
Basi di dati

Introduzione

Capitolo 0

Indice degli argomenti

1) <i>Definizioni introduttive</i>	1
1.1) Sistema informativo	
1.2) Sistema organizzativo	
1.3) Sistema informatico	
1.4) Gestione delle informazioni	
1.5) Informazione	
1.6) Base di dati	
2) <i>Archivio o Database?</i>	2
2.1) Il problema	
2.2) La soluzione	
2.3) La creazione di sottosistemi	
2.4) La convenienza dei costi	
3) <i>Database Management System (DBMS)</i>	3
3.1) Il ruolo del DBMS	
3.2) Le componenti del DBMS	
4) <i>L'architettura standard: schema logico e schema interno</i>	4
5) <i>Modelli logici</i>	5
5.1) Il modello dei dati	
5.2) Schema esemplificativo di un DBMS	
6) <i>Tabella comparativa: Archivi vs DBMS</i>	6
7) <i>Schema metodologico di sviluppo di un sistema informatico</i>	7
7.1) Lo schema concettuale	
7.2) Il modello concettuale	
8) <i>Panoramica sui livelli logici</i>	8
8.1) L'evoluzione dei DBMS	

1) Definizioni introduttive

1.1) Sistema informativo

Dicesi sistema informativo la componente di una organizzazione che si occupa della gestione delle informazioni di interesse. Il sistema informativo ha un importanza notevole poiché ci fornisce il contesto in cui stiamo lavorando.

Ogni organizzazione deve avere un sistema informativo, il quale è necessariamente di supporto ad altri sistemi.

1.2) Sistema organizzativo

Dicesi sistema organizzativo l'insieme di **risorse** e di **regole** per lo svolgimento coordinato delle attività per compiere scopi comuni a tutta l'organizzazione. Il sistema organizzativo lavora ed opera su differenti campi, possiamo pensare che si occupi di soldi, di documenti, ed anche di informazioni.

Per questo il sistema informativo è incluso nel sistema organizzativo!

1.3) Sistema informatico

Una parte del sistema informativo può essere gestita in maniera più tecnica, automatizzata. Il sistema che si occupa di questa automatizzazione è detto sistema informatico.

1.4) Gestione delle informazioni

La gestione delle informazioni consta di varie fasi:

- Raccolta
- Archiviazione
- Elaborazione
- Distribuzione

1.5) Informazione

Nei sistemi informativi, le informazioni sono *interpretazioni e correlazioni di dati*. Tali dati sono concetti atomici, come un intero, una stringa, un relativo. Un dato di per sé non ha senso ma grazie al suo **metadato** (ovvero la sua "spiegazione") assume un significato preciso. Se ad esempio diciamo "5" è naturale chiedersi a cosa si riferisca. Può essere un'età, una distanza, una data: necessitiamo del suo metadato. Se quindi diciamo "oggi è il 5 di marzo" ecco che il 5 assume un senso chiaro e preciso.

Abbiamo quindi che: dato + metadato = informazione. Un metadato è anche detto **schema**.

1.6) Base di dati

Una base di dati si può definire in due maniere:

- È un insieme organizzato di dati utilizzato per il supporto allo svolgimento di una determinata mansione (ad esempio un circolo di bocce avrà il nome dei partecipanti, la data di iscrizione, ecc.).
- È un insieme di dati atomici strutturati e persistenti, raggruppati in insiemi omogenei in relazione, organizzati con minima ridondanza per essere utilizzati da diverse applicazioni in modo sicuro e controllato.

La seconda definizione, decisamente più formale e precisa, ci dice che i dati vengono strutturati, ovvero organizzati, secondo le maniere più appropriate, noi vedremo i record anche detti **tuple**. I dati vengono prima di tutto raggruppati secondo il sistema della **classificazione** ovvero per omogeneità. Tali dati devono poi poter essere messi in relazione fra loro.

Pensiamo ad esempio ad un ospedale: abbiamo dottori, pazienti, reparti, attrezzature mediche, cancelleria, ecc. Avremo quindi questi gruppi, con correlazione tra medici e reparti ma non tra attrezzature mediche e pazienti.

La parola chiave è **astrazione**.

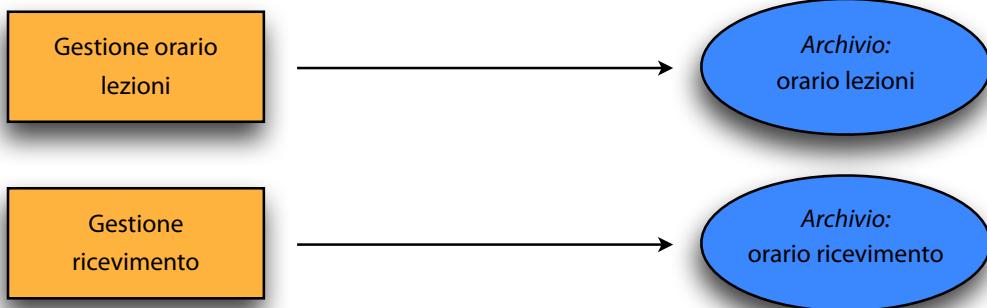
Possiamo quindi immaginare un sistema informatico che si occupa delle informazioni, le quali sono facenti parte del sistema informativo a sua volta compreso nel sistema organizzativo che è proprio di un ente/azienda.

2) Archivio o Database?

Prima dell'utilizzo dei database, si è sempre pensato ad un approccio unitario: ad ogni applicazione veniva fornito un archivio di dati su cui lavorare. Potrebbe sembrare una soluzione funzionale, ma con l'avvento della centralizzazione ci si è resi conto di quanto fosse una pessima idea. Vediamo il perché.

2.1) Il problema

Supponiamo di avere due applicazioni:

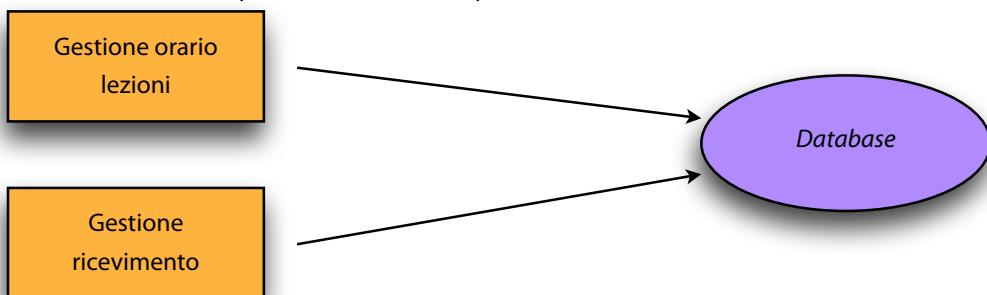


In tal caso ognuna delle due applicazioni ha un archivio. Nonostante le due applicazioni abbiano delle naturali correlazioni fra loro, i loro dati non sono in relazione e non v'è interazione tra i due applicativi. Volendo un giorno fondere i due archivi avremmo due problemi enormi:

- Ridondanza (tanti degli stessi dati sarebbero ripetuti)
- Probabile incoerenza (supponendo che vi siano stati degli errori, i dati potrebbero risultare incoerenti)

2.2) La soluzione

Ecco quindi che con un database possiamo risolvere la questione:



Possiamo portare un esempio reale in cui un problema del genere ha causato non poco scompiglio: nell'università di Torino, i 53 differenti dipartimenti avevano archivi diversi per salvare le informazioni riguardanti le forniture di materiali (computer, cancelleria, ecc.).

Quando è stato necessario stilare un resoconto totale delle spese, i codici dei prodotti, i nomi dei fornitori, e le altre informazioni non colimavano: questo intoppo ha avuto un costo di ben *due anni* di lavoro per rendere i dati consistenti.

2.3) La creazione di sottosistemi

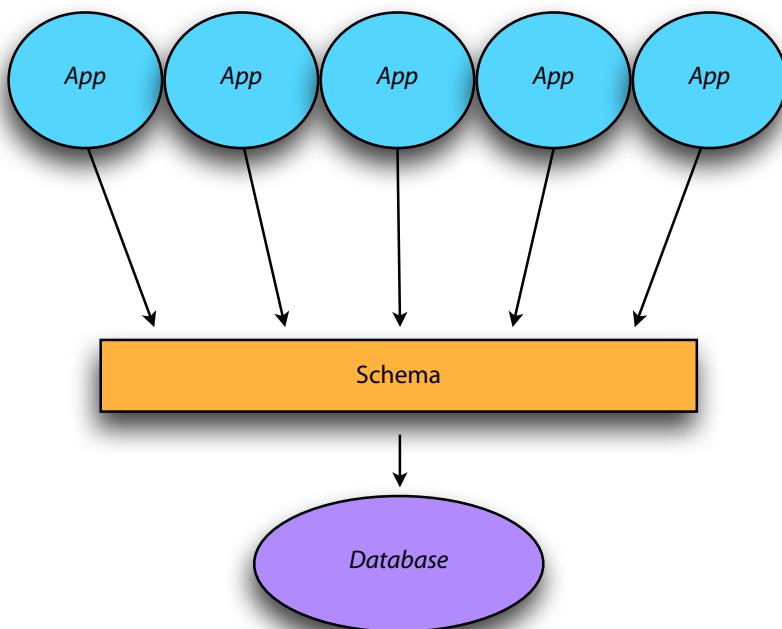
Quando andiamo a generare un vero e proprio database è necessario utilizzare alcune accortezze. Se pensiamo che il database verrà utilizzato da più utenti dobbiamo per esempio avere la possibilità di gestire **autorizzazioni** e di controllare l'**utilizzo concorrente** del database stesso.

2.4) La convenienza dei costi

Utilizzare un database non è soltanto intelligente ma è bensì scientificamente migliore rispetto all'utilizzo fallimentare degli archivi.

Infatti, utilizzando gli archivi l'aggiunta di una singola applicazione ha un costo **quadratico** rispetto al numero di applicazioni precedenti (bisogna verificare tutte le interazioni, gli eventuali scambi con gli altri archivi, ecc.)

Ma il database, invece, ha un costo di progettazione iniziale superiore rispetto a quello della singola applicazione, ma poi a lungo andare ha costo **lineare**: ogni applicazione non dovrà far altro che interagire con il database utilizzando uno schema.



3) Database Management System (DBMS)

Il DBMS è un sistema che ha diverse mansioni necessarie all'utilizzo effettivo e tecnico del database. Vediamo nel dettaglio.

3.1) Il ruolo del DBMS

- **Definisce gli schemi** (ovvero i metadati). A seconda del contesto gli schemi possono differire pur rappresentando la stessa cosa. In una università, lo studente potrebbe ad esempio avere il campo "matricola" mentre in un'ASL potrebbe avere il campo "patologia".
- Fa rispettare i **vincoli di integrità** ovvero arricchisce gli schemi con vincoli "di senso" dei dati, ad esempio un anno di nascita nel futuro non verrà permesso (è chiaramente erroneo).
- Sceglie le **strutture dati** in cui memorizzare i dati.
- Permette il **salvataggio**, il **recupero** e la **modifica** dei dati già salvati, ma solamente da utenti **autorizzati**.

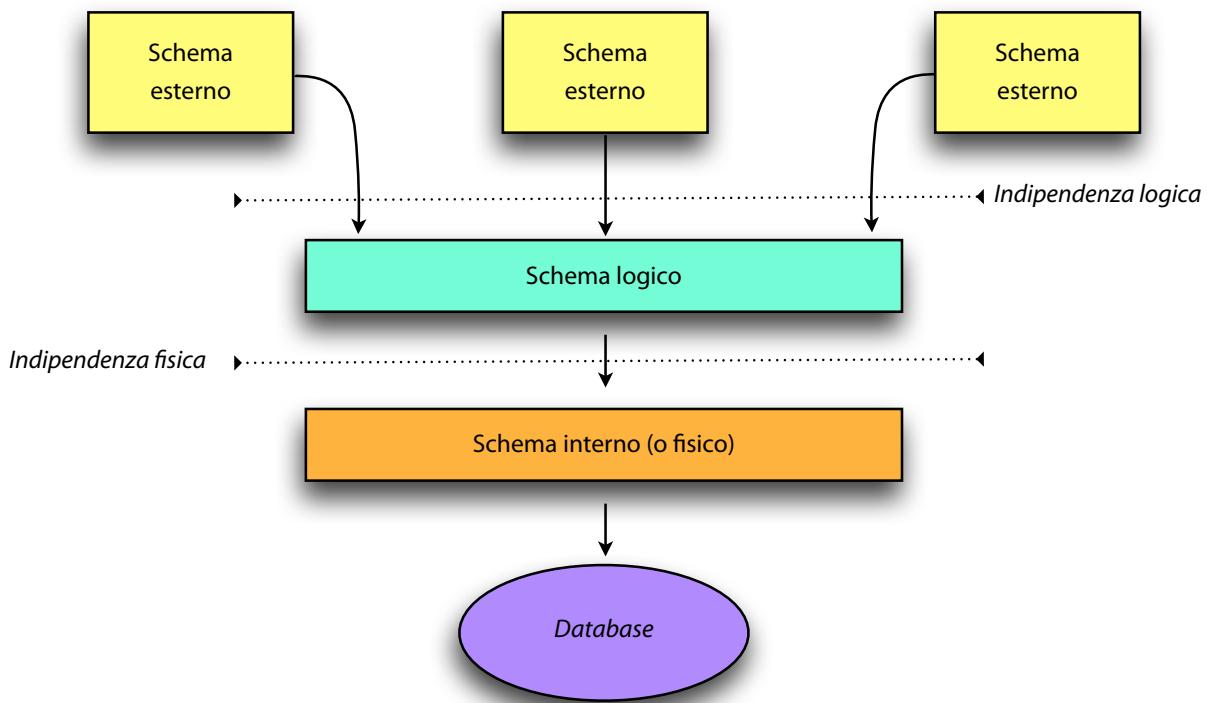
3.2) Le componenti del DBMS

Andando a vedere il contenuto vero e proprio dei DBMS troviamo:

- Linguaggi per la definizione di **schemi logici** (chiamati DDL: Data Definition Language).
- Vincoli di **integrità**.
- Linguaggi per la **manipolazione dei dati** (chiamati DML: Data Manipulation Language). Le operazioni fornite dai DML potrebbero essere riassunte in *Insert, Delete, Update*.
- Strutture per l'**accesso efficiente**.
- Gestione automatica delle **transazioni**.
- Gestione delle autorizzazioni.
- Gestione delle concorrenze.
- Ripristino dei guasti.

4) L'architettura standard: schema logico e schema interno

L'architettura standard (ANSI/SPARC) di un database vista più nel dettaglio è siffatta:



Lo **schema logico** costituisce la descrizione formale di tutto il patrimonio, si hanno quindi i *concetti percepiti*. Ad esempio possiamo immaginare di avere gli studenti, i banchi, ecc.

Lo **schema interno** è invece l'insieme degli strumenti che permettono la vera e propria sistemazione dei dati sulla *memoria*, quindi ad esempio abbiamo il mapping dei file (ovviamente relativi al DB) sulle periferiche.

Lo **schema esterno** è una porzione di schema logico, ed è solo una *vista* di ciò che lo schema logico rappresenta. Ad esempio un applicativo della segreteria studenti vedrà lo studente stesso in modo differente rispetto a ciò che potrebbe vedere un insegnante nel momento in cui registra un voto.

L'**indipendenza fisica** è quella condizione che ci assicura che l'eventuale cambiamento nella modalità di mappaggio o salvataggio dei dati (quindi la variazione dello schema interno) non intacchi lo schema logico, quindi che quest'ultimo rimanga invariato. Al massimo si potrà avere una miglioría dal punto di vista prestazionale (tempi).

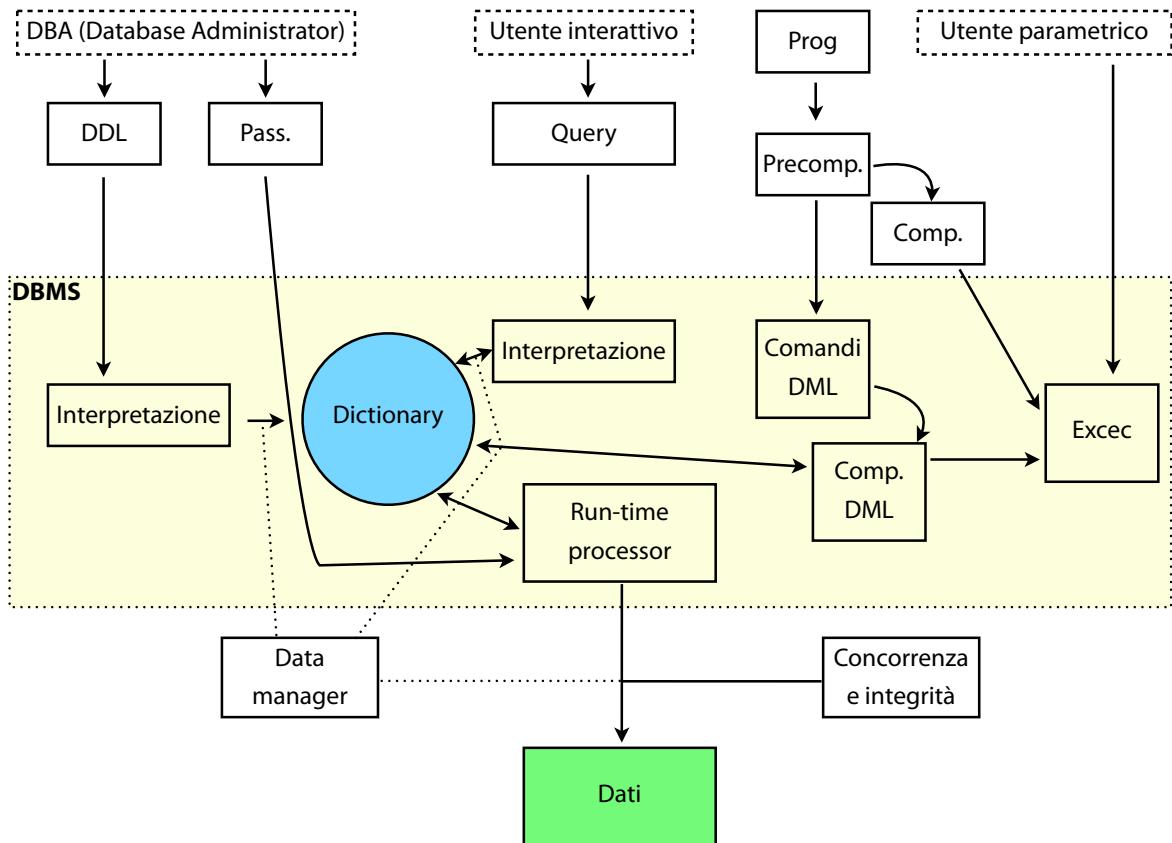
L'**indipendenza logica** è quella condizione che assicura che il cambiamento di una parte dello schema logico non influisca sulle viste esterne nel caso in cui non vengano variate parti direttamente collegate alla vista in questione.

5) Modelli logici

5.1) Il modello dei dati

Il modello dei dati è un meccanismo di astrazione per definire la base di dati con operatori e vincoli di integrità. Tali modelli vengono adottati nei DBMS esistenti per l'organizzazione dei dati. Sono quindi usati nei programmi e sono indipendenti dalle strutture fisiche. Tra questi modelli vi sono il **relazionale**, reticolare, gerarchico, a oggetti...

5.2) Schema esemplificativo di un DBMS



Vi sono tre tipologie differenti di utenti che possono approcciarsi al database.

Il **DBA** (database administrator) che è colui che costituisce sia lo schema logico che quello fisico, ed è l'unico in grado di modificarli, l'**utente interattivo** che è invece in grado di scrivere le query direttamente da terminale ma non può modificare gli schemi ed infine il più comune, l'**utente parametrico** che non conosce neanche l'esistenza del database ma si appoggia ad un eseguibile creato da un programmatore (il quale ha scritto il file Prog). Il programma conterrà quindi le query necessarie all'utente parametrico, il quale, comunque, è all'oscuro di tutto (si pensi ad una segretaria in un ufficio).

La parte fondamentale è il **dizionario**, salvato su memoria stabile, il quale contiene sia lo schema logico (con viste relative) che quello fisico (quindi il mapping su periferica). È quindi interpellato per il controllo di ogni singola query, sia essa contenuta in un programma che direttamente da terminale.

Lo schema ci fa vedere come le due indipendenze, fisiche e logiche siano rispettate, poiché i passaggi di interpretazione sono sempre staccati rispetto al dizionario stesso che può quindi essere modificato senza intaccare la struttura. L'**interpretazione** non fa altro che verificare se una certa query è scritta in maniera sensata (ad esempio SELECT FROM 'studenti' quando studenti non è una tabella del DB verrà segnalata come erronea).

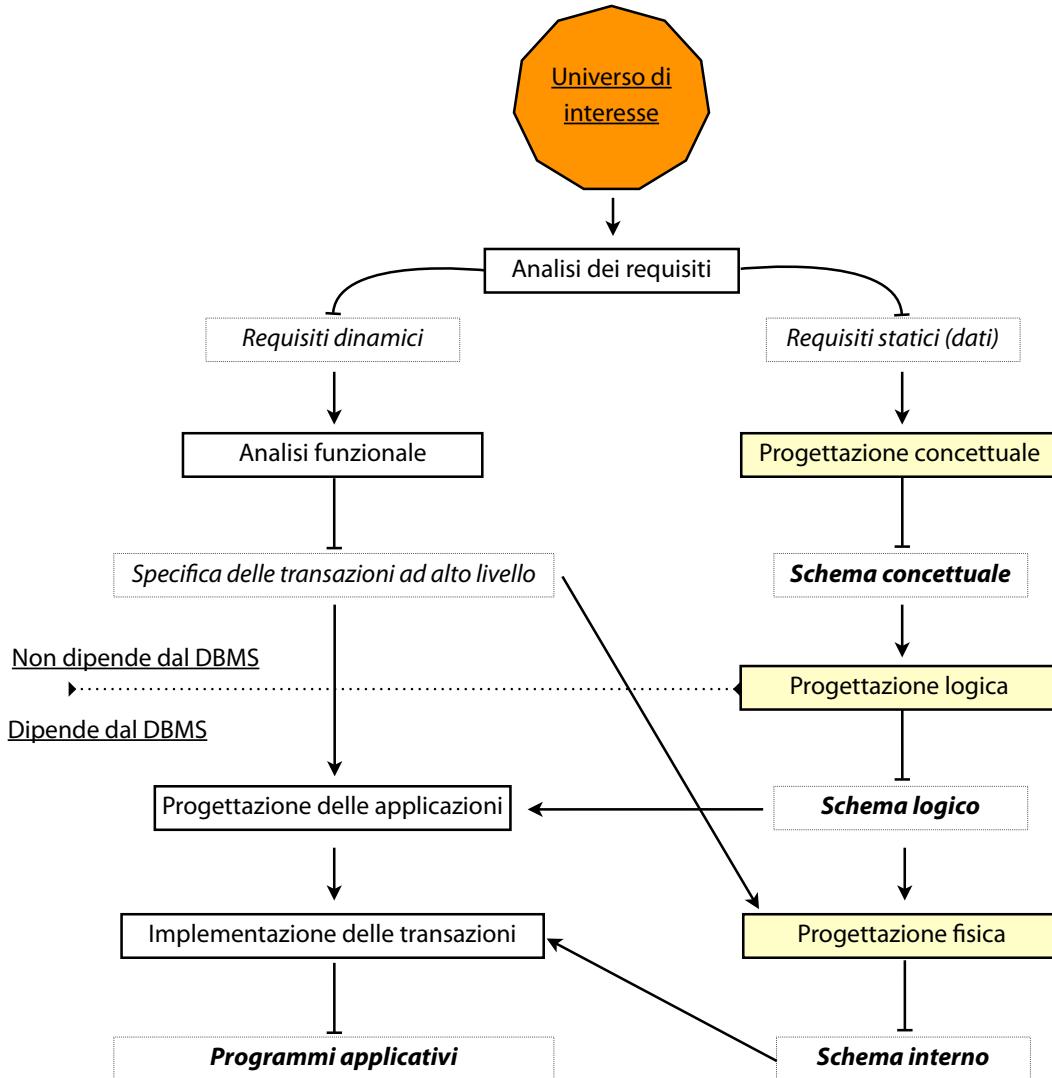
Nel caso di compilazione di un eseguibile le query vengono controllate durante la precompilazione.

6) Tabella comparativa: Archivi vs DBMS

Archivio	DBMS	Vantaggi
Applicazioni ed archivi generano ridondanza	Dati condivisi	Integrazione
Stesse informazioni aggiornate in tempi differenti	Aggiornamento unico e quindi simultaneo	Consistenza
Ogni applicazione deve garantire l'integrità	Controllo centralizzato dell'integrità	Integrità
Variazioni di tipo/formato devono essere riportate in ogni applicazione	Descrizione centralizzata: solo i programmi interessati subiscono variazioni	Indipendenza logica
Modalità di accesso ed organizzazione fisica	Accesso indipendente	Indipendenza fisica
Sinonimia nel riferire gli stessi dati	Solo nomi definiti nello schema (unico)	Standardizzazione
Accesso solo via applicativo	Accesso eventualmente interattivo	Facilità d'uso
Sicurezza gestita dalle applicazioni	Meccanismi unici di sicurezza	Sicurezza
Procedure di protezione da malfunzionamenti	Meccanismi specifici di recupero	Affidabilità
Uso esclusivo da ogni applicazione	Accesso simultaneo	Concorrenza

Si hanno quindi ben 10 aspetti "vincenti" nell'utilizzo dei database rispetto agli archivi. La partita è conclusa, vincono i database.

7) Schema metodologico di sviluppo di un sistema informatico



In grassetto appaiono le parti per il nostro corso di studi rilevanti, così come i quadrati gialli.

Ogni analisi/progettazione che viene effettuata produce dell'output necessario alla prosecuzione dei lavori di progettazione.

Inizialmente si esaminano i requisiti che il database deve avere, discernendo tra i requisiti statici (ovvero i **dati**) e quelli dinamici, ovvero il **fluire delle attività** (pensiamo ad esempio alle statistiche e gli studi per capire quale delle funzionalità implementabili sia più usata o meno, come ad esempio l'inserimento di un esame rispetto alla modifica di una residenza di uno studente). Tale analisi verte per la maggior parte sullo studio di singole attività.

7.1) Lo schema concettuale

Vediamo poi che i dati vengono utilizzati per la **progettazione concettuale**, un nuovo livello che si pone tra i dati stessi e la programmazione logica. Il passaggio dalla realtà alla formalizzazione precisa dello schema logico è un salto troppo grande: è quindi necessario introdurre un passaggio intermediario, con la relativa creazione di uno schema: lo **schema concettuale**. Tale schema è anch'esso formale, ma non basato sul sistema relazionale poi utilizzato nello schema logico. Lo schema concettuale rappresenta la conoscenza ed è vicino alla realtà. Si tratta di uno schema basato sul modello concettuale **entità-associazione**. La traduzione da schema concettuale a logico è quasi meccanica.

7.2) Il modello concettuale

Un **modello concettuale** è un sistema che viene spesso utilizzato nelle prime fasi della progettazione e si prefigge, come intuibile, di descrivere i concetti del mondo reale.

8) Panoramica sui livelli logici

Ne abbiamo già parlato prima ed il modello che noi adotteremo nello studio sarà il **modello relazionale**. Ma a seconda dei dati potrebbe rendersi più utile avere altri modelli, ad esempio il CAD per il disegno tecnico, il GIS per la rappresentazione geografica, ecc.

Possiamo raggruppare in tre differenti classi le applicazioni DBMS:

- **Applicazioni gestionali**: molti dati strutturalmente semplici.
- **Applicazioni navigazionali complesse**: vi sono grandi relazioni (si pensi ad una assonometria in AutoCAD) da dover essere espresse con efficienza.
- **Applicazioni multimediali**: la manipolazione di molti GB di file necessita di operazioni specifiche.

Non esiste al momento un DBMS in grado di soddisfare tutte e tre le caratteristiche.

8.1) L'evoluzione dei DBMS

Dalla nascita del file system nel **1970** vi sono stati due approcci iniziali, basati sul voler rappresentare in maniera efficiente strutture dati "famose" già nella programmazione:

- *Modello reticolare*: rappresentazione di una lista concatenata circolare.
- *Modello gerarchico*: rappresentazione di un albero.

L'idea dell'informatico Codd, nel **1971** è rivoluzionaria: nasce il modello relazionale che è un modo tutto diverso di concepire il database.

Nel **1990** la nascita di nuovi supporti e nuove necessità (CAD ad esempio) porta alla creazione di database *Object Oriented*, anche ispirati dai neonati linguaggi di programmazione orientati agli oggetti.

Nasce poi anche il modello *OLAP*, utilizzato per il raccoglimento di statistiche.

Basi di dati

Il modello relazionale

Capitolo 1

Enrico Mensa

Indice degli argomenti

1) Dalle strutture di classificazione al modello relazionale	1
1.1) Definizione di modello	1
1.2) Le tre strutture di classificazione: classificazione, aggregazione e generalizzazione	1
1.3) La relazione in forma di tabella	1
2) Le componenti del modello relazionale	2
2.1) Componenti della relazione	2
2.2) Attributo $A_i : T_i$	2
2.3) Schema di relazione: R	2
2.4) Istanza di relazione (stato): r	2
2.5) Una relazione è quindi una coppia $\langle R, r \rangle$ (notazione)	3
2.6) La rappresentazione di una singola tupla: t	3
2.7) La relazione matematica	3
2.8) Proprietà della relazione	3
2.9) Definizione di tupla t come funzione (rispetta il punto 8)	3
2.10) Vincoli locali di $R(A)$	4
Vincoli di dominio	
Vincolo $t[A_i] \text{ NOT NULL}$	
Vincolo di restrizione	
Vincolo di identificazione (le chiavi relazionali)	
2.11) Nozione di correttezza di una istanza $r(A)$	6
2.12) Schema di una base di dati S	6
Il criterio del buon progetto	
2.13) Vincoli globali in S	7
Vincolo esterno di Integrità Referenziale	
Vincoli generali globali	
2.14) Istanza (stato) di una base di dati	8

1) Dalle strutture di classificazione al modello relazionale

1.1) Definizione di modello

Un modello è una rappresentazione semplificata della realtà che consente di racchiudere in un numero finito di categorie tutti gli elementi di interesse del mondo reale.

1.2) Le tre strutture di classificazione: classificazione, aggregazione e generalizzazione

Per poter progettare è necessario fare un forte uso dell'astrazione. Per questo si adoperano le **strutture di classificazione**, anche dette **astrazioni**, le quali permettono di rappresentare gli oggetti della realtà raggruppandoli in classi.

L'astrazione è un procedimento mentale basato sul raggruppamento di oggetti date le loro caratteristiche comuni e tramite l'esclusione di altre giudicate non rilevanti. Si giunge così alla creazione di un nuovo oggetto che è unitario secondo le proprietà prese in considerazione.

Vi sono tre tipi di astrazione:

- **Classificazione**: si definisce una classe tramite le proprietà comuni degli oggetti (ad esempio un atomo ha in comune le proprietà di numero atomico, quantità di elettroni, ecc.).
- **Aggregazione**: è l'astrazione tramite la quale si giunge alla definizione di un concetto a partire da altri che ne costituiscono le proprietà (ad esempio il concetto di molecola a partire dal concetto di unione di più atomi, si pensi altrimenti ad una tupla).
- **Generalizzazione**: avendo due oggetti basati sull'astrazione di classificazione e con caratteristiche comuni, è possibile creare un nuovo oggetto che ne rappresenta l'unione (ad esempio la classe delle persone è aggregazione della classe degli uomini e della classe delle donne).

Nel modello relazionale, tali astrazioni prendono nomi particolari. La classificazione è detta **relazione**, l'aggregazione è detta **attributo strutturato**.

1.3) La relazione in forma di tabella

Prendiamo questo esempio grafico.

MEDICI					→ Relazione "medici"
COD	Cognome	Nome	Residenza	Reparto	Attributi della relazione "medici"
203	Neri	Piero	AL	A	
547	Bisi	Mario	MI	B	
461	Bargio	Sergio	TO	B	→ Singolo dato "MI"

REPARTI			→ Relazione "reparti"
Codice	Nome	Primario	Attributi della relazione "reparti"
A	Chirurgia	203	
B	Pediatria	574	
C	Medicina	530	→ Tupla

Osserviamo con attenzione questa rappresentazione. Notiamo che una singola tabella rappresenta una classe di classificazione, ovvero una relazione.

L'insieme degli attributi della tabella insieme alla relazione definiscono uno **schema di relazione**.

Un singolo dato ha significato solo quando è posizionato da qualche parte, altrimenti ne è privo.

Leggere dalla tabella un dato in corrispondenza di una colonna significa interpretarne il significato, ovvero leggerne lo schema, ovvero trasformare il dato in informazione. Ad esempio, nella relazione "REPARTI" leggere 574 all'interno della colonna primaria non ci dice nulla. L'interpretazione del metadata ci dice però che il primario del reparto con codice B è quel medico che corrisponde alla matricola 574.

La tupla è un **record**, ovvero una linea della tabella.

2) Le componenti del modello relazionale

Proviamo a fare un elenco degli elementi che ritroviamo nel modello relazionale.

2.1) Componenti della relazione

Una relazione è costituita da:

- Schema di relazione (*stabile, non viene modificato*).
- Istanza (*variabile*).

2.2) Attributo $A_i : T_i$

Un'attributo è costituito da:

- Nome dell'attributo (nell'esempio, "residenza" all'interno della relazione "medici").
- Tipo dell'attributo (*anche detto dominio*) fra cui:
 - Tipi primitivi (*Integer, String, Real, Boolean, Date, Tipi utente*)
 - $\text{NULL} \in T_i$ (è accettato avere una tupla con dei "buchi"). NULL è un valore vero e proprio, e può significare tre cose:
 - Il dato è sconosciuto (ad esempio una data di nascita non ancora fornita)
 - Il dato è inesistente (ad esempio un campo coniuge)
 - Mancanza di informazione (non si sa se il dato sia inesistente oppure sconosciuto)

2.3) Schema di relazione: R

Lo schema di relazione è un insieme di attributi { $A_1 : T_1 , A_2 : T_2 , \dots , A_n : T_n$ }. In qualità di insieme gli elementi in esso contenuti sono necessariamente distinti, ovvero il nome di ogni attributo deve essere diverso. I tipi sono invece condivisibili. Gli attributi sono poi *rappresentabili* in forma tabellare, ottenendo così la riga:

COD	Cognome	Nome	Residenza
-----	---------	------	-----------

Lo schema può essere con o senza nome. Nel caso sopra lo schema risulta essere: PAZIENTE(COD, Cognome, Nome, Residenza). Uno schema generico può essere rappresentato con questa notazione:

$R(A_1 : T_1 , A_2 : T_2 , \dots , A_n : T_n)$ o sottintendendo i tipi: $R(A_1 , A_2 , \dots , A_n)$.

Vi è poi la notazione $A : \{A_1 , A_2 , \dots , A_n\}$ che rappresenta il mero insieme di attributi, e quindi lo schema diviene $R(A)$ od eventualmente (A) (nel caso di schema senza nome).

Esiste poi la cardinalità $|A|$ = numero di attributi dello schema, ovvero il grado della relazione.

2.4) Istanza di relazione (stato): r

L'istanza di relazione è l'insieme delle tuple. Tale insieme è rappresentabile con la notazione $r(<v_1 , v_2 , \dots , v_n>, \dots)$ dove $<v_1 , v_2 , \dots , v_n>$ è una singola tupla e dove ogni $v_i \in \text{dom}(A_i)$ cioè sostanzialmente al tipo dell'attributo i -esimo.

La cardinalità di r è il numero di tuple che appartengono ad r , cioè in sostanza la quantità di record contenuti nella tabella.

2.5) Una relazione è quindi una coppia $\langle R, r \rangle$ (notazione)

Tale definizione coincide perfettamente con il punto 1.

Le notazioni per definire una relazione sono: $r(R)$, $r(A_1, A_2 \dots A_n)$, r .

La relazione è ben rappresentabile con una tabella, dove le righe sono tuple tranne la prima che rappresenta lo schema.

2.6) La rappresentazione di una singola tupla: t

Potrebbe essere utile rappresentare una singola tupla. Useremo quindi la notazione:

$t = \langle v_1, v_2, \dots, v_n \rangle \in r(A_1, A_2 \dots A_n)$ che è una singola tupla. Per arrivare ad un singolo dato contenuto nella tupla si usa la notazione $t[A_i] = t[A_i] = v_i$ ma volendo più dati anche $t.(A_i, A_j, A_k) = t[A_i, A_j, A_k] = \langle v_i, v_j, v_k \rangle$ che è una sotto-parte della tupla. Si noti che ovviamente $t.A$ o $t[A]$ equivale a tutta la tupla.

2.7) La relazione matematica

Sappiamo bene che, matematicamente parlando, una moltiplicazione fra insiemi equivale ad un sottoinsieme così definito: $s \subseteq \text{dom}(A_1) \times \text{dom}(A_2) \times \dots \times \text{dom}(A_n)$ dove però $A \times B = (a,b)$ è diverso da $B \times A = (b,a)$. Tale differenza non esiste in questo caso, quindi $A \times B = B \times A$.

2.8) Proprietà della relazione

- L'ordine degli attributi in R è irrilevante
- L'ordine delle tuple in r è irrilevante (all'interno di r , non all'interno delle tuple!)

Pertanto valgono tutte queste equivalenze:

Codice	Nome	Primario	=	Nome	Codice	Primario	=	Codice	Nome	Primario
A	Chirurgia	203		Chirurgia	A	203		A	Chirurgia	203
B	Pediatria	574		Pediatria	B	574		C	Medicina	530
C	Medicina	530		Medicina	C	530		B	Pediatria	574

Ma se la nozione d'ordine è davvero così irrilevante, come può esistere correlazione tra $(A_1, A_2 \dots A_n)$ e $r\{\langle v_1, v_2, \dots, v_n \rangle\}$? Cioè come si può essere certi che v_i sia effettivamente il dato contenuto in corrispondenza di A_i ?

2.9) Definizione di tupla t come funzione (rispetta il punto 8)

Per giustificare la domanda del punto 8 dobbiamo definire t come funzione. Una funzione manda un valore in un altro, quindi, prendendo la tupla $t < B543, Missoni, Nadia, TO >$ dobbiamo immaginarci una $t: A_i \rightarrow v_i$ che crea una correlazione univoca tra ogni attributo ed il dato della tupla corrispondente.

Pertanto $t: \{A_1, A_2, \dots, A_n\} \rightarrow D$ dove $D = \text{dom}(A_1) \cup \text{dom}(A_2) \cup \dots \cup \text{dom}(A_n)$ con il vincolo che $\forall A_i t(A_i) \in \text{dom}(A_i)$ cioè in sostanza D è l'insieme di tutti i domini ottenibili da tutti gli attributi, ma nel momento in cui il dato è sistemato in corrispondenza di un attributo deve ovviamente rispettarne il tipo. Possiamo immaginare quindi una funzione $t: A \rightarrow D$

A	D
COD	B543
Cognome	Missoni
Nome	Nadia
Residenza	TO

Si noti che t è una funzione totale (cioè ha sempre un risultato dato A_i) e definisce una tupla compatibile con lo schema A .

Data questa definizione di t , r è quindi un insieme di funzioni (t_1, t_2, \dots, t_n) tutte distinte fra loro (non può esistere una tupla identica all'altra!).

2.10) Vincoli locali di R(A)

Si definiscono vincoli locali quelle particolari condizioni che riguardano i dati e che vanno rispettate all'interno di un certo schema.

- Vincoli di dominio

Ogni D_i deve essere in prima forma normale cioè D_i deve essere *semplice*. Questo significa che una tupla non può avere elementi che sono insiemi oppure tabelle, ovvero devono essere dei "singoletti". Esempio di dominio non in prima forma normale è il seguente, in cui in cui i recapiti sono un insieme.

Cognome	Nome	Recapiti
Lamberti	Marco	{1012... , 292...}
Signoritto	Gino	{124... ,522...}

- Vincolo $t[A_i] \neq \text{NULL}$

Nel caso in cui questo vincolo sia attivo, nessuna tupla in corrispondenza all'attributo A_i può avere valore NULL.

- Vincolo di restrizione

Ad esempio vogliamo restringere un campo età dai 16 ai 69 anni (immaginiamo in una azienda dove vogliono registrare i lavoratori).

- Vincolo di identificazione (le chiavi relazionali)

~~~  
*Essendo un argomento ampio, vediamolo a parte!*  
~~~

La relazione è un modo formale per definire insiemi omogenei: ma come possiamo distinguere una tupla da un'altra (ovvero *identificarla*)? Una buona idea potrebbe essere quella di avere un attributo "codice".

Intuitivamente l'idea di codice è quella di essere univoco per ogni elemento cui assegnamo il codice stesso. Quindi avere codici uguali per tuple diverse strida con il significato stesso di codice. Prendiamo però questo esempio:

RICOVERI			
PAZ	Inizio	Fine	Reparto
A102	02/05/94	09/05/94	A
A102	02/12/94	02/01/95	A
S555	02/05/94	03/12/94	B

Ci possiamo subito rendere conto di come il codice paziente uguale non sia un problema, infatti un solo paziente può essere ricoverato due volte purché le date di inizio siano compatibili.

Capiamo quindi che non è tanto importante il concetto di "codice" in se, bensì il concetto che lo schema stesso vuole portare. Dobbiamo sostanzialmente entrare nel concetto di *Ricovero* per poter essere chiari su cosa possa identificare una tupla da un'altra.

Pensando ad un altro esempio ancora, gli attributi

Cognome	Nome	Data nascita	Luogo di nascita
---------	------	--------------	------------------

potrebbero essere tutti quanti utilizzati per identificare una persona singola! Infatti il sistema del codice fiscale si basa proprio su questi attributi.

Abbiamo quindi capito che ci sono parecchie complicazioni. Formalizziamo questi ragionamenti definendo una **chiave relazionale** che può essere di due tipi: **chiave candidata** e **chiave principale**. Per poter proseguire introduciamo la nozione di superchiave.

La nozione di superchiave

Una superchiave $sk \subseteq A$ è un sottoinsieme di attributi tale che:

$$\forall_{i,j} (t_i \in r(A)) \wedge (t_j \in r(A)) \wedge (t_i[sk] = t_j[sk]) \Rightarrow (t_i[A] = t_j[A])$$

Ovvero date due tuple appartenenti alla relazione, se tali tuple in corrispondenza degli attributi contenuti nella superchiave sono equivalenti allora devono essere necessariamente la stessa tupla. Ciò significa che non possono esistere due tuple con gli stessi dati nella superchiave a meno che non siano la stessa tupla.

La chiave candidata

Una chiave $k \subseteq A$ è detta candidata se è:

- Superchiave
- Superchiave *minimale* (ovvero nessun sottoinsieme proprio di k deve essere a sua volta superchiave, cioè in sostanza non ci devono essere attributi "inutili" compresi in k).

La chiave principale

Una chiave $pk \subseteq A$ è una chiave scelta dal progettista tra le chiavi candidate. Unico vincolo è che nessun dato sotto gli attributi inclusi in pk può essere NULL (questo per motivi di progettazione, non per motivi formali!). Verrà comunque precisato dopo l'utilizzo dell'elemento NULL).

Esempi di superchiavi, chiavi candidate e chiavi principali

RICOVERI

PAZ	Inizio	Fine	Reparto
A102	02/05/94	09/05/94	A
A102	02/12/94	02/01/95	A
S555	02/05/94	03/12/94	B
B444	01/12/94	02/01/95	B
S555	05/10/94	01/11/94	A

In questo caso abbiamo:

- PAZ \wedge Inizio \wedge Fine è sk
- PAZ \wedge Inizio \wedge Fine **non è** k (poiché PAZ \wedge Inizio è sk)
- PAZ \wedge Inizio è sk
- PAZ **non è** sk
- Inizio **non è** sk

STUDENTI

MATR	Cognome	Nome	Nascita	Luogo	CF
73342	Neri	Piero	02/05/94	TO	NPR...
74323	Bisi	Mario	03/07/94	RI	BMS...
73245	Bargio	Sergio	05/05/94	VC	BSE...

In questo caso abbiamo:

- MATR è sk e anche k
- Cognome \wedge Nome \wedge Nascita \wedge Luogo è sk
- CF è sk e anche k
- MATR \wedge CF è sk (ma **non è** k)

Precisazioni sui vincoli di identificazione

- $r(A)$ è anche una notazione per rappresentare insieme di tuple.
- $\neg \exists_{i,j} (t_i[sk] = t_j[sk]) \wedge (t_i[A] \neq t_j[A])$ è un altro modo di scrivere la notazione di superchiave, ovvero non esistono coppie di tuple coincidenti su sk e distinte. Ad esempio se arrivasse una richiesta con un COD uguale ad un COD già esistente, le due tuple dovrebbero coincidere. Chiaramente se il nuovo COD è diverso da tutti quelli già nel database il problema non si pone.
- Se sk è una superchiave allora $sk \subseteq w \subseteq A$ è una superchiave. Pertanto A è necessariamente una superchiave, anche per la definizione stessa di istanza (dove ogni tupla è distinta).
- Per la definizione sopra A è sk, quindi A contiene almeno una chiave candidata.

~~~

Dopo aver visto i vincoli locali ed in particolare dei vincoli riguardanti l'identificazione, torniamo alla lista di componenti.

~~~

2.11) Nozione di correttezza di una istanza $r(A)$

Una istanza $r(A)$ è corretta quando ogni tupla appartenente all'istanza è compatibile con A ovvero i campi della singola tupla rispettano tutti i vincoli.

2.12) Schema di una base di dati S

Abbiamo parlato di schemi di relazione, ma un database è certamente costituito da più relazioni. Definiamo perciò con $S = \{R_1, R_2, \dots, R_n\}$ l'insieme delle relazioni che creano una base di dati.

In questo caso ogni relazione deve avere un nome e ogni nome deve essere distinto, si richiede inoltre che ogni relazione abbia una chiave primaria.

Una buona rappresentazione può essere quella lineare:

PAZIENTE(COD, Cognome, Nome, Residenza)
 MEDICO(MATR, Cognome, Nome, Residenza, Reparto)
 REPARTO(COD, Nome, Primario)
 RICOVERO(PAZ, Inizio, Fine, Reparto)

Gli attributi sottolineati indicano la chiave primaria dello schema. Eventualmente si può adoperare la notazione di sottolineatura tratteggiata per indicare le altre chiavi candidate (non scelte per diventare chiave primaria).

