altri operatori utilizzabili

- Sort(O, $\{A_i\}$): serve per ordinare i record di O sull'attributo A_i. Viene usata prima di tutte quelle operazioni che richiedono che i dati siano già ordinati(come la distinct e la group by)
- GroupBy(O, A_i, F_i): la sintassi è la stessa della GroupBy logica, ovvero separa i record della tabella O in base all'attributo A_i e calcola su ogni sottogruppo le funzioni F_i. La semantica è leggermente diversa in quanto questa GroupBy richiede che i dati siano già ordinati sull'attributo A_i su cui deve raggruppare, dunque richiede un sort prima di essere chiamata

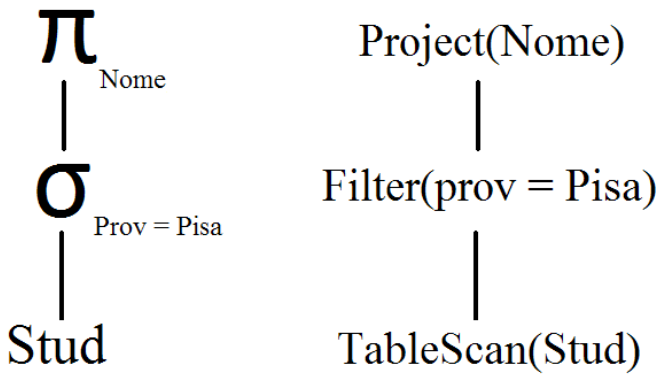


Figura 18: Albero logico(a sinistra) e albero fisico(a destra) dell'esempio 1

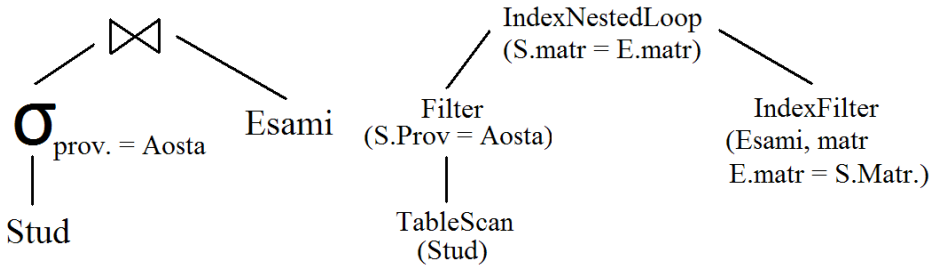


Figura 19: Albero logico(a sinistra) e albero fisico(a destra) dell'esempio 2

- Operatori di giunzione: servono per ricreare la giunzione e funzionano come già spiegato
  - NestedLoop
  - PageNestedLoop
  - IndexNestedLoop
  - SortMerge

Esempio 2 : supponiamo di avere l'albero logico come mostrato in figura 19, possiamo convertirlo nell'albero fisico sempre mostrato in figura 19 Notiamo che il sottoalbero a sinistra può avere una qualunque forma ma che il sottoalbero a destra deve contenere una IndexFilter perché abbiamo una IndexNestedLoop subito dopo. Infatti la condizione di uguaglianza presente nella IndexFilter deve essere la stessa presente nella IndexNested-Loop Attenzione a non mettere sia a destra che a sinistra la IndexFilter, è sbagliato perché non ha senso mettercelo

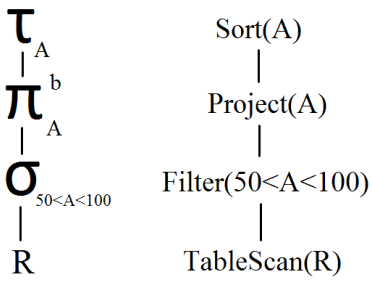


Figura 20: Albero logico(a sinistra) e albero fisico(a destra) dell'esempio 3

Esempio 3 : supponiamo di avere la seguente query SQL:

SELECT A
FROM R WHERE A BETWEEN 50 AND 100
ORDERBY A

Sia l'albero logico che quello fisico sono mostrati in figura 20. Se avessi avuto SELECT DISTINCT A a livello logico basta sostituire π^b_A con π_A. A livello fisico invece avrei dovuto aggiungere un'operazione DISTINCT sopra il Sort(A)

Esempio 4 : supponiamo di avere la seguente query SQL:

SELECT DISTINCT R.B
FROM R, S
WHERE R.B=S.B
ORDERBY R.B

supponiamo inoltre che R sia già ordinata mentre S no. Chiaramente devo fare un Sort(S.B) per ordinare anche S. Sia l'albero logico che quello fisico sono mostrati in figura 21.
Se invece avessi avuto un indice su S.B allora l'albero fisico sarebbe cambiato radicalmente perché l'indice mi consente di utilizzare Index Nested Loop. L'albero fisico è mostrato in figura 22

Notiamo che, in presenza di giunzioni in cui almeno una tabella presenta un indice l'albero a sinistra può essere complicato quanto ci pare mentre quello a destra deve contenere la IndexFilter. La struttura canonica di un albero fisico contenente più di una giunzione è quella mostrata in figura 23. Infatti adottando una struttura di quel tipo le tabelle $I_1 \ldots I_n$ possono essere tabelle con indice che mi consentono di effettuare la giunzione in modo ottimizzato