- Il criterio del buon progetto

Durante la progettazione di un database è importante seguire alcune regole di "buon senso", che non sono comunque leggi formali ma semplicemente consigli.

- 1 - In caso di attributi con nomi uguali fra relazioni diverse, si tenda a utilizzare tipi uguali.
- 2 - Evitare *omonimia*, ovvero dare nomi uguali ad attributi che hanno significato diverso, ad esempio:

PERSONA(..., Nome, ...)
 PRODOTTO(..., Nome, ...)

In questo caso, "Mario" e "Lavatrice" non sono accostabili nonostante nei singoli schemi abbiano senso.

- 3 - Evitare *sinonimia*, ovvero dare nomi diversi ad attributi che hanno uguale significato, ad esempio:

CC(..., Agenzia, ...)
(Succursale, ...)

In un ambiente bancario Agenzia e Succursale hanno lo stesso significato. Meglio usare solo una delle due.

Si cerca pertanto di attuare una **standardizzazione dei termini**.

2.13) Vincoli globali in S

- Vincolo esterno di Integrità Referenziale

Come già detto un database vuole rappresentare la realtà. Allora è necessario poter definire dei vincoli anche fra le relazioni all'interno del database. Ad esempio, prendiamo le due relazioni:

MEDICO(MATR, Cognome, Nome, Residenza, Reparto)

REPARTO(COD, Nome, Primario)

È abbastanza evidente che non può esistere un Reparto (attributo della relazione MEDICO) che non appartenga alla relazione REPARTO.

In maniera più formale, i valori che potranno apparire sotto l'attributo Reparto dovranno necessariamente figurare sotto l'attributo COD della relazione REPARTO.

Si tratta chiaramente di un vincolo direzionale ovvero non vale l'implicazione inversa.

Per denotare questo tipo di rapporto si utilizza una freccia che parte dal **referenziante** e che va al **referenziato**.

Si ha quindi:

MEDICO(MATR, Cognome, Nome, Residenza, Reparto)



REPARTO(COD, Nome, Primario)

Si noti che l'attributo referenziato deve sempre essere una chiave primaria.

Più in generale, considerando il caso:

$R_s(\dots, B, \dots)$
↓
 $R_n(PK, \dots)$

Il vincolo di integrità referenziale si esprime formalmente in questo modo:

$$\forall t_i (t_i \in r(R_s)) \Rightarrow \exists t_j (t_j \in r(R_n)) \wedge (t_i[B] = t_j[PK])$$

Ovvero per ogni dato sotto l'attributo B nella relazione R_s , deve esistere un dato uguale ad esso sotto l'attributo PK nella relazione R_n . Si ricorda che PK è necessariamente chiave primaria.

È ovvio che $\text{dom}(B) = \text{dom}(PK)$.

È comunque possibile che vi siano coppie di attributi che fanno riferimento a coppie di attributi che sono chiavi primarie come ad esempio:

ANALISI(CODICE-ANALISI, Paziente_Analizzato, Data_Ricovero, Tipo, ...)

RICOVERO(PAZ, Inizio, Fine, Reparto)

- Vincoli generali globali

Esaminiamo questo esempio di database in ambito bancario:

CLIENTE(CF, Nome, Indirizzo)

AGENZIA(N°-Ag, Città-Ag, Direttore, ...)

CC(N°-CC, N°-Ag, Città-Ag, Saldo, Data_Mov)

TITOLARE(CF, N°-CC)

MUTUO(CF, Ammontare, N°-Ag, Città-Ag)

Tra le integrità referenziali abbiamo:

* MUTUO(N°-Ag, Città-Ag) e CC(N°-Ag, Città-Ag) referenziano AGENZIA(N°-Ag, Città-Ag),

* TITOLARE(CF) referenzia CLIENTE(CF),

* TITOLARE(N°-CC) referenzia CC(N°-CC).

In questo esempio oltre ai vari vincoli di integrità referenziale abbiamo altri vincoli di senso logico, ad esempio non è possibile creare un mutuo se non si ha prima un conto corrente oppure non possono esistere agenzie nella stessa città con numeri non incrementali (non esiste l'agenzia 2 di Moncalieri senza l'agenzia 1 in Moncalieri).

Uno schema S di una base di dati è come definita nel punto 12 ma con l'aggiunta dei vincoli globali.

2.14) Istanza (stato) di una base di dati

Dato uno schema di base di dati $S = \{R_1, R_2, \dots, R_n\}$ possiamo definire il database come:

$$DB = (\langle R_1, r_1 \rangle, \langle R_2, r_2 \rangle, \dots, \langle R_k, r_k \rangle)$$

dove ogni r_i è corretta e sono soddisfatti tutti i vincoli locali e globali di ogni relazione e dello schema della base di dati stesso.

Possiamo chiaramente dichiarare che **il modello relazionale è basato su VALORI.**

Basi di dati

L'algebra relazionale

Capitolo 2

Enrico Mensa

Indice degli argomenti

1) <i>Introduzione all'algebra relazionale</i>	1
1.1) Accenni all'SQL	
1.2) Excursus sugli operatori	
1.3) Modello di database (per esempi successivi)	
2) <i>L'operatore base di selezione: σ</i>	3
2.1) Il predicato	
2.2) Il risultato	
2.3) Esempi	
3) <i>L'operatore base di proiezione: Π</i>	4
3.1) Il risultato	
3.2) Cardinalità del risultato	
3.3) Esempi	
3.4) Composizione fra selezione e proiezione	
3.5) Gli alberi sintattici	
4) <i>L'operatore base di prodotto cartesiano: x</i>	6
4.1) Condizione sugli operandi	
4.2) Il risultato	
4.3) Cardinalità del risultato	
4.4) La proprietà commutativa	
4.5) Esempi	
5) <i>L'operatore base di unione: \cup</i>	7
5.1) Condizione sugli operandi	
5.2) Il risultato	
5.3) Cardinalità del risultato	
5.4) Esempi	
6) <i>L'operatore base di differenza: -</i>	8

6.1) Condizione sugli operandi	
6.2) Il risultato	
6.3) Cardinalità del risultato	
6.4) Esempi	
7) <i>L'operatore base di ridenominazione: p</i>	9
7.1) Il risultato	
7.2) Esempi	
8) <i>L'operatore derivato di intersezione: ∩</i>	9
8.1) Condizione sugli operandi	
8.2) Il risultato	
8.3) Cardinalità del risultato	
8.4) Esempi	
9) <i>L'operatore derivato di Θ-join: ⋈</i>	10
9.1) La derivazione	
9.2) Condizione sugli operandi	
9.3) Il predicato	
9.4) Il risultato	
9.5) Cardinalità del risultato	
9.6) Esempio di applicazione semplice	
9.7) Esempio di applicazione con ridenominazione	
10) <i>Esercizi sull'applicazione degli operandi (visti fin'ora)</i>	13
10.1) Elencare i reparti diretti da medici che vi lavorano.	
10.2) Elencare i reparti diretti da medici che non vi lavorano.	
10.3) Ricoveri avvenuti nello stesso giorno del paziente A102	
10.4) Elencare i primari	
10.5) Trovare le MATR dei capi di impiegati che guadagnano più di 40 euro	
10.6) Trovare tutte le informazioni dei capi di impiegati che guadagnano più di 40 euro	

10.7) Trovare tutte le informazioni dei capi che guadagno meno di almeno uno dei loro subalterni	
10.8) Trovare i pazienti che hanno subito almeno due ricoveri	
11) Caso particolare del Θ -join: l'equi-join	18
11.1) 1° sottocaso: l'equi-join completo	
11.2) 2° sottocaso: confronto attributi/superchiave	
11.3) 3° sottocaso: confronto attributi/superchiave completo	
12) Caso particolare dell'equi-join: natural-join	18
12.1) Esempi di natural-join	
12.2) Natural-join degenerato	
13) Caso particolare del Θ -join: il semi-join	19
13.1) Cardinalità del risultato	
13.2) Esempio del semi-join	
14) Le interrogazioni negative (ed esempi)	20
14.1) Lo schema risolutivo delle interrogazioni negative	
14.2) Elencare i medici NON primari	
14.3) Elencare i pazienti NON residenti in città in cui risiede qualche medico	
14.4) Elencare i pazienti MAI ricoverati nel reparto A	
15) L'operatore derivato di quoziente: \div	21
15.1) Arrivare al quoziente empiricamente	
15.2) Definizione formale di quoziente	
15.3) Esempio di quoziente	
16) Semantica del valore NULL	23
16.1) La logica a tre valori	
16.2) Due nuovi predicati	
16.3) Esempio di query con valori nulli	
17) Caso particolare del Θ -join: l'outer-join	24
17.1) Il left-join	

17.2) Il right-join

17.3) Il full-join

18) Ulteriori esempi

26

18.1) Trovare gli autori che hanno pubblicato solo in collaborazione

1) Introduzione all'algebra relazionale

Lo studio del modello relazionale ci ha portato al chiarimento della parte detta **DDL** (Data Definition Language) del DBMS. Entriamo ora nel mondo del **DML** (Data Manipulation Language) che è quella famiglia di linguaggi che fornisce una grammatica formalizzata e precisa per poter trattare i dati contenuti nel database (tramite operazioni di inserimento, cancellazione e modifica).

1.1) Accenni all'SQL

Il DML viene utilizzato a livello più alto dai DMBS. In SQL (un DML) abbiamo quattro macro-tipi di comandi:

Inserimento di una tupla nella relazione

INSERT

INTO RELAZIONE(A₁, A₂, ..., A_k)

VALUE ("203","Enzo","Rossi")

Cancellazione di una tupla dalla relazione

DELETE

FROM RELAZIONE

WHERE Condizione di selezione della tupla da cancellare

Modifica di una tupla nella relazione

UPDATE RELAZIONE

SET (A₁ = espressione₁, A₂ = espressione₂)

WHERE Condizione di selezione della tupla da modificare

Interrogazione

SELECT A₁

FROM RELAZIONE

WHERE Condizione di selezione della tupla da cercare

Le condizioni di selezione sopracitate verranno espresse tramite precise regole formali, che sono però materia di studio del laboratorio di Basi di Dati.

1.2) Excursus sugli operatori

L'algebra relazionale è costituita da operatori di base e derivati, i quali sono applicati su uno o due relazioni (sono quindi operatori unari/binari) e che generano una relazione come risultato.

Ecco la lista degli operatori:

Operatori di base	
Nome	Notazione
Selezione	$\sigma_p(r(A))$
Proiezione	$\Pi_{A_1, \dots, A_j}(r(A))$
Prodotto cartesiano	$r_1(A) \times r_2(B)$
Unione	$r_1(A) \cup r_2(A)$
Differenza	$r_1(A) - r_2(A)$
Ridenominazione	$\rho_{\beta_1, \beta_2, \dots <- A_1, A_2, \dots}(r(A))$

Operatori derivati	
Nome	Notazione
Intersezione	$r_1(A) \cap r_2(A)$
Θ -join	$r_1(A) \bowtie_{\Theta} r_2(B)$
Quoziente	$r_1(A, B) \div r_2(B)$

Non resta che esaminarli tutti, uno ad uno, nel dettaglio.

Si noti che i risultati (relazioni) di queste operazioni sono relazioni senza nome.

L'algebra relazionale è inoltre composizionale ovvero è possibile la composizione di più funzioni fra loro.

1.3) Modello di database (per esempi successivi)

Nell'esaminare gli operatori durante gli esempi si farà riferimento a più relazioni:

PAZIENTI

<u>COD</u>	<u>Cognome</u>	<u>Nome</u>	<u>Residenza</u>
A102	Necchi	Luca	TO
B372	Rossigni	Piero	NO
B543	Missoni	Nadia	TO
B444	Missoni	Luigi	VC
S555	Rossetti	Gino	AT

RICOVERI

<u>PAZ</u>	<u>Inizio</u>	<u>Fine</u>	<u>Reparto</u>
A102	02/05/94	09/05/94	A
A102	02/12/94	02/01/95	A
S555	05/10/94	03/12/94	B
B444	01/12/94	02/01/95	B
S555	05/10/94	01/11/94	A

MEDICI

<u>MATR</u>	<u>Cognome</u>	<u>Nome</u>	<u>Residenza</u>	<u>Reparto</u>
203	Neri	Piero	AL	A
574	Bisi	Mario	MI	B
461	Bargio	Sergio	TO	B
530	Belli	Nicola	TO	C
405	Mizzi	Nicola	AT	A
501	Monti	Mario	VC	A

REPARTI

<u>COD</u>	<u>Nome</u>	<u>Primario</u>
A	Chirurgia	203
B	Pediatria	574
C	Medicina	530

IMPIEGATI

<u>MATR</u>	<u>Nome</u>	<u>Età</u>	<u>Stipendio</u>
100	Mario R.	34	40
103	Mario B.	23	35
104	Luigi N.	38	61
210	Pippo G.	40	50

E

<u>Matr</u>	<u>Corso</u>
400	Database
500	Linguaggi
500	Analisi
300	Database
500	Database
400	Linguaggi

P

<u>Corso</u>
Analisi
Database
Linguaggi

SUPERVISORI

<u>Capo</u>	<u>Impiegato</u>
210	100
210	103
210	104
231	105

2) L'operatore base di selezione: σ

Come già detto sopra l'operatore di selezione è un operatore che restituisce una relazione, quindi uno schema ed un'istanza. Permette di estrarre un sottoinsieme delle tuple di una relazione.

2.1) Il predicato

Come pedice dell'operatore viene espresso il predicato booleano eventualmente con parentesi tramite il quale si selezionano le tuple che appariranno poi nel risultato. Si sta quindi parlando dell'istanza del risultato.

Vi sono due tipologie di predici:

- $A_i \Theta$ costante

- $A_i \Theta A_j$

dove Θ rappresenta un operatore di confronto, infatti $\Theta = \{=, \neq, >, \geq, <, \leq\}$.

I vari predici, in qualità di espressioni booleane, possono poi essere combinati con i classici simboli \wedge e \neg .

2.2) Il risultato

Il risultato è uno schema senza nome che ha come attributi gli attributi della relazione su cui si opera la selezione, ma come tuple solamente quelle che rispettano il predicato espresso a pedice.

2.3) Cardinalità del risultato

La cardinalità del risultato è $0 \leq |\sigma_p(r(A))| \leq |r(A)|$ ovvero il numero delle tuple del risultato è minore/uguale a quello dell'operando. Eventualmente la cardinalità può essere zero (basta usare un predicato sempre falso).

2.3) Esempi

Supponiamo di voler *selezionare* tutti i pazienti che risiedono a TO.

Il predicato (residenza = "TO") definisce quali siano le istanze facenti parte del risultato. Abbiamo quindi:

$\sigma_{\text{Residenza} = \text{"TO"}}(\text{pazienti}) =$	COD	Cognome	Nome	Residenza
	A102	Necchi	Luca	TO
	B543	Missoni	Nadia	TO

Prendiamo invece i residenti in TO oppure in NO:

$\sigma_{\text{Residenza} = \text{"TO"} \vee \text{Residenza} = \text{"NO"} }(\text{pazienti}) =$	COD	Cognome	Nome	Residenza
	A102	Necchi	Luca	TO
	B372	Rossigni	Piero	NO
	B543	Missoni	Nadia	TO

Questa volta prendiamo tutti quelli che non risiedono a TO:

$\sigma_{\neg(\text{Residenza} = \text{"TO"})}(\text{pazienti}) =$	COD	Cognome	Nome	Residenza
	B372	Rossigni	Piero	NO
	B444	Missoni	Luigi	VC
	S555	Rossetti	Gino	AT

Si noti che lo stesso risultato sarebbe denotato dall'operazione $\sigma_{\text{residenza} \neq \text{"TO"} }(\text{pazienti})$. Questo ci fa capire come una operazione nell'algebra relazionale possa essere scritta in più modi.

3) L'operatore base di proiezione: Π

L'operatore di proiezione permette di estrarre "per attributi" una parte di una relazione.

3.1) Il risultato

Data l'operazione $\Pi_{A_i, \dots, A_j}(r(A))$ gli attributi del risultato saranno solo A_i, \dots, A_j . L'istanza comprende tutte le tuple dell'operando ma solamente rispetto agli attributi $A_i \dots A_j$. Si noti che non vi possono essere ripetizioni di tuple.

3.2) Cardinalità del risultato

La cardinalità del risultato è $0 \leq |\Pi_{A_i, \dots, A_j}(r(A))| \leq |r(A)|$ ovvero il numero delle tuple del risultato è minore/uguale a quello dell'operando. Eventualmente la cardinalità può essere zero (la relazione iniziale ha istanza vuota).

Di primo acchito potrebbe sembrare vero che $|\Pi_{A_i, \dots, A_j}(r(A))| = |r(A)|$ ovvero che il numero delle tuple risultato sia uguale a quello delle tuple dell'operando, ma non è sempre vero! Se proiettiamo verso un attributo che ha ripetizioni, le tuple risultanti non verranno riscritte due volte. Ad esempio:

$\Pi_{\text{Cognome}}(\text{pazienti}) =$	<table border="1"> <thead> <tr> <th>Cognome</th></tr> </thead> <tbody> <tr><td>Necchi</td></tr> <tr><td>Rossigni</td></tr> <tr><td>Missoni</td></tr> <tr><td>Rossetti</td></tr> </tbody> </table>	Cognome	Necchi	Rossigni	Missoni	Rossetti
Cognome						
Necchi						
Rossigni						
Missoni						
Rossetti						
	<p>In questo caso, $\text{pazienti} = 5$ e invece $\Pi_{A_i, \dots, A_j}(\text{pazienti}) = 4$.</p> <p>È quindi chiaro chiaro che l'equivalenza $\Pi_{A_i, \dots, A_j}(r(A)) = r(A)$ può valere <u>solo</u> se A_i, \dots, A_j è superchiave (non necessariamente minima).</p>					

3.3) Esempi

$\Pi_{\text{COD}, \text{Residenza}}(\text{pazienti}) =$	<table border="1"> <thead> <tr> <th>COD</th><th>Residenza</th></tr> </thead> <tbody> <tr><td>A102</td><td>TO</td></tr> <tr><td>B372</td><td>NO</td></tr> <tr><td>B543</td><td>TO</td></tr> <tr><td>B444</td><td>VC</td></tr> <tr><td>S555</td><td>AT</td></tr> </tbody> </table>	COD	Residenza	A102	TO	B372	NO	B543	TO	B444	VC	S555	AT	$\Pi_{\text{COD}, \text{Nome}}(\text{pazienti}) =$	<table border="1"> <thead> <tr> <th>COD</th><th>Nome</th></tr> </thead> <tbody> <tr><td>A102</td><td>Luca</td></tr> <tr><td>B372</td><td>Piero</td></tr> <tr><td>B543</td><td>Nadia</td></tr> <tr><td>B444</td><td>Luigi</td></tr> <tr><td>S555</td><td>Gino</td></tr> </tbody> </table>	COD	Nome	A102	Luca	B372	Piero	B543	Nadia	B444	Luigi	S555	Gino
COD	Residenza																										
A102	TO																										
B372	NO																										
B543	TO																										
B444	VC																										
S555	AT																										
COD	Nome																										
A102	Luca																										
B372	Piero																										
B543	Nadia																										
B444	Luigi																										
S555	Gino																										

Come è chiaro, risultano selezionate solo quelle colonne che appaiono come pedice dell'operando. Le tuple (a meno di equivalenze) sono tutte presenti nella relazione pazienti.

3.4) Composizione fra selezione e proiezione

Vediamo alcuni esempi.

$\Pi_{\text{COD}, \text{Residenza}}(\sigma_{\text{Residenza} = \text{"TO"}}(\text{pazienti})) =$	<table border="1"> <thead> <tr> <th>COD</th><th>Cognome</th><th>Nome</th><th>Residenza</th></tr> </thead> <tbody> <tr><td>A102</td><td>Necchi</td><td>Luca</td><td>TO</td></tr> <tr><td>B543</td><td>Missoni</td><td>Nadia</td><td>TO</td></tr> </tbody> </table>	COD	Cognome	Nome	Residenza	A102	Necchi	Luca	TO	B543	Missoni	Nadia	TO	$=$	<table border="1"> <thead> <tr> <th>COD</th><th>Residenza</th></tr> </thead> <tbody> <tr><td>A102</td><td>TO</td></tr> <tr><td>B543</td><td>TO</td></tr> </tbody> </table>	COD	Residenza	A102	TO	B543	TO
COD	Cognome	Nome	Residenza																		
A102	Necchi	Luca	TO																		
B543	Missoni	Nadia	TO																		
COD	Residenza																				
A102	TO																				
B543	TO																				

Si noti che in questo caso invertendo le due funzioni il risultato è non cambia.

Applicando quindi la regola $\sigma_{\text{Residenza} = \text{"TO"}}(\pi_{\text{COD}, \text{Nome}}(\text{pazienti}))$ otterremmo la stessa relazione risultato.

Ma questa non è una proprietà sempre vera, prendiamo infatti questo esempio:

$$\pi_{\text{COD}, \text{Nome}}(\sigma_{\text{Residenza} = \text{"TO"}}(\text{pazienti})) =$$

$$\pi_{\text{COD}, \text{Nome}}($$

<u>COD</u>	<u>Cognome</u>	<u>Nome</u>	<u>Residenza</u>
A102	Necchi	Luca	TO
B543	Missoni	Nadia	TO

$$) =$$

<u>COD</u>	<u>Nome</u>
A102	Luca
B543	Nadia

Ma invertendo le due operazioni...

$$\sigma_{\text{Residenza} = \text{"TO}}(\pi_{\text{COD}, \text{Nome}}(\text{pazienti})) =$$

$$\sigma_{\text{Residenza} = \text{"TO}}($$

<u>COD</u>	<u>Nome</u>
A102	Luca
B543	Nadia

Problema!

L'attributo 'Residenza' non esiste più nella tabella risultato dopo l'operazione di proiezione!

Risulta quindi chiaro che non sia valida l'equivalenza sopracitata. L'ordine in cui si effettuano le due operazioni è rilevante.

3.5) Gli alberi sintattici

Si noti che è possibile rappresentare le operazioni tramite alberi sintattici, ovvero generando un semplice grafo che ha come nodi gli operandi. Gli archi rappresentano invece le operazioni tra gli operandi.

4) L'operatore base di prodotto cartesiano: \times

Si tratta di un operatore particolare, detto operatore *tecnico*. Questo significa che da solo non è di grande aiuto, ma viene implementato nell'utilizzo del Θ -join. Questo operatore permette di combinare due schemi differenti in uno solo in tutti i modi possibili.

4.1) Condizione sugli operandi

È necessario che $r_1(A)$ e $r_2(B)$ abbiano schemi totalmente disgiunti ovvero $A \cap B = \emptyset$. Questo per motivi di ambiguità.

4.2) Il risultato

Il risultato è una relazione che ha come schema l'unione di A e B (si ricordi che l'ordine non conta) e come istanza la giustapposizione (combinazione) di tutte le tuple di $r_1(A)$ con tutte le tuple di $r_2(B)$.

4.3) Cardinalità del risultato

Data la definizione strutturale si ha che la cardinalità è: $0 \leq |r_1(A) \times r_2(B)| = |r_1(A)| \cdot |r_2(B)|$ ovvero la moltiplicazione delle due singole cardinalità.

4.4) La proprietà commutativa

Si noti che a differenza di quanto studiato per il prodotto cartesiano matematicamente definito, questo prodotto cartesiano gode della proprietà commutativa.

4.5) Esempi

Date le due relazioni $r_1(A_1, A_2, A_3)$ e $r_2(B_1, B_2)$ così definite:

r_1

A₁	A₂	A₃
a _{1,1}	a _{1,2}	a _{1,3}
a _{2,1}	a _{2,2}	a _{2,3}

r_2

B₁	B₂
b _{1,1}	b _{1,2}
b _{2,1}	b _{2,2}
b _{3,1}	b _{3,2}

Il risultato dell'operazione $r_1(A_1, A_2, A_3) \times r_2(B_1, B_2)$ risulta essere:

A₁	A₂	A₃	B₁	B₂
a _{1,1}	a _{1,2}	a _{1,3}	b _{1,1}	b _{1,2}
a _{1,1}	a _{1,2}	a _{1,3}	b _{2,1}	b _{2,2}
a _{1,1}	a _{1,2}	a _{1,3}	b _{3,1}	b _{3,2}
a _{2,1}	a _{2,2}	a _{2,3}	b _{1,1}	b _{1,2}
a _{2,1}	a _{2,2}	a _{2,3}	b _{2,1}	b _{2,2}
a _{2,1}	a _{2,2}	a _{2,3}	b _{3,1}	b _{3,2}

(stesse tuple hanno stessi colori)

Ogni riga di r_1 è ora associata ad ogni riga di r_2 .

5) L'operatore base di unione: U

Questa operazione calcola l'unione insiemistica tra le istanze di due relazioni.

5.1) Condizione sugli operandi

È necessario che $r_1(A)$ e $r_2(A)$ abbiano schemi con gli stessi attributi (in questo caso: A).

5.2) Il risultato

Il risultato dell'operazione di unione ha come schema lo schema A e come istanza tutte le tuple di r_1 e tutte le tuple di r_2 ovviamente senza ripetizioni.

5.3) Cardinalità del risultato

La cardinalità risulta essere $0 \leq |r_1(A) \cup r_2(A)| \leq (|r_1(A)| + |r_2(A)|)$ ovvero la cardinalità del risultato è al massimo la somma delle cardinalità dei due operandi.

5.4) Esempi

Prendiamo due proiezioni (per semplicità) e facciamone l'unione.

Cognome	Nome
Necchi	Luca
Rossigni	Piero
Missoni	Nadia
Missoni	Luigi
Rossetti	Gino
Neri	Piero
Bisi	Mario
Bargio	Sergio
Belli	Nicola
Mizzi	Nicola
Monti	Mario

$\Pi_{\text{Cognome}, \text{Nome}}(\text{pazienti}) \cup \Pi_{\text{Cognome}, \text{Nome}}(\text{medici}) =$

Si noti che in questo caso non essendoci alcuna ripetizione fra i nomi/cognomi dei pazienti e dei medici, l'istanza del risultato ha cardinalità pari alla somma delle cardinalità dei due operandi. In questo caso vi sono stati accorpamenti:

Residenza
TO
NO
VC
AT
MI
AL

$\Pi_{\text{Residenza}}(\text{pazienti}) \cup \Pi_{\text{Residenza}}(\text{medici}) =$

6) L'operatore base di differenza: -

Questa operazione calcola la differenza insiemistica tra le istanze di due relazioni.

6.1) Condizione sugli operandi

È necessario che $r_1(A)$ e $r_2(A)$ abbiano schemi con gli stessi attributi.

6.2) Il risultato

Il risultato dell'operazione di differenza ha come schema lo schema A e come istanza tutte le tuple di r_1 che non appartengono ad r_2 .

6.3) Cardinalità del risultato

La cardinalità risulta essere $0 \leq |r_1(A) - r_2(A)| \leq |r_1(A)|$ ovvero la cardinalità del risultato è al massimo la cardinalità del minuendo.

6.4) Esempi

Prendiamo due proiezioni (per semplicità) e facciamone la differenza.

$$\Pi_{\text{Cognome}, \text{Nome}}(\text{pazienti}) - \Pi_{\text{Cognome}, \text{Nome}}(\text{medici}) =$$

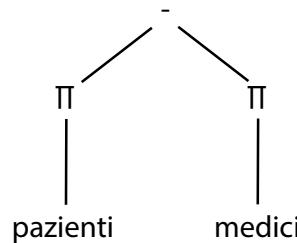
Cognome	Nome
Necchi	Luca
Rossigni	Piero
Missoni	Nadia
Missoni	Luigi
Rossetti	Gino

Non ci sono state sottrazioni poiché nessuna tupla di r_2 appartiene all'istanza di r_1 . Differenti questo caso:

$$\Pi_{\text{Residenza}}(\text{pazienti}) - \Pi_{\text{Residenza}}(\text{medici}) =$$

Residenza
NO

Volendo rappresentare quest'ultima operazione come albero sintattico avremmo:



7) L'operatore base di ridenominazione: ρ

Questo operatore molto semplicemente permette la ridenominazione di alcuni degli attributi (nel risultato!).

Viene utilizzato con l'operatore di Θ -join.

7.1) Il risultato

Il risultato dell'operazione di ridenominazione $\rho_{\beta_1, \beta_2, \dots <- A_1, A_2, \dots} (r(A))$ è identico alla relazione $r(A)$ ma con gli attributi A_1, A_2, \dots rinominati in β_1, β_2, \dots . Si noti che tale modifica avviene solo nella relazione risultato, $r(A)$ rimane quindi invariata (come d'altronde in ogni operazione vista fino ad ora).

7.2) Esempi

CodRep	NomeRep	PrimarioRep
A	Chirurgia	203
B	Pediatria	574
C	Medicina	530

$\rho_{\text{CodRep}, \text{NomeRep}, \text{PrimarioRep} <- \text{COD}, \text{Nome}, \text{Primario}} (\text{reparti}) =$

8) L'operatore derivato di intersezione: \cap

Questa operazione calcola l'intersezione insiemistica tra le istanze di due relazioni. Questo operando è composto poiché possiamo ottenerlo in questo modo: $r_1(A) \cap r_2(A) = r_1(A) - (r_1(A) - r_2(A))$.

8.1) Condizione sugli operandi

È necessario che $r_1(A)$ e $r_2(A)$ abbiano schemi con gli stessi attributi.

8.2) Il risultato

Il risultato dell'operazione di intersezione ha come schema lo schema A e come istanza tutte le tuple appartenenti sia a r_1 che a r_2 (ovvero le tuple che hanno in comune) ovviamente senza ripetizioni.

8.3) Cardinalità del risultato

La cardinalità risulta essere $0 \leq |r_1(A) \cup r_2(A)| \leq \min\{|r_1(A)|, |r_2(A)|\}$ ovvero la cardinalità del risultato è minore o uguale al minimo tra la cardinalità di r_1 ed r_2 .

8.4) Esempi

Prendiamo due proiezioni (per semplicità) e facciamone l'intersezione.

$\Pi_{\text{Residenza}} (\text{pazienti}) \cap \Pi_{\text{Residenza}} (\text{medici}) = \Pi_{\text{Residenza}} (\text{pazienti}) - ($	<table border="1"> <thead> <tr> <th>Residenza</th> </tr> </thead> <tbody> <tr> <td>NO</td> </tr> </tbody> </table>	Residenza	NO	$) =$	<table border="1"> <thead> <tr> <th>Residenza</th> </tr> </thead> <tbody> <tr> <td>TO</td> </tr> <tr> <td>VC</td> </tr> <tr> <td>AT</td> </tr> </tbody> </table>	Residenza	TO	VC	AT
Residenza									
NO									
Residenza									
TO									
VC									
AT									

9) L'operatore derivato di Θ -join: \bowtie

Questo operatore è indubbiamente fra i più importanti di tutta l'algebra relazionale, nonché il più utilizzato. L'operazione di join permette di mettere in comunicazione più relazioni dello stesso database.

Fare un'operazione di join è utile per ottenere una tabella unica contenente però informazioni mantenute da più relazioni.

Ad esempio avendo le relazioni PAZIENTE e RICOVERI potremmo desiderare un risultato che ci dia non solo il ricovero ma ci dica di seguito anche tutte le informazioni relative ai pazienti. Vediamolo meglio con un esempio.

9.1) La derivazione

L'operazione di join è scrivibile in termini di operatori di base in questo modo: $r_1(A) \bowtie_{\Theta} r_2(B) = \sigma_{\Theta}(r_1(A) \times r_2(B))$. Si ha quindi una selezione di un prodotto cartesiano tra i due operandi. La selezione viene effettuata seguendo il predicato Θ .

9.2) Condizione sugli operandi

È necessario che $r_1(A)$ e $r_2(B)$ abbiano schemi totalmente disgiunti ovvero $A \cap B = \emptyset$. Questo per motivi di ambiguità.

9.3) Il predicato

A pedice dell'operazione di Θ -join c'è un predicato Θ che è costituito da una congiunzione (ovvero una serie di AND [\wedge]) di predicati booleani del tipo $A_i \varphi B_j$ dove φ è un confronto e quindi $\varphi \in \{=, \neq, >, \geq, <, \leq\}$, mentre $A_i \in A$ (gli attributi di r_1) e $B_j \in B$ (gli attributi di r_2). Si noti che questo tipo di predicato è uno dei predicati base dell'operatore selezione.

9.4) Il risultato

La relazione ottenuta ha come schema l'unione degli schemi dei due operandi ovvero $A \cup B$. L'istanza è invece costituita dalle sole tuple che rispettano il predicato Θ .

9.5) Cardinalità del risultato

La cardinalità risulta essere la cardinalità delle operazioni da cui il Θ -join è composto, quindi:

$$0 \leq |r_1(A) \bowtie_{\Theta} r_2(B)| = |\sigma_{\Theta}(r_1(A) \times r_2(B))| \leq |r_1(A) \times r_2(B)| = |r_1(A)| \cdot |r_2(B)|.$$

9.6) Esempio di applicazione semplice

Prendiamo l'operazione: ricoveri $\bowtie_{PAZ=COD}$ pazienti la quale affiancherà per ogni PAZ la tupla corrispondente in paziente (guardando quindi COD).

L'operazione equivale a dire $\sigma_{PAZ=COD}$ (ricoveri x paziente). Il risultato è questo:

PAZ	Inizio	Fine	Reparto	COD	Cognome	Nome	Residenza
A102	02/05/94	09/05/94	A	A102	Necchi	Luca	TO
A102	02/12/94	02/01/95	A	A102	Necchi	Luca	TO
S555	05/10/94	03/12/94	B	S555	Rossetti	Gino	AT
B444	01/12/94	02/01/95	B	B444	Missoni	Luigi	VC
S555	05/10/94	01/11/94	A	S555	Rossetti	Gino	AT

A tale risultato si arriva generando prima di tutto la relazione ricoveri x pazienti:

PAZ	Inizio	Fine	Reparto	COD	Cognome	Nome	Residenza
A102	02/05/94	09/05/94	A	A102	Necchi	Luca	TO
A102	02/05/94	09/05/94	A	B372	Rossigni	Piero	NO
A102	02/05/94	09/05/94	A	B543	Missoni	Nadia	TO
A102	02/05/94	09/05/94	A	B444	Missoni	Luigi	VC
A102	02/05/94	09/05/94	A	S555	Rossetti	Gino	AT
A102	02/12/94	02/01/95	A	A102	Necchi	Luca	TO
A102	02/12/94	02/01/95	A	B372	Rossigni	Piero	NO
A102	02/12/94	02/01/95	A	B543	Missoni	Nadia	TO
A102	02/12/94	02/01/95	A	B444	Missoni	Luigi	VC
A102	02/12/94	02/01/95	A	S555	Rossetti	Gino	AT
S555	05/10/94	03/12/94	B	A102	Necchi	Luca	TO
S555	05/10/94	03/12/94	B	B372	Rossigni	Piero	NO
S555	05/10/94	03/12/94	B	B543	Missoni	Nadia	TO
S555	05/10/94	03/12/94	B	B444	Missoni	Luigi	VC
S555	05/10/94	03/12/94	B	S555	Rossetti	Gino	AT
B444	01/12/94	02/01/95	B	A102	Necchi	Luca	TO
B444	01/12/94	02/01/95	B	B372	Rossigni	Piero	NO
B444	01/12/94	02/01/95	B	B543	Missoni	Nadia	TO
B444	01/12/94	02/01/95	B	B444	Missoni	Luigi	VC
B444	01/12/94	02/01/95	B	S555	Rossetti	Gino	AT
S555	05/10/94	01/11/94	A	A102	Necchi	Luca	TO
S555	05/10/94	01/11/94	A	B372	Rossigni	Piero	NO
S555	05/10/94	01/11/94	A	B543	Missoni	Nadia	TO
S555	05/10/94	01/11/94	A	B444	Missoni	Luigi	VC
S555	05/10/94	01/11/94	A	S555	Rossetti	Gino	AT

Le linee più spesse rappresentano il set di "ricoveri" confrontato con tutti i pazienti. Per ogni set, dato che COD è chiave primaria, si è verificata solo una volta la condizione PAZ = COD.

Le linee evidenziate sono le facenti parte del risultato poiché sono le uniche che rispettano il predicato PAZ = COD.

9.7) Esempio di applicazione con ridenominazione

Può accadere che si voglia porre un join tra relazioni che non rispettano il necessario vincolo $A \cap B = \emptyset$.

Risulta quindi necessario adoperare l'operatore ρ di ridenominazione per ottenere due schemi totalmente disgiunti.

Pensiamo ad esempio di voler creare una semplice join tra reparto e medici con predicato "primario = MATR" (in formula $\text{reparti} \bowtie_{\text{primario} = \text{MATR}} \text{medici}$) ha un evidente problema di collisione per quanto concerne l'attributo "Nome" che appare in entrambe le relazioni. È quindi sufficiente applicare la ridenominazione $\rho_{\text{NR} \leftarrow \text{Nome}}(\text{reparti})$ per ovviare al problema, ottenendo così uno schema così costituito:

COD	NR	Primario	MATR	Cognome	Nome	Residenza	Reparto
-----	----	----------	------	---------	------	-----------	---------

L'operazione risulta quindi essere $(\rho_{\text{NR} \leftarrow \text{Nome}}(\text{reparti})) \bowtie_{\text{primario} = \text{MATR}} (\text{medici})$.

La dot-notation

Una tecnica usuale per rinominare gli schemi è quella della dot-notation ovvero si tende a rinominare l'attributo x di una relazione y in $y.x$ (in modo da capire subito di che attributo si stia parlando). Nel caso sopra avremmo: $\rho_{\text{reparti.Nome} \leftarrow \text{Nome}}(\text{reparti})$. Talvolta si tende a sottintendere l'utilizzo dell'operatore ρ , ma il suo utilizzo è esplicito tramite la dot-notation con cui gli attributi vengono utilizzati.

10) Esercizi sull'applicazione degli operandi (visti fin'ora)

10.1) Elencare i reparti diretti da medici che vi lavorano.

Presi tutti i reparti, joiniamo i primari con le matricole dei medici così da ottenere tutte le informazioni che ci servono. Ma facendo attenzione possiamo notare che, aggiungendo al predicato la condizione che il codice del reparto sia uguale a quello in cui il medico lavora, otteniamo direttamente il risultato. Abbiamo quindi:

$$\Pi_{\text{COD}, \text{reparti.nome}, \text{cognome}, \text{name}} ((\text{reparti}) \bowtie_{\text{primario} = \text{MATR} \wedge \text{COD} = \text{reparto}} (\text{medici}))$$

Dove:

$$(\text{reparti}) \bowtie_{\text{primario} = \text{MATR} \wedge \text{COD} = \text{reparto}} (\text{medici}) =$$

COD	reparti.nome	Primario	MATR	Cognome	Nome	Residenza	Reparto
A	Chirurgia	203	203	Neri	Piero	AL	A
B	Pediatria	574	574	Bisi	Mario	MI	B
C	Medicina	530	530	Belli	Nicola	TO	C

E quindi applicando la proiezione $\Pi_{\text{COD}, \text{reparti.nome}, \text{cognome}, \text{name}}$ si ha:

COD	reparti.nome	Cognome	Nome
A	Chirurgia	Neri	Piero
B	Pediatria	Bisi	Mario
C	Medicina	Belli	Nicola

10.2) Elencare i reparti diretti da medici che non vi lavorano.

Il ragionamento è uguale a prima, ma questa volta cerchiamo quelli che sono primari ma non lavorano nel reparto.

$$\Pi_{\text{COD}, \text{reparti.nome}, \text{cognome}, \text{name}} ((\text{reparti}) \bowtie_{\text{primario} = \text{MATR} \wedge \text{COD} \neq \text{reparto}} (\text{medici}))$$

Dove:

$$(\text{reparti}) \bowtie_{\text{primario} = \text{MATR} \wedge \text{COD} \neq \text{reparto}} (\text{medici}) =$$

COD	reparti.nome	Primario	MATR	Cognome	Nome	Residenza	Reparto
∅							

E quindi applicando la proiezione $\Pi_{\text{COD}, \text{reparti.nome}, \text{cognome}, \text{name}}$ si ha:

COD	reparti.nome	Cognome	Nome
∅			

10.3) Ricoveri avvenuti nello stesso giorno del paziente A102

Prima di tutto dobbiamo ottenere tutti i ricoveri del paziente A102 (una semplice selezione). Fatto questo è sufficiente fare una join tra il risultato della selezione e l'intera relazione ricoveri. Notiamo che dopo la selezione ci saranno ovvi problemi di ambiguità quindi esplicitiamo la ρ . Abbiamo:

$$(\rho_{PAZA102, \text{inizioA102}, \text{fineA102}, \text{repartoA102}}(\sigma_{PAZ = A102}(\text{reparti}))) \bowtie_{\text{inizioA102} = \text{inizio}} (\text{reparti})$$

Dove:

$$\rho_{PAZA102, \text{inizioA102}, \text{fineA102}, \text{repartoA102}}(\sigma_{PAZ = A102}(\text{reparti})) =$$

PAZA102	InizioA102	FineA102	RepartoA102
A102	02/05/94	09/05/94	A
A102	02/12/94	02/01/95	A

E quindi joinando $\bowtie_{\text{inizioA102} = \text{inizio}} (\text{reparti})$ si ha:

PAZA102	InizioA102	FineA102	RepartoA102	PAZ	Inizio	Fine	Reparto
A102	02/05/94	09/05/94	A	A102	02/05/94	09/05/94	A
A102	02/12/94	02/01/95	A	A102	02/12/94	02/01/95	A

È chiaramente ridondante l'informazione di A102 che è stato ricoverato lo stesso giorno di se stesso. È quindi saggio apportare una modifica al predicato della join divenendo così $\bowtie_{\text{inizioA102} = \text{inizio} \wedge PAZA102 \neq PAZ}$.

Si applica infine la proiezione $\Pi_{PAZ, \text{inizio}, \text{fine}, \text{reparto}}$ per ottenere il risultato (non ci sono persone che siano state ricoverate lo stesso giorno di A102):

PAZ	Inizio	Fine	Reparto
\emptyset			

10.4) Elencare i primari

È sufficiente dalla tabella dei medici joinare quelle occorrenze dei reparti che hanno primario = MATR, con la debita dot-notation per le ambiguità.

$$\Pi_{MATR, \text{cognome}, \text{medico.nome}, \text{reparto}}((\text{reparti}) \bowtie_{\text{primario} = \text{MATR}} (\text{medici})) =$$

MATR	Cognome	medico.nome	Residenza	Reparto
203	Neri	Piero	AL	A
574	Bisi	Mario	MI	B
530	Belli	Nicola	TO	C

Prendiamo le tabelle:

IMPIEGATI				SUPERVISORI	
MATR	Nome	Età	Stipendio	Capo	Impiegato
100	Mario R.	34	40	210	100
103	Mario B.	23	35	210	103
104	Luigi N.	38	61	210	104
210	Pippo G.	40	50	231	105

10.5) Trovare le MATR dei capi di impiegati che guadagnano più di 40 euro

Prima di tutto selezioniamo dalla relazione IMPIEGATI solo gli impiegati che hanno stipendio maggiore di 40. Dopodiché con un join dai supervisori agli impiegati otteniamo le informazioni sugli impiegati, infine ritagliamo i capi ottenendo l'informazione desiderata. Quindi:

$$\Pi_{\text{Capo}} ((\text{supervisori}) \bowtie_{\text{impiegato} = \text{MATR}} (\sigma_{\text{stipendio} > 40} (\text{impiegati})))$$

Dove:

$$\sigma_{\text{stipendio} > 40} (\text{impiegati}) =$$

MATR	Nome	Età	Stipendio
104	Luigi N.	38	61

E quindi joinando (* è il risultato sopra) (**supervisori**) $\bowtie_{\text{impiegato} = \text{MATR}}$ (*) si ha:

Capo	Impiegato	MATR	Nome	Età	Stipendio
210	104	104	Luigi N.	38	61

E con la proiezione Π_{Capo} :

Capo
210

10.6) Trovare tutte le informazioni dei capi di impiegati che guadagnano più di 40 euro

Partendo dalla tabella (prima della proiezione) dell'esercizio prima, possiamo semplicemente rejoinarla con IMPIEGATI ma questa volta cercando la corrispondenza tra capo e matricola. Si rende necessaria una ridenominazione. Si ha:

$$\Pi_{\text{MCapo}, \text{NCapo}, \text{ECapo}, \text{SCapo}} (\rho_{\text{MCapo}, \text{NCapo}, \text{ECapo}, \text{SCapo} \leftarrow \text{Matr}, \dots} (\text{impiegato})) \bowtie_{\text{MCapo} = \text{Capo}} (*)$$

dove il risultato dell'esercizio precedente (senza la proiezione):

$$(\text{supervisore}) \bowtie_{\text{impiegato} = \text{MATR}} (\sigma_{\text{stipendio} > 40} (\text{impiegati}))$$

è rappresentato con *.

Si ottiene quindi dalla join più esterna:

MCapo	NCapo	ECapo	SCapo	Capo	Impiegato	MATR	Nome	Età	Stipendio
210	Pippo G.	40	50	210	104	104	Luigi N.	38	61

E con la proiezione $\Pi_{MCapo, NCapo, ECapo, SCapo}$:

MCapo	NCapo	ECapo	SCapo
210	Pippo G.	40	50

10.7) Trovare tutte le informazioni dei capi che guadagno meno di almeno uno dei loro subalterni

Prima di tutto è necessario ottenere tutte le informazioni degli impiegati e tutte le informazioni dei capi per poter confrontare gli stipendi. Si tratta quindi di fare due join, la prima per le informazioni sugli impiegati, la seconda per le informazioni sui capi. Ragionando attentamente si nota però che la seconda join (che ci dà le informazioni dei capi) può essere completata con il controllo che lo stipendio del capo sia minore di quello dell'impiegato (già presente nella relazione per via della prima join). Abbiamo (sottintendendo la ρ) quindi:

$$\begin{aligned} & \Pi_{MCapo, NCapo, ECapo, SCapo, Stipendio} (\\ & \quad (\text{impiegato}) \bowtie_{MCapo = \text{capo} \wedge SCapo < \text{Stipendio}} ((\text{supervisori}) \bowtie_{\text{impiegato} = \text{MATR}} (\text{impiegati})) \\ &) \end{aligned}$$

Dove:

$$(\text{supervisori}) \bowtie_{\text{impiegato} = \text{MATR}} (\text{impiegati}) =$$

Capo	Impiegato	MATR	Nome	Età	Stipendio
210	101	100	Mario R.	34	40
210	103	103	Mario B.	23	35
210	104	104	Luigi N.	38	61
231	105

quindi il risultato delle due join è:

MCapo	NCapo	ECapo	SCapo	Capo	Impiegato	MATR	Nome	Età	Stipendio
210	Pippo G.	40	50	210	104	104	Luigi N.	38	61

E con la proiezione $\Pi_{MCapo, NCapo, ECapo, SCapo, Stipendio}$:

MCapo	NCapo	ECapo	SCapo	Stipendio
210	Pippo G.	40	50	61

10.8) Trovare i pazienti che hanno subito almeno due ricoveri

L'algebra relazionale non è aritmetica quindi non è facile contare. Ma è possibile tramite alcuni "trucchetti" contare fino a pochi risultati.

Pensiamo di far joinare con se stessa la tabella ricoveri laddove il paziente è lo stesso ma poi verificare anche che le due date di inizio sulla stessa tupla siano differenti (ovvero è stata trovata una corrispondenza fra i pazienti ma hanno date di inizio diverse quindi vi sono due ricoveri). Una proiezione su PAZ eliminerà i doppi. Si noti che è stata applicata una doppia ridenominazione con dot notation per evitare ambiguità. Abbiamo:

$$\Pi_{R1.PAZ} ((ricoveri_1) \bowtie_{R1.PAZ = R2.PAZ \wedge R1.Inizio \neq R2.Inizio} (ricoveri_2))$$

Dove dalla prima condizione del join ($R1.PAZ = R2.PAZ$) otteniamo:

R1.PAZ	R1.Inizio	R1.Fine	R1.Reparto	R2.PAZ	R2.Inizio	R2.Fine	R2.Reparto
A102	02/05/94	09/05/94	A	A102	02/05/94	09/05/94	A
A102	02/05/94	09/05/94	A	A102	02/12/94	02/01/95	A
A102	02/12/94	02/01/95	A	A102	02/05/94	09/05/94	A
A102	02/12/94	02/01/95	A	A102	02/12/94	02/01/95	A
S555	05/10/94	03/12/94	B	S555	05/10/94	03/12/94	B
S555	05/10/94	03/12/94	B	S555	05/10/94	01/11/94	A
B444	01/12/94	02/01/95	B	B444	01/12/94	02/01/95	B
S555	05/10/94	01/11/94	A	S555	05/10/94	03/12/94	B
S555	05/10/94	01/11/94	A	S555	05/10/94	01/11/94	A

ma poi la seconda esclude tutte le "ripetizioni dirette" della stessa tupla:

R1.PAZ	R1.Inizio	R1.Fine	R1.Reparto	R2.PAZ	R2.Inizio	R2.Fine	R2.Reparto
A102	02/05/94	09/05/94	A	A102	02/12/94	02/01/95	A
A102	02/12/94	02/01/95	A	A102	02/05/94	09/05/94	A

E con la proiezione $\Pi_{R1.PAZ}$:

R1.PAZ
A102

11) Caso particolare del Θ -join: l'equi-join

L'equi-join è un particolare caso del Θ -join, ovvero si ha che la condizione Θ è composta **soltamente da equivalenze**. Il predicato Θ è quindi una *congiunzione di uguaglianze* ovvero in generale $A_i = A_j \wedge \dots$

Si rappresenta come $r_1(A) \bowtie_{\Theta} r_2(B)$.

Il risultato dell'equi-join ha cardinalità $0 \leq |r_1(A) \bowtie_{\Theta} r_2(B)| \leq |r_1(A)| \cdot |r_2(B)|$, ovvero la stessa cardinalità del join normale. Ma vi sono tre tipi particolari di equi-join, vediamoli attentamente.

11.1) 1° sottocaso: l'equi-join completo

Un particolare equi-join è quello completo, che può essere rispetto ad r_1 piuttosto che ad r_2 .

- Un equi-join è completo rispetto ad r_1 quando ogni ennupla di r_1 trova almeno una corrispondenza in r_2 . In questo caso la cardinalità è: $|r_1| \leq |r_1 \bowtie_{\Theta} r_2| \leq |r_1| \cdot |r_2|$
- Un equi-join è completo rispetto ad r_2 quando ogni ennupla di r_2 trova almeno una corrispondenza in r_1 . In questo caso la cardinalità è: $|r_2| \leq |r_1 \bowtie_{\Theta} r_2| \leq |r_1| \cdot |r_2|$
- Un equi-join è completo rispetto ad r_1 ed r_2 quando ogni ennupla di r_1 trova almeno una corrispondenza in r_2 e viceversa. In questo caso la cardinalità è: $\max\{|r_1|, |r_2|\} \leq |r_1 \bowtie_{\Theta} r_2| \leq |r_1| \cdot |r_2|$

11.2) 2° sottocaso: confronto attributi/superchiave

Abbiamo in r_1 un insieme di attributi A' che confrontiamo (tutti) con degli attributi SK che sono in r_2 . SK è superchiave. Si ha quindi che l'equi-join ha come condizioni, dati $r_1(..., A' ...)$ ed $r_2(..., SK, ...)$, una serie di AND fra le equivalenze dei singoli attributi di A' e SK . Si ha quindi che la cardinalità è: $0 \leq |r_1 \bowtie_{\Theta} r_2| \leq |r_1|$. Il motivo è abbastanza intuitivo, infatti quando cerchiamo corrispondenza tra le tuple sotto A' e quelle SK possiamo al massimo trovare un risultato (per via della SK dall'altra parte!) quindi al massimo potremo soddisfare tutte le tuple di r_1 . Potrebbe comunque capitare che qualche tupla di r_1 non abbia corrispondenza in SK e quindi si ha solo un \leq .

11.3) 3° sottocaso: confronto attributi/superchiave completo

Proviamo ora a pensare al caso precedente (abbiamo in r_1 un insieme di attributi A' che confrontiamo (tutti) con degli attributi SK che sono in r_2) ma al posto di SK abbiamo una PK e tra A' e la PK vige un vincolo di integrità referenziale (non esistono valori sotto a A' che non siano anche sotto a PK). In questo caso è chiaro che valgano entrambe le cardinalità dei primi due sottocasi, quindi: $|r_1| \leq |r_1 \bowtie_{\Theta} r_2| \leq |r_1|$ ovvero $|r_1 \bowtie_{\Theta} r_2| = |r_1|$.
Esempio: $|(ricoveri) \bowtie_{PAZ=COD} (pazienti)| = |ricoveri|$

12) Caso particolare dell'equi-join: natural-join

Il natural join si rappresenta così: $r_1 \bowtie r_2$, ovvero non si specifica la Θ , questo perché nel natural-join si effettua un equi-join tra gli attributi che hanno lo stesso nome, non riproducendo però quegli attributi ma "fondendoli" in uno solo. Vediamo più formalmente la questione.

Si ha una $r_1(A)$ e $r_2(B)$ con $A \cap B \neq \emptyset$ ovvero con qualche attributo comune. Si ha quindi che:

$\Pi_{A \cup B}(r_1(A) \bowtie_{r_1.c1=r_2.c1 \wedge \dots} r_2(B)) = r_1(A) \bowtie r_2(B)$. Ovvero si pone l'equivalenza su tutti gli attributi con nome comune e poi si uniscono in proiezione (diventando quindi un tutt'uno).

12.1) Esempi di natural-join

Per illustrare il natural-join modifichiamo leggermente i nomi degli attributi del nostro esempio d'ambito sanitario.

PAZIENTI(COD, Cognome, Nome, Residenza)
 MEDICI(MAT, Cognome, Nome, Residenza, RepAff)
 REPARTO(REP, Nome, Reparto, MAT)
 RICOVERO(COD, Inizio, Fine, REP)

Abbiamo quindi, per esempio un natural join tra pazienti e medici (non ha molto senso!):

$$\begin{aligned}
 & \Pi_{\text{COD}, \text{Cognome}, \text{Nome}, \text{Residenza}, \text{MAT}. \text{RepAff}}(\\
 & \quad (\text{pazienti}) \bowtie_{p.\text{cognome} = m.\text{cognome} \wedge p.\text{nome} = m.\text{nome} \wedge p.\text{residenza} = m.\text{residenza}} (\text{medici}) \\
 &) = (\text{pazienti}) \bowtie (\text{medici})
 \end{aligned}$$

Tale relazione avrà come schema:

<u>COD</u>	<u>Cognome</u>	<u>Nome</u>	<u>Residenza</u>	<u>MATR</u>	<u>Reparto</u>
------------	----------------	-------------	------------------	-------------	----------------

Molto più sensato è l'esempio (ricoveri) \bowtie (pazienti) che presenta come schema:

<u>COD</u>	<u>Inizio</u>	<u>Fine</u>	<u>REP</u>	<u>Cognome</u>	<u>Nome</u>	<u>Residenza</u>
A102	02/05/94	09/05/94	A	Necchi	Luca	TO
A102	02/12/94	02/01/95	A	Rossigni	Piero	NO
S555	05/10/94	03/12/94	B	Missoni	Nadia	TO
B444	01/12/94	02/01/95	B	Missoni	Luigi	VC
S555	05/10/94	01/11/94	A	Rossetti	Gino	AT

che ha quindi fuso COD di ricoveri con COD di pazienti.

12.2) Natural-join degenerato

Data $r_1(A) \bowtie r_2(B)$ con $A \cap B = \emptyset$, allora il natural-join degenera nel rapporto cartesiano $r_1(A) \times r_2(B)$ poiché le uguaglianze non esistono, ovvero il Θ sarebbe vuoto.

13) Caso particolare del Θ -join: il semi-join

Il semi-join è un join particolare che permette di eliminare parti di una relazione sfruttandone un'altra. In formule si traduce come $r_1(A) \bowtie_\Theta r_2(B) = \Pi_A(r_1(A) \bowtie_\Theta r_2(B))$.

13.1) Cardinalità del risultato

La cardinalità del semi-join è $\emptyset \leq |r_1(A) \bowtie_\Theta r_2(B)| \leq |r_1|$ poiché è al massimo la cardinalità dello schema che si considera.

13.2) Esempio del semi-join

L'esempio visto prima (10.4) "Elencare i primari" è un semi-join poiché a noi non interessano le informazioni sui reparti se non l'attributo primario che ci dice a quali medici siamo interessati. Perciò abbiamo:
 $(\text{medici}) \bowtie_{\text{Med.MAT} = \text{Rep.MAT}} (\text{reparti})$.

14) Le interrogazioni negative (ed esempi)

Per ora abbiamo sempre e solo risolto interrogazioni *positive*.

Un'altra classe di interrogazioni è detta "interrogazioni negative", le quali contengono parole chiave come "mai", "non", ecc. Vi sono due tipi di interrogazioni negative, quelle **effettive** e quelle **non effettive**. Le prime sono quelle che sono davvero negative, le seconde sono quelle che invece sono interrogazioni positive "mascherate" da negative, o comunque delle negative riconducibili a positive. Prendiamo ad esempio "i pazienti che non risiedono a Torino", in realtà è scrivibile come "i pazienti che risiedono ovunque tranne che a Torino".

14.1) Lo schema risolutivo delle interrogazioni negative

Una interrogazione negativa si effettua utilizzando un semplice schema, si calcola l'universo (U) che è l'insieme di tutti gli elementi che potrebbero essere introdotti nella soluzione, dopodiché si formula l'interrogazione in modo positivo (P) dopodiché il risultato (R) non è altro che $U - P$.

Da ciò si deduce che U e P debbano essere necessariamente con lo stesso schema (per poter effettuare la differenza).

14.2) Elencare i medici NON primari

- Universo: i medici, ma possiamo trattarli dalla matricola quindi $\Pi_{\text{MATR}}(\text{medici})$
- Formulare P: "Elencare i medici primari", ma avendo definito come universo le matricole dobbiamo trattare con matricole, pertanto $\Pi_{\text{primario}}(\text{reparti})$
- $R = U - P = \Pi_{\text{MATR}}(\text{medici}) - (\rho_{\text{MATR}<\text{-primario}}(\Pi_{\text{primario}}(\text{reparti})))$, ed è l'elenco dei primari in matricola.

Dopodiché si effettua $R \bowtie \text{medici}$ per ottenere le informazioni necessarie dalla relazione medici.

È indispensabile che U e P abbiano lo stesso schema!

Avremmo potuto anche affrontare il problema diversamente, considerando un universo più ampio.

- Universo: i medici, quindi (**medici**)
- Formulare P: "Elencare i medici primari", ma avendo definito come universo tutto lo schema medici dobbiamo ottenere un risultato con lo schema uguale a medici. $\Pi_{\text{Attributi_di_medico}}(\text{reparti}) \bowtie_{\text{primario} = \text{MATR}} (\text{medici})$
- $R = U - P = (\text{medici}) - \Pi_{\text{Attributi_di_medico}}(\text{reparti}) \bowtie_{\text{primario} = \text{MATR}} (\text{medici})$, ed è l'elenco dei primari con già tutte le informazioni.

È quindi irrilevante l'approccio poiché il numero di operazioni da fare è sempre lo stesso.

14.3) Elencare i pazienti NON residenti in città in cui risiede qualche medico

- Universo: i pazienti, quindi (**pazienti**)
- Formulare P: "Elencare i pazienti residenti in città in cui risiede qualche medico", ed il risultato deve essere con lo schema di pazienti. $\Pi_{\text{Attributi_di_pazienti}}(\text{pazienti}) \bowtie_{\text{paz.res} = \text{med.res}} (\text{medici})$
- $R = U - P = (\text{pazienti}) - \Pi_{\text{Attributi_di_pazienti}}(\text{pazienti}) \bowtie_{\text{paz.res} = \text{med.res}} (\text{medici})$, ed è l'elenco dei pazienti in questione con già tutte le informazioni.

14.4) Elencare i pazienti MAI ricoverati nel reparto A

- Universo: i pazienti, quindi (pazienti)
- Formulare P: "Elencare i pazienti ricoverati almeno una volta nel reparto A", ed il risultato deve essere con lo schema di pazienti. $\prod_{\text{Attributi_di_pazienti}} (\sigma_{\text{reparto} = 'A'}(\text{ricoveri})) \bowtie_{\text{PAZ} = \text{COD}} (\text{pazienti})$
- $R = U - P = (\text{pazienti}) - \prod_{\text{Attributi_di_pazienti}} (\sigma_{\text{reparto} = 'A'}(\text{ricoveri})) \bowtie_{\text{PAZ} = \text{COD}} (\text{pazienti})$, ed è l'elenco dei pazienti in questione con già tutte le informazioni.

15) L'operatore derivato di quoziente: \div

Il quoziente è un operatore che è utile per risolvere query che richiedono **"per ogni"** ecc. Il quoziente è così definito:

$$R(A,B) \div S(B).$$

Per arrivare al quoziente utilizziamo prima un esempio.

15.1) Arrivare al quoziente empiricamente

Prendiamo questi due schemi:

E	P
MATR	Corso
400	Database
500	Linguaggi
500	Analisi
300	Database
500	Database
400	Linguaggi

La tabella E (Esami) contiene le matricole degli studenti che hanno passato un certo esame.
La tabella P elenca la lista dei corsi.
Avendo passato tutti i corsi si può andare alla laurea.

Elencare gli studenti che hanno superato tutti gli esami

Come potremmo agire? Proviamo a pensare di creare una relazione dove tutti gli studenti hanno fatto tutti gli esami e da quella sottraiamo le informazioni della relazione E. A questo punto abbiamo in mano gli studenti che devono ancora dare degli esami. Non resta che togliere dalla lista di tutti gli studenti quest'ultimo risultato per ottenere tutti gli studenti che hanno superato tutti gli esami. Abbiamo quindi: $\prod_{\text{MATR}}(E) - \prod_{\text{MATR}}(\prod_{\text{MATR}}(E) \times \prod_{\text{corso}}(P) - E)$ dove:

$\prod_{\text{MATR}}(E) \times \prod_{\text{corso}}(P) - E$ (tutti gli studenti con tutti gli esami meno gli esami effettivamente dati dagli studenti, ci dà gli esami che gli studenti devono ancora dare):

MATR	Corso
400	Analisi
300	Analisi
300	Linguaggi

Poi si riprende E e si proietta su MATR (per ottenere l'insieme di tutti gli studenti) e si sottrae la relazione sopra (anch'essa proiettata su MATR chiaramente), si ha quindi tutti gli studenti meno quelli che devono ancora dare qualche esame:

MATR
400
500
300

Il risultato è chiaramente

MATR
500

che è effettivamente l'unico studente che ha passato tutti gli esami.

Possiamo quindi trarre questa regola generale:

15.2) Definizione formale di quoziente

$$R(A,B) \div S(B) = \Pi_A(R) - \Pi_A((\Pi_A(R) \times \Pi_B(S)) - R)$$

15.3) Esempio di quoziente

Prendiamo questo esempio.

Pazienti che sono stati ricoverati in TUTTI i reparti

$R(A,B) =$ è la coppia (PAZ, Reparto), ovvero: $\Pi_{PAZ, Reparto}(\text{ricoveri})$

$R(A,B) =$ è l'elenco dei codici dei reparti ovvero: $\Pi_{COD}(\text{reparti})$

perciò la relazione diventa (con la ridenominazione di COD in reparto da fare!):

$\Pi_{PAZ}(\text{ricoveri}) - \Pi_{PAZ}((\Pi_{PAZ}(\text{ricoveri}) \times \Pi_{COD}(\text{reparti})) - \Pi_{PAZ, Reparto}(\text{ricoveri}))$ dove:

$\Pi_{PAZ}(\text{ricoveri}) \times \Pi_{COD}(\text{reparti}) - \Pi_{PAZ, Reparto}(\text{ricoveri})$ (tutti i pazienti ricoverati in tutti i reparti meno quelli effettivamente ricoverati nei rispettivi reparti che ci da quelli che tutti i reparti nei quali un certo paziente non è stato ricoverato):

PAZ	Reparto		PAZ	Reparto		PAZ	Reparto
A102	A	-	A102	A	=	A102	B
A102	B		S555	B		A102	C
A102	C		B444	B		S555	C
S555	A		S555	A		B444	A
S555	B					B444	C
S555	C						
B444	A						
B444	B						
B444	C						

Sottraiamo quindi la nostra tabella proiettata su pazienti da quel risultato sopra (ovvero tutti i reparti nei quali un certo paziente non è stato ricoverato) sempre proiettato su pazienti, in sostanza sottraiamo da tutti i pazienti quei pazienti che sono non sono stati in almeno un reparto (altrimenti non sarebbero nel risultato sopra!):

PAZ	PAZ		PAZ
A102	A102	-	\emptyset
S555	S555	=	
B444	B444		

Effettivamente non ci sono pazienti che siano stati ricoverati in tutti i reparti.

Si noti che nell'ultima interrogazione c'era una **informazione implicita** ovvero si dava per scontato che ogni reparto fosse luogo di ricovero. Se questo dovesse non valere (ad esempio se nei reparti avessimo Laboratorio e Radiografia, che sono chiaramente reparti dove non avvengono ricoveri) allora l'interrogazione sarebbe equivalsa: $\Pi_{PAZ, Reparti}(ricoveri) \div (\rho_{Reparto < COD}(\Pi_{COD}(reparti))) = \emptyset$ poiché nessun paziente sarebbe stato mai ricoverato in laboratorio e radiografia.

In questi casi, potrebbe essere meglio utilizzare come $S(B) \Pi_{Reparti}(ricoveri)$ che non potrà che contenere reparti utilizzati (da almeno un paziente). Questa scelta potrebbe però essere problematica se ci fossero reparti senza neanche un paziente (quindi non inclusi nella proiezione di reparti su ricoveri).

16) Semantica del valore NULL

Abbiamo ragionato finora ignorando i possibili valori nulli contenuti nelle tuple. Codd sviluppa però alcune regole per poterli considerare. Si ricorda che il concetto di valore nullo è che il valore è **sconosciuto** ovvero c'è mancanza di informazione piuttosto che il valore è inesistente. Quando c'è un valore nullo, siamo nel caso in cui il confronto è del tipo $A_i = \text{costante}$ dove la costante è, per l'appunto, NULL.

16.1) La logica a tre valori

Soltamente nelle espressioni booleane si applicano due soli valori T (true) e F (false). Ma in questo caso, con il NULL, dobbiamo introdurre un terzo valore, U (unknown). Una espressione vale U se (con Θ un qualsiasi confronto):

- $t[A_i] \Theta \text{cost} = U$ se $t[A_i]$ è nullo.
- $t[A_i] \Theta t[A_j] = U$ se $t[A_i]$ oppure $t[A_j]$ sono nulli (quindi anche se entrambi lo sono).

Come ben sappiamo, però, le espressioni booleane non sono sempre sole ma sono a loro volta rapportate con i classici operatori booleani. Vediamo quindi le tabelle dei vari operatori.

AND	F	U	T
F	F	F	F
U	F	U	U
T	F	U	T

OR	F	U	T
F	F	U	T
U	U	U	T
T	T	T	T

NOT	-
F	T
U	U
T	F

Una volta ottenuto il risultato finale dell'espressione booleana totale dell'operando, allora si applica la funzione detta di **collasso** $C(P(T))$ (dove $P(T)$ è il risultato dell'espressione booleana) così definita:

P(T)	C(P(T))
F	F
U	F
T	T

La funzione di collasso ci dice semplicemente che se la condizione risulta essere U, allora non va inclusa nel risultato (ovvero il risultato è F).

16.2) Due nuovi predicati

Talvolta può essere utile estrapolare tutte le tuple che hanno un certo attributo a NULL oppure tutte quelle che non lo hanno. Nascono così i due predicati **A_i IS NULL** e **A_i IS NOT NULL**.

16.3) Esempio di query con valori nulli

Modifichiamo la tabella ricoveri

PAZ	Inizio	Fine	Reparto
A102	02/05/94	09/05/94	A
A102	02/12/94	02/01/95	-
S555	05/10/94	03/12/94	B
B444	01/12/94	02/01/95	-
S555	05/10/94	01/11/94	A

Elencare i diversi pazienti che sono stati ricoverati ricoverati nello stesso reparto

La query è semplicemente:

(ricoveri) $\bowtie_{PAZ1 \neq PAZ2 \wedge REP1=REP2}$ (ricoveri)

ed il risultato è:

PAZ1	Inizio1	Fine1	Reparto1	PAZ2	Inizio2	Fine2	Reparto2
A102	02/05/94	09/05/94	A	S555	05/10/94	01/11/94	A
S555	05/10/94	01/11/94	A	A102	02/05/94	09/05/94	A

17) Caso particolare del Θ -join: l'outer-join

L'ultima applicazione del join è l'outer join. Si tratta di un join normale, ma vengono incluse tutte le tuple delle relazioni a sinistra o destra o entrambe coinvolte nel join. Se una tupla dovesse non trovare corrispondenza allora verranno aggiunti i valori NULL a tutti gli altri attributi.

Esistono tre tipi di outer-join. Se vogliamo mantenere la relazione che si trova a sinistra del simbolo avremo un **left-join**, se vogliamo mantenere la relazione che si trova a destra del simbolo avremo un **right-join**, se invece vogliamo mantenerle entrambe avremo un **full-join**. Vediamoli empiricamente con tre immediati esempi.

Consideriamo alcune tabelle modificate (per rendere sensato l'esempio):

RICOVERI				REPARTI		
PAZ	Inizio	Fine	Reparto	COD	Nome	Primario
A102	02/05/94	09/05/94	A	A	Chirurgia	203
A102	02/12/94	02/01/95	P	B	Pediatria	574
B444	01/12/94	02/01/95	B	C	Medicina	530
S555	05/10/94	01/11/94	A	L	Laboratorio	461
B372	02/10/94	02/10/94	D			

Dove il reparto D nella tabella RICOVERI rappresenta il Day-Hospital e P il reparto di Pronto-Soccorso (che non sono reparti veri e propri quindi non hanno record nella relazione REPARTI)

17.1) Il left-join

Con il left-join includiamo nel risultato tutta la relazione di sinistra con le tuple associate se presenti, e laddove il join non trovi corrispondenza si aggiungono valori nulli. Pertanto, (reparti) $\bowtie_L COD = \text{Reparto} \text{ (ricoveri)}$ risulta:

PAZ	Inizio	Fine	Reparto	COD	Nome	Primario
A102	02/05/94	09/05/94	A	A	Chirurgia	203
S555	05/10/94	01/11/94	A	A	Chirurgia	203
B444	01/12/94	02/01/95	B	B	Pediatria	574
A102	02/12/94	02/01/95	P	-	-	-
B372	02/10/94	02/10/94	D	-	-	-

17.2) Il right-join

Con il right-join includiamo nel risultato tutta la relazione di destra con le tuple associate se presenti, e laddove il join non trovi corrispondenza si aggiungono valori nulli. Pertanto, (reparti) $\bowtie_R COD = \text{Reparto} \text{ (ricoveri)}$ risulta:

PAZ	Inizio	Fine	Reparto	COD	Nome	Primario
A102	02/05/94	09/05/94	A	A	Chirurgia	203
S555	05/10/94	01/11/94	A	A	Chirurgia	203
B444	01/12/94	02/01/95	B	B	Pediatria	574
-	-	-	-	C	Medicina	530
-	-	-	-	L	Laboratorio	461

17.3) Il full-join

Con il right-join includiamo nel risultato tutta la relazione di sinistra con le tuple associate se presenti, e tutta la relazione di destra con le tuple associate se presente. Laddove il join non trovi corrispondenza si aggiungono valori nulli. Pertanto, (reparti) $\bowtie_F COD = \text{Reparto} \text{ (ricoveri)}$ risulta:

PAZ	Inizio	Fine	Reparto	COD	Nome	Primario
A102	02/05/94	09/05/94	A	A	Chirurgia	203
S555	05/10/94	01/11/94	A	A	Chirurgia	203
B444	01/12/94	02/01/95	B	B	Pediatria	574
A102	02/12/94	02/01/95	P	-	-	-
B372	02/10/94	02/10/94	D	-	-	-
-	-	-	-	C	Medicina	530
-	-	-	-	L	Laboratorio	461

18) Ulteriori esempi

18.1) Trovare gli autori che hanno pubblicato solo in collaborazione

Consideriamo questa relazione F(A,P):

A	P
a ₁	p ₁
a ₂	p ₁
a ₃	p ₂
a ₁	p ₃

A è una lista di autori e P è una lista di pubblicazioni. La relazione indica quali autori hanno lavorato a quali pubblicazioni.

Potremmo pensare di estrapolare quali pubblicazioni sono state fatte da più autori. Per fare questo abbiamo la query: $\prod_{F1.A, F1.P} ((F1) \bowtie_{F1.P = F2.P \wedge F1.A \neq F2.A} (F2))$ che da come risultato:

F1.A	F1.P	F2.A	F2.P		F1.A	F1.P
a ₁	p ₁	a ₂	p ₁	$\prod \rightarrow$	a ₁	p ₁
a ₂	p ₁	a ₁	p ₁		a ₂	p ₁

Ora, prendendo l'elenco delle nostre pubblicazioni (sostanzialmente la tabella F) possiamo sottrarre il risultato ottenuto sopra ottenendo così le pubblicazioni firmate da un unico autore (tutte le pubblicazioni meno quelle da più autori). Applicando quindi una proiezione sugli autori otteniamo quegli autori che hanno pubblicato anche singolarmente. Abbiamo perciò (con * il risultato sopra) $\prod_{F1.A} (F) - *$ che da come risultato:

F1.A	F2.P	-	F1.A	F1.P	=	F1.A	F2.P	$\prod \rightarrow$	F1.A
a ₁	p ₁	-	a ₁	p ₁	=	a ₃	p ₂		a ₃
a ₂	p ₁		a ₂	p ₁		a ₁	p ₃		a ₁
a ₃	p ₂								
a ₁	p ₃								

È ora banalmente sufficiente sottrarre da tutti gli autori (nuovamente la tabella F proiettata su A) quegli autori che hanno pubblicato anche singolarmente. Abbiamo perciò (con * il risultato sopra) $\prod_A (F) - *$ che da:

A	-	A	=	A
a ₁	-	a ₃	=	
a ₂		a ₁		
a ₃				

Quindi a₂ ha pubblicato *soltanto* in collaborazione.

La query totale risulta essere: $\prod_A (F) - (\prod_{F1.A} (F - (\prod_{F1.A, F1.P} ((F1) \bowtie_{F1.P = F2.P \wedge F1.A \neq F2.A} (F2)))))$

Basi di dati

Ottimizzazione delle operazioni

Capitolo 3

Enrico Mensa

Indice degli argomenti

1) Regole di equivalenza tra operatori	1
1.1) Atomizzazione delle selezioni	
1.2) Idempotenza delle proiezioni	
1.3) Anticipazione della selezione rispetto al join	
1.4) Anticipazione della proiezione rispetto al join	
1.5) Inglobamento di una selezione in un prodotto cartesiano a formare un join	
1.6) Distributività della selezione rispetto all'unione	
1.7) Distributività della selezione rispetto alla differenza	
1.8) Distributività della proiezione rispetto all'unione	
1.9) Regole di corrispondenza tra operatori insiemistici e condizioni di selezione complesse	
1.10) Proprietà distributiva del join rispetto all'unione	
2) L'ottimizzazione del DBMS	2
2.1) Algoritmo di ottimizzazione logica	
3) Analisi dei costi	4
3.1) Modello di costo della selezione	
3.2) Stima della cardinalità dell'equi-join	
4) La minimizzazione dei costi	6
4.1) Calcolo del costo senza ottimizzazione	
4.2) Calcolo del costo con ottimizzazione (senza il 7° passo)	
4.3) Spiegazione del 7° passo dell'algoritmo di ottimizzazione logica	
4.4) Calcolo del costo con ottimizzazione (con il 7° passo)	
4.5) Calcolo di VAL con selezione nei casi di uguaglianza tra attributo e predicato	

1) Regole di equivalenza tra operatori

Il DBMS utilizza alcune regole (derivanti dall'algebra stessa) per poter invertire/scambiare/scomporre gli operatori ed ottimizzare così l'esecuzione delle query. Vediamole raggruppate per operatore.

1.1) Atomizzazione delle selezioni

$\sigma_{F1 \wedge F2}(r) \equiv \sigma_{F1}(\sigma_{F2}(r))$ ovvero l'AND tra due condizioni di selezione è la selezione del primo sulla selezione del secondo (e viceversa chiaramente).

1.2) Idempotenza delle proiezioni

$\Pi_X(\Pi_{X,Y}(r) \equiv \Pi_X(r)$ ovvero una proiezione fatta prima su X e Y e poi su X è tranquillamente sostituibile da una sola su X.

1.3) Anticipazione della selezione rispetto al join

$\sigma_F(r_1 \bowtie r_2) \equiv (\sigma_F(r_1)) \bowtie r_2$ se F fa riferimento solo ad attributi di r_1 . È possibile anche la versione su r_2 .

1.4) Anticipazione della proiezione rispetto al join

Avendo due relazioni: r_1 definito su A e r_2 definito su B; se Y è incluso in B e se Y è incluso nell'intersezione fra A e B (Y è quindi tra gli attributi in comune di r_1 e r_2) (in sostanza se gli attributi (B - Y) non sono coinvolti nel join) allora vale:

$$\Pi_{A,Y}(r_1 \bowtie r_2) \equiv r_1 \bowtie (\Pi_Y(r_2))$$

Combinando questa regola con la 1.2 possiamo ottenere:

$$\Pi_Y(r_1 \bowtie r_2) \equiv \Pi_Y(\Pi_{Y1}(r_1) \bowtie_F (\Pi_{Y2}(r_2)))$$

In sostanza possiamo eliminare subito tutti quegli attributi che non fanno parte del join e non saranno inclusi nel risultato finale (per evitare di appesantire il join).

Esempio

Abbiamo le due relazioni:

IMPIEGATI

MATR	Nome	Età	Stipendio
100	Mario R.	34	40
103	Mario B.	23	35
104	Luigi N.	38	61
210	Pippo G.	40	50

SUPERVISORI

Capo	Impiegato
210	100
210	103
210	104
231	105

Avendo la seguente query: $\Pi_{Capo}(\sigma_{Eta < 30}(Impiegati) \bowtie_{Matr = Imp} (Supervisori))$ possiamo usare la regola sopra per eliminare dal primo argomento del join (Impiegati) gli attributi non necessari ottenendo così:

$$\Pi_{Capo}(\Pi_{Matr}(\sigma_{Eta < 30}(Impiegati)) \bowtie_{Matr = Imp} (Supervisori)).$$

In effetti non avrebbe senso fare il join usando tutti gli attributi di Impiegati quando a noi serve solo MATR (si noti però gli altri attributi verranno successivamente eliminati dalla proiezione su Capo, ed è quindi per questo che possiamo "anticiparne" la cancellazione).

1.5) Inglobamento di una selezione in un prodotto cartesiano a formare un join

Si basa sulla definizione del join per cui un join vuoto è un join che è solo un rapporto cartesiano, si ha quindi:

$$\sigma_F(r_1 \bowtie r_2) \equiv r_1 \bowtie_F r_2$$

1.6) Distributività della selezione rispetto all'unione

$\sigma_F(r_1 \cup r_2) \equiv \sigma_F(r_1) \cup \sigma_F(r_2)$ ovvero la selezione dell'unione di due relazioni è l'unione delle selezioni delle singole relazioni.

1.7) Distributività della selezione rispetto alla differenza

$\sigma_F(r_1 - r_2) \equiv \sigma_F(r_1) - \sigma_F(r_2)$ ovvero la selezione della differenza di due relazioni è la differenza delle selezioni delle singole relazioni.

1.8) Distributività della proiezione rispetto all'unione

$\Pi_F(r_1 \cup r_2) \equiv \Pi_F(r_1) \cup \Pi_F(r_2)$ ovvero la proiezione dell'unione di due relazione è l'unione delle proiezioni delle singole relazioni.

1.9) Regole di corrispondenza tra operatori insiemistici e condizioni di selezione complesse

- $\sigma_{F \vee G}(r) \equiv \sigma_F(r) \cup \sigma_G(r)$
- $\sigma_{F \wedge G}(r) \equiv \sigma_F(r) \cap \sigma_G(r) \equiv \sigma_F(r) \bowtie \sigma_G(r)$
- $\sigma_{F \wedge \neg G}(r) \equiv \sigma_F(r) - \sigma_G(r)$

1.10) Proprietà distributiva del join rispetto all'unione

$$r \bowtie (r_1 \cup r_2) \equiv (r \bowtie r_1) \bowtie (r \bowtie r_2)$$

Si ricordano poi le varie proprietà commutative di tutti gli operatori binari tranne la differenza.

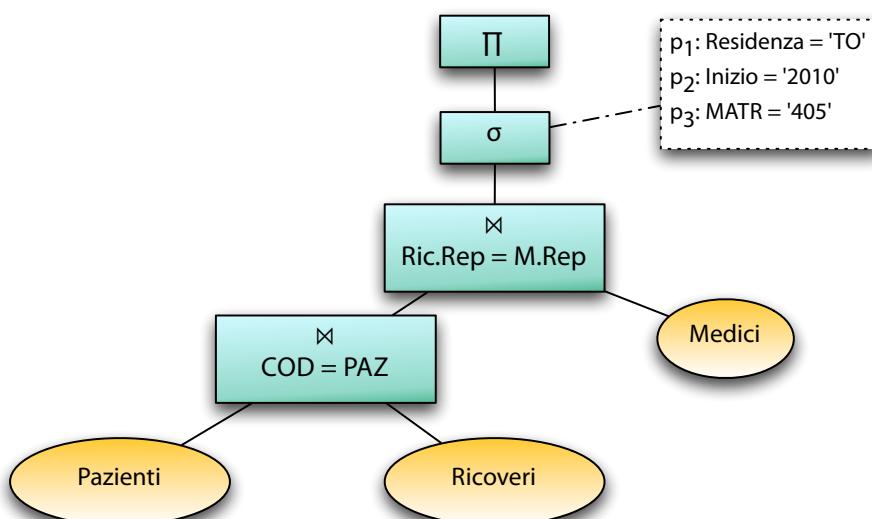
2) L'ottimizzazione del DBMS

Il DBMS effettua delle operazioni effettive sfruttando le regole sopra per poter migliorare le prestazioni delle query.

Avviene questo passaggio *interrogazione -> ottimizzazione logica -> ottimizzazione fisica*.

Possiamo visualizzare l'interrogazione con il suo albero per poterla "lavorare".

$$\Pi_{...} (\sigma_{\text{res} = 'TO' \wedge \text{Inizio} = '2010' \wedge \text{Matr} = '405'} ((\text{pazienti} \bowtie_{\text{COD} = \text{PAZ}} \text{ricoveri}) \bowtie_{\text{Ric.Rep} = \text{Med.rep}} (\text{medici}))$$



Diciamo per ipotesi che le tuple della relazione pazienti e reparti siano 10^5 e quelle della relazione medici siano 10^2 .

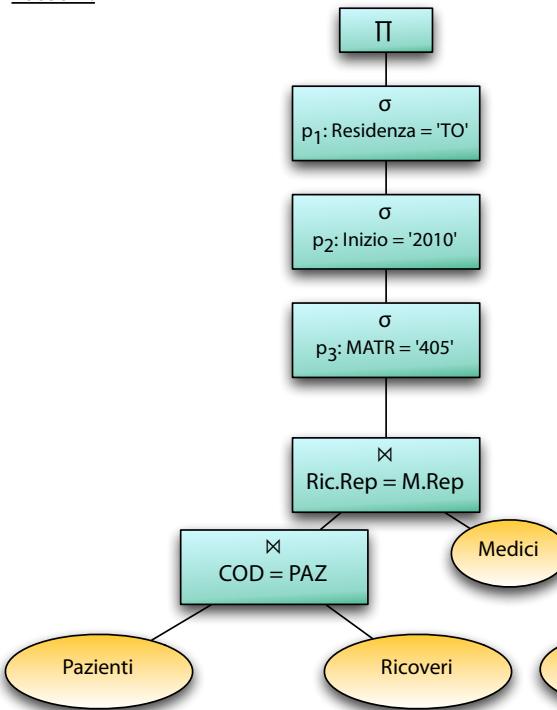
2.1) Algoritmo di ottimizzazione logica

L'algoritmo di ottimizzazione logica perpetuato dal DBMS segue questi punti (con una proposizione di selezione in forma congiuntiva):

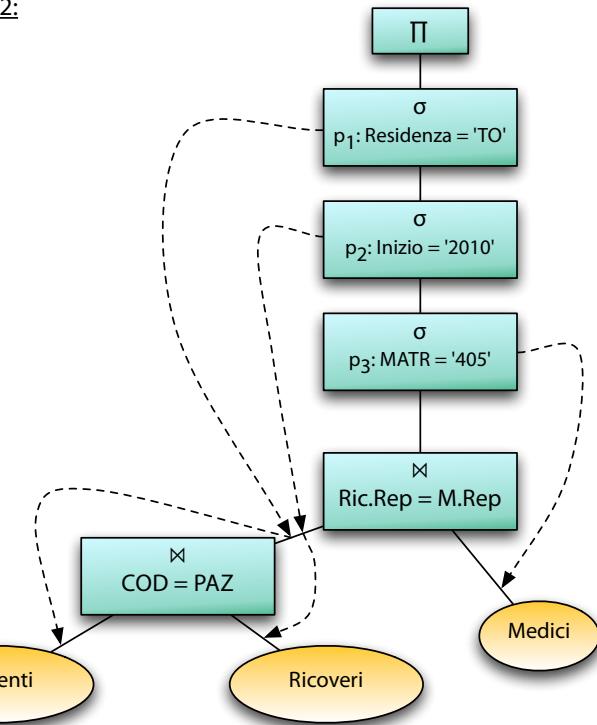
- 1) Decomposizione delle selezioni (regola: $\sigma_{F1} \wedge F2(r) \equiv \sigma_{F1}(\sigma_{F2}(r))$)
- 2) Spostamento delle selezioni verso le foglie fino a che è possibile impiegando le proprietà distributive della selezione
- 3) Spostare le proiezioni verso le foglie fino a quando è possibile utilizzando la regola distributiva della proiezione
- 4) Ricomporre le selezioni (usando l'inverso della regola di prima)
- 5) Accoppare le proiezioni (regola: $\Pi_X(\Pi_{X,Y}(r) \equiv \Pi_X(r))$)
- 6) Riconoscimento del join (applicando la definizione base del join)
- 7) Utilizzare le proprietà associative degli operatori binari cercando la configurazione ottimale dell'albero di parsificazione.

Si lasci il punto 7 a spiegazioni successive e vediamo una applicazione dell'algoritmo sull'albero di prima.

Passo 1:



Passo 2:



Spiegazione del passo 2:

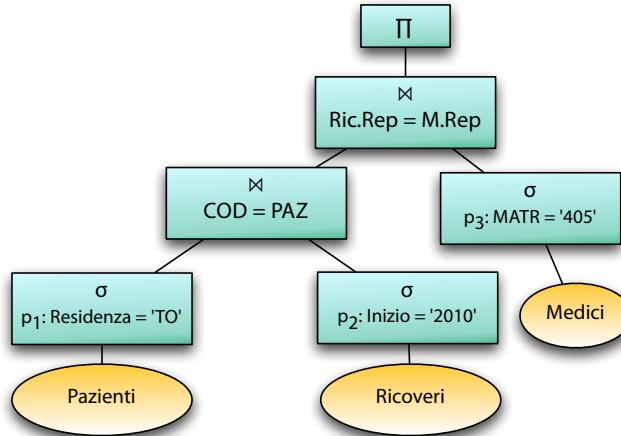
Durante lo spostamento delle selezioni verso il basso si è utilizzata la regola che se $\sigma_p \subseteq A$ nella query $\sigma_p(r(A) \bowtie S(B))$ allora vale la regola $\sigma_p(r(A) \bowtie S(B)) \equiv \sigma_p(r(A)) \bowtie S(B)$ (regola 1.3).

In pratica la selezione P3 (Matr = '405') vede coinvolto solo MATR che è incluso in MEDICI, quindi vale $\sigma_p \subseteq B$ e allora possiamo far scendere la selezione perché vale la regola $\sigma_p(r(A) \bowtie S(B)) \equiv r(A) \bowtie \sigma_p(S(B))$.

Essendo scesa tra il join e medici la selezione P3, la selezione P2 è alle porte del join. In questo caso vale $\sigma_p \subseteq A$ poiché "Inizio" è incluso in A (che in questo caso sono tutti gli attributi da pazienti e reparti). Allora possiamo far scendere a sinistra e frapporre la selezione tra i due join. A questo punto dobbiamo ancora chiederci se P2 può scendere, ed è così poiché "Inizio" è incluso in RICOVERI e vale quindi $\sigma_p \subseteq B$. A questo punto P2 scende a destra.

Non resta che esaminare P1, per il quale vale prima $\sigma_p \subseteq A$ proprio come per P2 (infatti Res è incluso nel join tra PAZIENTI e RICOVERI) e poi vale $\sigma_p \subseteq A$ (Res appartiene a PAZIENTI) e quindi scende a sinistra due volte.

Per questo il risultato finale è:



Nei passi successivi si dovrebbero semplificare le produzioni, quindi ricomporre le selezioni, ecc. ma in questo caso non c'è il problema perché non ci sono operazioni di questo genere da svolgere.

Si noti che il passo 6 è utile perché il DBMS ha algoritmi specifici per l'esecuzione ottimale del join.

3) Analisi dei costi

Per analizzare i costi il DBMS utilizza funzioni statistiche per effettuare i conti. Esaminiamole:

- **CARD(t)**: restituisce la cardinalità della relazione t.
- **VAL(A_i, t)**: restituisce il numero di valori distinti assegnati all'attributo A_i nella relazione t. Se ne deduce che:
 - $VAL(A_i, t) = CARD(\Pi_{A_i}(t))$
 - $VAL(A_i, t) \leq CARD(t)$
 - $VAL(A_i, t) = CARD(t)$ se A_i è chiave primaria
- **MAX(A_i, t)**: restituisce il valore massimo assegnato all'attributo A_i nella relazione t.
- **MIN(A_i, t)**: restituisce il valore minimo assegnato all'attributo A_i nella relazione t.

3.1) Modello di costo della selezione

Data la cardinalità $0 \leq |\sigma_p(r)| \leq |r|$ si ha che il modello di costo della selezione è $|\sigma_p(r)| = f_p \cdot |r|$ dove f_p è il fattore di selezione del predicato p, pertanto f_p è compreso fra zero e uno poiché esprime una probabilità. Si tratta di una percentuale stimata di quanto il predicato p diverrà vero. Si guardi ora la tabella delle corrispondenze predicato/f_p supponendo che il confronto sia di tipo A_i Θ cost.

Si ipotizza, nel creare la tabella, che la distribuzione sia uniforme. Si adotta quindi la tecnica casi favorevoli su casi possibili.

p	Predicato	f_p
p ₁	A _i = v	1/VAL(A _i , t)
p ₂	A _i ≤ v	(v - MIN(A _i , t)) / (MAX(A _i , t) - MIN(A _i , t))
p ₃	A _i ≥ v	(MAX(A _i , t) - v) / (MAX(A _i , t) - MIN(A _i , t))
p ₄	v ₁ ≤ A _i ≤ v ₂	(v ₁ - v ₂) / (MAX(A _i , t) - MIN(A _i , t))

Chiaramente ogni singola condizione può ancora essere messa in AND, OR oppure NOT. Quindi, ricordando l'ipotesi brutale degli eventi indipendenti:

Predicato	f_p
$P_1 \wedge P_2 \wedge P_3 \wedge \dots \wedge P_k$	$f_p = f_{p1} \cdot f_{p2} \cdot f_{p3} \cdot \dots \cdot f_{pk}$
$P_1 \vee P_2$	$f_p = f_{p1} + f_{p2} - (f_{p1} \cdot f_{p2})$
$\neg P_1$	$f_p = 1 - f_{p1}$

Si noti che applicando *De Morgan* possiamo ottenere la formula generica per l'or, infatti:

$$P_1 \vee P_2 \vee P_3 \vee \dots \vee P_k = \neg(\neg P_1 \wedge \neg P_2 \wedge \neg P_3 \wedge \dots \wedge \neg P_k)$$

e quindi

$$f_p = 1 - ((1 - f_{p1}) \cdot (1 - f_{p2}) \cdot (1 - f_{p3}) \cdot \dots \cdot (1 - f_{pk}))$$

3.2) Stima della cardinalità dell'equi-join

Quanto vale la cardinalità stimata dell'equi-join? in simboli: $| r_1(A) \bowtie_{A_i=B_j} r_2(B) | = ?$

Pensiamo di prendere una tupla da A_i ed associarla con tutte le tuple di B_j , questo significa praticamente $A_i = v$ (una tupla specifica) che abbiamo detto valere $1/\text{VAL}(B_j, r_2)$. Questo va fatto però con ogni tupla di B_j pertanto abbiamo $\text{CARD}(r_2) \cdot 1/\text{VAL}(B_j, r_2)$. Ma questa operazione va fatta anche per ogni tupla v che appartiene ad r_1 quindi in totale si ha: $1/\text{VAL}(B_j, r_2) \cdot \text{CARD}(r_1) \cdot \text{CARD}(r_2)$.