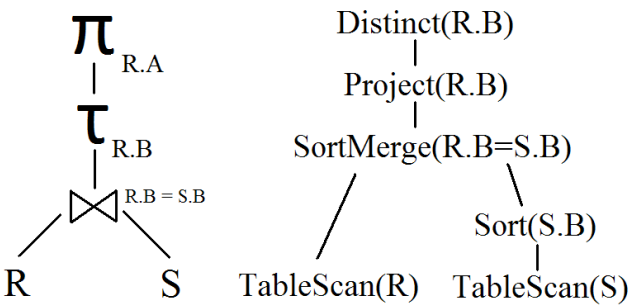


Figura 21: Albero logico(a sinistra) e albero fisico(a destra) dell'esempio 4

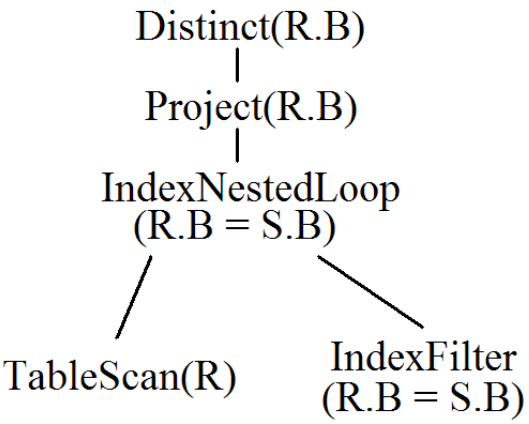


Figura 22: Albero fisico con Index Nested Loop

6.3 Transazioni

2/12/2015

def. Transazione : una transazione è un programma sequenziale costituito da operazioni che il sistema deve eseguire garantendo atomicità, serializzabilità e persistenza

In modo più specifico

- Atomicità: se la transazione fallisce a metà è annullata per intero, se ne cancellano gli effetti come se non fosse mai avvenuta
- Serializzabilità: l'esecuzione concorrente di transazioni è equivalente ad un'esecuzione sequenziale
- Persistenza: garantisce la preservazione dei dati in caso di malfunzionamenti

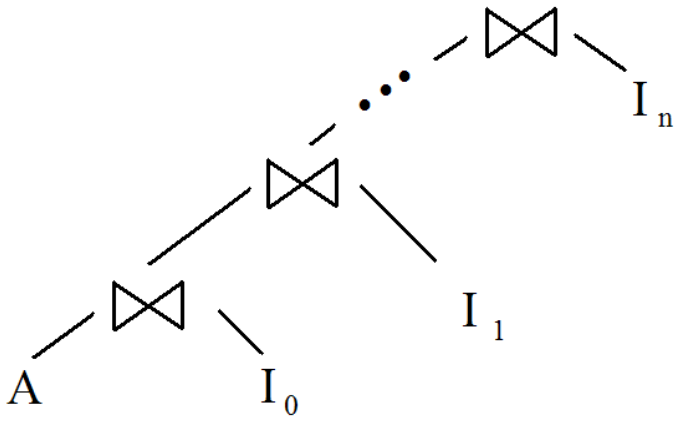


Figura 23: Albero fisico con più di una giunzione

I malfunzionamenti sono di diverso tipo:

1. Fallimento di transazione: non comporta perdita di dati né in memoria temporanea né in memoria persistente. Un esempio di fallimento di transazione potrebbe essere il tentativo di un utente di ritirare più soldi di quanti ne possiede
2. Fallimento di sistema: comporta la perdita di dati in memoria temporanea ma non in memoria permanente. Un'interruzione di corrente elettrica è un esempio di fallimento di sistema.
3. Disastro: comporta la perdita di dati in memoria permanente. Un esempio è un danno fisico a un disco

Uno strumento di supporto alle basi di dati è il log, un file seriale a cui scrive sempre in coda. Il log è amministrato dal gestore affidabilità e ad ogni modifica del log è associato un numero.
Scrivere la modifica sul log è l'ultimo atto di una transazione, ciò che la conclude.
Per essere sicuri che le transazioni siano andate a buon fine si utilizzano dei checkpoint. Il modo più semplice per fare un checkpoint consiste nel sospendere momentaneamente le transazioni e completare quelle in sospeso.
Nel caso si verifichi un fallimento di sistema dopo il checkpoint basta eseguire nuovamente tutte le transazioni successive all'ultimo checkpoint che avevo impostato. Anche se avessi un nuovo fallimento durante un ripristino da un fallimento di sistema potrei tentare di eseguire il ripristino più volte poiché disfare e rifare transazioni è un'operazione idempotente.
Le modifiche alla base di dati vengono fatte in due tempi: prima modifico il log, poi vado a modificare la pagina. Se facessi il contrario potrebbero verificarsi delle situazioni di inconsistenza(questa politica è detta Log Ahead Rule)