Non è difficile fare il ragionamento dall'altro lato ovvero si sceglie una tupla in r_2 dopodiché si accoppia con tutte quelle in r_1 e si ripete l'operazione per tutte le tuple in r_2 , in questo caso abbiamo: $1/\text{VAL}(A_i, r_1) \cdot \text{CARD}(r_1) \cdot \text{CARD}(r_2)$.

Perciò abbiamo che $1/\text{VAL}(B_j, r_2) \cdot \text{CARD}(r_1) \cdot \text{CARD}(r_2)$ e $1/\text{VAL}(A_i, r_1) \cdot \text{CARD}(r_1) \cdot \text{CARD}(r_2)$ **rappresentano lo stesso join!** Come è chiaro le due operazioni differiscono solamente per il primo termine perciò DBMS sceglierà in base a questa semplice operazione: $\min\{1/\text{VAL}(A_i, r_1), 1/\text{VAL}(B_j, r_2)\} \cdot \text{CARD}(r_1) \cdot \text{CARD}(r_2)$.

4) La minimizzazione dei costi

L'analisi dei costi ha una sola funzione: quella di minimizzarli e quindi di minimizzare il movimento di pagine che è ovviamente direttamente proporzionale al numero di tuple che si "maneggiano" durante le query.

Diminuire il numero di tuple diventa quindi fondamentale. Proviamo a farci un'idea di quanto l'ottimizzazione (logica) possa effettivamente migliorare i tempi di esecuzione.

4.1) Calcolo del costo senza ottimizzazione

Proveremo a calcolare i costi minimizzati e non dello schema usato in precedenza come esempio. Consideriamo prima di tutto tutti i vari dati forniti dal problema:

DATI DEL PROBLEMA				
Cardinalità	$ pazienti = 10^5$	$ ricoveri = 10^5$	$ medici = 10^2$	
VAL	VAL(Inizio, Ricoveri) = 10	VAL(Reparto, Ricoveri) = 10	VAL(Residenza, Pazienti) = 10	
	VAL(COD, Pazienti) = 10^5	VAL(MATR, Medici) = 10^2	VAL(PAZ, Ricoveri) = 10^5	
		VAL(Reparto, Medici) = 10^2		

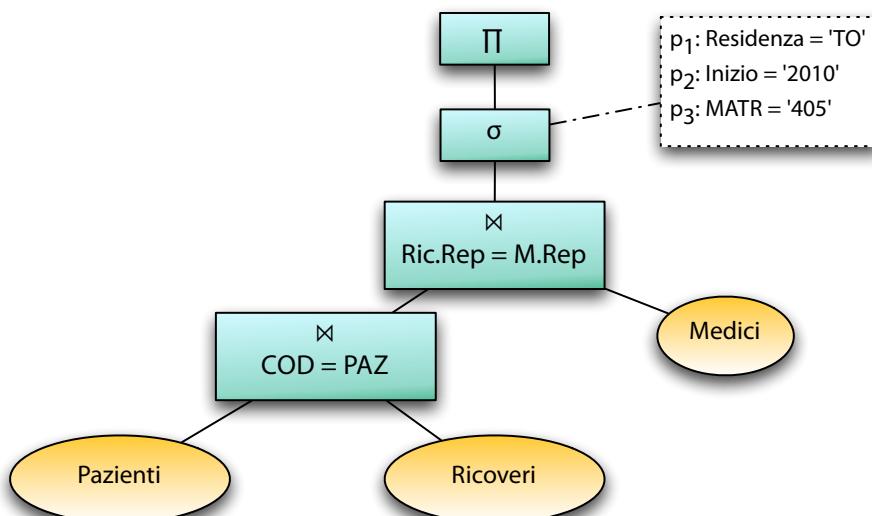
Vogliamo eseguire la query:

$\Pi \dots (\sigma_{Residenza = 'TO' \wedge Inizio = '2010' \wedge Matr = '405'} ((pazienti \bowtie_{COD = PAZ} ricoveri) \bowtie_{Ric.rep = M.rep} (medici))$

e ci calcoliamo quindi subito i tre f_p delle selezioni:

p	Predicato	f_p	Valore numerico
p_1	Residenza = 'TO'	$1/VAL(Residenza, Pazienti)$	$1/10^2$
p_2	Inizio = '2010'	$1/VAL(Inizio, Ricoveri)$	$1/10$
p_3	Matr = '405'	$1/VAL(Matr, Medici)$	$1/10^2$

Non resta che iniziare a fare i conti, ricordando che l'albero della query è:

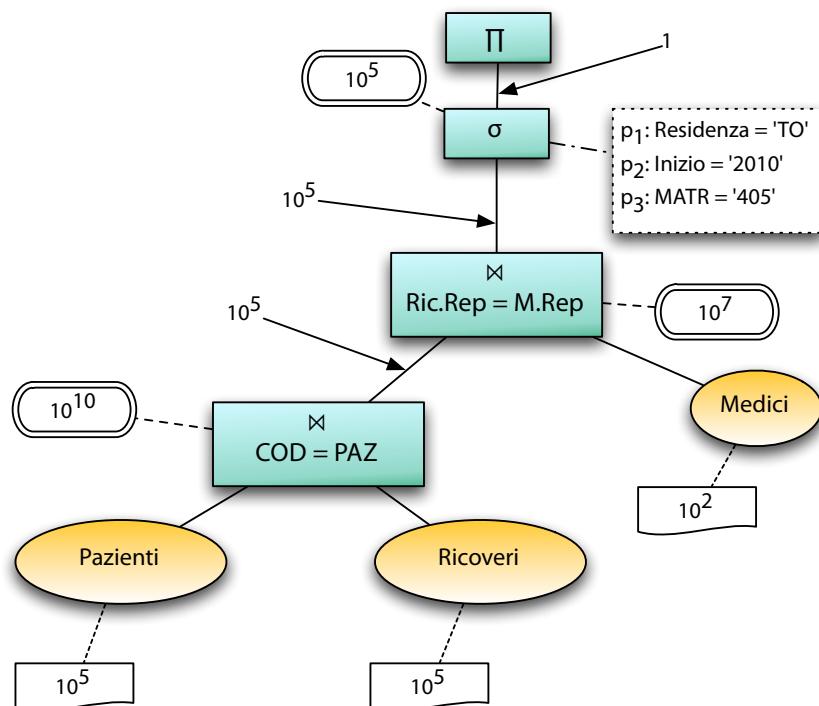


Nei disegni che seguono i rettangoli rappresentano operazioni, le frecce incidenti sugli archi sono cardinalità di risultati, le linee tratteggiate uscenti dai quadrati rappresentano invece il costo computazionale dell'operazione. I numeri uscenti dai cerchi (che rappresentano le relazioni) indicano invece la cardinalità delle relazioni stesse.

Si ricordano le due formule (che chiameremo F1 e F2 da qui in poi) :

- **F1:** Calcolare la cardinalità di un equi-join con $A_i = B_j$: $\min\left\{\frac{1}{VAL(A_i, R)}, \frac{1}{VAL(B_j, S)}\right\} \cdot CARD(R) \cdot CARD(S)$

- **F2:** La cardinalità di un equi-join, se c'è vincolo di integrità referenziale fra A_i e B_j , è uguale alla cardinalità della relazione che ha l'attributo *referenziante* ovvero colui che fa riferimento alla chiave primaria.



Partendo dal join in basso, cerchiamo di capire il perché di questi numeri.

Il costo computazionale del join 'COD = PAZ' è banalmente **10¹⁰** poiché è la combinazione fra tutte le tuple dei due operandi del join ($10^5 \cdot 10^5 = 10^{10}$).

La cardinalità del risultato, essendo un equi-join, potremmo ottenerla usando la formula F1 ma facendo attenzione notiamo che è necessario usare **F2** poiché vale un vincolo di integrità referenziale tra COD e PAZ dove Pazienti è *referenziata* e Ricoveri è *referenziante*. Data F2, quindi, la cardinalità del risultato è la cardinalità della relazione referenziante ovvero Ricoveri e quindi, in definitiva, **10⁵**.

Il join 'Ric.Rep = M.Rep' prende come operatore di sinistra il risultato del join sopra esaminato, quindi cardinalità 10⁵ e come operando destro la relazione Medici che ha cardinalità 10² e quindi il costo di computazionale dell'operazione risulta $10^5 \cdot 10^2 = 10^7$.

La cardinalità del risultato, questa volta, è quella ottenuta da **F1** e quindi:

$$\begin{aligned} & \min\{1/VAL(Reparto, Ricoveri), 1/VAL(Reparto, Medici)\} \cdot CARD(Ricoveri) \cdot CARD(Medici) = \\ & \min\{1/10, 1/10^2\} \cdot 10^5 \cdot 10^2 = 1/10^2 \cdot 10^5 \cdot 10^2 = 10^5 \end{aligned}$$

Il costo computazionale della selezione è ovviamente uguale alla quantità di tuple in entrata, quindi 10⁵.

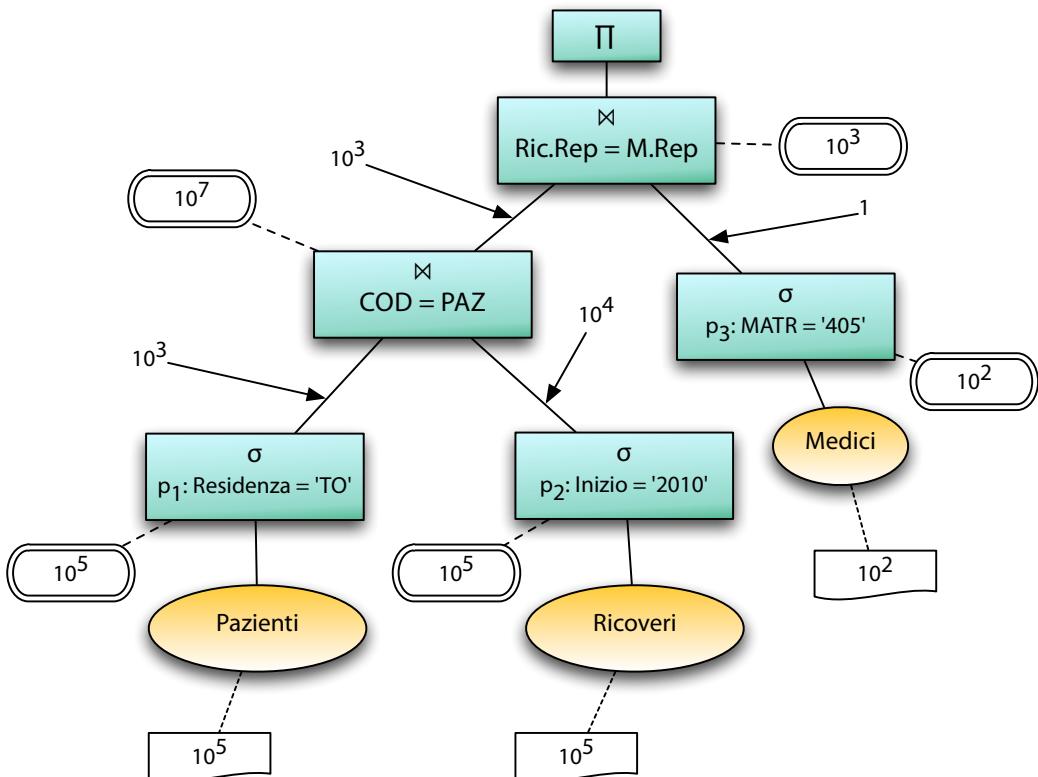
Il risultato della selezione dipende dai suoi f_p . Trattandosi di una congiunzione tra condizioni, si ha che la cardinalità del risultato è: $f_{p1} \cdot f_{p2} \cdot f_{p3} \cdot CARD(Operando)$ e quindi $1/10^2 \cdot 1/10 \cdot 1/10^2 \cdot 1/10^5 = 1$.

Il costo totale di computazione è quindi: $10^{10} + 10^7 + 10^5$ (i due join e la selezione) ma dato che noi guardiamo solamente l'ordine di grandezza notiamo che si tratta di 10¹⁰ computazioni, ovvero 10 miliardi!

4.2) Calcolo del costo con ottimizzazione (senza il 7° passo)

Ripetiamo ora l'esercizio sopra, ma questa volta applicando l'ottimizzazione. Notiamo quindi l'albero modificato con le selezioni verso le foglie e aggiungiamo una ulteriore formula necessaria per i conti:

- **F3:** il numero di valori distinti assegnati ad un attributo sul quale è stata applicata una selezione è il minimo tra il VAL senza la selezione e la cardinalità della selezione, ovvero in formule: $VAL(A_i, \sigma_p(R)) = \min\{VAL(A_i, R), |\sigma_p(R)|\}$. Questo vale solo se A_i non è coinvolto nel predicato p di selezione (si veda a fondo paragrafo per gli altri casi).



Anche qui, vediamo di esaminare i numeri.

Il costo computazionale delle tre selezioni è ovviamente lo stesso delle tre relazioni in input, quindi, da sinistra verso destra, **$10^5, 10^5, 10^2$** .

Il risultato della prima selezione è 10^3 poiché si applica la formula del fattore di selettività (paragrafo 3.1)

$$- |\sigma_{p1}(\text{Pazienti})| = f_{p1} \cdot |\text{Pazienti}| = 1/10^2 \cdot 10^5 = 10^3$$

Il risultato delle altre due selezioni è esattamente la stessa cosa e quindi:

$$- |\sigma_{p2}(\text{Ricoveri})| = f_{p2} \cdot |\text{Ricoveri}| = 1/10 \cdot 10^5 = 10^4$$

$$- |\sigma_{p3}(\text{Medici})| = f_{p3} \cdot |\text{Medici}| = 1/10^2 \cdot 10^2 = 1$$

Il join 'COD = PAZ' ha il solito costo computazionale che è la moltiplicazione dei due input, $10^3 \cdot 10^4 = \mathbf{10^7}$

Il risultato del join 'COD=PAZ' è invece più complesso da calcolare: in input non abbiamo due relazioni 'semplici' ma abbiamo relazioni filtrate da selezioni, quindi non possiamo applicare F1 direttamente poiché $VAL(COD, \text{Pazienti})$ e $VAL(PAZ, \text{Ricoveri})$ non sono i dati corretti (come detto sopra, sono stati filtrati!). È necessario calcolare F3. Calcoliamo quindi $VAL(COD, \sigma_{p1}(\text{Pazienti}))$ e $VAL(PAZ, \sigma_{p2}(\text{Ricoveri}))$ che utilizzeremo al posto di $VAL(COD, \text{Pazienti})$ e $VAL(PAZ, \text{Ricoveri})$ in F1. Perciò:

$$- VAL(COD, \sigma_{p1}(\text{Pazienti})) = \min\{VAL(COD, \text{Pazienti}), |\sigma_{p1}(\text{Pazienti})|\} = \min\{10^5, 10^3\} = 10^3$$

$$- VAL(PAZ, \sigma_{p2}(\text{Ricoveri})) = \min\{VAL(PAZ, \text{Ricoveri}), |\sigma_{p2}(\text{Ricoveri})|\} = \min\{10^5, 10^4\} = 10^4$$

Possiamo ora applicare: $\min\left\{\frac{1}{VAL(A_i, \sigma_{p1}(R))}, \frac{1}{VAL(B_j, \sigma_{p2}(S))}\right\} \cdot CARD(\sigma_{p1}(R)) \cdot CARD(\sigma_{p2}(S))$

che è la versione modificata della F1 utilizzando la F3.

Risulta essere: $\min\{ 1/\text{VAL}(\text{COD}, \sigma_{p1}(\text{Pazienti})), 1/\text{VAL}(\text{PAZ}, \sigma_{p2}(\text{Ricoveri})) \} \cdot \text{CARD}(\sigma_{p1}(\text{Pazienti})) \cdot \text{CARD}(\sigma_{p2}(\text{Ricoveri}))$ e quindi $\min\{1/10^3, 1/10^4\} \cdot 10^3 \cdot 10^4 = 10^3$ che è la effettiva cardinalità finale del risultato.

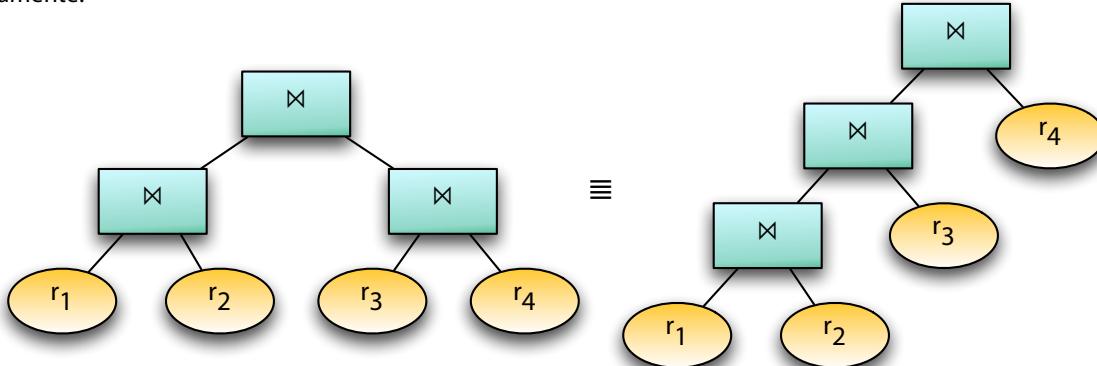
Il join 'Ric.Rep = M.Rep' ha il solito costo computazionale che è la moltiplicazione dei due input, $10^3 \cdot 1 = 10^3$

Il join 'Ric.Rep = M.Rep' ha come risultato una cardinalità calcolabile tramite le formule usate prima, ma non è rilevante per i costi.

Il costo totale di computazione è quindi: $10^5 + 10^5 + 10^2 + 10^7 + 10^3$ (tre selezioni e due join) ma dato che noi guardiamo solamente l'ordine di grandezza notiamo che si tratta di 10^7 computazioni, ovvero 10 milioni! Notiamo un risparmio di un **fattore mille** il che significa che se per fare la query ci impiegasse 1 sec, ottimizzandola impiegherebbe 1msec.

4.3) Spiegazione del 7° passo dell'algoritmo di ottimizzazione logica

Come sappiamo il join gode di proprietà associativa, ovvero $(r_1 \bowtie r_2) \bowtie (r_3 \bowtie r_4) \equiv (r_1 \bowtie r_2) \bowtie r_3 \bowtie r_4$ e graficamente:



Ma perché potremmo desiderare di "riassociare" i join?

Dato che il join (in generale) diminuisce la cardinalità si tendono a portare i join tra relazioni che hanno cardinalità più piccola verso il fondo dell'albero. Questo significa che i join con operandi che hanno cardinalità più piccola verranno portati a fondo albero.

I join in questo modo, risalendo dall'albero, diminuiscono sempre più le cardinalità e quindi i costi computazionali.

Applichiamo ora il settimo punto al nostro esempio di prima.

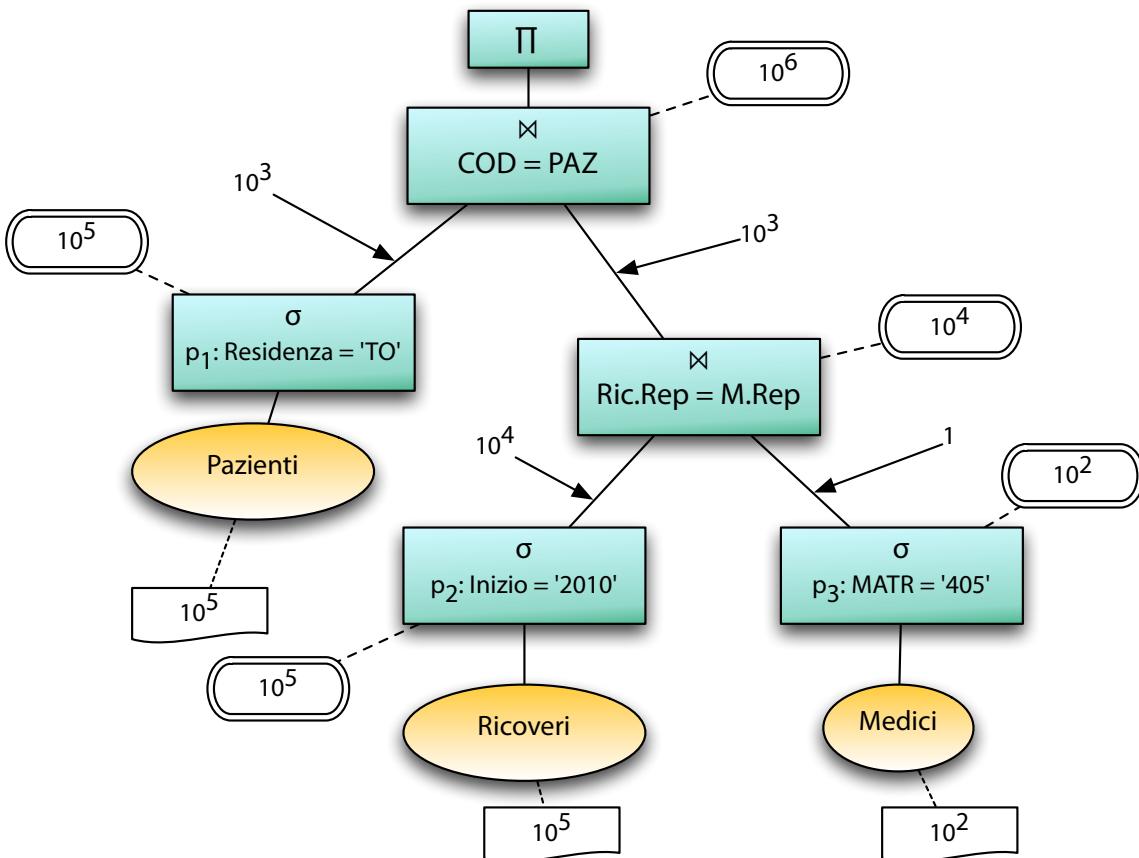
4.4) Calcolo del costo con ottimizzazione (con il 7° passo)

Possiamo subito accorgerci, guardando l'albero (paragrafo 4.2) che il join con cardinalità minima è $\bowtie_{\text{Ric.rep} = \text{M.rep}}$ che ha come cardinalità dell'operando destro il numero 1 che è molto piccolo quindi si presta bene per essere portato a fondo albero.

La nostra query viene quindi re-associata così:

$$\Pi_{...} ((\sigma_{\text{Residenza} = 'TO'} (\text{pazienti})) \bowtie_{\text{COD} = \text{PAZ}} ((\sigma_{\text{Inizio} = '2010} (\text{Ricoveri})) \bowtie_{\text{Ric.rep} = \text{M.rep}} (\sigma_{\text{Matr} = '405'} (\text{Medici})))$$

Notiamo subito che il join di cui abbiamo parlato sopra è stato "affossato" nelle parentesi al massimo livello possibile.



Cerchiamo, anche qui, di capire i numeri.

I risultati uscenti dalle selezioni (10^3 , 10^4 , 1) così come i loro costi computazionali (10^5 , 10^5 , 10^2) sono ovviamente gli stessi di prima.

Notiamo la prima miglioria nel join 'Ric.Rep = M.Rep' dove il costo computazionale risulta essere 10^4 poiché è la moltiplicazione delle cardinalità dei due operatori (come al solito).

Il risultato del join 'Ric.Rep = M.Rep' risulta essere 10^3 . Questo perché abbiamo applicato la F2 modificata, ovvero:

$$\min\left\{\frac{1}{VAL(A_i, \sigma_{p_1}(R))}, \frac{1}{VAL(B_j, \sigma_{p_2}(S))}\right\} \cdot CARD(\sigma_{p_1}(R)) \cdot CARD(\sigma_{p_2}(S))$$

dove abbiamo:

- $VAL(Ric.Rep, \sigma_{p_2}(Ricoveri)) = \min\{VAL(Ric.Rep, Ricoveri), |\sigma_{p_2}(Ricoveri)|\} = \min\{10, 10^4\} = 10$
- $VAL(M.Rep, \sigma_{p_3}(Medici)) = \min\{VAL(M.Rep, Medici), |\sigma_{p_3}(Medici)|\} = \min\{10^2, 1\} = 1$

E quindi applichiamo F2:

$$\min\{1/VAL(Ric.Rep, \sigma_{p_2}(Ricoveri)), 1/VAL(M.Rep, \sigma_{p_3}(Medici))\} \cdot CARD(\sigma_{p_2}(Ricoveri)) \cdot CARD(\sigma_{p_3}(Medici)) = \\ \min\{1/10, 1/1\} \cdot 10^4 \cdot 1 = 10^3$$

La vera miglioria si nota nel join 'COD = PAZ' che ha costo computazionale 10^6 invece che 10^7 poiché abbiamo modificato gli operandi prima rispetto a ciò che abbiamo fatto nell'esercizio precedente.

I costi totali di computazione è quindi: $10^5 + 10^5 + 10^2 + 10^4 + 10^6$ (tre selezioni e due join) ma dato che noi guardiamo solamente l'ordine di grandezza notiamo che si tratta di 10^6 computazioni, ovvero 1 milione!

Quindi, con le nostre migliorie, abbiamo ridotto di un **fattore diecimila** il costo computazionale portandolo da dieci miliardi ad un milione.

4.5) Calcolo di VAL con selezione nei casi di uguaglianza tra attributo e predicato

Come visto nello studio della F3 (paragrafo 4.2) abbiamo visto che quella formula vale solo se A_i e predicato sono disgiunti. Vediamo quindi una tabella che indica quali sono i valori di $VAL(A_i, \sigma_p(r))$ se A_i e p non sono disgiunti.

p	Predicato	VAL($A_i, \sigma_p(R)$)	MAX($A_i, \sigma_p(R)$)	MIN($A_i, \sigma_p(R)$)
p_1	$A_i = v$	1	v	v
p_2	$A_i \leq v$	$f_{p2} \cdot VAL(A_i, R)$	v	MIN(A_i, R)
p_3	$A_i \geq v$	$f_{p3} \cdot VAL(A_i, R)$	MAX(A_i, R)	v
p_4	$v_1 \leq A_i \leq v_2$	$f_{p4} \cdot VAL(A_i, R)$	v_2	v_1

Basi di dati

Calcolo relazionale

Capitolo 4

Enrico Mensa

Indice degli argomenti

1) <i>Introduzione al calcolo relazionale</i>	1
1.1) Il procedimento di traduzione di una query	
2) <i>Le componenti del calcolo relazionale</i>	1
2.1) L: Range List	
2.2) T: Target List	
2.3) F: Funzione (anche detta formula)	
3) <i>Esempi di calcolo relazionale</i>	2
3.1) Trovare i pazienti residenti in Torino (si richiede nome e cognome)	
3.2) Nome dei pazienti ricoverati nel reparto 'A' (si vuole anche la data del ricovero)	
3.3) Nome dei pazienti ricoverati nel reparto 'A'	
3.4) Medici curanti del paziente 'A102'	
3.5) Cognome e nome di pazienti e medici omonimi	
3.6) Cognome e nome di pazienti e medici non omonimi	
3.7) I capi (MATR, nome) cui i dipendenti guadagnano tutti più di 40	
3.8) Gli studenti che hanno superato tutti gli esami	
4) <i>Conclusioni sugli insiemi delle query e ulteriori informazioni</i>	3
4.1) Le interrogazioni ricorsive	

1) Introduzione al calcolo relazionale

Il calcolo relazionale è un approccio **dichiarativo** e non procedurale che fornisce un paradigma logico utile allo sviluppo delle interrogazioni.

Vi sono principalmente tre tipi di calcolo relazionale:

- Calcolo sui domini
- **Calcolo su tuple con range**
- Datalog -> Prolog

Noi ci concentreremo sullo studio del "calcolo su tuple con range" per diversi motivi, il primo è che è alla base dell'SQL, il secondo è che mappato sull'algebra relazionale che abbiamo appena studiato.

1.1) Il procedimento di traduzione di una query

Una query viene quindi filtrata tramite questi "stand":

Query -> Calcolo Relazionale -> Procedura -> Ottimizzazione Logica + Fisica -> Algebra Relazionale.

2) Le componenti del calcolo relazionale

Il calcolo relazionale si sviluppa tramite una tripletta {T | L | F}. Questa tripletta è in grado di rappresentare una query dell'algebra relazionale in maniera molto più compatta ma altrettanto precisa. Vediamo ora le tre parti separatamente.

2.1) L: Range List

La Range List definisce delle variabili sulle tavole relazionali $x(r_1)$, $y(r_2)$, ecc. Le variabili assumono il valore delle tuple nella relazione.

Esempio

L : P(Pazienti)

La variabile P assumerà il valore di tutte le tuple di pazienti.

2.2) T: Target List

La target list, date le variabili nella range list, estrapola alcuni degli attributi delle tuple contenute nelle variabili.

Notazioni

- **P.nome** estrae i valori di nome dalla relazione su cui P è definita.
 - **P.(nome, cognome)** estrae i valori sia di nome che di cognome dalla relazione su cui P è definita.
 - **CognomePersona : P.cognome** rinomina l'attributo cognome in CognomePersona
 - **(NC, CG) : P.(nome, cognome)** rinomina l'attributo nome in NC e cognome in CG
 - **(NC) : P.(nome, cognome)** rinomina solo l'attributo nome in NC
- Inoltre P.Cognome sottintende la rinomina Cognome : P.Cognome.

2.3) F: Funzione (anche detta formula)

F è un predicato nella forma $x.A_i \Theta y.B_j$ dove Θ è un confronto booleano $\{\wedge, \vee, \neg, \Rightarrow\}$ ed $x, y \in L$.

Sono infine ammessi gli operatori esistenziali e universali quindi $\exists s(r)(f')$, $\forall s(r)(f')$ con $s \in L$.

3) Esempi di calcolo relazionale

Vediamo ora alcuni esempi di calcolo relazionale (si usano le relazioni usate nel passato).

3.1) Trovare i pazienti residenti in Torino (si richiede nome e cognome)

L: p(PAZIENTI)

T: {p.Cognome, P.Nome} ≡ p.(Cognome, Nome)

F: p.Residenza = 'TO'

Anche scrivibile: {p.(Cognome, Nome) | p(PAZIENTI) | p.Residenza = 'TO'}

3.2) Nome dei pazienti ricoverati nel reparto 'A' (si vuole anche la data del ricovero)

L: p(PAZIENTI), r(RICOVERI)

T: p.(Cognome, Nome), r.(Inizio)

F: (p.COD = p.PAZ) ∧ p.Reparto = 'A'

p.(Cognome, Nome), r.(Inizio)

Anche scrivibile: { p.(Cognome, Nome), r.(Inizio) | p(PAZIENTI), r(RICOVERI) | (p.COD = p.PAZ) ∧ (r.Reparto = 'A') }

3.3) Nome dei pazienti ricoverati nel reparto 'A'

L: p(PAZIENTI)

T: p.(Cognome, Nome)

F: $\exists r(RICOVERI) | ((r.COD = p.PAZ) \wedge p.Reparto = 'A')$

Anche scrivibile: { p.(Cognome, Nome) | p(PAZIENTI) | **$\exists r(RICOVERI)$** | ((r.COD = p.PAZ) ∧ (p.Reparto = 'A')) }

Da questo esempio si evince che non è necessario includere tutte le variabili nella range list, ma tramite gli operatori esistenziale e universale è possibile introdurre una nuova variabile, in questo caso r (in grassetto).

3.4) Medici curanti del paziente 'A102'

{ m.(MATR, Cognome) | m(MEDICI) | $\exists r(RICOVERI) | ((r.PAZ = 'A102') \wedge (r.Reparto = m.Reparto))$ }

3.5) Cognome e nome di pazienti e medici omonimi

{ p.(Cognome, Nome) | p(PAZIENTI) | $\exists m(MEDICI) | ((m.Nome = r.Nome) \wedge (m.Cognome = r.Cognome))$ }

3.6) Cognome e nome di pazienti e medici non omonimi

{ p.(Cognome, Nome) | p(PAZIENTI) | $\exists m(MEDICI) | \neg((m.Nome = r.Nome) \wedge (m.Cognome = r.Cognome))$ }

3.7) I capi (MATR, nome) cui i dipendenti guadagnano tutti più di 40

L: i(IMPIEGATI), s(SUPERVISORI)

T: i.(MATR, Nome)

F: (s.Capo = i.Matr) $\wedge \forall s'(SUPERVISORI) | (s.Capo = s'.Capo) \Rightarrow \exists i'(IMPIEGATI) | (s'.Impiegato = i'.Impiegato) \wedge (i'.Stipendio > 40)$

Questa volta la F risulta parecchio complicata! Esaminiamola:

1° (**s.Capo = i.Matr**) Dato un impiegato che è capo (quindi un capo, in sostanza)

2° $\forall s'(SUPERVISORI) | (s.Capo = s'.Capo)$ Per ogni supervisore che è quel capo (ovvero per ogni occorrenza del capo)

3° $\exists i'(IMPIEGATI) | (s'.Impiegato = i'.Impiegato)$ Deve esistere un impiegato tale che sia collegato a quel capo

4° (**i.Stipendio > 40**) e abbia stipendio maggiore di 40 (quarto punto).

Prima abbiamo l'occorrenza del capo (**primo** punto), poi si deve avere che ogni altra occorrenza di quel capo (**secondo** punto) debba essere (debba esistere) collegata con un impiegato (**terzo** punto) il quale deve avere stipendio maggiore di 40 (**quarto** punto).

3.8) Gli studenti che hanno superato tutti gli esami

L: $e(E)$

T: $e.Matr$

F: $\forall c(P) (\exists es(E) | (es.Matr = e.Matr) \wedge (es.CORSO = c.CORSO))$

1° $\forall c(P)$ Per ogni corso (c)

2° $\exists es(E)$ Deve esistere un esame (es)

3° $(es.Matr = e.Matr)$ sostenuto da e

4° $(es.CORSO = c.CORSO)$ che corrisponda al corso c

4) Conclusioni sugli insiemi delle query e ulteriori informazioni

Possiamo finalmente trarre delle conclusioni.

Possiamo notare che non abbiamo mai usato l'operatore di unione, questo poiché l'SQL non lo implementa. Ma questo non è un problema, poiché possiamo sfruttare il sovra-insieme del calcolo relazionale, il calcolo su domini.

In sostanza il calcolo su domini è generalissimo e vede le variabili come effettivi valori booleani che sono 'true' solo se la variabile assume il valore della tupla. Questo approccio è pericoloso, ma ci permette di implementare l'unione fra due singole query SQL.

Insiematicamente abbiamo: *Domini* che includono *Formule Sicure* (algebra relazionale) che includono *Calcolo su tuple con range*, sul quale, come detto, SQL si basa.

4.1) Le interrogazioni ricorsive

L'insieme delle query che possiamo scrivere tramite il calcolo basato su tuple con range manca di quelle interrogazioni dette *ricorsive*.

Pensiamo ad esempio ad una relazione Famiglia(Genitore, Figlio) allora non potremmo ottenere tutta la genealogia di un certo elemento della famiglia (ovvero non possiamo percorrere l'albero genealogico in maniera ricorsiva).

Possiamo certo tramite una join ottenere i nonni, tramite due join ottenere i trisavoli e così via, ma non tutti insieme. Questo tipo di problema è risolto nel calcolo **datalog**, di cui noi, però non ci interessiamo.

Basi di dati

La teoria della normalizzazione

Capitolo 5

Enrico Mensa

Indice degli argomenti

1) <i>Introduzione: le transazioni</i>	1
1.1) Le componenti di una transazione	
1.2) Le proprietà (astratte) di una transazione	
1.3) DBMS e transazioni: l'automa a stati finiti	
2) <i>Perché normalizzare?</i>	2
2.1) Le anomalie più frequenti	
2.2) Esempi delle anomalie	
2.3) Idea intuitiva di molecole informative	
3) <i>Dipendenza funzionale: dal concetto al formalismo</i>	4
3.1) Definizione formale di dipendenza funzionale	
3.2) Altro esempio di dipendenza funzionale	
3.3) Operare sulle dipendenze funzionali	
4) <i>La teoria di Armstrong</i>	5
4.1) Assioma di riflessività	
4.2) Assioma dell'unione	
4.3) Assioma della transitività	
4.4) Teorema dell'espansione	
4.5) Teorema della decomposizione	
4.6) Teorema della pseudo-transitività e teorema del prodotto (poco usati)	
4.7) Teorema del prodotto (poco usato)	
5) <i>Applicazioni della teoria di Armstrong sulle dipendenze funzionali: le chiusure</i>	7
5.1) Chiusura di un insieme F	
5.2) La nozione di equivalenza	
5.3) La chiusura di un insieme di attributi	
5.4) Algoritmo per il calcolo della chiusura di un insieme di attributi	
5.5) Riflessioni finali (importanti) in merito alle chiusure	

6) Applicazioni della teoria di Armstrong sulle dipendenze funzionali: Individuare le chiavi	11
6.1) Dalle dipendenze alle chiavi: somiglianza fra definizioni	
6.2) La ricerca delle chiavi: un approccio metodologico	
6.3) Caso particolare: attributi non citati in F	
6.4) Stesse chiavi da insiemi equivalenti	
7) La decomposizione delle relazioni	13
7.1) Il fenomeno delle tuple spurie	
7.2) La decomposizione con join senza perdita di informazioni	
7.3) Teorema per la decomposizione con join senza perdita di informazioni	
8) La decomposizione delle dipendenze funzionali	15
8.1) Restrizioni di F in $R_i(A_i)$	
9) Approccio alle forme normali	16
10) Boyce Codd Normal Form (BCNF)	16
10.1) Condizioni di appartenenza BCNF	
10.2) Esempi di applicazione empirica della definizione	
10.3) Algoritmo per la trasformazione in BCNF	
10.4) Considerazioni sull'algoritmo per la trasformazione in BCNF	
10.5) Fattori positivi e negativi dell'uso della BCNF	
11) La terza forma normale	21
11.1) Condizioni di appartenenza alla III°NF	
11.2) Algoritmo per la sintesi in III°NF	
Insieme F' di copertura minimale	
Algoritmo per il calcolo della chiusura minimale ed esempio	
11.3) Caratteristica 1: in $R(X,A)$ X è chiave primaria	
11.4) Caratteristica 2: le dipendenze funzionali sono conservate	
11.5) Caratteristica 3: garanzia del join senza perdita di informazioni (JSP)	
11.6) Caratteristica 4: ogni relazione generata è a sua volta in terza forma normale	
11.7) Caratteristica 5: Lo schema ammette "semplificazioni" che mantengono la terza forma normale	

1) Introduzione: le transazioni

Per poter parlare di normalizzazione dobbiamo prima di tutto capire cosa siano le transazioni.

Come già detto in precedenza un'applicazione è una serie di transazioni che applicano operazioni sul database. Vediamo di definire più formalmente una transazione.

1.1) Le componenti di una transazione

Una transazione tipo è siffatta:

* *begin*

- Azione 1
 - Azione 2
 - ...
 - Azione n
- * *Commit Work / Rollback*

Le azioni sono comandi, operazioni sul database, tra cui:

- INSERT
- DELETE
- UPDATE
- SELECT

Alla fine di una transazione può essere fatto un **Commit Work** (si rendono effettive e durature le modifiche apportate dalle azioni della transazione) oppure un **Rollback** (si annullano tutte le azioni effettuate dalla transazione). Capiremo meglio il perché di queste due opportunità in seguito.

1.2) Le proprietà (astratte) di una transazione

Una transazione deve rispettare le proprietà base, **ACID**.

A = Atomica, (una transazione deve essere eseguita nella sua interezza, altrimenti non ha senso)

C = Consistente, (una transazione deve rispettare in ogni sua azione i vari vincoli)

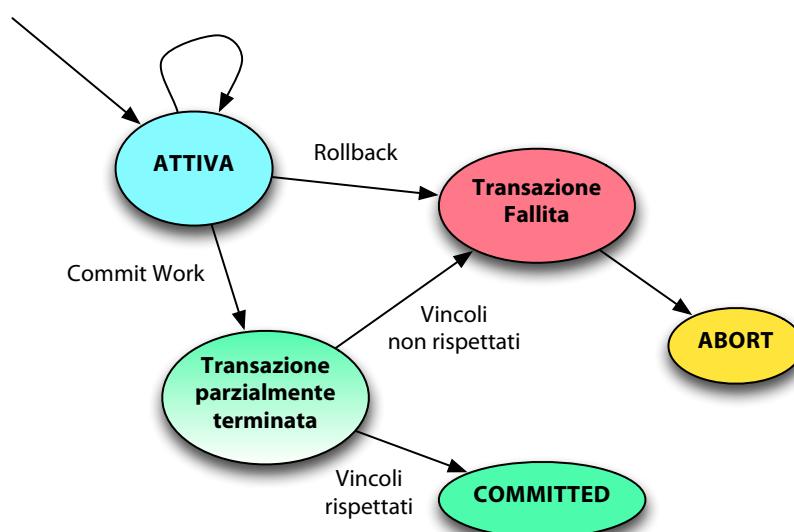
I = Isolata (una transazione viene considerata come 'eseguita da sola' senza doversi occupare delle concorrenze)

D = Durabile (gli effetti di una transazione devono poter essere persistenti)

1.3) DBMS e transazioni: l'automa a stati finiti

Quando il DBMS riceve una transazione T_i non guarda da che applicazione gli sia arrivata ma si pone l'unico obiettivo di portarla a termine.

Il DBMS segue l'intero ciclo di vita della transazione tramite questo automa a stati finiti:



Ogni transazione rimane **attiva** durante la sua durata. Quando tutte le azioni sono concluse allora se il DBMS riceve dall'applicazione un'ordine di Commit Work passa la transazione allo stato **transazione parzialmente terminata**, dopodiché il DBMS effettua un controllo sui vincoli (*consistenza*), se essi sono rispettati allora la transazione viene resa effettiva (*durabilità*) e quindi **committed**. Se invece l'applicazione manda un **rollback** allora lo stato evolve in **transazione fallita** e parte tutto il procedimento di disfacimento della transazione, al termine del quale la transazione passa allo stato **abort**. Se dallo stato di transazione parzialmente terminata il controllo sui vincoli dovesse dare esito falso, allora lo stato in cui la transazione finirebbe sarebbe quello di **transazione fallita**.

La componente che si occupa di mantenere veritiero il concetto di *isolazione* è un modulo a parte che si prefigge l'obiettivo della **gestione concorrente**.

La componente che si occupa del rollback e del disfacimento è detta **crash recovery**.

Potremmo domandarci perché il controllo sui vincoli venga fatto solo a fine transazione e non per ogni azione, ma tramite questo esempio possiamo capire la motivazione. Prendiamo due relazioni:

$R_1(PK_1, \dots, A_1, \dots)$ e $R_2(PK_2, \dots, B_1, \dots)$ e supponiamo di avere un vicolo di integrità referenziale da A_1 a PK_2 e da B_1 a PK_1 .

Allora possiamo facilmente capire che, partendo dalle relazioni vuote, non potremmo mai popolarle con tuple dato che ogni inserimento non sarebbe legittimo (mancherebbe il corrispondente nell'altra relazione).

Permettendo invece più azioni, solo al termine dell'inserimento verificheremo che effettivamente ci siano state le accoppiate corrette A_1-PK_2 e B_1-PK_1 e quindi "daremo il tempo" al DBMS di inserire le tuple tranquillamente.

Da tutto questa analisi si determina una ipotesi implicita, ovvero che le transazioni devono essere relativamente brevi perché altrimenti diverrebbero ingestibili (si pensi a un rollback molto difficoltoso, ecc.).

2) Perché normalizzare?

Una volta introdotto il concetto di transazione possiamo passare alla normalizzazione.

Come sappiamo, lo schema ER viene tradotto in schema logico.

Ma la creazione di uno schema non è così semplice, e dobbiamo porre l'attenzione ad alcune anomalie frequenti che la progettazione potrebbe ignorare.

La normalizzazione è quindi un processo volto all'eliminazione della ridondanza e del rischio di incoerenza del database.

2.1) Le anomalie più frequenti

Tra le anomalie che vogliamo scongiurare abbiamo:

- Anomalia di update
- Anomalia di cancellazione
- Anomalia di inserzione

2.2) Esempi delle anomalie

Per poter toccare con mano le anomalie sopracitate vediamo un esempio piuttosto complesso. Prendiamo una relazione Esame:

Esame(MATR, NomeS, IndirizzoS, CAPS, CodiceFiscaleS, DataNascitaS, Corso, Voto, DataEsame, CodProf, NomeProf, Qualifica, TipoUfficio)

abbreviabile:

MATR	NS	IS	CAP	CFS	DN	Co	Vo	DE	CDP	NP	Q	TU
------	----	----	-----	-----	----	----	----	----	-----	----	---	----

Prendiamo dunque questa tupla d'esempio:

MATR	NS	IND	CAP	CFS	DN	Co	Vo	DE	CDP	NP	Q	TU
341	Piero	NO	28100	P73	1975	Prog	27	3/5/79	B1	Bono	Prof.Ass.	TipA

Ci aspettiamo che all'inserimento di un nuovo esame dato da Piero, la sua matricola, il suo nome, il suo indirizzo, il suo CAP, il suo CF e la sua data di nascita siano sempre gli stessi.

L'inserimento di dati *non consistenti* (ad esempio in seguito a un cambio di indirizzo) creerebbe disparità nel database il quale, soprattutto, non rispecchierebbe la realtà.

Un aggiornamento di indirizzo va fatto su **tutte le tuple** non solamente su parte. Ecco quindi l'**anomalia dell'update**. Si noti che lo stesso discorso si potrebbe fare se dovessimo cambiare ufficio alla Bono (tutti gli uffici andrebbero aggiornati!).

Supponiamo poi che si aggiunga un nuovo professore, allora vorremmo inserire una tupla che contiene solo lui.

MATR	NS	IND	CAP	CFS	DN	Co	Vo	DE	CDP	NP	Q	TU
341	Piero	NO	28100	P73	1975	Prog	27	3/5/79	B1	Bono	Prof.Ass.	TipA
-	-	-	-	-	-	-	-	-	G1	Giolito	Prof.Ass.	TipB

Non potremmo farlo poiché vi sono chiavi primarie tra gli altri campi, abbiamo una **anomalia di inserzione**.

In ultimo luogo, se volessimo modificare una porzione di tupla cancellando per esempio un professore che è deceduto allora annulleremo il concetto di esame ed ecco l'**anomalia di cancellazione**.

2.3) Idea intuitiva di molecole informative

Come potremmo ovviare alle anomalie? Prima di tutto dobbiamo chiarire il fatto che quelle anomalie dipendono dal mondo reale che definisce cosa ha senso e cosa no.

Non resta che specificare (dapprima in modo non formale) delle regole di dipendenza (*dipendenza funzionale*) tra gli attributi. Ogni dipendenza specificata è detta **molecola informativa**.

Nell'esempio di sopra potremmo citare:

- MATR → NS, IND, CAP, CFS, DN (*dato la stessa matricola ci aspettiamo lo stesso nome, lo stesso indirizzo, lo stesso CAP, lo stesso codice fiscale e la stessa data di nascita*)
- CDP → NP, Q, TU (*dato un codice professore ci aspettiamo lo stesso nome, la stessa qualifica e lo stesso ufficio*)
- IND → CAP (*dato una città ci aspettiamo lo stesso CAP*)
- Q → TU (*dato una qualifica ci aspettiamo lo stesso tipo di ufficio*)

Queste dipendenze sono unilaterali e non valgono automaticamente nell'altro verso. Invece, potrebbe essere sensato specificare il senso biunivoco di alcune dipendenze funzionali:

- MATR → CFS così come CFS → MATR

Si noti infine che vi possono essere dipendenze con più attributi a sinistra, quindi

MATR, Co → Vo, DE, CDP (*dato una matricola e un corso si ha un esame unico, un unico voto un'unica data d'esame ed un unico codice professore*)

3) Dipendenza funzionale: dal concetto al formalismo

Abbiamo avuto un assaggio di cosa siano le molecole informative e quindi le dipendenze funzionali. Facciamo il punto della situazione con questa tabella.

Obiettivi	Criticità	Azioni
Esprimibilità delle informazioni	Anomalie di intersezione/ cancellazione	<u>Normalizzazione</u> e progettazione tramite entity - relationship
Efficienza	Anomalie di update	
Leggibilità	Sinonimi e omonimi	Standardizzazione dei nomi

3.1) Definizione formale di dipendenza funzionale

Scriviamo in maniera formale il concetto di dipendenza funzionale:

$X \rightarrow Y$ significa che $\forall t_1, t_2 \in r (t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y])$

ovvero se due tuple hanno una parte 'X' in comune allora devono avere anche la parte 'Y' in comune.

È chiaro che quindi una relazione è definibile come $(r(A), F)$ con F come insieme di dipendenze funzionali.

3.2) Altro esempio di dipendenza funzionale

Prendiamo la seguente tabella di Agenzie (che stabilisce la correlazione con il proprio direttore):

N°A	Città	codDirett	nomeDirett	ProvAg
1	TO	10	Rossi	TO
2	TO	11	Bianchi	TO
3	TO	10	Rossi	TO

Ma attenzione! Se ci fosse la regola che un direttore può lavorare solo in agenzie della stessa città, dovremmo includere il vincolo 'codDirett \rightarrow Città'

3.3) Operare sulle dipendenze funzionali

Prendiamo alcune dipendenze funzionali basate sull'esempio sopra. Questa è una visione della realtà che un certo progettista potrebbe fare.

- MATR \rightarrow NS, IS, CAP, CFS, DN
- CDP \rightarrow NP, Q
- IND \rightarrow CAP
- Q \rightarrow TU
- MATR \rightarrow CFS
- CFS \rightarrow MATR
- MATR, Co \rightarrow Vo, DE, CDP

Ma volendo, un altro progettista, potrebbe interpretare la realtà così:

MATR \rightarrow NS, IN, CAP

MATR \rightarrow CFS, DN

...

Le due visioni sono del tutto equivalenti poiché da entrambe possiamo dedurre che:

MATR \rightarrow NS, IS, CAP, CFS, DN, CAP. E poi, dato che CFS \rightarrow MATR si ha che CFS \rightarrow MATR, NS, IS, CAP, DN, CAP.

Questo ci porta ad un 'indizio': se possiamo ottenere due cose equivalenti e possiamo soprattutto dedurre qualcosa da qualcos'altro è chiaro che sotto queste dipendenze funzionali ci sia un **calcolo**. Di questo si occupa la **teoria di Armstrong**.

4) La teoria di Armstrong

La teoria di Armstrong definisce alcuni assiomi che noi applichiamo alle dipendenze funzionali. Si noti che, mentre la teoria è dimostrabile, la nostra applicazione (al modello delle dipendenze funzionali) è solamente verificabile ovvero si può verificare che le dipendenze funzionali sono un modello su cui è possibile applicare la teoria di Armstrong.

Nella teoria di Armstrong vi sono tre assiomi principali ed altri 5 teoremi derivati. Vediamoli. Intenderemo X e Y come insiemi di attributi.

4.1) Assioma di riflessività

$$\text{Se } Y \subseteq X \text{ allora } X \rightarrow Y$$

Questo assioma è utile per poter dimostrare i teoremi successivi.

Esempio

Se $X = \{\text{MATR}, \text{UF}\}$ e $Y = \{\text{MATR}\}$ allora $\text{MATR}, \text{UF} \rightarrow \text{MATR}$

4.2) Assioma dell'unione

$$\text{Se } X \rightarrow Y \text{ e } X \rightarrow Z \text{ allora } X \rightarrow YZ$$

Dove con ZY si intende l'unione dell'insieme Y con Z.

Esempio

Se $\text{MATR} \rightarrow \text{NS}, \text{IS}$, e $\text{MATR} \rightarrow \text{CAP}, \text{CFS}, \text{DN}$ allora $\text{MATR} \rightarrow \text{NS}, \text{IS}, \text{CAP}, \text{CFS}, \text{DN}$

4.3) Assioma della transitività

$$\text{Se } X \rightarrow Y \text{ e } Y \rightarrow Z \text{ allora } X \rightarrow Z$$

Esempio

Se $\text{CDP} \rightarrow \text{Q}$ e $\text{Q} \rightarrow \text{TU}$ allora $\text{CDP} \rightarrow \text{TU}$

4.4) Teorema dell'espansione

$$\text{Se ho } X \rightarrow Y \text{ e un'insieme qualunque } W \text{ allora } WX \rightarrow WY$$

Dimostrazione:

Parto da WX . $WX \rightarrow X$ (per riflessività), quindi dato che $X \rightarrow Y$ (per ipotesi) si ha che $WX \rightarrow Y$ (per transitività).

Inoltre, $WX \rightarrow W$ (per riflessività). Infine, dato $WX \rightarrow W$ e $WX \rightarrow Y$, si ottiene $WX \rightarrow WY$ (per unione).

Esempio:

Se $\text{MATR} \rightarrow \text{NS}$ e $W = \{\text{CDP}, \text{CFS}\}$ allora $\text{MATR}, \text{CDP}, \text{CFS} \rightarrow \text{NS}, \text{CDP}, \text{CFS}$

4.5) Teorema della decomposizione

$$\text{Se ho } X \rightarrow YZ \text{ allora } X \rightarrow Y \text{ e } X \rightarrow Z$$

Dimostrazione:

Parto da $X \rightarrow YZ$. Sappiamo che $YZ \rightarrow Y$ (per riflessività), e anche che $YZ \rightarrow Z$ (per riflessività). Da ognuna delle due dipendenze possiamo ottenere, dato $X \rightarrow YZ$, rispettivamente $X \rightarrow Y$ e $X \rightarrow Z$ (entrambe per transitività).

Esempio:

Se $\text{MATR} \rightarrow \text{NS}, \text{CFS}$ allora $\text{MATR} \rightarrow \text{NS}$ e $\text{MATR} \rightarrow \text{CFS}$

4.6) Teorema della pseudo-transitività e teorema del prodotto (poco usati)*Se ho $X \rightarrow Y$ e $WY \rightarrow Z$ allora $WX \rightarrow Z$* *Dimostrazione:*

Parto da WX . Abbiamo che $WX \rightarrow X$ (per riflessività) e dato che $X \rightarrow Y$ (per ipotesi) ottengo $WX \rightarrow Y$ (per transitività). Dopodiché espando con W ed ottengo $WWX \rightarrow WY$ ovvero $WX \rightarrow WY$ (per espansione). Ora, avendo $WX \rightarrow WY$ e $WY \rightarrow Z$ (per ipotesi) ottengo $WX \rightarrow Z$ (per transitività).

Esempio:

Se $CFS \rightarrow MATR$ e $MATR, Co \rightarrow Vo$ allora $Co \rightarrow Vo$.

4.7) Teorema del prodotto (poco usato)*Se ho $X \rightarrow Y$ e $W \rightarrow Z$ allora $XW \rightarrow YZ$* *Dimostrazione:*

Parto da WX . Abbiamo che $WX \rightarrow X$ (per riflessività) e dato che $X \rightarrow Y$ (per ipotesi) ottengo $WX \rightarrow Y$ (per transitività). Riparto da WX e ottengo $WX \rightarrow W$ (per riflessività) e dato che $W \rightarrow Z$ (per ipotesi) ottengo $WX \rightarrow Z$ (per transitività). Non resta che, dato $WX \rightarrow Y$ e $WX \rightarrow Z$ applicare l'unione ed ottenere $WX \rightarrow YZ$ (per unione).

Esempio:

Se $MATR \rightarrow CFS$ e $Q \rightarrow TU$ allora $MATR, Q \rightarrow CFS, TU$.

Si noti che potremmo invertire l'assioma di unione con quello di espansione, infatti il teorema dell'unione è dimostrabile usando l'espansione come assioma.

Unione: Se $X \rightarrow Y$ e $X \rightarrow Z$ allora $X \rightarrow YZ$

Parto da $X \rightarrow Y$ (ipotesi). Per espansione ottengo $ZX \rightarrow ZY$. Dopodiché da $X \rightarrow Z$ (ipotesi) ottengo $X \rightarrow ZX$ (per riflessione). Quindi per transitività tra $X \rightarrow ZX$ e $ZX \rightarrow YZ$ ottengo $X \rightarrow YZ$.

5) Applicazioni della teoria di Armstrong sulle dipendenze funzionali: le chiusure

Una volta definita cosa sia la teoria di Armstrong, è importante vederne le applicazioni rispetto al nostro argomento di studio: le dipendenze funzionali.

5.1) Chiusura di un insieme F

Un insieme F di dipendenze funzionali ha una chiusura F^+ così definita:

$$F^+ = \{X \rightarrow Y \mid F \vdash X \rightarrow Y\}$$

ovvero è l'insieme delle dipendenze che sono deducibili partendo da F, ovvero date le dipendenze funzionali di una relazione si hanno tutte le varie dipendenze funzionali ottenibili utilizzando le regole della teoria di Armstrong su F (abbiamo quindi tutte le riflessive, ecc.)

Esistono vari algoritmi per ottenere la F^+ di un certo F, ma sono algoritmi non interessanti poiché esponenziali nel numero di dipendenze di F.

5.2) La nozione di equivalenza

Consideriamo un semplice esempio, abbiamo due progettisti vedono il mondo in questo modo:

Progettista 1 (F)	Progettista 2 (G)	Poniamoci ora una semplice domanda: le due visioni sono in qualche modo "uguali"? Cioè, descrivono formalmente lo stesso mondo? E quindi ancora, possiamo passare da una visione all'altra?
ABC → D B → C	AB → D B → C	

Proviamo a passare dalla visione 1 (F) alla visione 2 (G). Ovvamente B → C non è da provare, essendo già presente in entrambe le visioni. Occupiamoci quindi di ottenere AB → D.

Partendo da AB abbiamo AB → B (per riflessività). Poi, dato B → C (per ipotesi da F) possiamo ottenere AB → C (per transitività). Quindi possiamo dire che AB → ABC (per espansione con AB). Possiamo quindi affermare che AB → D (per transitività AB → ABC e ABC → D (per ipotesi da F)).

È possibile dimostrare la cosa in senso inverso e quindi partendo da G arrivare ad F? Cerchiamo di dedurre ABC → D. Partendo da ABC e posso affermare che ABC → AB (per riflessività). Dato che so che AB → D (per ipotesi) allora è facile ottenere ABC → D (per transitività ABC → AB e AB → D).

Se possiamo passare da G ad F, possiamo dire che F e G sono **equivalenti** (in notazione $F \equiv G$). Questo ci dice una cosa ben più importante ovvero che $F^+ = G^+$ (d'altronde la chiusura comprende tutte le trasformazioni possibili quindi è ovvio che sia così).

Quindi,

se da $F \vdash G$ (da F deduco G) significa che $G^+ \subseteq F^+$

se da $G \vdash F$ (da G deduco F) significa che $F^+ \subseteq G^+$

Pertanto se $F \vdash G$ e $G \vdash F$ è chiaro che $F^+ = G^+$.

5.3) La chiusura di un insieme di attributi

Sia X un insieme di attributi ($X \subseteq A$) in una relazione $(r(A), F)$. Allora la chiusura di un insieme di attributi X è:

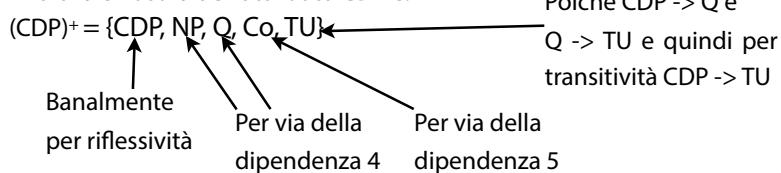
$$X^+_F = \{X_i \mid F \vdash X \rightarrow X_i\}$$

Ovvero la chiusura di un insieme di attributi X è quell'insieme di attributi che posso dedurre secondo le regole di F partendo da X.

Riprendiamo l'insieme F dell'esempio iniziale (leggermente ricostituito):

- 1) MATR \rightarrow NS, IND
- 2) MATR \rightarrow CFS, DN
- 3) IND \rightarrow CAP
- 4) CDP \rightarrow NP, Q
- 5) CDP \rightarrow Co
- 6) Q \rightarrow TU
- 7) CFS \rightarrow MATR
- 8) MATR, Co \rightarrow Vo, DE, CDP

Allora la chiusura dell'attributo CDP è:



Le proprietà

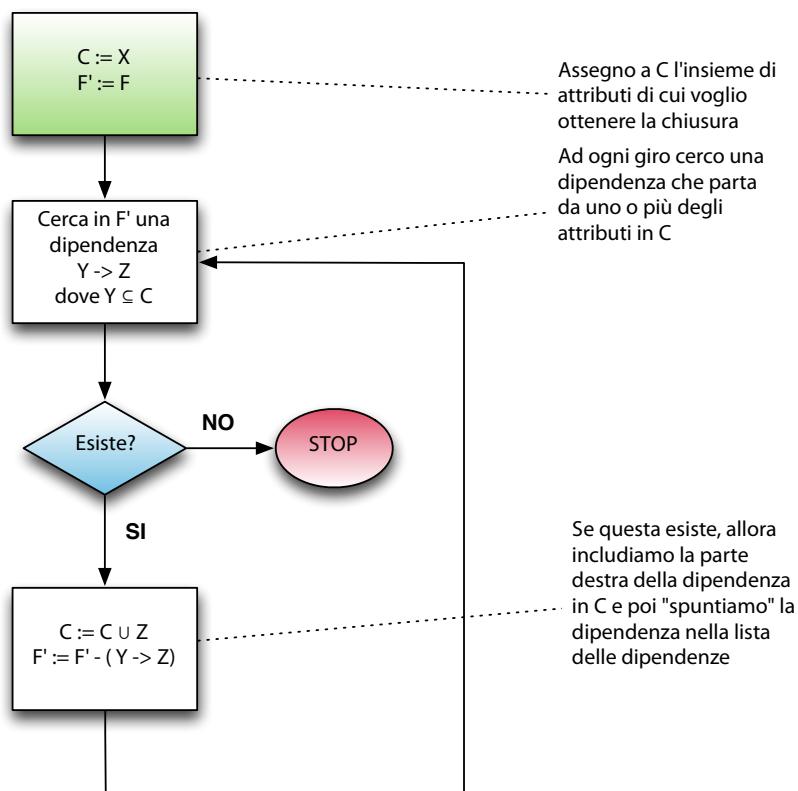
Possiamo dedurre due semplici proprietà di X^+ :

- $X \subseteq X^{+F}$ (banalmente per riflessività)
- Date le varie $X \rightarrow X, X \rightarrow X_1, X \rightarrow X_2, \dots, X \rightarrow X_n$ possiamo affermare che $X \rightarrow X_1X_2\dots X_n$

5.4) Algoritmo per il calcolo della chiusura di un insieme di attributi

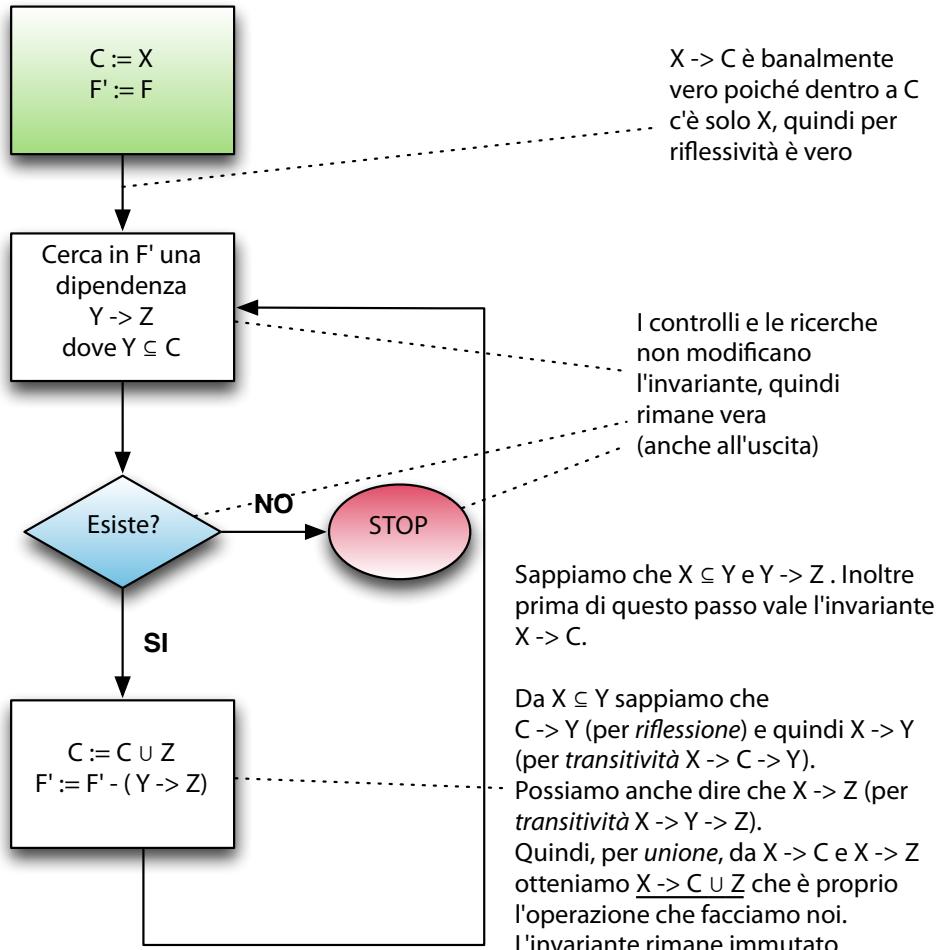
La chiusura di un insieme di attributi diventa utile quando troviamo un meccanismo meccanico con cui calcolarla.

Vediamo questo algoritmo:



Alla fine dell'algoritmo otterremo che $C = X^+$.

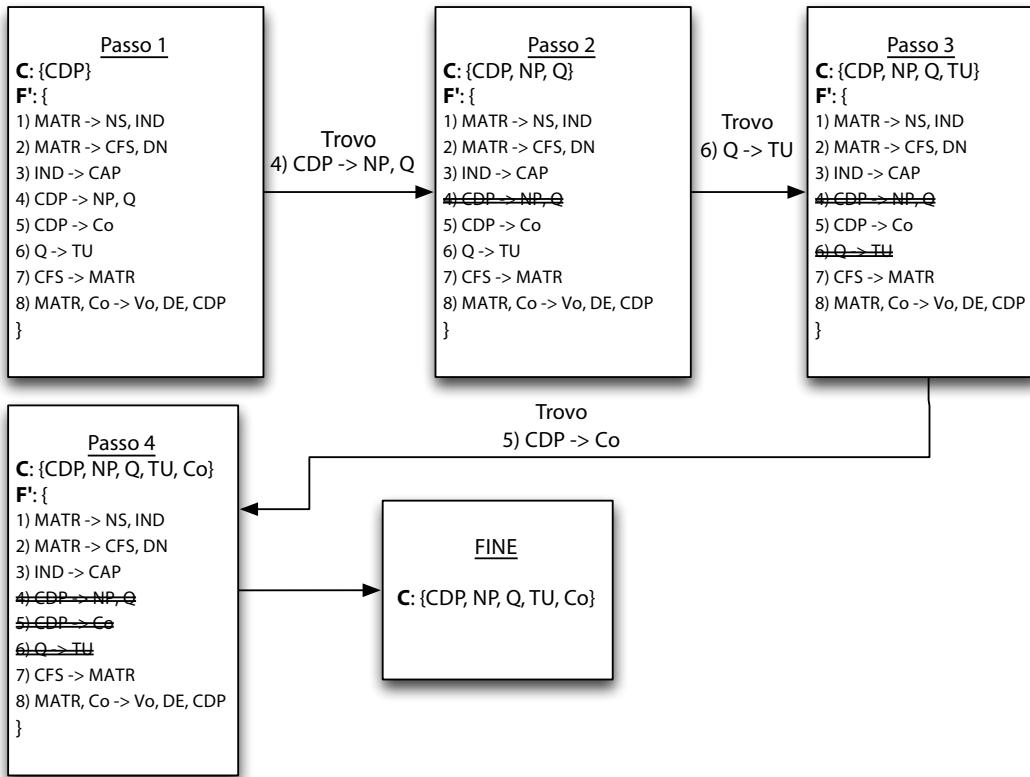
Questo è facilmente intuibile, ma chiarifichiamolo tramite l'invariante di ciclo ovvero che $X \rightarrow C$ (infatti se $X \rightarrow C$ allora vale la seconda proprietà della chiusura di un insieme di attributi ovvero che $X \rightarrow X^+$ e quindi $C = X^+$).



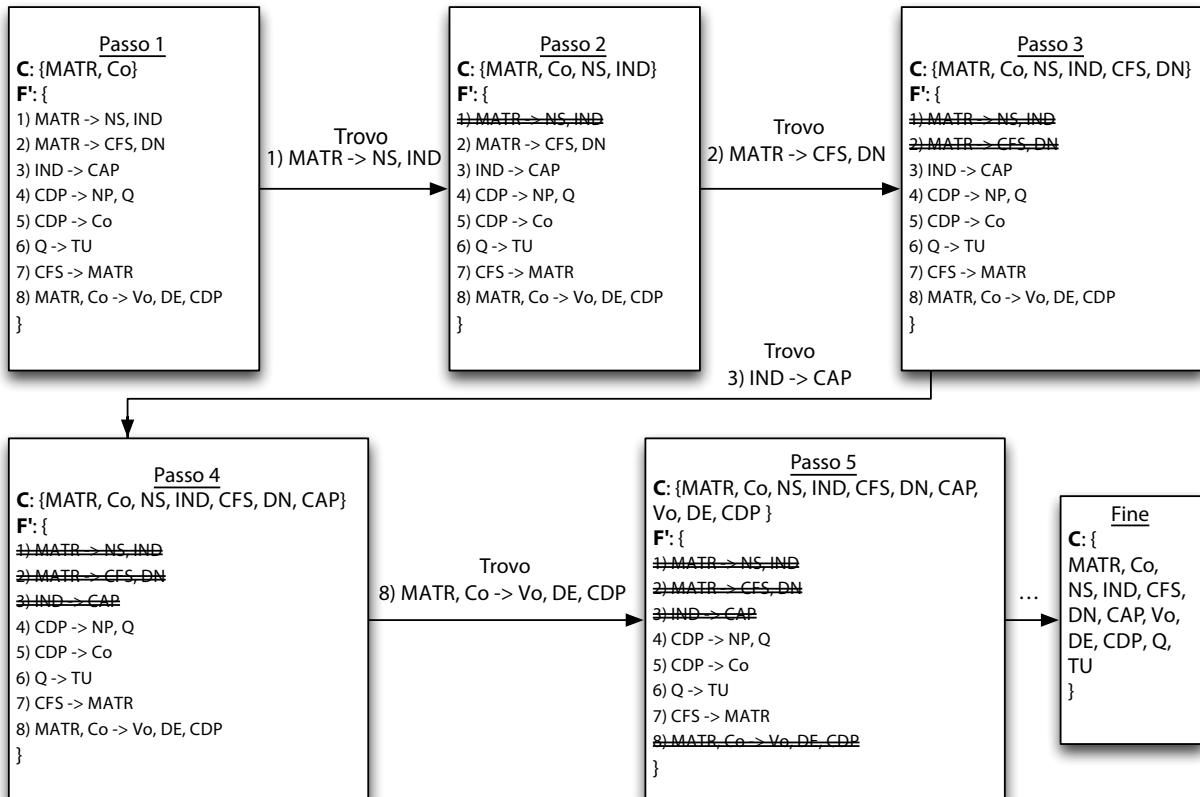
La complessità dell'algoritmo è **polinomiale** nel numero di elementi di F .

Esempio 1 di applicazione dell'algoritmo: (CDP)⁺

Riprendiamo (CDP)⁺ e otteniamo lo stesso risultato di prima.



Esempio 2 di applicazione dell'algoritmo: (MATR, Co)⁺



Si può facilmente notare come questo esempio sia un caso particolare: questa accoppiata "fa dipendere" tutti gli attributi. Potremmo forse collegarla al concetto di chiave primaria? (vedremo dopo!) Notiamo che anche $(CFS, Co)^+$ ha la stessa caratteristica.

5.5) Riflessioni finali (importanti) in merito alle chiusure

In seguito a quanto detto, possiamo affermare che:

$$F \vdash X \rightarrow Y \equiv Y \subseteq X^+_{\mathcal{F}}$$

Ovvero dire che da F è deducibile che Y dipende da X , è come dire che Y è incluso nella chiusura di X e soprattutto **viceversa!**

La grande potenza è che abbiamo trasformato un problema *logico* in un problema *meccanico*.

Esempio

Ci chiediamo se CAP dipenda da MATR ($MATR \rightarrow CAP$?)

Dato che $(MATR)^+ = \{MATR, NS, IN, DN, CFS, CAP\}$ si ha che $CAP \subseteq (MATR)^+$ e allora possiamo affermare che è vero che $MATR \rightarrow CAP$.

6) Applicazioni della teoria di Armstrong sulle dipendenze funzionali: Individuare le chiavi

Un'altra importante applicazione della teoria di Armstrong è quella della ricerca delle chiavi in una relazione.

6.1) Dalle dipendenze alle chiavi: somiglianza fra definizioni

Prendiamo la definizione di chiave:

Un insieme di attributi $K \subseteq A$ è chiave candidata se è superchiave ed è minimale.

Un insieme di attributi $SK \subseteq A$ è superchiave se:

$$\forall t_1, t_2 \in r (t_1[SK] = t_2[SK] \Rightarrow t_1[A] = t_2[A])$$

Prendiamo ora la definizione di dipendenza funzionale:

Se vale la dipendenza funzionale $X \rightarrow Y$ allora:

$$\forall t_1, t_2 \in r (t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y])$$

Con le conoscenze che abbiamo ora, possiamo chiaramente affermare che essere superchiavi è un caso particolare delle dipendenze funzionali dove $Y = A$.

6.2) La ricerca delle chiavi: un approccio metodologico

Grande vantaggio dell'aver notato la definizione sopra è il fatto che ora possediamo un approccio metodologico alla ricerca delle chiavi.

Il progettista può congetturare una chiave, ovvero supporre che $F \vdash ? K \rightarrow A$.

Allora per verificare se ciò che dice è vero, è sufficiente controllare che $X^+_{\mathcal{F}}$ sia uguale ad A (paragrafo 5.5).

Esempio I

Congetturiamo $F \vdash ? (CFS, NP) \rightarrow \{ MATR, Co, NS, IND, CFS, DN, CAP, Vo, DE, CDP, Q, TU \}$ cioè che (CFS, NP) sia superchiave.

Calcoliamo la chiusura: $(CFS, NP)^+ = \{ CFS, NP, MATR, NS, DN, IND, CAP \}$. Dato che questo insieme non è A , allora (CFS, NP) non è superchiave.

Esempio II

Congetturiamo $F \vdash ? (MATR, Co, DE) \rightarrow \{ MATR, Co, NS, IND, CFS, DN, CAP, Vo, DE, CDP, Q, TU \}$

Andando al calcolo: $(MATR, Co, DE)^+ = \{ MATR, Co, NS, IND, CFS, DN, CAP, Vo, DE, CDP, Q, TU \}$ e quindi possiamo affermare che $(MATR, Co)$ sia **superchiave**.

Ma come arriviamo alla chiave candidata? Non resta che verificare tutte le varie combinazioni.

$(\text{MATR}, \text{DE}) = \{ \text{MATR}, \text{DE}, \text{NS}, \text{IND}, \text{CFS}, \text{DN}, \text{CAP} \}$ non è superchiave

$(\text{DE}, \text{Co}) = \{ \text{DE}, \text{Co} \}$ non è superchiave

$(\text{MATR}, \text{Co}) = \{ \text{MATR}, \text{Co}, \text{NS}, \text{IND}, \text{CFS}, \text{DN}, \text{CAP}, \text{Vo}, \text{DE}, \text{CDP}, \text{Q}, \text{TU} \}$ è **superchiave!**

Ma sarà minima?

$(\text{MATR}) = \{ \text{MATR}, \text{DE}, \text{NS}, \text{IND}, \text{CFS}, \text{DN}, \text{CAP} \}$ non è superchiave

$(\text{Co}) = \{ \text{Co} \}$ non è superchiave

Abbiamo quindi dedotto che (MATR, Co) sia superchiave minimale: una chiave candidata.

Si noti che nell'esempio di prima ne abbiamo altre (anche "insospettabili"):

- (CFS, Co)
- $(\text{MATR}, \text{CDP})$
- (CFS, CDP)

Si noti che l'algoritmo ha tempo di esecuzione esponenziale nel numero di attributi considerati.

6.3) Caso particolare: attributi non citati in F

Prendiamo questo piccolo esempio:

$r(A, B, C, D, E)$ che ha come $F: \{ B \rightarrow CD, CD \rightarrow E \}$

Notiamo subito che l'attributo **A** non è coinvolto in F . Da questo possiamo dedurre che non esisterà mai nessun insieme di chiusura su alcun attributo che includerà A , e data la definizione di chiave (ovvero una dipendenza che include tutti gli attributi) non potremo mai trovare una chiave candidata per questa relazione.

In questi casi si aggiunge all'insieme di cui si sta facendo la chiusura l'insieme degli attributi non coinvolti. Quindi, per esempio, $(B)^+ = \{ B, C, D, E \}$ diventa $(B)^+ = \{ A, B, C, D, E \}$.

Ulteriore caso particolare di questo caso particolare è un **insieme delle dipendenze vuoto**. In questo caso è chiave candidata l'insieme degli attributi stesso poiché tutti gli attributi non sono coinvolti e quindi tutti gli attributi vanno aggiunti alla chiave.

6.4) Stesse chiavi da insiemi equivalenti

L'ultimo aspetto da considerare è quello che deriva da una situazione classica: due progettisti vedono la realtà in modo differente e definiscono i due insiemi di dipendenze funzionali F e G .

Si calcola che $F \equiv G$ (cioè $F^+ = G^+$) poiché i due hanno fatto un buon lavoro: questo significa che dato un certo insieme di attributi X , la sua chiusura in F è uguale alla sua chiusura in G , ovvero in simboli $X^+_F = X^+_G$. Dimostriamolo!

Per dimostrare una uguaglianza si dimostrano separatamente le due parti della congiunzione: $X^+_F \subseteq X^+_G \wedge X^+_G \subseteq X^+_F$

La prima parte è facilmente dimostrabile, infatti:

Prendiamo un insieme di attributi $X_i \in X^+_F$, allora per definizione $F \vdash X \rightarrow X_i$ (altrimenti X_i non farebbe parte di X^+_F !) e da ciò si ottiene che, certamente, $X \rightarrow X_i \in F^+$. Ma dato che $F^+ = G^+$ allora è anche vero che $X \rightarrow X_i \in G^+$. Dato questo fatto, $G \vdash X \rightarrow X_i$ e quindi si può dire che $X_i \in X^+_G$.

La seconda parte, speculare alla prima:

Prendiamo un insieme di attributi $X_i \in X^+_G$, allora per definizione $G \vdash X \rightarrow X_i$ (altrimenti X_i non farebbe parte di X^+_G !) e da ciò si ottiene che, certamente, $X \rightarrow X_i \in G^+$. Ma dato che $G^+ = F^+$ allora è anche vero che $X \rightarrow X_i \in F^+$. Dato questo fatto, $F \vdash X \rightarrow X_i$ e quindi si può dire che $X_i \in X^+_F$.

Allora è vero sia che $X^+_F \subseteq X^+_G$ che $X^+_G \subseteq X^+_F$. Dunque, $X^+_F = X^+_G$.

7) La decomposizione delle relazioni

L'operazione di normalizzazione si concretizza tramite la suddivisione di una relazione in relazioni più piccole e semplici, questo per evitare le anomalie. Prendiamo un nuovo database su cui lavorare per gli esempi:

Esame(MATR, NomeS, Voto, Corso, Voto, CodCorso, Titolare), abbreviabile:

MATR	NomeS	Voto	Corso	CodC	Titolare
31	Piero	27	Database	1	Giolito
31	Piero	26	Algoritmi	2	Bono
37	Giovanni	25	Database	3	Demo

7.1) Il fenomeno delle tuple spurie

Pensiamo di dividere questa relazione in due relazioni usando l'attributo Corso in comune fra i due, per poi, con un join, ricomporre la tabella. Abbiamo rispettivamente $r_1(\text{MATR}, \text{NomeS}, \text{Voto}, \text{Corso})$ e $r_2(\text{Corso}, \text{CodC}, \text{Titolare})$.

MATR	NomeS	Voto	Corso	Corso	CodC	Titolare
31	Piero	27	Database	Database	1	Giolito
31	Piero	26	Algoritmi	Algoritmi	2	Bono
37	Giovanni	25	Database	Database	3	Demo

A questo punto ci aspetterebbe che con un natural-join (equi-join senza ripetizione di attributi) possiamo ricomporre la tabella, perciò ci aspettiamo che $r_1 \bowtie r_2 = \text{Esame}$. Ma provando ad eseguirlo...

MATR	NomeS	Voto	Corso	CodC	Titolare
31	Piero	27	Database	1	Giolito
31	Piero	27	Database	3	Demo
31	Piero	26	Algoritmi	2	Bono
37	Giovanni	25	Database	1	Giolito
37	Giovanni	25	Database	3	Demo

Si presenta il fenomeno delle **tuple spurie** ovvero ci sono tuple che prima non c'erano, e queste non sono un valore aggiunto ma bensì distruggono le informazioni (infatti ora risulta che Piero e Giovanni abbiano dato Database con entrambi i professori Demo e Giolito!)

Pertanto, presa una relazione $r(A)$ dividiamo gli attributi in due insiemi A_1 ed A_2 dove $A_1 \cup A_2 = A$. Dopodiché definiamo:

- $r_1(A_1) = \Pi_{A_1} r(A)$
- $r_2(A_2) = \Pi_{A_2} r(A)$

E quindi, come abbiamo appena visto, possiamo affermare che in generale

$$|r(A)| \leq |r_1(A_1) \bowtie r_2(A_2)|$$

e quindi possiamo dire che $r_1(A_1) \bowtie r_2(A_2) - r(A) \rightarrow \text{tuple spurie}$.

7.2) La decomposizione con join senza perdita di informazioni

Dunque siamo interessati a trovare un modo per evitare questo fenomeno delle tuple spurie. Più formalmente:

Dato $R(A)$ ottenere due relazioni $R_1(A_1)$ ed $R_2(A_2)$ con $A_1 \cup A_2 = A$ e con

- $R_1(A_1) = \Pi_{A_1} R(A)$
- $R_2(A_2) = \Pi_{A_2} R(A)$

dove però valga la regola: $r(A) = r_1(A_1) \bowtie r_2(A_2)$

Quindi, in parole, $R_1(A_1)$ e $R_2(A_2)$ è una decomposizione con join senza perdita di informazioni se per ogni $r(A)$ corretta si verifica l'uguaglianza $r(A) = \Pi_{A_1} R(A) \bowtie \Pi_{A_2} R(A)$.

Per risolvere questo problema si adoperano le dipendenze funzionali, quindi consideriamo $(R(A), F)$.

7.3) Teorema per la decomposizione con join senza perdita di informazioni

Condizione sufficiente e necessaria affinché una decomposizione sia senza perdita di informazioni è:

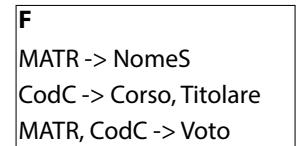
$$F \vdash A_1 \cap A_2 \rightarrow A_1 \vee F \vdash A_1 \cap A_2 \rightarrow A_2$$

Cioè $A_1 \cap A_2$ deve essere superchiave di $R_1(A_1)$ oppure $A_1 \cap A_2$ deve essere superchiave di $R_2(A_2)$.

Esempio

Riprendiamo la relazione di prima e aggiungiamo le dipendenze:

MATR	NomeS	Voto	Corso	CodC	Titolare
------	-------	------	-------	------	----------



Una buona decomposizione può essere:

MATR	NomeS	Voto	CodC	Corso	CodC	Titolare
------	-------	------	------	-------	------	----------

infatti in questo caso $(\text{MATR}, \text{NomeS}, \text{Voto}, \text{CodC}) \cap (\text{Corso}, \text{CodC}, \text{Titolare}) = \text{CodC}$

e come si legge da F, $\text{CodC} \rightarrow \text{Corso, Titolare}$ e banalmente per espansione $\text{CodC} \rightarrow \text{Corso, Titolare, CodC}$ che è la totalità degli attributi di E_2 . Quindi è verificata la regola sopracitata.

Dimostrazione

Per semplicità dimostreremo solamente la condizione di sufficienza, anche se la condizione è anche necessaria.

Ipotesi: $F \vdash A_1 \cap A_2 \rightarrow A_1 \vee F \vdash A_1 \cap A_2 \rightarrow A_2$

Tesi: ogni $r(A)$ corretta è componibile con join senza perdita di informazioni in $R_1(A_1) = \Pi_{A_1} R(A)$ e $R_2(A_2) = \Pi_{A_2} R(A)$

Utilizzeremo una tecnica visiva per dimostrare questo teorema. Tracciamo le due relazioni R_1 ed R_2 con tuple t_1 e t_2 .

Indichiamo con ... e /// attributi che non fanno parte dell'intersezione $A_1 \cap A_2$, mentre quelle facenti parte con ~~~.

R ₁		R ₂	
...	~~~	~~~	///
...	~~~	~~~	///

Dopo il join avremo certamente questa situazione:

R		
...	~~~	///
...	~~~	
	~~~	///

Il nostro obiettivo è dimostrare queste due condizioni:

$$R(A) \subseteq r_1(A_1) \bowtie r_2(A_2) \wedge r_1(A_1) \bowtie r_2(A_2) \subseteq R(A)$$

per poter quindi affermare che  $R(A) = r_1(A_1) \bowtie r_2(A_2)$ .

Ma la prima metà è ovviamente provata perché  $R(A) \subseteq r_1(A_1) \bowtie r_2(A_2)$  è vero per la definizione del join stesso!

Non resta che provare che  $r_1(A_1) \bowtie r_2(A_2) \subseteq R(A)$ .

Simuliamo il join tra le nostre due tuple  $t_1$  e  $t_2$ , abbiamo:

...	~~~	✖	~~~	///
-----	-----	---	-----	-----

quindi: 

...	~~~	///
-----	-----	-----

Dimostrando che questa tupla  $\in r(A)$  abbiamo dimostrato il tutto.

Entrano ora in gioco le **dipendenze funzionali!** Dato che  $A_1 \cap A_2 \rightarrow A_1$  siamo certi del fatto che  $~~~$  sia in comune alle due tuple. Ma allora  $R$  non è altro che:

come volevasi dimostrare.

...	~~~	///
...	~~~	///

## 8) La decomposizione delle dipendenze funzionali

Quando una relazione viene scissa è necessario dividere anche le dipendenze funzionali per mantenere l'aspetto della realtà. Potrebbe sembrare banale, ovvero le relazioni ereditano quelle dipendenze che coinvolgono gli attributi delle relazioni stesse (quindi  $F_1 \cup F_2 = F$ ), ed è così, ma ci sono dei casi particolari.

### Caso particolare I°

Prendiamo una relazione  $R(..., A, ..., B, ..., C, ...)$  e consideriamo le dipendenze funzionali:

$$A \rightarrow B \quad B \rightarrow C \quad A \rightarrow C$$

Scindiamo poi rispetto ad  $A \rightarrow B$ , allora avremmo  $R_1(..., A, ..., C, ...)$  ed  $R_2(A, B)$ . Abbiamo  $F_1 = \{ A \rightarrow C \}$   $F_2 = \{ A \rightarrow B \}$  e notiamo che  $B \rightarrow C$  è sparito quindi  $F_1 \cup F_2 \neq F$ . Allora  $B \rightarrow C$  dovrà essere implementato come **vincolo globale**.

### Caso particolare II°

Prendiamo una relazione  $R(..., A, ..., B, ..., C, ...)$  e consideriamo le dipendenze funzionali:

$$A \rightarrow B \quad B \rightarrow C$$

Scindiamo rispetto a  $A \rightarrow B$  e troviamo  $R_1(..., A, ..., C, ...)$  ed  $R_2(A, B)$ . Ma questa volta  $F_1 = \{ \emptyset \}$   $F_2 = \{ A \rightarrow B \}$ . In questo caso abbiamo un **ERRORE** poiché  $F_1$  è vuoto ma in realtà vige  $A \rightarrow C$  anche se non esplicitamente scritto in  $F$ .

### 8.1) Restrizioni di $F$ in $R_i(A_i)$

Date le cose dette prima possiamo sostenere che:

1)  $F_i = \{ X \rightarrow Y \mid F \vdash X \rightarrow Y \wedge X, Y \subseteq A_i \}$  (ovvero  $F_i$  coinvolge quegli attributi che sono negli attributi di  $R_i$ ) e che

2) Una decomposizione mantiene le dipendenze se  $(F_1 \cup F_2)^+ = F^+$

*Dimostrazione*

Se  $(F_1 \cup F_2)^+ = F^+$  allora  $(F_1 \cup F_2)^+ \subseteq F^+$  e  $F^+ \subseteq (F_1 \cup F_2)^+$ . La seconda è banalmente vera per il punto 1. Per la definizione di chiusura, per provare la prima parte è sufficiente dimostrare che  $(F_1 \cup F_2) \vdash F$ .

### 9) Approccio alle forme normali

Ora che disponiamo di tutti i mezzi possiamo finalmente parlare delle forme normali, che sono come "ricette" da seguire. Facciamone prima una lista e poi esaminiamo i più rilevanti separatamente.

- **I° forma normale:** ne abbiamo già parlato, una relazione è in prima forma normale se non contiene attributi multivalore o che contengono a loro volta tabelle
- **II° forma normale:** importante storicamente, la tralasciamo
- **III° forma normale:** la studieremo attentamente in seguito
- **Boyce Codd Normal Form (BCNF):** la studieremo attentamente in seguito
- **IV° forma normale**  
ed altre ancora.

La cosa importante è che ognuna delle forme normali elencate implica tutte le precedenti (ovvero sono affinamenti continui).

### 10) Boyce Codd Normal Form (BCNF)

Boyce e Codd inventano questa forma normale.

#### **10.1) Condizioni di appartenenza BCNF**

Una relazione  $(R(A), F)$  è in BCNF se per ogni  $X \rightarrow Y \in F$  vale almeno una delle seguenti condizioni:

- 1)  $X \rightarrow Y$  è **banale**, ovvero  $Y \subseteq X$  (si tratta quindi di una forma riflessiva)
- 2)  $X$  è **superchiave**

#### **10.2) Esempi di applicazione empirica della definizione**

##### *Esempio 1*

Consideriamo la relazione Persona(CF, Nome, DataNascita, Indirizzo) con  $F = \{CF \rightarrow \text{Nome}, \text{DataNascita}, \text{Indirizzo}\}$ . Dato che CF è chiaramente chiave (infatti da CF dipende tutta A) allora la relazione è in BCNF.

##### *Esempio 2*

Consideriamo la relazione Studente(MATR, CF, Nome, DataNascita, Indirizzo) con

$$\begin{aligned} F = \{ & CF \rightarrow \text{Nome}, \text{DataNascita}, \text{Indirizzo} \\ & CF \rightarrow \text{MATR} \\ & \text{MATR} \rightarrow CF \\ \} \end{aligned}$$

Controlliamo che sia BCNF verificando le chiusure.

$$(CF)^+ = \{CF, \text{Nome}, \text{DataNascita}, \text{Indirizzo}, \text{MATR}\} = A \quad \text{OK}$$

$$(\text{MATR})^+ = \{\text{MATR}, CF, \text{Nome}, \text{DataNascita}, \text{Indirizzo}\} = A \quad \text{OK}$$

Tutte e tre le produzioni rispettano la seconda condizione, quindi Studente è BCNF.

##### *Esempio 3*

Consideriamo la relazione Esame(MATR, NS, Co, Vo, CDP, Q, TU) con

$$\begin{aligned} F = \{ & \text{MATR}, \text{Co} \rightarrow \text{Vo}, \text{CDP} \\ & \text{MATR} \rightarrow \text{NS} \\ & \text{CDP} \rightarrow \text{Q} \\ & \text{Q} \rightarrow \text{CDP} \\ \} \end{aligned}$$

Per via delle dipendenze  $CDP \rightarrow Q$ ,  $Q \rightarrow CDP$  ed  $CDP \rightarrow Q$  le cui chiusure sono

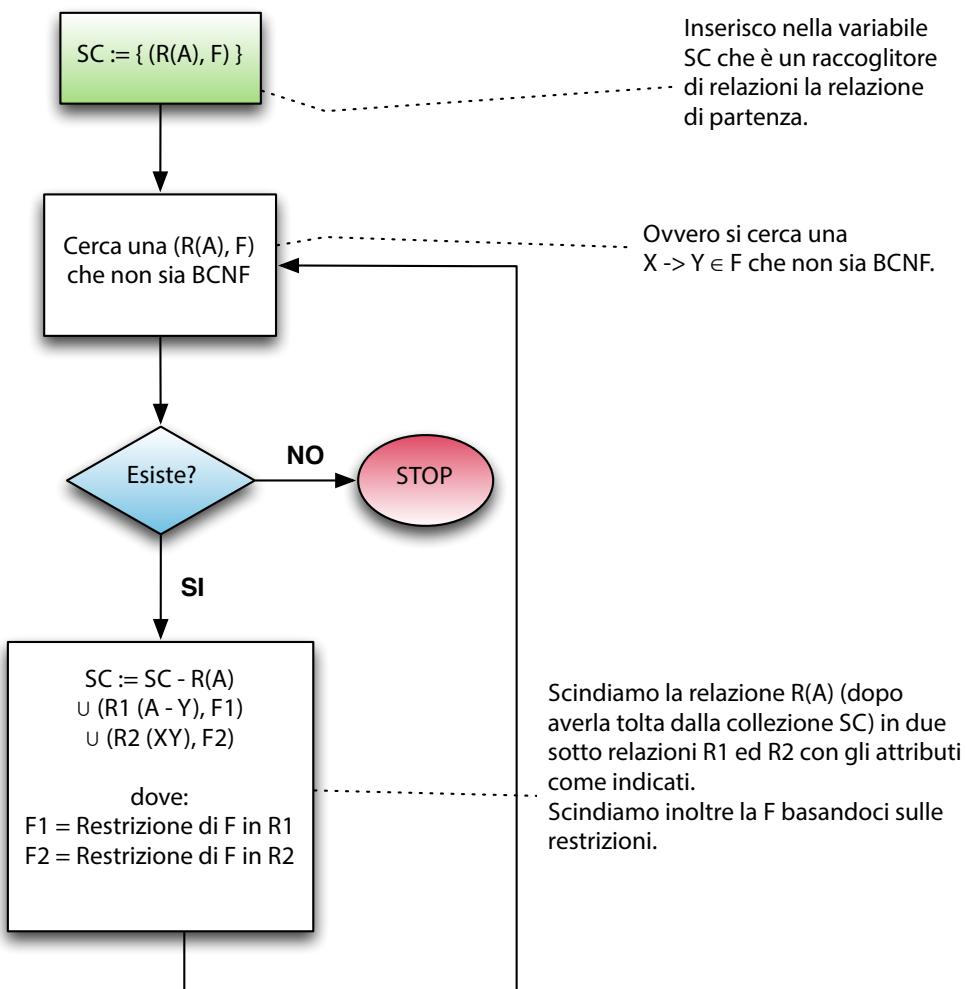
$$(Q)^+ = \{ Q \}$$

$$(CDP)^+ = \{ CDP \}$$

$$(MATR)^+ = \{ MATR, NS \}$$

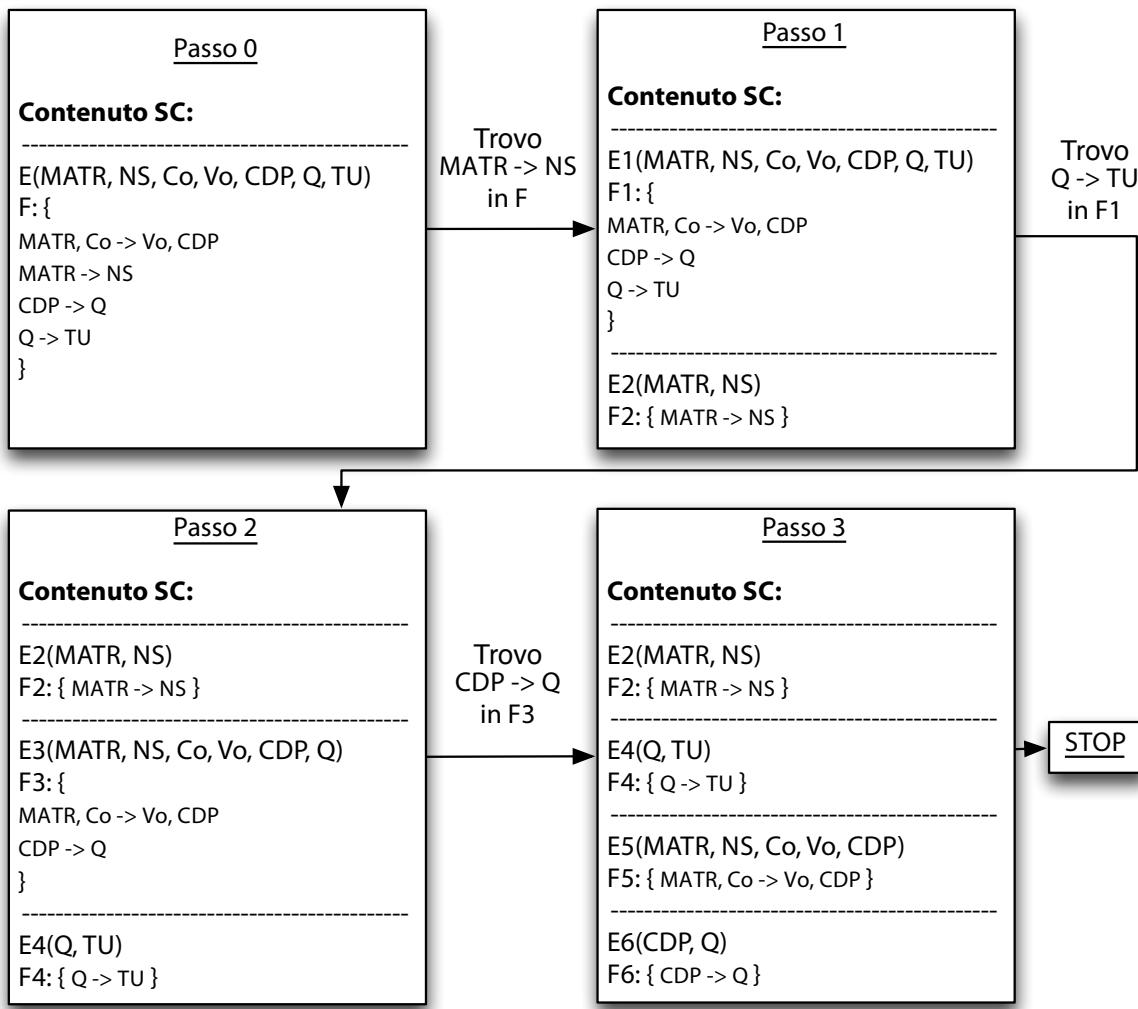
si ha che tale relazione non è BCNF poiché tali chiusure non sono A. Come risolvere la questione?

### 10.3) Algoritmo per la trasformazione in BCNF



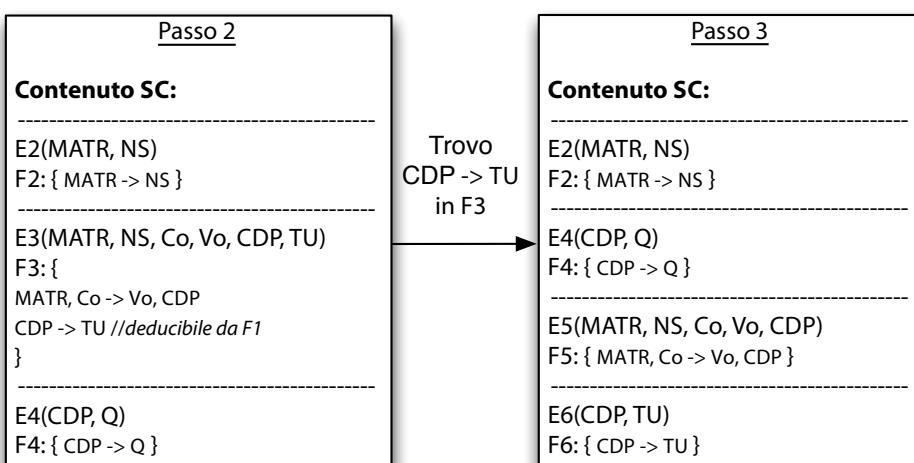
Notiamo che l'algoritmo è molto semplice ed ha costo **esponenziale**.

Eseguiamo ora l'algoritmo sul problema posto prima, dopodiché faremo alcune considerazioni sull'algoritmo stesso.

**Riflessioni in merito all'algoritmo**

Potrebbe sembrare, a prima vista, che vi siano due congettive che sono vere solo in questo caso.

- 1 - *"Ogni volta che effettuiamo una decomposizione otteniamo una relazione BCNF"*. Ma questo non è vero.
- 2 - *"L'algoritmo mantiene le dipendenze in ogni caso"*. Ma anche questo non è vero, poiché se al passo 1 avessimo scelto CDP -> Q rispetto a Q -> TU allora avremmo ottenuto a fine algoritmo:



Ma attenzione! Se ora proviamo a ricomporre con  $F_6 \cup F_4$  non troveremo più  $CDP \rightarrow Q$  e  $Q \rightarrow TU$  ( $F_1$ ) ma bensì  $CDP \rightarrow TU$ ,  $Q$ . La prima è ancora deducibile ma la seconda invece no, infatti provando a calcolare  $(Q)^+$  troviamo solo  $\{Q\}$  e non  $\{Q, TU\}$  che ci avrebbe indicato che era deducibile  $Q \rightarrow TU$  (che, appunto, avevamo all'inizio).  $Q \rightarrow TU$  va aggiunto come vincolo globale.

#### 10.4) Considerazioni sull'algoritmo per la trasformazione in BCNF

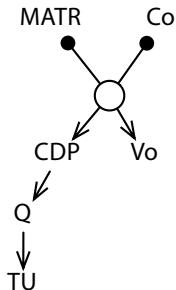
Formalizziamo le cose che abbiamo appreso:

- * L'algoritmo termina ed è corretto (termina perché mano a mano nella fase di split riduciamo la quantità di attributi delle relazioni).
- * L'algoritmo non è in grado di mantenere le dipendenze in ogni caso (non ha criterio di scelta ma è sensibile all'ordine di selezione).
- * La decomposizione con join è senza perdita di informazioni, infatti  $(A - Y) \cap (XY) = X$  e dato che  $X \rightarrow Y$  (l'abbiamo trovata nell'algoritmo) si ha per espansione che  $X \rightarrow XY$ . Abbiamo quindi verificato la condizione  $F \vdash A_1 \cap A_2 \rightarrow A_2$ .
- * L'algoritmo ha costo esponenziale.

Il secondo punto è semi-risolubile con un metodo **euristico** se vediamo le dipendenze funzionali con un grafo, nell'esempio di prima (leggermente modificato in Esame(MATR, Co, Vo, CDP, Q, TU) con

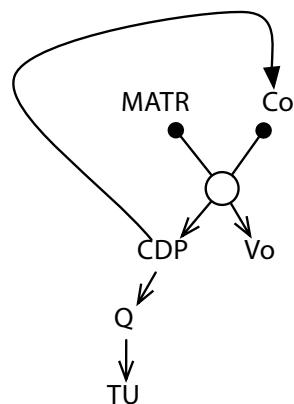
$$F = \{ \begin{array}{l} \text{MATR, Co} \rightarrow \text{Vo, CDP} \\ \text{CDP} \rightarrow \text{Q} \\ \text{Q} \rightarrow \text{CDP} \end{array} \}$$

avremmo:



Non è difficile capire che, **partendo dalle foglie**, evitiamo quel problema di prima perché non perdiamo attributi "in mezzo".

Questa tecnica risolve il problema in ogni caso? La risposta purtroppo è no, infatti se considerassimo anche una eventuale dipendenza  $CDP \rightarrow Co$  il nostro grafo muterebbe in:



E quindi a un certo passo avremo:

$E_1(\text{MATR, Co, Vo, CDP})$	$E_2(Q, TU)$	$E_3(CDP, Q)$
$F_1 \{$	$F_2 \{ Q \rightarrow TU \}$	$F_3 \{ CDP \rightarrow Q \}$
$\text{MATR, Co} \rightarrow \text{Vo, CDP}$		
$\text{CDP} \rightarrow \text{Co}$		
}		

E quindi creeremo ancora le due relazioni:

$E_4(\text{MATR, Vo, CDP})$	$E_5(CDP, Co)$
$F_4 \{ \text{MATR, CDP} \rightarrow \text{Vo} \} // \text{deducibile da } F_1$	$F_5 \{ CDP \rightarrow Co \}$

Chiediamoci ora se  $\text{MATR, Co} \rightarrow \text{Vo, CDP}$  è ancora deducibile da  $F_4 \cup F_5$ . Dato che  $(\text{MATR, Co})^+ = \{\text{MATR, Co}\}$  allora deduciamo che non lo è. Abbiamo perso la dipendenza per sempre.

### 10.5) Fattori positivi e negativi dell'uso della BCNF

La BCNF ha principalmente due problemi:

- Le relazioni ottenute alla fine sono potenzialmente poco naturali
- Potrebbe richiedere l'inclusione di vincoli globali (che sono pesanti)

Vediamolo tramite un esempio.

Consideriamo la relazione conto corrente con dipendenze F:

$CC(\text{Titolare}, \text{N}^{\circ}\text{C}, \text{N}^{\circ}\text{A}, \text{Città_A}, \text{Provincia}, \text{Direttore}, \text{Saldo}, \text{DataM}) // DataM sta per data ultimo movimento$

$$\begin{aligned} F = \{ & \quad \text{Città_A}, \text{N}^{\circ}\text{A} \rightarrow \text{Direttore} \\ & \quad \text{Città_A} \rightarrow \text{Provincia} \\ & \quad \text{N}^{\circ}\text{C} \rightarrow \text{N}^{\circ}\text{A}, \text{Città_A} \\ & \quad \text{N}^{\circ}\text{C} \rightarrow \text{Saldo}, \text{DataM} \\ \} \end{aligned}$$

Non è difficile capire che la chiave è (Titolare, N[°]C).

Come è chiaro non ci troviamo in BCNF. Applichiamo l'algoritmo in maniera sensata (partendo dalle foglie) per evitare la perdita di dipendenze funzionali ed otteniamo le seguenti relazioni:

$R_2(\text{Città_A}, \text{Provincia})$  con  $F_2 = \{ \text{Città_A} \rightarrow \text{Provincia} \}$

$R_3(\text{Città_A}, \text{N}^{\circ}\text{A}, \text{Direttore})$  con  $F_3 = \{ \text{Città_A}, \text{N}^{\circ}\text{A} \rightarrow \text{Direttore} \}$

$R_4(\text{N}^{\circ}\text{C}, \text{N}^{\circ}\text{A}, \text{Città_A})$  con  $F_4 = \{ \text{N}^{\circ}\text{C} \rightarrow \text{N}^{\circ}\text{A}, \text{Città_A} \}$

$R_5(\text{N}^{\circ}\text{C}, \text{Saldo}, \text{DataM})$  con  $F_5 = \{ \text{N}^{\circ}\text{C} \rightarrow \text{Saldo}, \text{DataM} \}$

$R_6(\text{Titolare}, \text{N}^{\circ}\text{C})$  con  $F_6 = \{ \}$  (ecco perché ogni attributo fa parte della chiave)

Non resta che fare alcune osservazioni:

La BCNF risolve l'anomalia di inserzione e minimizza quella di update, poiché la ridondanza è minima.

L'algoritmo di costruzione della BCNF non ha una unica soluzione, dipende dall'ordine di valutazione delle dipendenze

Può essere utile pensare di accoppare più relazioni, come la  $R_4$  e la  $R_5$  avendo una  $R_7(\text{N}^{\circ}\text{C}, \text{N}^{\circ}\text{A}, \text{Città_A}, \text{Saldo}, \text{DataM})$ . Perché dovremmo fare una scelta del genere? Se le operazioni si dovessero orientare su alcuni attributi piuttosto che altri è saggio splittare, ma se le informazioni sono spesso usate insieme allora può essere più comodo unire due relazioni (per esempio  $R_2 + R_3$  per una stampa anagrafica di una agenzia). Questo tipo di operazione è detta **de-normalizzazione**.

ATTENZIONE! Se unissimo  $R_2$  ed  $R_3$  otterremmo  $R_7(\text{Città_A}, \text{N}^{\circ}\text{A}, \text{Direttore}, \text{Provincia})$  con

$$\begin{aligned} F = \{ & \quad \text{Città_A}, \text{N}^{\circ}\text{A} \rightarrow \text{Direttore} \\ & \quad \text{Città_A} \rightarrow \text{Provincia} \\ \} \end{aligned}$$

Si vede subito che tale relazione non è in BCNF poiché "Città_A → Provincia" non rispetta il vincolo per cui l'antecedente deve essere superchiave della relazione!

Inoltre, de-normalizzare aumenta il rischio di anomalia di update (si pensi ad un cambio di provincia).

Non solo, se avessimo una di quelle dipendenze circolari, (in questo caso potremmo pensare Direttore → Città_A), avremmo una  $R_3(\text{Città_A}, \text{N}^{\circ}\text{A}, \text{Direttore})$  con  $F_3 = \{ \text{Città_A}, \text{N}^{\circ}\text{A} \rightarrow \text{Direttore}, \text{Direttore} \rightarrow \text{Città_A} \}$  che non è ancora in BCNF, quindi dovremmo dividerla in:

$R_9(\text{Direttore}, \text{Città_A})$  con  $F_9 = \{ \text{Direttore} \rightarrow \text{Città_A} \}$

$R_8(\text{N}^{\circ}\text{A}, \text{Città_A})$  con  $F_8 = \{ \}$

Pare non ci siano problemi ma invece,  $(F_8 \cup F_9) \not\vdash F_3$  da cui eravamo partiti!  $\text{N}^{\circ}\text{A}, \text{Città_A} \rightarrow \text{Direttore}$  non è più deducibile ed andrebbe integrata come vincolo globale.

Come possiamo ovviare a tutti questi problemi? Usando la **terza forma normale**.

## 11) La terza forma normale

Come abbiamo notato, la BCNF è un po' troppo "restrittiva", quindi diminuiamo leggermente l'affinamento studiando la terza forma normale (III°NF).

### **11.1) Condizioni di appartenenza alla III°NF**

Una relazione  $(R(A), F)$  è in III°NF se per ogni  $X \rightarrow Y \in F$  vale una delle seguenti condizioni:

- 1)  $X \rightarrow Y$  è **banale**, ovvero  $Y \subseteq X$  (si tratta quindi di una forma *riflessiva*)
- 2)  $X$  è **superchiave**
- 3)  $Y$  è un insieme di **attributi primi**

*Un attributo primo è un attributo che è parte di una chiave di  $R(A)$ .*

Quindi, ad esempio, nel caso sopra riportato come "critico" per la BCNF:

$R_3(\text{Città_A, N}^{\circ}\text{A, Direttore})$  con  $F_3 = \{ \text{Città_A, N}^{\circ}\text{A} \rightarrow \text{Direttore}, \text{Direttore} \rightarrow \text{Città_A} \}$  vediamo che la dipendenza funzionale  $\text{Direttore} \rightarrow \text{Città_A}$  è legittima poiché  $\text{Città_A}$  fa parte delle chiavi della relazione (che è  $\text{Città_A, N}^{\circ}\text{A}$ )!

### **11.2) Algoritmo per la sintesi in III°NF**

Anche per questa forma normale abbiamo un algoritmo piuttosto semplice. Data una relazione  $(R(A), F)$ :

- 1) Ricavare una **copertura minimale** dell'insieme delle dipendenze funzionali  $F' = \{ X_1 \rightarrow A_1, X_2 \rightarrow A_2, \dots, X_n \rightarrow A_n \}$
- 2) **Creare** tante relazioni quante sono le dipendenze funzionali, tali che se si ha  $X_n \rightarrow A_n$  si crei la relazione  $R_n(X_n, A_n)$
- 3) Se nessun  $R_i$  contiene almeno una chiave di  $R(A)$  allora si aggiunga una relazione  $R_{n+1}(K)$  dove  $K$  è chiave di  $R(A)$

Come si è visto si introduce il concetto di *copertura minima*. Dobbiamo studiarlo nel dettaglio.

#### Insieme $F'$ di copertura minimale

Dato un insieme  $F$  di dipendenze funzionali, il suo insieme  $F'$  di copertura minimale ha le seguenti caratteristiche:

- 1)  $F' \equiv F$  (i due insiemi sono equivalenti, quindi  $(F')^+ = (F)^+$ )
- 2) Per ogni dipendenza  $X \rightarrow A_i$ , si ha che  $A_i$  è un attributo soltanto (l'insieme  $F'$  è detto in **forma canonica**)
- 3)  $F'$  è privo di **attributi estranei**
- 4)  $F'$  è privo di **dipendenze ridondanti**

Si noti che la forma canonica è necessaria per comodità di applicazione e non per ragioni teoriche.

Definiamo chiaramente:

- **Attributo estraneo:** Dato  $B \in X$  e  $(X \rightarrow Y) \in F$ , se  $F \vdash (X-B) \rightarrow Y$ , possiamo dire che  $B$  è un attributo estraneo. Ovvero: presa una dipendenza che contiene un certo attributo  $B$ , se togliendo  $B$  dalla dipendenza possiamo ancora dedurre la parte destra (partendo da  $F$ ), possiamo dire che  $B$  è attributo estraneo.
- **Dipendenza ridondante:** Data  $(X \rightarrow Y) \in F$ , se  $F - (X \rightarrow Y) \vdash X \rightarrow Y$  allora  $X \rightarrow Y$  è una dipendenza ridondante. Ovvero: presa una dipendenza  $X \rightarrow Y$  da  $F$ , se togliendola possiamo ancora dedurla dall' $F$  rimasto possiamo dire che quella dipendenza è una dipendenza ridondante.

In sostanza la copertura minimale è una "semplificazione" dell'insieme  $F$  di partenza, se ne cancellano infatti le ridondanze (le parti inutili / deducibili da altre). Per esempio, dato l' $F$  usato sopra:

$$\begin{aligned} F = \{ & \text{Città_A, N}^{\circ}\text{A} \rightarrow \text{Direttore} \\ & \text{Città_A} \rightarrow \text{Provincia} \\ & \text{N}^{\circ}\text{C} \rightarrow \text{N}^{\circ}\text{A, Città_A} \\ & \text{N}^{\circ}\text{C} \rightarrow \text{Saldo, DataM} \} \end{aligned}$$

La sua chiusura minimale risulta essere:

$$\begin{aligned} F' = \{ & \text{ Città_A, N\textdegree A } \rightarrow \text{ Direttore} \\ & \text{ Città_A } \rightarrow \text{ Provincia} \\ & \text{ N\textdegree C } \rightarrow \text{ N\textdegree A} \\ & \text{ N\textdegree C } \rightarrow \text{ Città_A} \\ & \text{ N\textdegree C } \rightarrow \text{ Saldo} \\ & \text{ N\textdegree C } \rightarrow \text{ DataM} \end{aligned}$$

}

#### Algoritmo per il calcolo della chiusura minimale ed esempio

Come detto sopra, è necessario avere una chiusura minimale per poter applicare l'algoritmo di III°NF. Vediamo quindi un algoritmo per la chiusura minimale. Per ipotesi immaginiamo di avere una  $F$  in forma canonica.

1)  $F' := F$

2) **Per ogni** dipendenza  $(X \rightarrow A_i) \in F'$

**Per ogni**  $B \in X$ , se  $A_i \in (X - B)^+_{F'}$  **cancella**  $B$  dalla produzione ed aggiorna  $F'$

3) **Per ogni** dipendenza  $(X \rightarrow A_i) \in F'$

**Crea** una  $F^* := (F' - (X \rightarrow A_i))$ , se  $A_i \in (X)^+_{F^*}$  **aggiorna**  $F' := F^*$

In parole povere, il punto 2 risolve gli attributi estranei, poiché calcola la chiusura di una parte sinistra di una dipendenza togliendone un certo attributo "presunto estraneo". Se da quella chiusura riotteniamo la parte destra, allora significa che lo era effettivamente. Questa è in sostanza l'applicazione della definizione, infatti se è vero che  $A_i \in (X - B)^+_{F'}$  allora è vero che  $F' \vdash (X - B) \rightarrow A_i$  che è proprio la definizione di attributo estraneo.

Il punto tre invece risolve le dipendenze ridondanti, prova infatti a creare un nuovo insieme di dipendenze senza quella "presunta ridondante", se da quell'insieme possiamo riavere la dipendenza stessa, quella dipendenza è ridondante (questo è verificato poiché la chiusura della parte sinistra della produzione, basandosi su  $F^*$  che non ha più la produzione incriminata, riottiene la parte destra).

I due punti non sono invertibili ed il costo dell'algoritmo è **polinomiale**.

#### Esempio

Prendiamo questa relazione  $R(ABCDE)$  con

$$\begin{aligned} F = \{ & \text{ ABCD } \rightarrow E \\ & \text{ B } \rightarrow \text{ C} \\ & \text{ BC } \rightarrow \text{ E} \end{aligned}$$

}

Eseguiamo in maniera metodica l'algoritmo per ottenere la  $F'$  minimale.

1) Consideriamo **A** nella produzione  $\text{ABCD} \rightarrow E$ , calcoliamo  $(BCD)^+ = \{ \text{BCDE} \}$ . Abbiamo ritrovato  $E$ , quindi  $A$  è attributo estraneo. Aggiorniamo  $F = \{ \text{BCD} \rightarrow E, \text{B} \rightarrow \text{C}, \text{BC} \rightarrow \text{E} \}$

2) Consideriamo **B** nella produzione  $\text{BCD} \rightarrow E$ , calcoliamo  $(CD)^+ = \{ \text{CD} \}$ . Non abbiamo ritrovato  $E$ , quindi  $B$  non è attributo estraneo.

3) Consideriamo **C** nella produzione  $\text{BCD} \rightarrow E$ , calcoliamo  $(BD)^+ = \{ \text{BDCE} \}$ . Abbiamo ritrovato  $E$ , quindi  $C$  è un attributo estraneo. Aggiorniamo  $F = \{ \text{BD} \rightarrow E, \text{B} \rightarrow \text{C}, \text{BC} \rightarrow \text{E} \}$

4) Consideriamo **D** nella produzione  $\text{BD} \rightarrow E$ , calcoliamo  $(B)^+ = \{ \text{BCE} \}$ . Abbiamo ritrovato  $E$ , quindi  $D$  è un attributo estraneo.  $F = \{ \text{B} \rightarrow \text{E}, \text{B} \rightarrow \text{C}, \text{BC} \rightarrow \text{E} \}$

5) Consideriamo **B** nella produzione  $\text{BC} \rightarrow E$ , calcoliamo  $(C)^+ = \{ \text{C} \}$ . Non abbiamo ritrovato  $E$ , quindi  $B$  non è un attributo estraneo.

6) Consideriamo **C** nella produzione  $\text{BC} \rightarrow E$ , calcoliamo  $(B)^+ = \{ \text{CE} \}$ . Abbiamo ritrovato  $E$ , quindi  $C$  è un attributo estraneo. Quindi, l' $F'$  risulta:  $\{ \text{B} \rightarrow \text{E}, \text{B} \rightarrow \text{C}, \text{B} \rightarrow \text{E} \}$  quindi ovviamente  $F' = \{ \text{B} \rightarrow \text{E}, \text{B} \rightarrow \text{C} \}$ .

Abbiamo quindi ottenuto  $R(\underline{\text{ABCDE}})$  con chiave  $K = \text{ABD}$ .

Si noti che l'algoritmo non produce sempre lo stesso risultato, dipende infatti dall'ordine in cui vengono valutate le produzioni.

Ora che abbiamo introdotto la nozione di copertura minima, possiamo proseguire con l'esempio di prima provando ad applicare il punto 2° dell'algoritmo per la trasformazione in III°F.

Data:

$R(\underline{ABCDE})$  con chiave  $K = (ABD)$

$F = \{ \begin{array}{l} B \rightarrow E \\ B \rightarrow C \end{array} \}$

otteniamo due relazioni,  $R_1(\underline{B}, E)$  ed  $R_2(\underline{B}, C)$ , dato che però la chiave ABD non è totalmente coinvolta in una delle relazioni, dobbiamo aggiungere  $R_3(\underline{A}, \underline{B}, D)$  come ci dice il 3° passo dell'algoritmo.

### 11.3) Caratteristica 1: in $R(X, A)$ X è chiave primaria

Ogni relazione generata è di tipo  $R_j(X_j, A_j)$  con  $F = \{ X_j \rightarrow A_j \}$ . Banalmente possiamo sapere che  $X_j$  è chiave di  $R_j$ , infatti calcolando  $(X_j)^+ = \{ X_j, A_j \}$  cioè tutti gli attributi di  $R_j$  e quindi è chiave.

*Dimostrazione per assurdo se  $X_j$  fosse superchiave*

Se  $X_j$  fosse superchiave (ovvero non minima) allora ci sarebbe una  $K \subseteq X_j$  che è chiave pertanto deve essere vero che  $F' \vdash K \rightarrow A_j$  (altrimenti non sarebbe chiave!). Ma dunque se questo è vero abbiamo che  $X_j \rightarrow K$  (per riflessione, dato che  $K \subseteq X_j$ ) e anche che  $K \rightarrow A_j$ , quindi per transizione  $X_j \rightarrow A_j$ .

Ma se avessimo una situazione del genere avremmo una dipendenza ridondante poiché  $X_j \rightarrow A_j$  ed  $K \rightarrow A_j$  sono ridondanti fra loro! Infatti, se eliminiamo  $X_j \rightarrow A_j$  da F, possiamo comunque dedurlo poiché  $(X_j)^+ = \{ X_j, A_j \}$  (la chiusura ci da tutto poiché  $X_j$  include K e da lì possiamo riottenere  $A_j$ !). Quindi,  $F - (X_j \rightarrow A_j) \vdash X_j \rightarrow A$  e pertanto c'è ridondanza verificata.

Ma non può esistere ridondanza perché siamo partiti da una copertura minima! Quindi, dimostrato.

### 11.4) Caratteristica 2: le dipendenze funzionali sono conservative

Le **dipendenze** funzionali sono banalmente **conservative** per costruzione (per ogni dipendenza viene creata una relazione!)

### 11.5) Caratteristica 3: garanzia del join senza perdita di informazioni (JSP)

Riprendiamo il nostro esempio

- $R_1(\underline{B}, E) \quad F = \{ B \rightarrow E \}$
- $R_2(\underline{B}, C) \quad F = \{ B \rightarrow C \}$
- $R_3(\underline{A}, \underline{B}, D) \quad F = \{ \emptyset \}$

e proviamo empiricamente che c'è JSP.

$R_4 = R_3 \bowtie R_2 = (ABCD)$ . Dal punto di vista delle dipendenze funzionali si ha  $F, R_3 \cap R_4 = B \rightarrow C$  quindi  $R_3 \cap R_4 = R_2$ .

Quindi,  $R_5 = R_4 \bowtie R_1 = (ABCDE)$ . Ecco la nostra relazione iniziale.

Proviamo a dimostrarlo con il calcolo letterale.

A fine algoritmo abbiamo:

- una serie di  $R_j(X_j, A_j)$  dove  $X_j$  è chiave (si è dimostrato nel punto 11.3)
- $F' \vdash K \rightarrow KA_1A_2...A_n$  (questo è intuitivamente vero perché una chiave K deve esserci, piuttosto aggiunta nella  $R_{n+1}$  come detto nel punto 3° dell'algoritmo). Vogliamo dimostrare questo punto, ovvero che  $(K)^+ = \{ K, A_1, A_2, \dots, A_n \}$ . Dimostriamolo per passi.

1° passo)  $(K)^{+_{F'}} = \{ K \}$  (banalmente), ma dato che in un qualche  $R_j$  (eventualmente  $R_{n+1}$ ) abbiamo considerato  $K$  nella sua interezza, abbiamo che  $(K)^{+_{F'}} = \{ K \} = R_{n+1}(K)$  (supponiamo di aver aggiunto la  $R_{n+1}$  perché  $K$  non era stata considerata prima, così come ci dice la terza regola di sintesi in III°NF).

2° passo) Ora che abbiamo incluso  $K$  nella chiusura, ci sarà necessariamente  $K$  nelle  $X_1, \dots, X_n$  (ovvero  $X_k \subset K$ ) da cui dipendono gli  $A_1, \dots, A_n$  nelle relazioni  $R_1, \dots, R_n$ . Allora possiamo sostenere che  $(K)^{+_{F'}} = \{ K, A_1 \}$  (supponendo di considerare che il primo  $X_n$  che troviamo a far parte di  $K$  sia proprio  $X_1$ ) e che quindi abbiamo una relazione parziale che chiameremo  $R_p$ . In formule:  $R_1(X_1, A_1) \bowtie R_{n+1}(K) = R_{p1}(K, A_1)$ .

Vale il join senza perdita di informazioni?

Sì, sappiamo che  $R_1 \cap R_{n+1} = X_1$  (poiché  $X_1 \subset K$ ). Dato che  $X_1 \rightarrow A_1$ , allora  $(X_1)^+ = \{ X_1, A_1 \}$  e questo insieme è equivalente a tutto  $R_1$ . Il teorema JSP è convalidato e quindi il join è senza perdita (per il teorema si veda il paragrafo 7.3).

3° passo) Proseguiamo con la chiusura, diciamo di trovare  $KA_1$  nella  $X_2$  (cioè  $X_2 \subset KA_1$ ) e quindi possiamo dire che  $(K)^{+_{F'}} = \{ K, A_1, A_2 \}$  e la nostra relazione parziale  $R_{p2}$  si ottiene come  $R_2(X_2, A_2) \bowtie R_{p1}(K, A_1) = R_{p2}(K, A_1, A_2)$ .

Vale il join senza perdita di informazioni?

Sì, sappiamo che  $R_1 \cap R_{p1} = X_2$  e  $X_2 \rightarrow A_2$ , quindi  $(X_2)^+ = \{ X_2, A_2 \}$  e dato che questo insieme equivale a tutto  $R_2$  il teorema è convalidato e quindi abbiamo un join senza perdita.

Così facendo si possono ottenere  $n$  passi, dove ogni volta avviene un join senza perdita di informazioni. Così, alla fine  $(K)^{+_{F'}} = \{ K, A_1, A_2, \dots, A_n \}$  e si ha una relazione  $R(K, A_1, A_2, \dots, A_n)$  che è la relazione prima di trasformarla in terza forma normale.

#### 11.6) Caratteristica 4: ogni relazione generata è a sua volta in terza forma normale

Il processo di sintesi garantisce che ogni relazione  $R_i$  generata sia a sua volta in III°FN. Utilizziamo una tecnica dimostrativa combinatoria, ovvero variamo tutti i casi. Prendiamo come relazione di base una  $R_j$  siffatta:

$R_j$	
$X_j$	$A_j$

##### 1° Caso

$R_j$	
...	$W$
...	$A_j$

Abbiamo fra gli attributi  $X_j$  un certo  $W$ , e vale la dipendenza  $A_j \rightarrow W$ .

Questo tipo di dipendenza è **legittima** perché siamo nel caso di un attributo primo, ovvero  $W$  è parte della chiave primaria.

##### 2° Caso

$R_j$	
...	$Y$
...	$A$
...	

Abbiamo fra gli attributi  $X_j$  un certo  $Y$ , e fra gli attributi  $A_j$  un certo  $A$ .

Vale la dipendenza  $Y \rightarrow A$ .

Questo tipo di dipendenza è **impossibile** che si verifichi poiché genera **ridondanza**, infatti per riflessività dato che  $Y \subseteq X_j$  si può dire che  $X_j \rightarrow Y$  e dato che  $Y \rightarrow A$ ,  $X_j \rightarrow A$ . Ma questo è un classico caso di ridondanza tra  $X_j \rightarrow A$  ed  $Y \rightarrow A$ .

##### 3° Caso

$R_j$	
...	$Y$
...	$B$
...	$A_j$

Abbiamo che  $Y$  e  $B$  sono entrambi inclusi in  $X_j$ . Vale la dipendenza  $Y \rightarrow B$ .

Questo caso è **impossibile** che si verifichi infatti  $(X_j - B) \rightarrow Y \rightarrow B$  (per riflessività e poi transitività), poi per espansione con  $X_j$  si ha  $(X_j - B) \rightarrow X_j$  e sapendo che  $X_j \rightarrow A_j$  allora  $(X_j - B) \rightarrow A_j$ . Abbiamo quindi sia che  $(X_j - B) \rightarrow A_j$  e sia che  $(X_j - B) \rightarrow B$  ovvero che  $(X_j - B) \rightarrow BA_j$  ma a questo punto,  $X_j$  sarebbe solo più superchiave (non minima) e non chiave (poiché  $X_j - B$  che è banalmente incluso in  $X_j$  è chiave!).

### 11.7) Caratteristica 5: Lo schema ammette "semplificazioni" che mantengono la terza forma normale

Ultima caratteristica è quella che, volendo, è possibile accorpate gli schemi senza il rischio di perdere la III°FN. Consideriamo ad esempio: E(Matricola, Corso, Voto, DataE, Prof, Qualifica)

$$\begin{aligned} F = \{ & \text{Matr, Corso} \rightarrow \text{Voto} \\ & \text{Matr, Corso} \rightarrow \text{DataE} \\ & \text{Matr, Corso} \rightarrow \text{Prof} \\ & \text{Prof} \rightarrow \text{Qualifica} \\ \} \end{aligned}$$

Applicando la sintesi abbiamo:

- E₁(Matr, Corso, Voto)
- E₂(Matr, Corso, DataE)
- E₃(Matr, Corso, Prof)
- E₄(Prof, Qualifica)

Tutte le relazioni che condividono la stessa chiave possono essere istantaneamente accorpate, ad esempio E₅(Matr, Corso, Voto, DataE, Prof) può sostituire le relazioni E₁, E₂, E₃.

Dimostriamo che questa operazione mantiene la III° NF.

Consideriamo una relazione R(X,A,B,C,D) con F = { X → A, X → B, X → C, X → D } allora otteniamo dopo la sintesi delle relazioni R₁(X,A) R₂(X,B) R₃(X,C) R₄(X,D). Ricomponendo tali relazioni, troveremmo mai delle dipendenze del tipo B → C? (infrangendo così la III°FN?). No, perché avremmo ridondanza, infatti X → B → C cioè X → C. Quindi avremmo che B → C ed X → C che è chiaramente ridondante.

### 12) Lo schema ER: una normalizzazione intrinseca

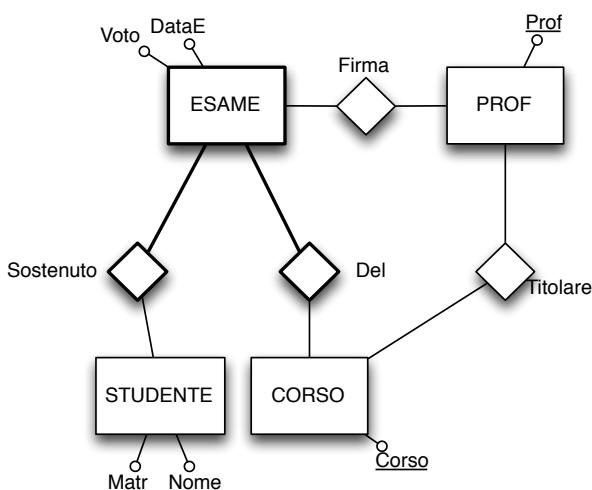
Mentre creiamo lo schema ER non ci siamo mai preoccupati di normalizzare o altro, perché?

Perché la traduzione da schema ER a relazione genera automaticamente relazioni in Boyce Codd Normal Form! I vincoli vengono semplicemente sottintesi durante l'intera lavorazione.

Ma quindi il modello ER è in grado di esprimere ogni sorta di relazione? Se abbiamo detto che la BCNF non garantisce il mantenimento delle dipendenze funzionali in caso di dipendenze cicliche non banali, allora qualcosa non torna!

Il motivo è che il modello ER, in realtà, non è in grado di esprimere ogni situazione, sarà infatti compito del bravo programmatore completare lo schema con vincoli globali, laddove necessari.

*Esempio*



Abbiamo un  

$$F = \{$$

- Matr, Corso → Voto
- Matr, Corso → DataE
- Matr, Corso → Prof
- Prof → Corso**

$$\}$$

Dove vediamo che la dipendenza che non possiamo rappresentare in ER è quella in grassetto: nessuno ci garantisce che il professore che firma un certo esame sia anche il titolare del corso, cosa che invece volevamo dalla dipendenza. Sarà necessario aggiungere un vincolo globale.

---

# **Basi di dati**

## *Moduli del DBMS*

### Capitolo 6

---

*Enrico Mensa*

## Indice degli argomenti

1) <i>L'esecuzione delle transazioni</i>	1
1.1) <b>La memoria stabile</b>	
1.2) <b>La comunicazione fra memoria centrale e periferiche</b>	
1.3) <b>L'architettura di una pagina</b>	
2) <i>Il gestore del buffer</i>	3
2.1) <b>Le operazioni possibili</b>	
3) <i>Organizzazione di una tipica tabella relazionale</i>	4
3.1) <b>Strategia ordinata</b>	
3.2) <b>Strategia heap</b>	
4) <i>I metodi di accesso</i>	6
5) <i>Il metodo di accesso sequenziale</i>	6
5.1) <b>Analisi dei costi</b>	
6) <i>Il metodo di accesso tabellare</i>	6
6.1) <b>Introduzione agli indici</b>	
6.2) <b>Classificazione riassuntiva degli indici</b>	
6.3) <b>I B - Tree</b>	
6.4) <b>L'aspetto computazionale dei B - Tree</b>	
6.5) <b>I B+ Tree: dal B - Tree agli indici</b>	
6.6) <b>Uso del B+ Tree</b>	
6.7) <b>Il costo di split potrebbe influenzare le prestazioni del B - Tree?</b>	
6.8) <b>Il costo di accesso</b>	
Calcolo del CI	
Calcolo del CD	
6.9) <b>Criteri di scelta di un buona chiave di ricerca</b>	
7) <i>Il metodo di accesso diretto (cenni)</i>	14
7.1) <b>Bucket e funzioni di hash</b>	

**7.2) Il problema della saturazione dei bucket**

**7.3) Analisi dei costi**

8) Considerazioni finali sui metodi d'accesso ed euristica 15

9) Algoritmi di join 17

**9.1) Prima categoria: Nested Loop**

**9.2) Prima categoria: Block Loop**

**9.3) Seconda categoria: Hash - Join**

**9.4) Terza categoria: Sort - Merge**

### 1) L'esecuzione delle transazioni

Ricordiamo le proprietà di una transazione (Capitolo 5, paragrafo 1.2):

Una transazione deve rispettare le proprietà base, **ACID**.

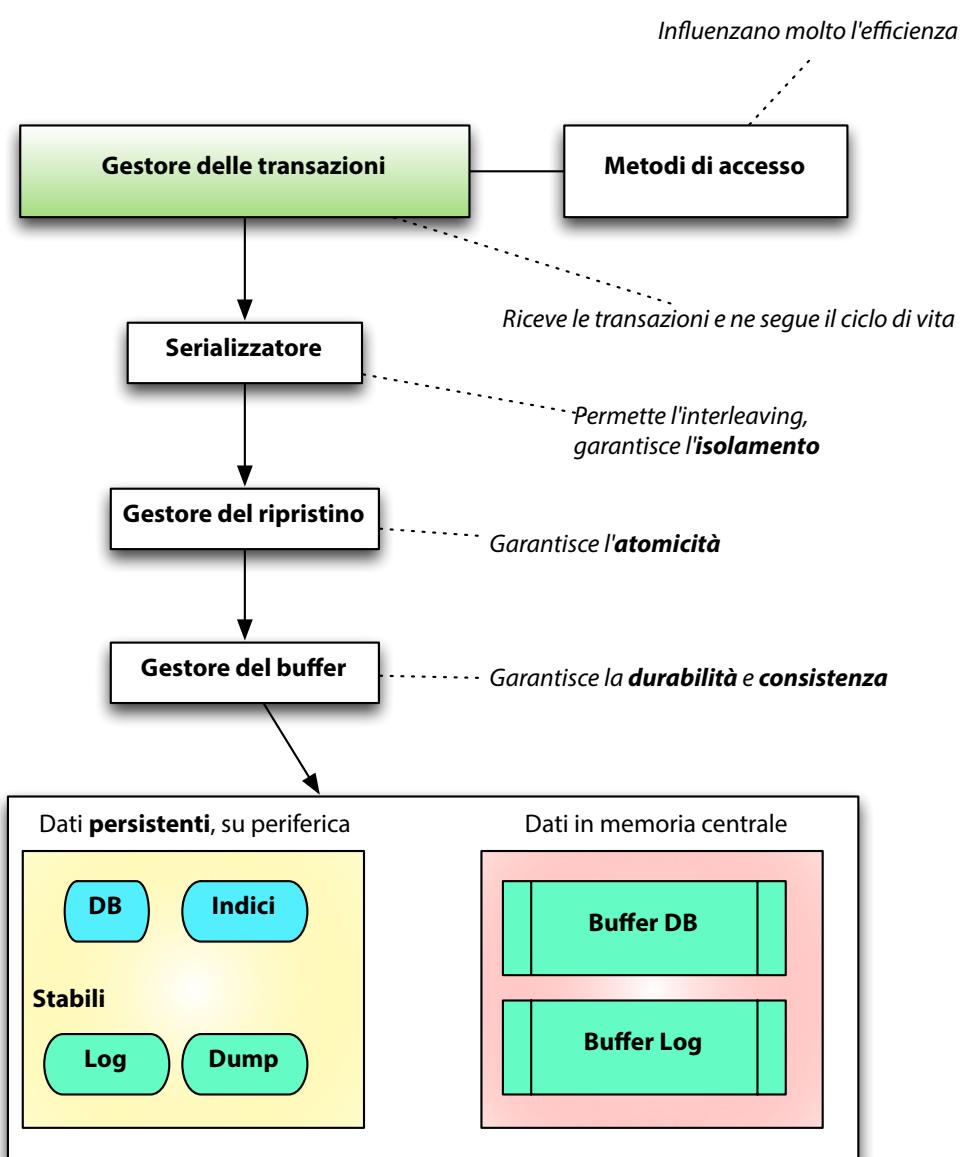
A = Atomica, (una transazione deve essere eseguita nella sua interezza, altrimenti non ha senso)

C = Consistente, (una transazione deve rispettare in ogni sua azione i vari vincoli)

I = Isolata (una transazione viene considerata come 'eseguita da sola' senza doversi occupare delle concorrenze)

D = Durabile (gli effetti di una transazione devono poter essere persistenti)

Qual'è il sistema che ci permette di far valere tutte queste proprietà? Vediamolo sommariamente ed entriamo successivamente nel dettaglio.



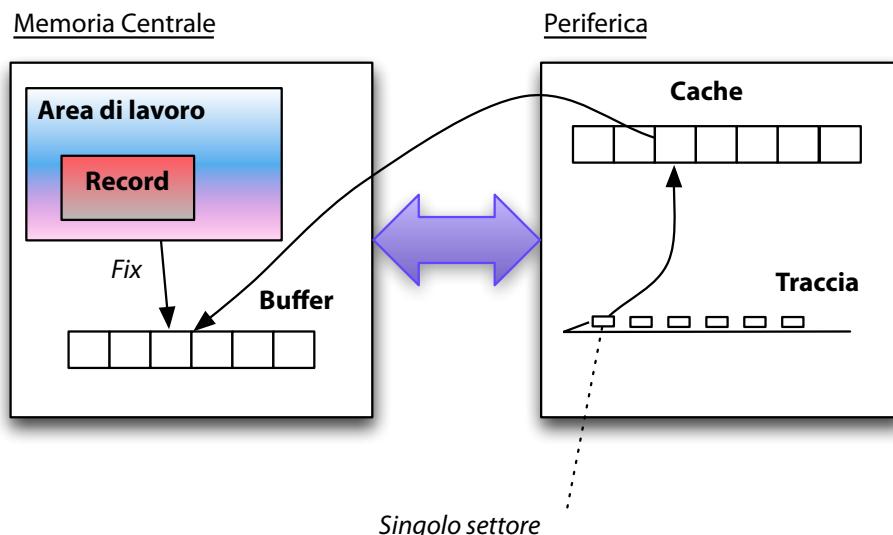
Il **log** contiene le ultime operazioni effettuate, il **dump** invece contiene una copia aggiornata ogni tempo T dell'intero DB.

### 1.1) La memoria stabile

La memoria stabile è ciò che più idealmente si avvicina ad una memoria "indistruttibile". C'è quindi una forte incongruenza con la realtà, poiché sappiamo bene che nessun dato è eterno. Ma i sistemi implementati dal Gestore del buffer garantiscono la durabilità. Il concetto è quello di poter ricostruire i dati in caso di errori/problemi/guasti.

### 1.2) La comunicazione fra memoria centrale e periferiche

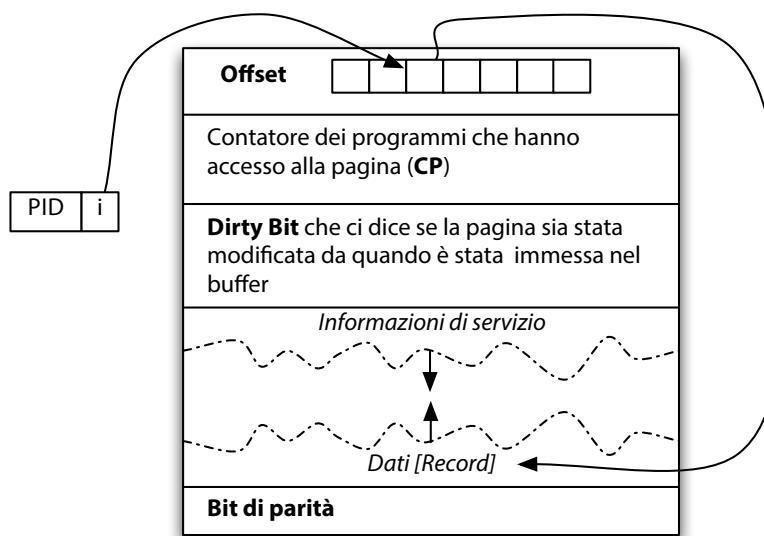
Possiamo pensare alla memoria centrale come una DRAM e una periferica come un Hard Disk.



Abbiamo che un singolo settore contenente il record viene copiato in cache dalla periferica, quindi messo nel buffer della memoria centrale attendendo il FIX da parte dell'area di lavoro che prende in carico il record e prosegue con le sue azioni. "L'unità di misura" con cui vengono scambiati i settori sono le pagine, ovvero è applicato un sistema di **paging**. Ogni pagina ha un suo PID identificativo (Page ID). Solitamente la dimensione di una pagina è proporzionale alla dimensione di un settore. Saranno i metodi di accesso a far sì che venga scelta la pagina corretta in relazione ad una certa richiesta.

### 1.3) L'architettura di una pagina

Giusto per avere un'idea di cosa stiamo parlando, guardiamo il dettaglio di una pagina.



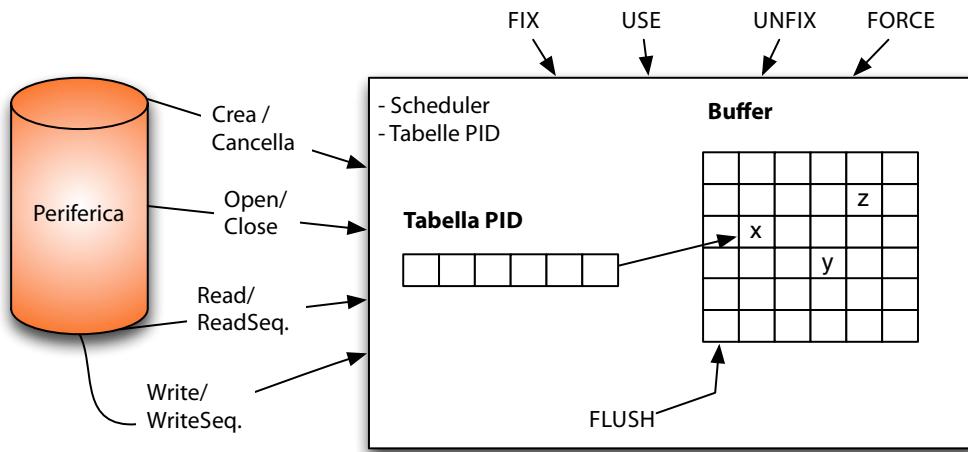
Una pagina mantiene i vari record in un array di offset. Così, al momento della richiesta, è sufficiente fornire il proprio PID ed un i che identifica la posizione dell'array nel quale si trova l'indirizzo del settore desiderato.

Tramite indirizzamento indiretto abbiamo il nostro record.

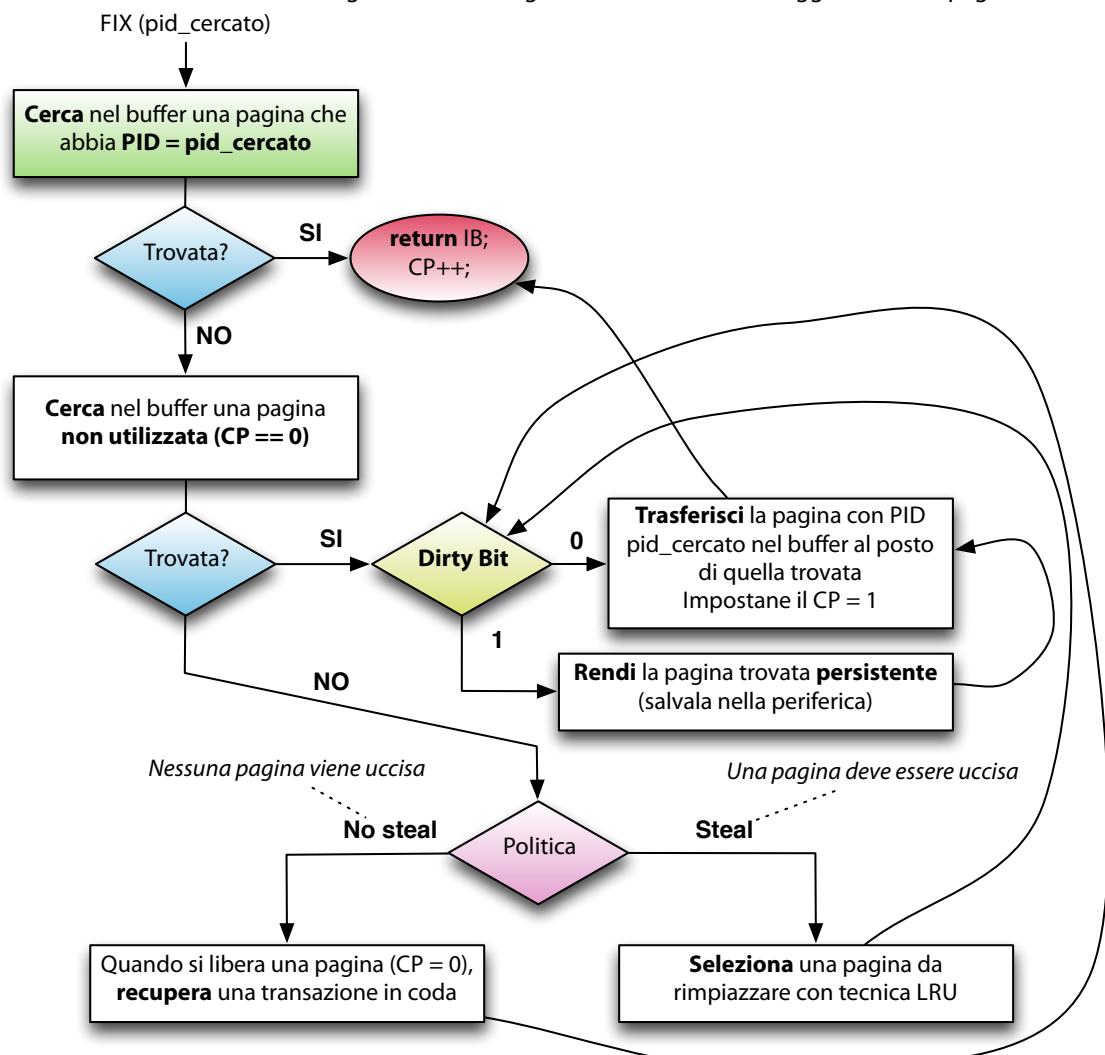
Abbiamo poi informazioni aggiuntive come il CP ed il dirty bit.

## 2) Il gestore del buffer

Come funziona il gestore del buffer? Vediamone una rappresentazione.



Le frecce entranti sono le operazioni ricevute dal gestore del buffer, il quale tiene un collegamento tra le pagine e i settori. Data un'accoppiata (PID, i) vogliamo ottenere l'indirizzo nella tabella (o matrice) buffer che ci darà l'effettivo indirizzo del settore. Vediamo ora l'algoritmo che svolge una **FIX** cioè ottiene legge una certa pagina dal buffer.



Come si vede dall'algoritmo, viene ritornato un IB cioè un Indirizzo Buffer dove è contenuta la pagina che si vuole FIXare.

### 2.1) Le operazioni possibili

Le altre operazioni sono:

- **UNFIX** semplicemente esegue CP--,
- **USE** va a pescare il record desiderato nella pagina,
- **FORCE** la transazione costringe la ri-copia su memoria periferica di una certa pagina,
- **FLUSH** come la force, ma è eseguita da parte del gestore del buffer.

### 3) Organizzazione di una tipica tabella relazionale

Una singola relazione può essere rappresentata in memoria con differenti strategie. In ogni caso ogni record è atomico e viene inserito in una pagina, avendo quindi una struttura del genere:

*Ogni riga è una singola tupla* **Pagina 1**

A1	...	...

### 3.1) Strategia ordinata

Potremmo pensare di inserire le tuple in maniera ordinata, e quindi avere qualcosa del genere (si supponga di trattare la relazione Esame(Matricola, ...):

**Pagina 1**

300	...	...
305	...	...
450	...	...
500	...	...

Chiaramente è necessario decidere un attributo su cui basare l'ordinamento, tale attributo è detto **chiave di ricerca** (che non ha nulla a che vedere con le chiavi relazionali!).

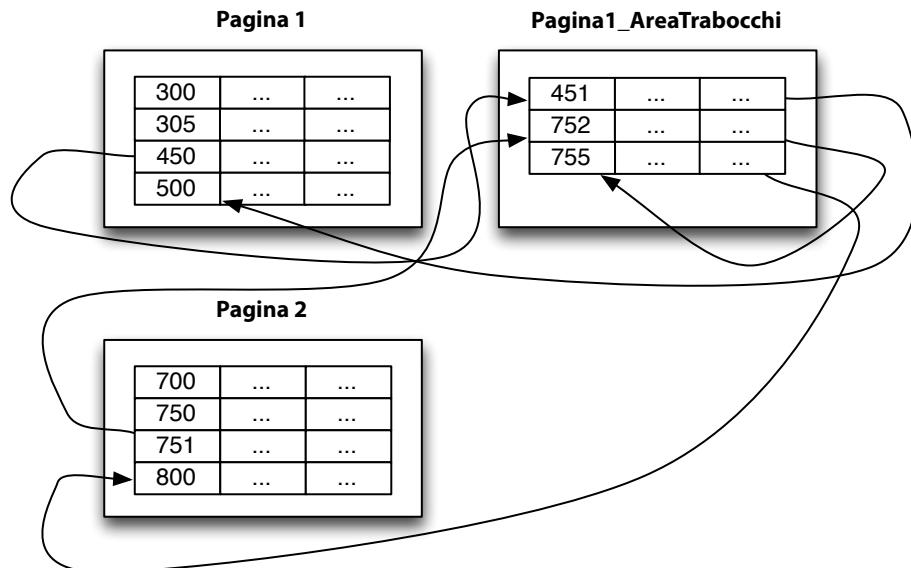
**Pagina 2**

700	...	...
750	...	...
751	...	...
800	...	...

Questo tipo di impostazione però è scomoda nel caso di inserimento di nuovi record. Se volessimo inserire la matricola 451, ad esempio, non avendo lo spazio ci troveremmo a dover traslare tutti i record, operazione per nulla comoda.

Per risolvere questa situazione si è introdotta l'**area trabocchi** cioè un insieme pagine dove inserire i record che non possono essere inseriti in maniera sequenziale per motivi di spazio.

Vi sono due tipologie di area trabocchi, l'**area trabocchi statica** e l'**area trabocchi dinamica**.

*Area trabocchi statica*

In sostanza si inseriscono dei puntatori per mantenere "sensato" il susseguirsi delle pagine nell'ordinamento.

Nella versione statica è possibile inserire in una pagina trabocchi record provenienti da ogni altra pagina.

*Area trabocchi dinamica*

L'area trabocchi dinamica è del tutto simile a quella statica, ma ogni pagina ha una sua area trabocchi riservata. Il costo computazionale è migliore ma c'è più segmentazione interna.

Entrambe le implementazioni dell'area trabocchi sono costose poiché percorrere tutti i puntatori è poco efficiente.

**3.2) Strategia heap**

La strategia heap, molto più usata, è innanzi tutto **clusterizzata** ovvero si tende ad inserire in posizioni ravvicinate le tuple che hanno stessa chiave. La strategia ordinata, quindi, non è clusterizzata.

Un esempio di heap (relazioni STUD(MATR, Nome, Indirizzo, DataN) e Esami(MATR, Corso, Voto, DataE) è:

Pagina 1		
300	Piero	...
300	DB	27
...	...	...
450	Anna	...
450	LFT	...

Nonostante più tabelle siano meschiate in una pagina sola, le tuple con MATR simile sono ravvicinate e questo chiaramente aiuta l'operazione di join.

La complessità di organizzazione è chiaramente maggiore.

#### 4) I metodi di accesso

Vi sono diverse metodologie di accesso:

- Sequenziale (implementa le tavole ad heap)
- Tabellare - *Utilizzo di indici*
- Diretto (hash) - *Utilizzo di indici*

Per poter comparare i vari metodi dobbiamo avere il concetto di costo.

$$\text{Costo di accesso: } \alpha * N_{\text{pag}} + \beta * N_{\text{tuple}} + \gamma * N_{\text{messaggi}}$$

Ovvero il costo di ricerca di una pagina*numero di pagine trasferite durante il processo(ordine dei msec), il costo relativo alla quantità di tuple elaborate ed il costo per la comunicazione con l'I/O (le ultime due nell'ordine dei nsec). Non avendo noi però informazioni sui secondi due parametri, considereremo il costo come  $\alpha * N_{\text{pag}}$ .

Inoltre, ogni metodo andrebbe analizzato per le operazioni di inserzione, update, ricerca ed eliminazione ma per semplicità calcoleremo solo il costo per l'operazione di ricerca.

#### 5) Il metodo di accesso sequenziale

Il metodo sequenziale percorre linearmente le pagine fino a trovare il record desiderato.

##### **5.1) Analisi dei costi**

Dato n il numero di pagine:

- * Costo di ricerca con successo:  $(n+1)/2$
- * Costo di ricerca con insuccesso: n

Come l'abbiamo calcolato? Non avendo informazioni storiche, abbiamo ipotizzato una distribuzione uniforme ovvero che ogni pagina abbia la stessa probabilità di contenere il record. Per questo, date n pagine, mediamente se ne trasferiscono  $n+1/2$  per trovarne quella con il record. Chiaramente nel caso di ricerca con insuccesso si ha un costo n poiché si devono guardare tutte le pagine.

Se ordinassimo con chiave di ricerca, allora i costi sarebbero:

- * Costo di ricerca con successo:  $n+1/2$
  - * Costo di ricerca con insuccesso:  $n+1/2$
- poiché una volta "scavalcato" il valore di MATR cercato, potremmo non proseguire (sappiamo che non esiste).

#### 6) Il metodo di accesso tabellare

Decisamente più interessante è il metodo tabellare, che sfrutta gli indici. Impiliamo le nostre tuple come se si avesse una tabella, dopodiché affianchiamo un file (anch'esso diviso in pagine, ma logicamente lo rappresentiamo tutto unito) dove vi sono i valori distinti della chiave di ricerca ma questa volta ordinati. Questo file, detto **indice**, contiene in ogni riga un valore di chiave ed il suo RID corrispondente (ovvero l'indirizzo dove si trova il record associato alla chiave di ricerca che ha quel determinato valore nella tupla).

##### **6.1) Introduzione agli indici**

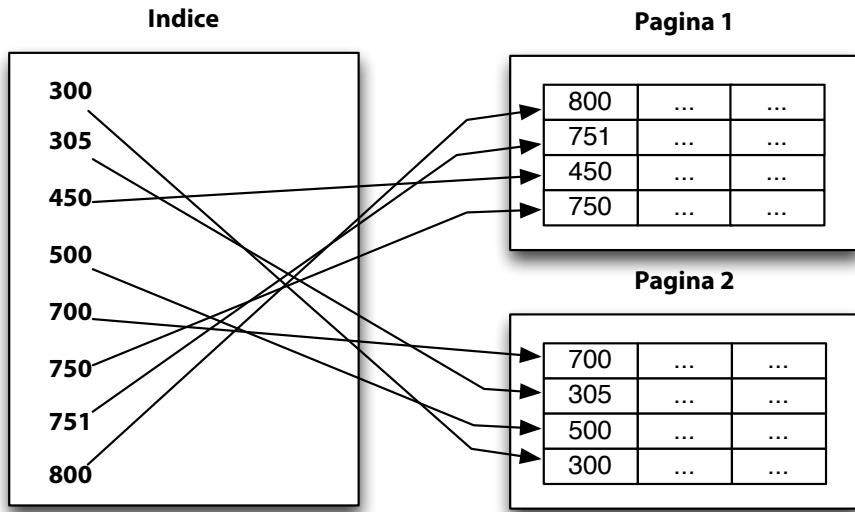
Vi sono due tipologie di indici:

- Indice **denso** (il file indice contiene tutti i valori distinti della chiave di ricerca)
- Indice **sparso** (il file indice non contiene tutti i valori distinti della chiave di ricerca)

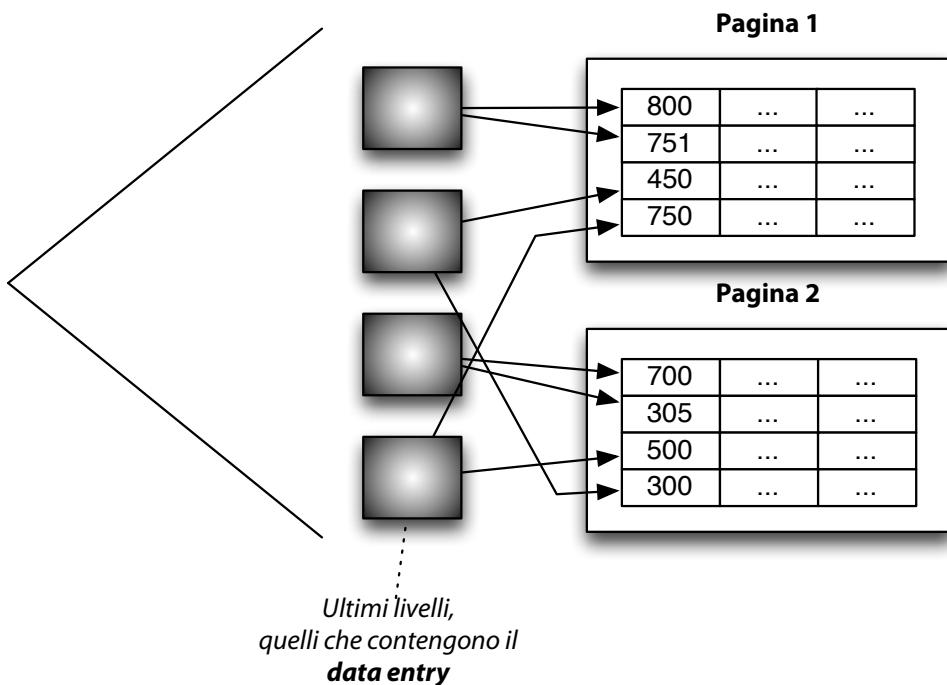
Come abbiamo detto, l'indice è un file, e quindi a sua volta è suddiviso in pagine. Se l'indice dovesse essere molto ampio, allora avremmo un **indice multilivello** (ed in via generale considereremo sempre questo tipo di indici) ovvero un indice di un indice.

Diamone una rappresentazione grafica.

Indice a unico livello, la freccia rappresenta il RID (puntatore al record corrispondente alla chiave)



Indice multilivello



Come vediamo, grazie agli indici multilivello possiamo introdurre il concetto di **data entry** ovvero la foglia dell'albero di livelli di indici, cioè il vero e proprio collegamento coi record. La data entry è definibile come l'informazione minima per reperire il record.

Definendo una data entry  $K^*$  nell'indice, abbiamo che può essere:

- 1)  $K^* = t[k]$  (la data entry è direttamente la tupla - si veda in seguito)
- 2)  $K^* = (K, RID)$  (una coppia chiave di ricerca, indirizzo del RID - come nelle foglie dell'albero degli indici)
- 3)  $K^* = (K, Lista_RID)$  (una coppia chiave di ricerca, lista di RID poiché ad una chiave corrispondono più tuple)

Questo ultimo caso ci fa capire che è possibile anche avere più record collegati ad una stessa chiave di ricerca, in tal caso si utilizza una lista di RID invece che il singolo puntatore al record.

È inoltre possibile che vi siano **più chiavi di ricerca**, in tal caso vi sarà un altro file indice che farà riferimento alla memoria.

È oltremodo possibile avere indici **clusterizzati** e non clusterizzati.

Un indice è clusterizzato quando i record nella memoria rispettano l'indice, ovvero sono ordinati anche loro. Il fatto che l'indice sia clusterizzato fa sì che sia possibile anche renderlo **sparso**, cioè si può dire che dato un RID abbiamo tutto un "pacchetto" di record consecutivi in memoria, quindi si può sfoltire l'indice ed includere solo i primi record di ogni pagina in memoria. Ognuna di quelle data entry si collegherà quindi ad un primo record di una pagina.

Quando si ha un indice clusterizzato allora è possibile introdurre l'intero sistema di indicizzazione nelle pagine stesse, ovvero il file e la memoria dove sono contenuti i record diventano un tutt'uno. Questo caso è quello in cui  $K^* = t[k]$  cioè la data entry è esattamente la tupla.

## 6.2) Classificazione riassuntiva degli indici

Riassumiamo quindi le varie nozioni.

Un **indice** è un file che contiene i valori di un attributo detto **chiave di ricerca**. Tali valori sono ordinati. Un indice può essere:

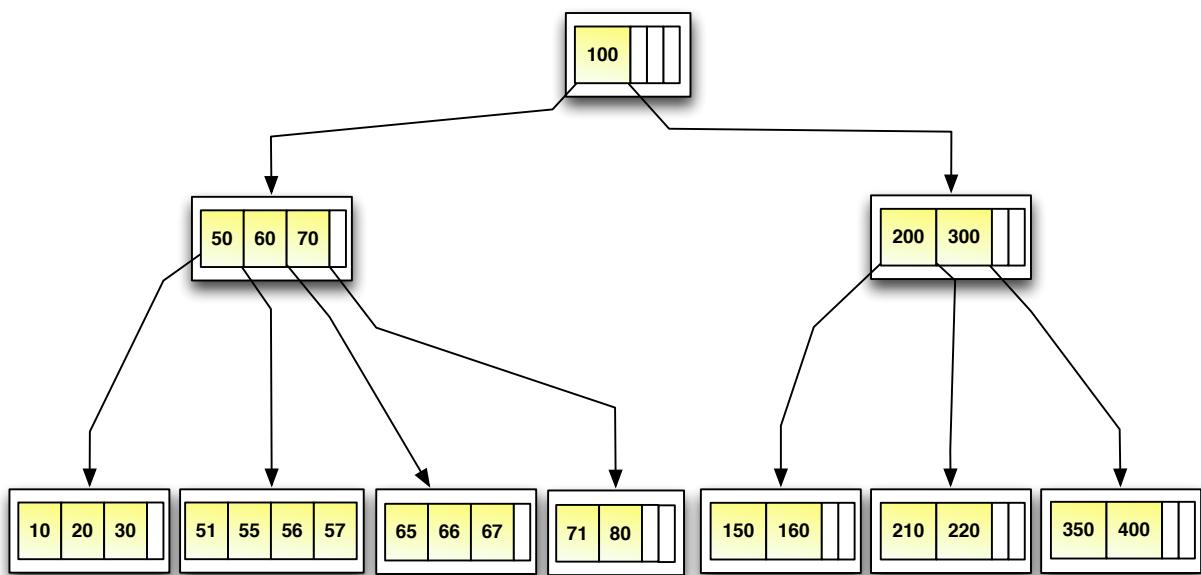
- **Denso**: un indice è denso quando contiene tutti i valori della chiave di ricerca
- **Sparso**: un indice è sparso quando non contiene tutti i valori della chiave di ricerca (presente quando l'indice è anche clusterizzato)
- **Clusterizzato**: l'ordine dell'indice è rispettato anche nella memorizzazione dei record stessi

Un indice definito su una chiave relazionale principale si chiama Indice principale, gli altri sono detti Indici secondari.

## 6.3) I B - Tree

Prima di proseguire dobbiamo introdurre il concetto di B - Tree, sul quale baseremo poi gli indici.

Questo è un B - Tree.



È un albero ordinato (ogni nodo ha nell'albero di sinistra gli indici più piccoli e a destra gli indici più grandi), dove ogni nodo ha un certo numero di chiavi (che non possono superare un certo massimo). Dobbiamo definire alcune regole secondo cui un albero è B - Tree. Vediamole.

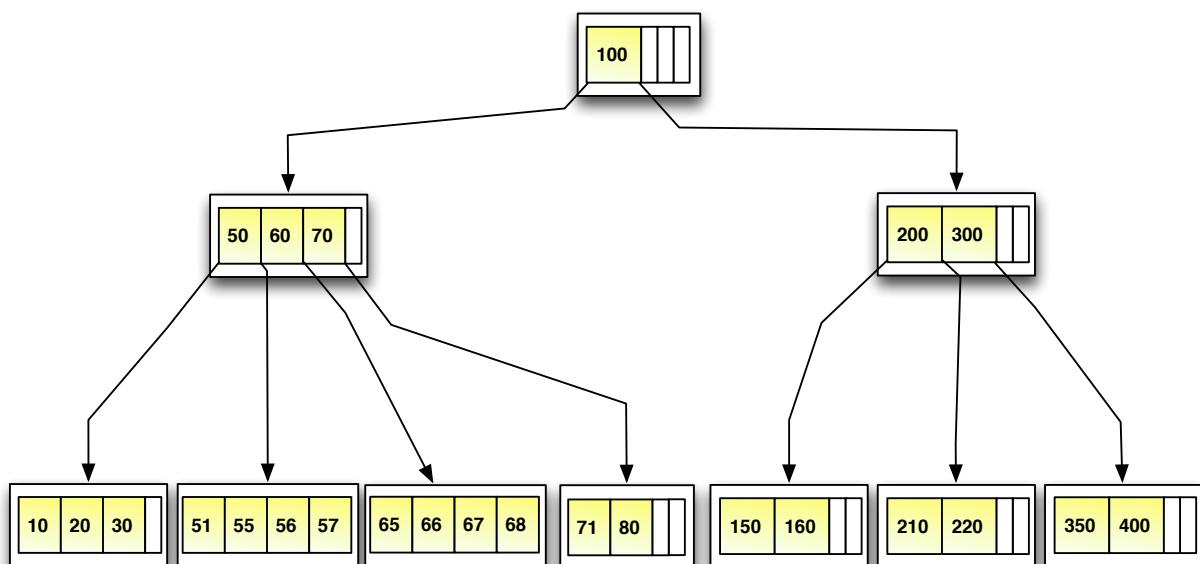
- **m**: il massimo numero di figli di un nodo
- **m - 1**: massimo numero di chiavi di un nodo
- **[m/2] - 1**: minimo numero di chiavi in un nodo interno ( $\lfloor \cdot \rfloor$  è l'operatore parte intera)
  - $1 \leq j \leq m - 1$  con  $j$  il numero di chiavi della radice
  - $\lfloor m/2 \rfloor - 1 \leq j \leq m - 1$  con  $j$  il numero di chiavi del nodo
  - Le foglie devono essere tutte allo stesso livello
  - Ogni figlio deve avere  $j+1$  nodi se il padre ha  $j$  nodi.

Nell'esempio sopra disegnato, abbiamo che  $m = 5$  (ogni nodo può avere al massimo 5 figli) e quindi che si possono avere al massimo 4 chiavi ogni nodo ( $m - 1$ ) e che il numero minimo di chiavi di ogni nodo è 2 ( $\lfloor m/2 \rfloor - 1$ ).

Una volta chiarito cosa sia un B - Tree possiamo chiederci in cosa consista l'inserimento. Vi sono due casistiche:

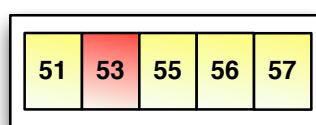
#### 1 - Inserimento con spazio vuoto:

Banalmente si percorre l'albero fino a trovare la posizione corretta della chiave mantenendo l'ordine, una volta trovata la posizione si inserisce. Se volessimo inserire **68** avremmo questa tipologia di inserimento.



#### 2 - Inserimento senza spazio vuoto:

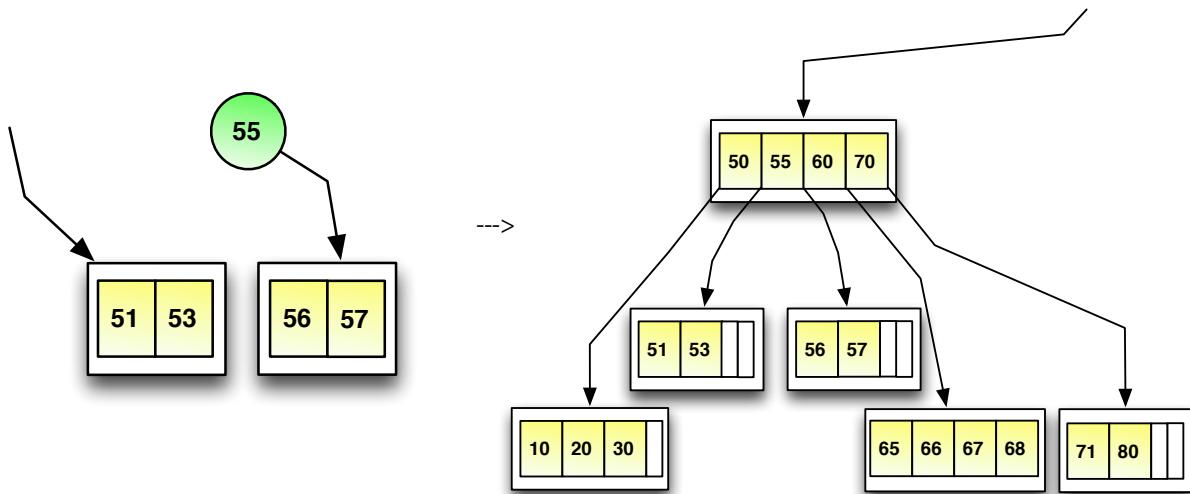
Se naturalmente non c'è spazio per la chiave, come accadrebbe volendo aggiungere la chiave **53**, ci troveremmo in una situazione del genere:



Essendo il numero di chiavi contenute maggiore a  $m-1$ , è necessario effettuare una operazione di **split** che consta di tre passi:

- identificare una chiave detta **separatrice** che è "al centro" dei valori (ed è indipendente da quella che vogliamo aggiungere), in questo caso 55.
- **estrarre** la chiave separatrice ed accorparla nel padre del nodo in questione
- **collegare** in maniera appropriata i figli rimasti (la parte destra segue la chiave separatrice, la parte sinistra segue la parte precedente alla chiave separatrice nel nodo padre).

Il risultato sull'albero di prima (zoom sul figlio sinistro) è quindi questo:



L'albero può crescere di livello perché, se la radice dovesse essere "piena", verrebbe spartita in due con selezionando come nuova radice la chiave separatrice.

#### 6.4) L'aspetto computazionale dei B - Tree

Consideriamo ora l'aspetto computazionale del B - Tree. Usiamo una tabella per calcolare iterativamente ogni passo: Si vogliano inserire N chiavi, e si indichi con  $h = m - 1$  (per semplicità). Vogliamo riempire ogni livello al massimo, pertanto per ogni livello  $L$  abbiamo :

$L$	N° nodi	N° chiavi
1	1	$h$ (il massimo di chiavi inseribili)
2	$h + 1$ (ogni $h$ sopra ne crea 1)	$(h+1) \cdot h$ (i nodi attuali per il massimo di chiavi inseribili)
3	$(h + 1)^2$ (ognuno degli $h+1$ ne fa altri $h+1$ )	$(h+1)^2 \cdot h$ (i nodi attuali per il massimo di chiavi inseribili)
...	...	...
$L$	$(h + 1)^{L-1}$	$(h + 1)^{L-1} \cdot h$

Quindi, sommando sui termini "N° chiavi" abbiamo  $h \cdot \sum_{i=0}^{L-1} (h+1)^i \geq N$  (la diseguaglianza c'è perché il

numero di chiavi contenibili dall'albero deve essere al minimo  $N$  (quelle che volevamo inserire noi) ma volendo

anche di più. Quindi, risolvendo la formula: e quindi  $h \cdot \frac{(h+1)^L - 1}{h+1 - 1} \geq N$  ovvero  $(m-1+1)^L \geq N$  e quindi

andando a risolvere con i logaritmi:  $L \geq \log_m(N+1)$

Questo numero indica il numero di livelli per avere nel caso migliore (cioè inserendo al meglio le chiavi cioè riempendo del tutto ogni nodo)  $N$  chiavi.

Possiamo provare a risolvere lo stesso problema ma questa volta vogliamo inserire il minimo numero possibile di chiavi dentro ai nodi, ovvero usare "al peggio" la struttura dati. Chiamiamo sempre  $N$  la quantità di chiavi che vogliamo inserire e  $k = \lceil m/2 \rceil - 1$  per semplicità di scrittura.

L	N° nodi	N° chiavi
1	1	1 (il minimo di chiavi inseribili)
2	2 (il precedente più la chiave creata prima)	2 · k (i nodi attuali per il minimo di chiavi inseribili)
3	2 · (k+1) (ognuno degli 2 ne fa altri k+1)	2 · (k+1) · k (i nodi attuali per il massimo di chiavi inseribili)
...	...	...
L	$(k + 1)^{L-2}$	$(k + 1)^{L-2} \cdot k$

Da questo, risolvendo in maniera del tutto simile a prima, otteniamo che  $L \leq \log_{\frac{m}{2}}(\frac{N+1}{2})$

e quindi abbiamo, unendo i due risultati:  $\log_{\frac{m}{2}}(\frac{N+1}{2}) \geq L \geq \log_m(N+1)$  che è semplificabile intuitivamente

in:  $\log_m(N) \geq L \geq \log_m(N)+1$

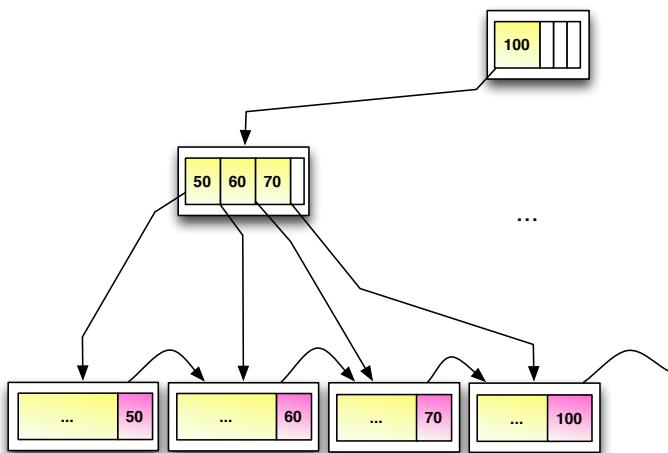
Questo significa che in tempo **logaritmico** possiamo percorrere l'albero cioè avere un numero di livelli non elevati. Proviamo a considerare un caso reale, con  $m$  circa 100 (in 8KB ci stanno comodamente 100 record). Allora, abbiamo che, con tre livelli,  $100^3 = 1.000.000$  cioè con un indice a tre livelli possiamo tenere **un milione di chiavi!** Questo è importante poiché con tre soli accessi arriviamo al record desiderato.

### 6.5) I B+ Tree: dal B - Tree agli indici

Per ora abbiamo parlato solo intuitivamente dell'applicazione dei B - Tree alle chiavi. Dobbiamo trasformare il B - Tree in un B+ Tree, con 3 passaggi:

- 1) La chiave separatrice di un nodo che come figli ha delle foglie, finisce anche nelle foglie stesse
- 2) Ogni foglia è linkata con la successiva
- 3) Ad ogni chiave nelle foglie viene appesa la lista dei RID.

Il nostro albero diventa quindi:



Come si vede, nelle foglie abbiamo **tutte** le chiavi con i RID associati, che è proprio quello che volevamo.

Si noti che l'inserimento con split non è problematico, ma è sufficiente inserire un if che controlli se ci troviamo in una foglia: in tal caso invece di eliminare la chiave separatrice ne lasciamo una copia nella foglia lasciata a sinistra, così rimane direttamente al suo posto. Nel caso non sia una foglia, si procede con l'algoritmo normale.

### 6.6) Uso del B+ Tree

In caso di semplici selezioni basati su una chiave, si percorre l'albero in tempo logaritmo e poi si accede al RID. Quando invece facciamo una selezione con range siamo avvantaggiati dai link foglia/foglia che abbiamo messo nella fase di trasformazione in B+ Tree. Ci basterà scorrere fino al valore desiderato per avere tutti i record che vogliamo.

### 6.7) Il costo di split potrebbe influenzare le prestazioni del B - Tree?

La risposta è no.

Infatti, se abbiamo  $N_0$  nodi, allora potremmo pensare che  $n_{\text{split}} = N_0 - 1$  (cioè i nodi li abbiamo ottenuti per split successivi, in effetti è l'unico modo). Ma dobbiamo anche considerare le radici che quando si spartono producono due nodi! Quindi  $n_{\text{split}} \leq N_0 - 1$ .

Prendiamo ora il conteggio dei nodi in un albero "poco pieno": si ha  $1 + ([m/2] - 1) * (N_0 - 1) \leq N$

Quindi, uguaglio le due formule su  $N_0 - 1$  ed ho  $n_{\text{split}} \leq (N_0 - 1) \leq (N - 1) / ([m/2] - 1)$  e ottengo quindi il rapporto  $(n_{\text{split}}/N) \leq (1 / (m/2 - 1))$ . Questo ci dice quanto vale il numero medio di split.

In un caso reale, abbiamo che avviene circa uno split ogni 50 chiavi.

### 6.8) Il costo di accesso

Proviamo ora a quantificare tutte queste ottimizzazioni. Dobbiamo eseguire un semplice conto, calcolare il costo di accesso: in breve  $C_A$ .

Il  $C_A = C_I + C_D$  ovvero il costo di accesso è uguale al costo di accesso all'indice più il costo di accesso ai dati.

*Nel catalogo sono contenute varie informazioni a riguardo delle relazioni, ecc. Fra queste abbiamo: CARD(T) (numero di tuple di una certa tabella T), Npag (numero di pagine occupate da una certa tabella), VAL(A, T) (quantità di valori distinti in un certo attributo A, di una tabella T), Nleaf (numero di pagine foglie di un certo indice).*

#### Calcolo del $C_I$

Il costo di accesso ad un indice si può calcolare facilmente, ma dobbiamo ricordarci che abbiamo due tipi di ricerche, quelle puntuali e quelle per range. Considereremo il caso per range, che è più generale e comprende comunque la ricerca puntuale.

In una ricerca con range ( $\sigma_{V1 \leq A \leq V2}$ ), abbiamo che  $f_s(P) = V_2 - V_1 / \text{MAX}(A) - \text{MIN}(A)$ . Si ricorda che  $f_s(P)$  è il fattore di selettività di P ovvero la probabilità che il predicato P (in questo caso  $V_1 \leq A \leq V_2$ ) diventi vero.

Pertanto, la stima dei valori distinti che soddisfano la richiesta è:  $f_s(P) \cdot \text{VAL}(A, T)$  cioè la probabilità che il predicato sia vero per la totalità dei dati distinti possibili (ognuno ha la probabilità  $f_s(P)$  di essere selezionato).

Ricordiamo che il concetto di costo non è altro che la movimentazione delle pagine. Ma allora, quante foglie ho dovuto leggere per raggiungere tutti i valori richiesti? (come sappiamo in una ricerca con range una volta arrivati alla foglia iniziale bisogna percorrere le successive fino a fine range).

Stima di foglie che devo movimentare =  $[f_s(P) \cdot \text{VAL}(A, T) / \text{n}^{\circ} \text{ medio di valori distinti per foglia}]$ . Questa formula è banale, abbiamo infatti calcolato "indici che devo percorrere" / "indici per ogni foglia" = "numero di foglie che devo percorrere". Il fattore che abbiamo a denominatore è però  $\text{VAL}(A, T) / \text{Nleaf}$ ! (il totale dei valori distinti / foglie cioè come ho distribuito tutti i valori che avevo sulle foglie, ovvero quanti valori ogni foglia, che è proprio il dato che mi serviva.)

Da ciò abbiamo:  $f_s(P) \cdot \text{VAL}(A, T) / (\text{VAL}(A, T) / \text{Nleaf}) = f_s(P) \cdot \text{Nleaf}$ .

Quindi,  $C_I = f_s(P) \cdot \text{Nleaf}$ .

Calcolo del  $C_D$ 

Ora consideriamo il costo di accesso ai dati.

Banalmente, potremmo dire  $C_D = f_s(P) \cdot \text{CARD}(T)$  ovvero la probabilità di avere un successo (avere proprio il dato che mi serviva) per il totale delle tuple poiché ogni tupla ha probabilità  $f_s(P)$  di essere quella giusta.

Ma questo sarebbe una sovrastima poiché se cerchiamo di seguito due dati che si trovano nella stessa pagina, la pagina stessa verrà movimentata solo la prima volta e non la seconda (poiché sarà già ovviamente nel buffer). Usiamo quindi la **formula di Cardenas**. Chiamiamo il  $C_D$  sovrastimato di sopra come  $E_{ris}$  cioè la media delle tuple risposta. Cardenas ci dice  $\Phi(E_{ris}, N_{pag})$ . Dimostriamolo, ma per comodità chiameremo  $E_{ris}$  (media delle tuple che devo estrarre) con la lettera 'k', e il  $N_{pag}$  occupate dalla relazione con la lettera 'n'. Perciò  $\Phi(k, n)$ . Vediamo il ragionamento.

- Dato un RID, **1/n** è la probabilità che quel RID si trovi in una data pagina. (casi favorevoli / casi possibili)
- Allora, **(1 - 1/n)** è la probabilità che quel RID **non** si trovi in una data pagina.
- Quindi **(1 - 1/n)^k** è la probabilità che ogni pagina non contenga ognuno dei k RID! (cioè non nella prima pagina, non nella seconda, non nella terza... k volte  $(1 - 1/n)$ ).
- E quindi **1 - [(1 - 1/n)^k]** è la probabilità che una data pagina contenga **qualche RID** (almeno uno)!

Ma allora possiamo dire che  $\Phi(k, n) = n \cdot \{ 1 - [(1 - 1/n)^k] \}$  (ogni pagina per la probabilità che quella pagina abbia almeno un RID cercato!)

Questa funzione ha un andamento asintotico del tipo  $\Phi(k, n) \approx \min(k, n)$ . Questo significa che per k molto alto, n è il valore scelto, ovvero n è il numero di pagine da leggere per ottenere tutti i RID, quindi praticamente dobbiamo leggerle tutte dato che n =  $N_{pag}$  totali della tabella. Il DBMS fa queste considerazioni e se il caso non adopera neanche l'indice ma utilizza direttamente la ricerca sequenziale.

### 6.9) Criteri di scelta di un buona chiave di ricerca

Non resta che capire quali siano le caratteristiche per definire una buona chiave di ricerca. Una buona chiave di ricerca deve essere:

- **Selettiva** (non deve avere valori ripetuti per molte tuple, altrimenti avremmo un indice molto corto con liste lunghissime: ci ridurremmo al caso della sequenzialità)
- **Poco volatile** (deve essere cambiata molto poco / mai perché altrimenti il costo di aggiornamento dell'indice sarebbe frequente quindi sconveniente)

## 7) Il metodo di accesso diretto (cenni)

Facciamo alcuni cenni sul metodo di accesso diretto. Tale metodo si basa sulle funzioni di hash. Questo tipo di metodi funzionano **solo con ricerche puntuale**.

### 7.1) Bucket e funzioni di hash

All'interno del sistema abbiamo alcune pagine chiamate bucket, che sono indicizzate. Diciamo nel nostro esempio che sono indicate da 0 a m-1 (quindi ci sono in totale m bucket). Dato il valore di una chiave (k), si "fa passare" all'interno di una funzione detta di hash che è fatta così:  $h = (k \cdot a + b) \bmod(m)$  dove a e b sono dei numeri primi. Questo fa sì che si ottenga un certo indice di bucket (ecco il perché modulo m) dove verrà salvata la chiave. A fianco della chiave avremo i nostri RID.

**Bucket1**

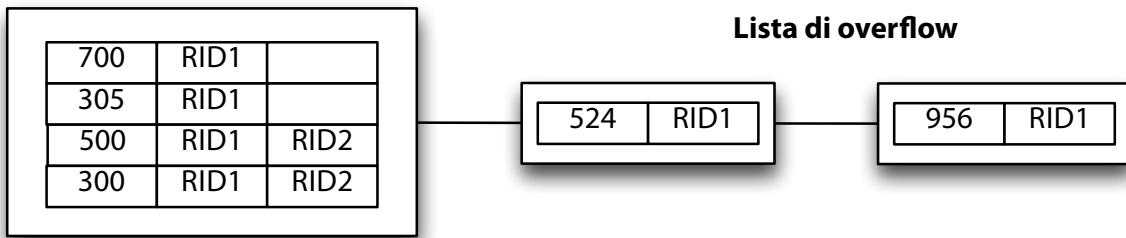
800	RID1	RID2
751	RID1	

Nel caso di una ricerca, viene nuovamente seguita la funzione di hash e così si ritrova l'indice necessario per reperire la chiave. Questo metodo permette un accesso in tempo O(1) cioè con un solo passaggio. Se con C rappresentiamo la capacità di un bucket,  $m \cdot C = \text{capacità totale del sistema}$ .

### 7.2) Il problema della saturazione dei bucket

La funzione di hash per come è fatta distribuisce in modo "saggio" le chiavi, cioè mediamente le "spalma" in equal misura su tutti i bucket. Ma può capitare che non vi sia sufficiente spazio per salvare una chiave in un bucket detto **saturo** ( pieno). In tal caso si "attacca" con un puntatore la chiave, avendo quindi una **lista di overflow**.

**Bucket2**



È facile capire come questa cosa però incida sulle prestazioni, infatti non avremo più un solo accesso ma, incrementalmente, sempre di più.

Chiaramente questo problema si pone solo quando  $m \cdot C < N$  dove N è il numero di chiavi che vogliamo serbare. Si noti che questa è solo una delle soluzioni implementabili ma per brevità vedremo solo questa.

### 7.3) Analisi dei costi

Facciamo ora un'analisi più precisa dei costi. Chiamiamo  $d = N / m \cdot C$  il **fattore di carico** ovvero il numero di chiavi medio in ogni bucket. Un fattore di carico 0,5 significa che il 50% della pagina è piena. Mettiamo ora sulle righe la capacità dei bucket e sulle colonne il fattore di carico. Andiamo quindi a calcolare il numero di accessi (sarà spiegata meglio dopo la tabella).

C \ d	0,5	0,7	0,9
20	1,0	1,02	1,27
50	1,0	1,0	1,08
100	1,0	1,0	1,04

Notiamo subito come un'alto livello di fattore di carico porti gli accessi a valori superiori ad 1.

1,27 accessi significa che con 100 accessi mi aspetto di fare 127 spostamenti di pagine.

Da questi dati si deduce che un buon fattore di carico è intorno all'80%.

Una scelta implementativa potrebbe portare alla coesistenza di indici e di funzioni di hash, i primi usati per le ricerche con range, i secondi invece per la ricerca puntuale. A questo punto potremmo anche pensare di inserire direttamente la tupla vicino alla chiave, ottenendo così che i bucket sarebbero direttamente l'area dove sono salvati i record.

### 8) Considerazioni finali sui metodi d'accesso ed euristica

- Il numero di indici su una relazione è raramente maggiore di quattro, questo poiché altrimenti ogni modifica sarebbe appesantita dall'aggiornamento di qualche indice
- I DBMS tengono traccia di una distribuzione più precisa di quella uniforme che abbiamo ipotizzato noi (se la distribuzione, comunque, non è vicina all'essere uniforme allora l'indice scelto non è buono poiché ad una sola chiave possiamo collegare molte più tuple rispetto a quanto faremmo con le altre chiavi)
- Tipicamente si adotta come indice una chiave esterna (utile per il join)

Le euristiche dipendono dall'architettura, noi esamineremo molto in breve le scelte del DB2 (IBM) e del DBMS di Oracle.

#### In DB2

Prendiamo una selezione generica,  $\sigma_{\psi} \wedge p_1 \wedge p_2 \wedge \dots \wedge p_n$  dove rappresentiamo con  $\psi$  tutti gli attributi non favoriti da un indice, e con  $p_1 \dots p_n$  tutti quegli attributi che, invece, un'indice ce l'hanno. L'ottimizzazione che fa il DBMS è quella di scegliere l'indice fra i  $p_1 \dots p_n$  che convenga di più (cioè il cui utilizzo consti della minore movimentazione delle pagine possibile, ovvero il  $p_i$  che abbia costo di accesso minore).

Una volta rintracciato l'attributo che ha  $C_A$  minore, lo utilizzo ed ottengo l'insieme di tuple desiderate. A questo punto applico le altre uguaglianze e decido quali delle tuple fanno parte del risultato e quali no.

#### Esempio

CC(Titolare, N_Conto, Città_A, Saldo, DataM). Le chiavi sono su Titolare e Città_A. Effettuando ad esempio una  $\sigma_{Titolare = 'xyz' \wedge Città_A = 'Torino' \wedge Saldo = '200'}$  sicuramente il saldo non verrà scelto come attributo (non ha la chiave!). Poi, una volta calcolati i  $C_A$ , si sceglierà l'attributo migliore tra Titolare e Città_A. Ad intuito possiamo supporre che sia Titolare la chiave migliore, poiché è molto più selettiva, cioè ha un  $f_s$  minore (e ricordiamo che  $C_A = f_s(P) \cdot N_{leaf} + \Phi(E_{ris}, N_{pag})$ ).

In Oracle

Differente l'approccio di Oracle che cerca di usare tutte le chiavi intersecando poi i risultati (cioè intersecando i RID comuni ottenuti dalle singole ricerche sulle chiavi). Si nota però che con l'utilizzo di OR piuttosto che != si ha che i vantaggi sono simili tra i due metodi.

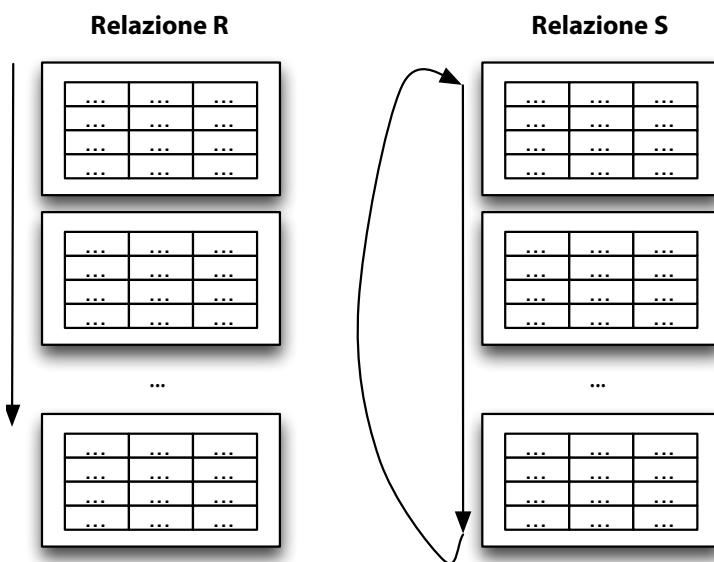
## 9) Algoritmi di join

Essendo il join una delle operazioni più ricorrenti ma al tempo stesso una delle più costose, sono state implementate ben sette categorie differenti di algoritmi di join per risolvere ogni situazione. Per brevità ne vedremo solo tre (e solo alcuni degli algoritmi nelle tre categorie).

### 9.1) Prima categoria: Nested Loop

Il primo algoritmo che vediamo è parecchio semplice.

Consideriamo due relazioni inserite nelle loro pagine. Useremo  $N_{pag}(X)$  per dire il numero totale di pagine della relazione X.



*Algoritmo:*

Presata una pagina della relazione R, si confronta con tutte le pagine della relazione S.

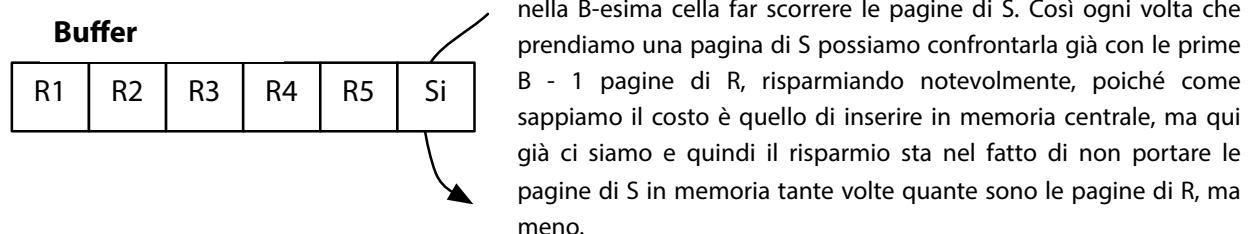
Pertanto mentre R verrà scandita una volta sola, S verrà scandita esattamente  $N_{pag}(R)$  volte.

Ecco perché si ha:

**Costo:**  $N_{pag}(R) + N_{pag}(R) \cdot N_{pag}(S)$

Potrebbe sembrare una tecnica scadente, ma non abbiamo considerato la struttura che supporta tutto questo! Infatti abbiamo un buffer che ci può essere di grande aiuto.

Supponiamo di avere un buffer con capacità B. Allora potremmo mettere le prime  $B - 1$  pagine di R dentro al buffer e nella B-esima cella far scorrere le pagine di S. Così ogni volta che prendiamo una pagina di S possiamo confrontarla già con le prime  $B - 1$  pagine di R, risparmiando notevolmente, poiché come sappiamo il costo è quello di inserire in memoria centrale, ma qui già ci siamo e quindi il risparmio sta nel fatto di non portare le pagine di S in memoria tante volte quante sono le pagine di R, ma meno.



**Costo:**  $N_{pag}(R) + (N_{pag}(R) / (B - 1)) \cdot N_{pag}(S)$ . Le volte in cui percorriamo tutte le pagine di S sono esattamente la quantità di "gruppetti" di  $B - 1$  pagine della relazione R riesco a fare.

Si noti che si sceglierà come R la relazione che ha numero di pagine più basso poiché abbiamo un fattore sommato  $N_{pag}(R)$  che è bene ridurre al minimo.

## 9.2) Prima categoria: Block Loop

Supponiamo ora di avere un join siffatto:  $\sigma_p(r) \bowtie_{A=B} \sigma_q(s)$

Applichiamo questo algoritmo:

*Per ogni  $t_1 \in \sigma_p(r)$*

*Per ogni  $t_2 \in \sigma_q \wedge (B = t_1[A]) (s)$*

**join  $t_1$  e  $t_2$**

Cosa significa? Per ogni tupla uscente dalla prima selezione con predicato p, considera le tuple uscenti dalla selezione del predicato q e che soddisfino il join ( $B = t_1[A]$  è proprio la condizione del join  $A = B$ ). Fai il join tra le due tuple.

Calcoliamo il costo e poi vediamo di capire meglio con un esempio.

**Costo:**  $C_A(r) + [f_s(p) \cdot \text{CARD}(R)] \cdot C_A(s)$

Il primo addendo ci dà il costo di accesso ad R necessario per il primo "per ogni". Tale costo di accesso include la selezione, quindi sarebbe a sua volta  $f_s(p) \cdot \text{Nleaf}$ . In buona sostanza è quindi il costo di accesso per avere il risultato della prima selezione. Il secondo addendo invece è la quantità di tuple che una selezionata su R (dalla p) per il costo di accesso ad S, infatti il secondo "per ogni" accade per ognuno dei risultati della prima selezione, cioè per ognuna delle  $t_1$ .

*Esempio*

Consideriamo: CC(Titolare, N_Conto, Città_A, Saldo, DataM) e CLIENTI(CF, Nome, ...) e consideriamo questo join:

$\sigma_{CF='xyz'}(\text{CLIENTI}) \bowtie_{CF = \text{Titolare}} \sigma_{Città_A = 'To'} \wedge \text{Saldo} = '200'(\text{CC})$ . Facendo qualche conto:

-  $C_A(\text{clienti})$  è circa 1, poiché il fattore di selettività del codice fiscale è 1 (è una chiave) per il numero di foglie coinvolte (sempre uno perché non è una ricerca con range) si ha 1 (trattasi del costo di indirizzamento). Per quanto concerne il costo di raggiungimento dei dati invece possiamo dire che le pagine contengono direttamente le tuple e quindi vale zero (una volta trovato l'indice ho la tupla).

-  $C_A(\text{CC})$  è invece circa 3 poiché ci sono circa tre conti a persona (titolare). Questo valore di  $C_A$  è basso perché applichiamo l'algoritmo e quindi non usiamo  $\sigma_{Città_A = 'To'} \wedge \text{Saldo} = '200'(\text{CC})$  bensì  $\sigma_{Città_A = 'To'} \wedge \dots \wedge \text{Titolare} = CF$  ovvero abbiamo sfruttato il titolare nella ricerca per arrivare più in fretta alla tupla che desideravamo. Con la Città_a avremmo avuto un fattore più alto e quindi un costo di accesso più alto!

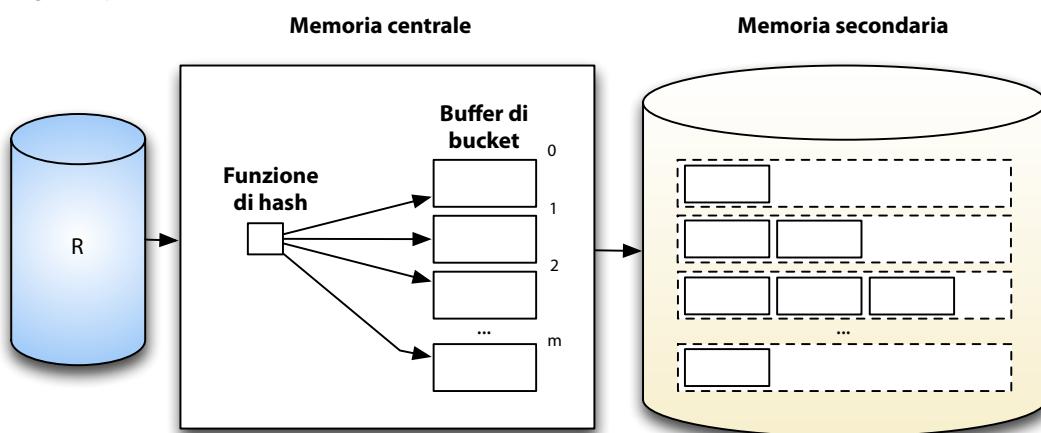
In totale, si ha  $1 + 1 \cdot 3 = 5$  ( $f_s(p) \cdot \text{CARD}(R)$  vale uno perché il CF è una chiave quindi il suo fattore di selettività per la cardinalità fa 1).

### 9.3) Seconda categoria: Hash - Join

Vediamo questo metodo, molto più creativo e divertente. Funziona solo per gli equi - join.

Abbiamo le nostre relazione in memoria secondaria (genericamente, R). In memoria principale ci riserviamo un buffer di bucket (dimensione m) e abbiamo una funzione di hash. In un'altra zona di memoria secondaria riserviamo degli spazi per conservare dei bucket.

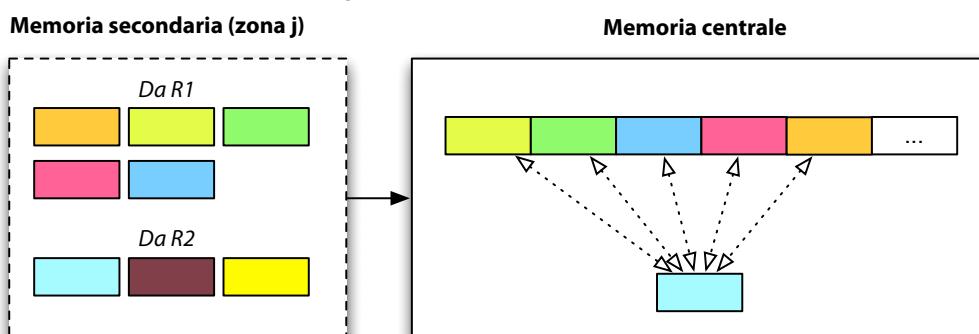
**Fase 1)** Prendiamo in esame  $R_1$  e poi  $R_2$ . Ogni relazione viene letta nella sua totalità, indice per indice. Tramite la funzione di hash ogni indice trovato viene ridirezionato in uno degli  $m$  bucket. Per costruzione, la funzione di hash trovando valori uguali li porterà nello stesso bucket. Quando uno dei bucket è saturo, viene nuovamente copiato in memoria secondaria, ma senza dimenticare da quale posizione arrivasse. Quindi in sostanza in memoria secondaria abbiamo  $m$  zone che tengono tutti i bucket pieni. Una volta conclusa la lettura delle pagine, anche gli ultimi  $m$  bucket vengono spostati in memoria secondaria.



**Costo della fase:**  $2 \cdot (N_{\text{pag}}(R_1) + N_{\text{pag}}(S_2))$ . Ogni pagina viene letta una volta da memoria secondaria e riscritta (sotto forma di bucket, quindi spezzettata in giro) in memoria secondaria (ecco perché 2).

**Fase 2)** Ora si prendono in esame, una per una, le singole zone della memoria secondaria dove ci sono tutti i bucket che contengono chiavi ridirezionate dalla funzione di hash. Per esempio, prendiamo una zona  $j$ . All'interno di una zona avremo bucket con chiavi provenienti da  $R_1$  e chiavi provenienti da  $R_2$ . Separiamole.

A questo punto, portiamo in memoria secondaria tutti i bucket provenienti da  $R_1$  della zona  $j$ . Questi bucket li salviamo in un buffer (che sarà certamente grande a sufficienza).



Lasciamo poi una celletta dove invece facciamo scorrere i bucket provenienti da  $R_2$ . Mentre li facciamo scorrere chiaramente li confrontiamo con tutto il buffer, eseguendo quindi il confronto chiave per chiave  $A_i = A_j$ . Se la corrispondenza è trovata, la tupla fa parte del risultato.

**Costo della fase:**  $N_{\text{pag}}(R_1) + N_{\text{pag}}(S_2)$ , ogni pagina viene ricoppiata nuovamente in memoria centrale.

**Costo totale:**  $3 \cdot (N_{\text{pag}}(R_1) + N_{\text{pag}}(S_2))$

#### 9.4) Terza categoria: Sort - Merge

Ultima categoria che vediamo è il sort - merge. Funziona solo per gli equi - join.

Supponiamo di avere le due relazioni ordinate basandosi sulla chiave. In tal caso potremmo prendere una chiave di  $R_1$  e confrontarla con  $R_2$ . Quando su  $R_2$  "sforiamo" (ed è qui che l'ordinamento è utile), allora ci fermiamo. Proseguiamo su  $R_1$  e continuamo esattamente da dove eravamo rimasti con  $R_2$  finché non sforiamo di nuovo, e così via. Ogni relazione viene letta una sola volta.

**Costo:**  $\text{ORD}(R_1) + \text{ORD}(R_2) + \text{Npag}(R) + \text{Npag}(S)$

Questo metodo non è molto usato poiché i costi di ordinamento sono, talvolta, molto elevati.

---

# **Basi di dati**

## *Gestione delle concorrenze e crash recovery*

### Capitolo 7

---

*Enrico Mensa*

# Indice degli argomenti

## Parte I: Gestione delle concorrenze

1) L'esecuzione simultanea delle transazioni	1
1.1) Storie e notazioni	
1.2) Transazioni con interleaving	
2) Equivalenza tra storie (view - equivalenza)	3
2.1) Condizioni di equivalenza	
2.2) Princípio di serializzabilità (IMPORTANTE!)	
3) Classificazioni delle anomalie nell'interleaving	4
3.1) Perdita di aggiornamento	
3.2) Lettura sporca	
3.3) Lettura non ripetibile	
3.4) Aggiornamento fantasma	
3.5) Inserimento fantasma	
4) Grafo dei conflitti	5
4.1) I possibili conflitti	
4.2) Costruzione del grafo	
4.3) Teorema: test di serializzabilità	
5) Il lock	8
5.1) L'idea intuitiva del lock	
5.2) La gestione dei lock (tavola di compatibilità e tabella dei lock)	
5.3) L'applicazione reale dei lock	
Il protocollo a due fasi	
5.4) Il teorema di serializzabilità basato sul protocollo a due fasi	
5.5) Correttivo del protocollo a due fasi: il protocollo a due fasi stretto / forte	
6) La gestione del deadlock	12
6.1) Prima soluzione: grafo di attesa	

**6.2) Seconda soluzione: Timeout**

**6.3) Terza soluzione: Timestamp**

7) *Riflessioni finali*

13

**7.1) Granularità del lock**

**Parte II: Crash Recovery**

1) *Contestualizzazione del crash recovery*

14

2) *Le quattro tipologie di crash recovery (crash del sistema)*

14

**2.1) Il file di log**

**2.2) UNDO / REDO**

UNDO(T₁, ..., T_k)

REDO(T₁, ..., T_k)

**2.3) Caso uno: steal / no-flush**

Uso del file di log

Sistema del crash recovery

**2.4) Il checkpoint nei file di log**

**2.5) Caso due: no-steal / no-flush**

Uso del file di log

Sistema del crash recovery

**2.6) Caso tre: steal / flush**

Uso del file di log

Sistema del crash recovery

**2.7) Caso quattro: no-steal / flush**

3) *Crash recovery con pagine ombra (DB multimediali)*

20

4) *Dump Restore (crash della periferica)*

21

5) *Rivisitazione della tabella iniziale*

21

## Parte I: Gestione delle concorrenze

### 1) L'esecuzione simultanea delle transazioni

**Ricordiamo le proprietà di una transazione (Capitolo 5, paragrafo 1.2):**

Una transazione deve rispettare le proprietà base, **ACID**.

A = Atomica, (una transazione deve essere eseguita nella sua interezza, altrimenti non ha senso)

C = Consistente, (una transazione deve rispettare in ogni sua azione i vari vincoli)

I = Isolata (una transazione viene considerata come 'eseguita da sola' senza doversi occupare delle concorrenze)

D = Durabile (gli effetti di una transazione devono poter essere persistenti)

		Consideriamo due transazioni T ₁ e T ₂ .
T ₁	T ₂	
read(A)	read(A)	Notiamo che la proprietà di queste transazioni è quella di mantenere costante la somma tra i valori di A e B, quindi A + B = <i>costante</i> .
A := A - 50	temp = A · 0,1	Prendiamo ad esempio due valori da cui partire, - A = 1.000 - B = 2.000 e quindi A + B = 3.000
write(A)	A := A - temp	Allora con l'esecuzione sequenziale di T ₁ e poi quella di T ₂ otterremmo A = 855 B = 2145 con somma A + B = 3.000
read(B)	write(A)	Consideriamo invece l'esecuzione sequenziale di T ₂ e poi quella di T ₁ avremmo: A = 850 B = 2150 con somma A + B = 3.000
B := B + 50	read(B)	
write(B)	B := B + temp	
	write(B)	

Come si vede dai valori trovati la somma rimane costante quindi le due combinazioni T₁T₂ e T₂T₁ sono legittime.

**L'esecuzione sequenziale di due transazioni è sempre legittima.** Dal punto di vista del DBMS quindi queste due sequenze sono "legali" e permesse (al di là del fatto che i valori siano diversi!). Potremmo pensare ad esempio ad un deposito/prelievo bancario per effettuare un bonifico. L'ordine è irrilevante ma è essenziale il risultato.

### 1.1) Storie e notazioni

Quando due transazioni compiono una serie di azioni, possiamo scriverle di seguito e chiamarla **storia**. Possiamo considerare azioni di due tipi: r_i(X) e w_i(X). Con la prima si intende che la transazione i-esima legga X, con la seconda che la transazione i-esima scriva X. Si noti che nella storia si considerano solo le letture e le scritture poiché ci mettiamo dal punto di vista del DBMS che, appunto, vede solo quelle.

Pertanto le transazioni T₁ e T₂ con le due storie che abbiamo provato a fare prima appaiono rispettivamente come:

**Storia (T₁T₂):** r₁(A), w₁(A), r₁(B), w₁(B), r₂(A), w₂(A), r₂(B), w₂(B)

**Storia (T₂T₁):** r₂(A), w₂(A), r₂(B), w₂(B), r₁(A), w₁(A), r₁(B), w₁(B)

### 1.2) Transazioni con interleaving

Purtroppo l'utilizzo sequenziale delle transazioni non è sensato né tantomeno efficace. Pertanto si adotta la tecnica dell'interleaving, che non è così banale dato il fatto che dobbiamo mantenere il principio di isolamento ovvero per la transazione deve essere "come se fosse da sola". Vediamo un paio di esempi.

Esempio 1

Vediamo il susseguirsi delle azioni in ogni istante di tempo.

t	T ₁	T ₂
1	<b>read(A)</b>	
2	A := A - 50	
3		<b>read(A)</b>
4		temp = A · 0,1
5		A := A - temp
6		<b>write(A)</b>
7		<b>read(B)</b>
8	<b>write(A)</b>	
9	<b>read(B)</b>	
10	B := B + 50	
11	<b>write(B)</b>	
12		B := B + temp
13		<b>write(B)</b>

Con questo susseguirsi di azioni, abbiamo questi risultati:

- A = 950

- B = 2.100

e quindi una somma A + B = 3.050, **inconsistenza!**

La somma non vale più 3.000, quindi questo interleaving non è accettabile.

Questo accade perché all'istante 8 sovrascrivo ciò che, precedentemente, T₂ aveva scritto all'istante 6 in A.

Stessa cosa accade poi per B all'istante 13, sovrascrivendo la write di T₁ dell'istante 11.

Questa storia è:

S₁: r₁(A), r₂(A), w₂(A), r₂(B), w₁(A), r₁(B), w₁(B), w₂(B)

In definitiva questo risultato è, per il DBMS, inconsistente.

Esempio 2

Con questo susseguirsi di azioni, abbiamo questi risultati:

- A = 855

- B = 2.159

e quindi una somma A + B = 3.000, **è consistente!**

La somma vale 3.000, quindi questo interleaving è accettabile.

Questa storia è:

S₂: r₁(A), w₁(A), r₂(A), w₂(A), r₁(B), w₁(B), r₂(B), w₂(B)

In definitiva questo risultato è, per il DBMS, consistente.

t	T ₁	T ₂
1	<b>read(A)</b>	
2	A := A - 50	
3	<b>write(A)</b>	
4		<b>read(A)</b>
5		temp = A · 0,1
6		A := A - temp
7		<b>write(A)</b>
8	<b>read(B)</b>	
9	B := B + 50	
10	<b>write(B)</b>	
11		<b>read(B)</b>
12		B := B + temp
13		<b>write(B)</b>

## 2) Equivalenza tra storie (view - equivalenza)

Per poter capire quali storie "vadano bene" e quali no, dobbiamo introdurre la nozione di equivalenza tra storie.

Ovvero, quando possiamo dire che  $S_i \equiv S_j$  ?

### 2.1) Condizioni di equivalenza

Notazione: se i puntini che precedono/seguono una azione  $r_i(X)/w_i(X)$  significano "nessuno ha modificato X".

Una certa storia  $S_1$  è equivalente ad una storia  $S_2$  ( $S_1 \equiv S_2$ ) se valgono queste tre condizioni:

1)  $S_1$  ed  $S_2$  condividono le medesime azioni (anche in ordine diverso)

2) controllo sull'input: se

- $S_1: \dots r_i(X) \text{ allora } S_2: \dots r_i(X)$  (*viene letto X senza che sia stato modificato prima in entrambe le storie*)  
oppure
- $S_1: \dots w_j(X) \dots r_i(X) \text{ allora } S_2: \dots w_j(X) \dots r_i(X)$  (*viene letto X dopo una modifica di una certa transazione j in entrambe le storie*)

3) controllo sull'output: se

- $S_1: \dots w_i(X) \dots \text{ allora } S_2: \dots w_i(X) \dots$  (*la transazione i scrive per ultima X in entrambe le storie*)

Detto questo, possiamo dire che  $S_2 \equiv T_1 T_2$ . Vediamo se le tre proprietà sono soddisfatte:

**T₁T₂:**  $r_1(A), w_1(A), r_1(B), w_1(B), r_2(A), w_2(A), r_2(B), w_2(B)$

**S₂:**  $r_1(A), w_1(A), r_2(A), w_2(A), r_1(B), w_1(B), r_2(B), w_2(B)$

- ✓ Banalmente contengono le stesse azioni, (proprietà 1)
- ✓  $r_1(A)$  viene effettuato senza che prima venga scritto A in entrambe le storie, (proprietà 2 caso 1)
- ✓  $r_1(B)$  viene effettuato senza che prima venga scritto B in entrambe le storie, (proprietà 2 caso 1)
- ✓  $r_2(A)$  viene effettuato in seguito a  $w_1(A)$  in entrambe le storie, (proprietà 2 caso 2)
- ✓  $r_2(B)$  viene effettuato in seguito a  $w_1(B)$  in entrambe le storie, (proprietà 2 caso 2)
- ✓  $w_2(B)$  è l'ultima transazione a scrivere B in entrambe le storie, (proprietà 3)
- ✓  $w_2(A)$  è l'ultima transazione a scrivere A in entrambe le storie, (proprietà 3)

Invece non è vero che  $S_1 \equiv T_1 T_2$ , infatti:

**T₁T₂:**  $r_1(A), w_1(A), r_1(B), w_1(B), r_2(A), w_2(A), r_2(B), w_2(B)$

**S₁:**  $r_1(A), r_2(A), w_2(A), r_2(B), w_1(A), r_1(B), w_1(B), w_2(B)$

- ✓ Banalmente contengono le stesse azioni, (proprietà 1)
- ✓  $r_1(A)$  viene effettuato senza che prima venga scritto A in entrambe le storie, (proprietà 2 caso 1)
- ✓  $r_1(B)$  viene effettuato senza che prima venga scritto B in entrambe le storie, (proprietà 2 caso 1)
- *  $r_2(A)$  **NON** viene effettuato in seguito a  $w_1(A)$  in entrambe le storie, (proprietà 2 caso 2) (accade solo in  $T_1 T_2$ , mentre in  $S_1$   $r_2(A)$  non è preceduto da  $w_1(A)$ )
- *  $r_2(B)$  **NON** viene effettuato in seguito a  $w_1(B)$  in entrambe le storie, (proprietà 2 caso 2) (accade solo in  $T_1 T_2$ , mentre in  $S_1$   $r_2(B)$  non è preceduto da  $w_1(B)$ )
- ✓  $w_2(B)$  è l'ultima transazione a scrivere B in entrambe le storie, (proprietà 3)
- *  $w_2(A)$  **NON** è l'ultima transazione a scrivere A in entrambe le storie, (proprietà 3) (in  $T_1 T_2$  l'ultima a scrivere è  $w_2(A)$  mentre in  $S_1$  l'ultima a scrivere A è  $w_1(A)$ )

Dato che almeno una delle delle condizioni non è rispettata, non vale l'equivalenza.

## 2.2) Principio di serializzabilità (IMPORTANTE!)

Una storia S è corretta se è equivalente ad una storia seriale (una qualsiasi!).

Si noti che con  $T_1 \dots T_n$  si hanno ben  $n!$  storie seriali differenti. Quindi è sufficiente che S sia equivalente ad una di esse per essere corretta.

## 3) Classificazioni delle anomalie nell'interleaving

Tutte le analisi che abbiamo fatto ora sono però a posteriori. Cerchiamo di trovare qualche tecnica a priori per evitare "il danno", e quindi salvare la situazione prima di arrivare ad un interleaving erroneo. Ma prima cerchiamo di esaminare la problematica dal punto di vista teorico.

Ci sono cinque anomalie "classiche" che ci preniamo di evitare. Vediamole.

### 3.1) Perdita di aggiornamento

Supponiamo di avere una semplice transazione che legge X, lo incrementa e poi lo riscrive. Supponiamo di lanciare la transazione due volte per ottenere "+2".

t	T ₁	T ₂
1	<b>read(X)</b>	
2	X := X + 1	
3		<b>read(X)</b>
4		X := X + 1
5		<b>write(X)</b>
6	<b>write(X)</b>	

La scrittura all'istante 6 sovrascrive la scrittura dell'istante 5, rendendo l'aggiornamento non effettivo.

Quindi invece di incrementare la variabile due volte, la incrementeremmo solo una.

### 3.2) Lettura sporca

Ci troviamo nel caso:

.....w₁(X).....r₂(X).....Abort(T₁)

Talvolta capita che una transazione riceva un Abort e quindi venga uccisa. Ma se questa transazione aveva già intaccato i dati (w₁(X)) allora le successive letture da parte di altre transazioni (r₂(X)) saranno sporiose, nonostante l'abort che riporterà il valore di X allo stato originale (ma T₂ oramai aveva letto il vecchio!).

### 3.3) Lettura non ripetibile

Ci troviamo nel caso:

.....r₁(X).....w₂(X).....r₁(X)

Anche se un po' stupido, supponiamo che T₁ legga più volte il dato dal DB. In mezzo alle due letture viene piazzata una scrittura (w₂(X)): chiaramente le due letture danno dati differenti e quindi il principio di isolamento è perso dato il no-sense della situazione (se veramente la T₁ fosse isolata i dati non dovrebbero cambiare senza che sia lei a toccarli!).

### 3.4) Aggiornamento fantasma

Ci troviamo nel caso in cui ci sia una consistenza dettata dal fatto che  $X + Y = \text{costante}$ .

t	T ₁	T ₂
1	<b>read(X)</b>	
2		<b>read(X)</b>
3		$X := X - 100$
4		<b>read(Y)</b>
5		$Y := Y + 100$
6		<b>write(X)</b>
7		<b>write(Y)</b>
8	<b>read(Y)</b>	
9	<u>check X + Y</u>	

Leggiamo una certa X, dopodiché un'altra transazione si infila e modifica le due variabili (mantenendo tra l'altro il vincolo  $X + Y = \text{costante}$ ). A questo punto quando T₁ torna ad eseguire si trova una Y nuova, che sommata alla precedente X ovviamente non può che dare un risultato errato rispetto alla costante attesa dalla somma.

Questo tipo di anomalia è molto particolare poiché il DBMS non si accorge di nulla, infatti il vincolo all'interno del database è rispettato, ma è il dato in possesso di T₁ che è inconsistente!

### 3.5) Inserimento fantasma

Abbiamo un insieme  $I = \{X_1, \dots, X_n\}$  di oggetti. Si compiono poi tre azioni:

- * T₁ esegue un'operazione di conto di quanti elementi vi sono in I,
- * T₂ aggiunge un nuovo oggetto
- * T₁ calcola nuovamente gli elementi di I

Anche qui i due valori sono differenti, quindi abbiamo una violazione dell'isolamento.

## 4) Grafo dei conflitti

L'approccio dell'equivalenza fra storie non è molto buono dal punto di vista algoritmico (abbiamo visto che dobbiamo esaminare n! casi possibili per verificare che sia serializzabile), cerchiamone quindi un'altro.

### 4.1) I possibili conflitti

Per poter capire se vi sono conflitti in una situazione è necessario compilare il grafo dei conflitti, e per compilarlo dobbiamo prima di tutto capire quali siano le combinazioni "illegali".

Prese le operazioni di lettura e scrittura abbiamo quattro possibili combinazioni in una storia:

- 1) S: .....r_i(X).....w_j(X).....
- 2) S: .....w_j(X).....r_i(X).....
- 3) S: .....w_j(X).....w_i(X).....
- 4) S: .....r_j(X).....r_i(X).....

I primi tre casi sono casi conflittuali. Fra questi, i primi due sono azioni in conflitto poiché la loro inversione creerebbe delle differenze nell'input corrispettivo mentre la terza non va bene poiché sono in grado di alterare lo stato (l'ultima scrittura sovrascrive tutte le altre vecchie).

Il quarto caso non è un problema poiché non va a toccare i dati ma li legge soltanto.

#### 4.2) Costruzione del grafo

La costruzione del grafo  $G(S_i)$  è molto semplice:

- 1) Nodi: Ogni transazione coinvolta diventa un nodo  $T_i$
- 2) Archi:

- Data la storia del tipo S: .....  $w_i(X) \dots r_{j1}(X) \dots r_{j2}(X) \dots w_k(X)$  considero tutte le letture tra due scritture (oppure da una scrittura fino alla fine) e creo un arco da  $T_i$  a  $T_{j1}$  e da  $T_i$  a  $T_{j2} \dots$  da  $T_i$  a  $T_k$  (si intende quindi rappresentare che la scrittura di  $T_i$  "ha fatto dipendere" tutte le letture da parte di  $T_{j1}, T_{j2}, \dots$  e la scrittura di  $T_k$ )  
[abbiamo rappresentato i conflitti due e tre]
- Data la storia del tipo S: .....  $r_i(X) \dots w_j(X)$  considero la scrittura di  $T_j$  e quindi creo un arco da  $T_i$  a  $T_j$  (si intende rappresentare la dipendenza della lettura di  $T_i$  dalla scrittura di  $T_j$ )  
[abbiamo rappresentato il conflitto uno]

##### Esempio 1 - $S_1$

Consideriamo la storia  $S_1$ :  $r_1(A), r_2(A), w_2(A), r_2(B), w_1(A), r_1(B), w_1(B), w_2(B)$

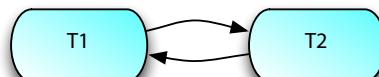
- Creiamo prima di tutto due nodi,  $T_1$  e  $T_2$



- Leggiamo partendo dalla prima azione, troviamo  **$r_1(A)$ ,  $r_2(A)$ ,  $w_2(A)$** , ... e ci troviamo quindi nel secondo caso per cui si deve creare un arco. Creiamo quindi l'arco  $T_1 \rightarrow T_2$ .



- Osservando la seconda azione, troviamo ...,  **$r_2(A)$ ,  $w_2(A)$ ,  $r_2(B)$ ,  $w_1(A)$** , ... e ci troviamo quindi nel primo caso per cui si deve creare un arco. Creiamo quindi l'arco  $T_2 \rightarrow T_1$ .



Essendosi creato un **ciclo**, ci fermiamo qui (capiremo meglio dopo cosa significa avere un ciclo).

##### Esempio 2 - $S_2$

Consideriamo la storia  $S_2$ :  $r_1(A), w_1(A), r_2(A), w_2(A), r_1(B), w_1(B), r_2(B), w_2(B)$

- Creiamo prima di tutto due nodi,  $T_1$  e  $T_2$



- Leggiamo partendo dalla prima azione, troviamo  **$r_1(A)$ ,  $w_1(A)$ ,  $r_2(A)$ ,  $w_2(A)$** , ... e ci troviamo quindi nel secondo caso per cui si deve creare un arco. Creiamo quindi l'arco  $T_1 \rightarrow T_2$ .



- Osservando la seconda azione, troviamo ...,  **$w_1(A)$ ,  $r_2(A)$ ,  $w_2(A)$**  ... e ci troviamo quindi nel primo caso per cui si deve creare un arco. Creiamo quindi l'arco  $T_1 \rightarrow T_2$  (ma c'è già! quindi rimane inalterato).

- La terza azione  $r_2(A)$  non da problemi poiché non è seguita da scritture fatte da altre transazioni

- La quarta azione  $w_2(A)$  non da problemi poiché non è seguita da altre scritture su A

- Osservando la quinta azione, troviamo ...,  **$r_1(B)$ ,  $w_1(B)$ ,  $r_2(B)$ ,  $w_2(B)$**  ... e ci troviamo quindi nel secondo caso per cui si deve creare un arco. Creiamo quindi l'arco  $T_1 \rightarrow T_2$  (ma c'è già! quindi rimane inalterato).

- Osservando la sesta azione, troviamo ...,  **$w_1(B)$ ,  $r_2(B)$ ,  $w_2(B)$**  ... e ci troviamo quindi nel primo caso per cui si deve creare un arco. Creiamo quindi l'arco  $T_1 \rightarrow T_2$  (ma c'è già! quindi rimane inalterato).

Il secondo esempio non ha cicli.

#### 4.3) Teorema: test di serializzabilità

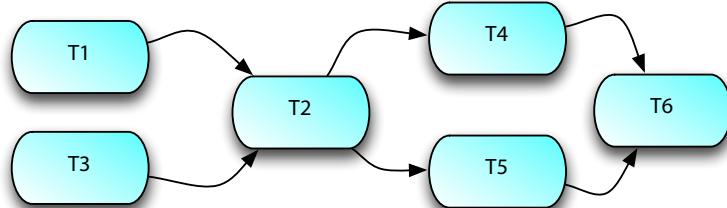
"Se il grafo dei conflitti  $G(S)$  di una storia  $S$  è aciclico, allora  $S$  è serializzabile".

Si noti che questa condizione è necessaria, quindi se il grafo è aciclico necessariamente la storia è serializzabile ma non è detto, dato un grafo ciclico, che la storia sia non serializzabile.

*Dimostrazione per induzione del teorema*

- Passo base:  $G(T_1)$  è banalmente aciclico (data una transazione soltanto, il suo grafo è aciclico poiché è un nodo solitario)
- Poniamo vero il test su  $n-1$  transazioni e poi dimostriamo che è vero per  $n$  transazioni.

Prendiamo un grafo generico aciclico:



Allora ci deve essere almeno un nodo che non ha archi entranti (altrimenti il grafo non sarebbe aciclico!). In questo caso ne abbiamo due,  $T_1$  e  $T_3$ . Consideriamo ad esempio  $T_1$ . Nella storia mi imbatterò nelle varie azioni compiute da  $T_1$ , genericamente potremmo dire  $\dots r_1(X) \dots w_1(Y) \dots w_1(Z)$ . Prendiamo tutte queste azioni e le portiamo in testa alla storia  $S$ , ottenendo quindi qualcosa del genere:  $r_1(X), w_1(Y), w_1(Z), \dots$ .

Potremmo supporre di aver ottenuto:  $S \equiv T_1 S'$  cioè aver scisso la storia  $S$  in una sottostoria  $S'$  a cui fa capo tutta  $T_1$ .

Sarà vera questa equivalenza? Verifichiamola applicando la regola della view - equivalenza!

Come sappiamo abbiamo due tipologie di conflitti preoccupanti:

- .....  $r_i(X) \dots w_1(X)$
- .....  $w_i(X) \dots w_1(X)$

Ma entrambe queste situazioni non possono verificarsi poiché se così fosse nel grafo avremmo avuto un arco  $T_i \rightarrow T_1$  e quindi un arco entrante in  $T_1$  ma noi sappiamo che questo non può essere vero! Abbiamo quindi verificato l'equivalenza  $S \equiv T_1 S'$ .

Non resta che sfruttare un'altra proprietà dei grafi secondo la quale eliminando un nodo da un grafo aciclico otteniamo sempre un grafo aciclico. Da questo, possiamo togliere un nodo alla volta ed applicare ogni volta l'equivalenza garantendo quindi di avere una storia serializzabile.

Quindi, otterremmo che  $S \equiv T_1 T_3 T_4 T_5 T_6$  ma anche  $S \equiv T_3 T_1 T_2 T_4 T_5 T_6$ .

Abbiamo insomma tutti gli ordini topologici del grafo come possibili serializzazioni.

Ecco quindi dimostrato che se il grafo è aciclico, possiamo ritrovare una storia serializzabile equivalente.

## 5) Il lock

Ora possiamo finalmente introdurre una tecnica preventiva per evitare i conflitti. Dobbiamo stilare dei protocolli che garantiscano storie serializzabili.

### 5.1) L'idea intuitiva del lock

Il lock si basa su un semplice concetto, prima di fare una lettura o una scrittura chiedi al DBMS se puoi farla. Ve ne sono due tipi:

- * **LS - Lock Shared** (richiesto dalle transazioni quando vogliono leggere, permette l'accesso comunitario)
  - * **LX - Lock Exclusive** (richiesto dalle transazioni quando vogliono scrivere, permette solo l'accesso esclusivo)
- Abbiamo poi l'operazione di UN(X) che elimina un certo lock sulla risorsa X.

### 5.2) La gestione dei lock (tavola di compatibilità e tabella dei lock)

Ma come funzionano questi lock? Vediamo una tabella detta tavola di compatibilità. Il raffronto da fare è "se posseggo un certo lock e mi arriva una richiesta di un'altro lock, che faccio?"

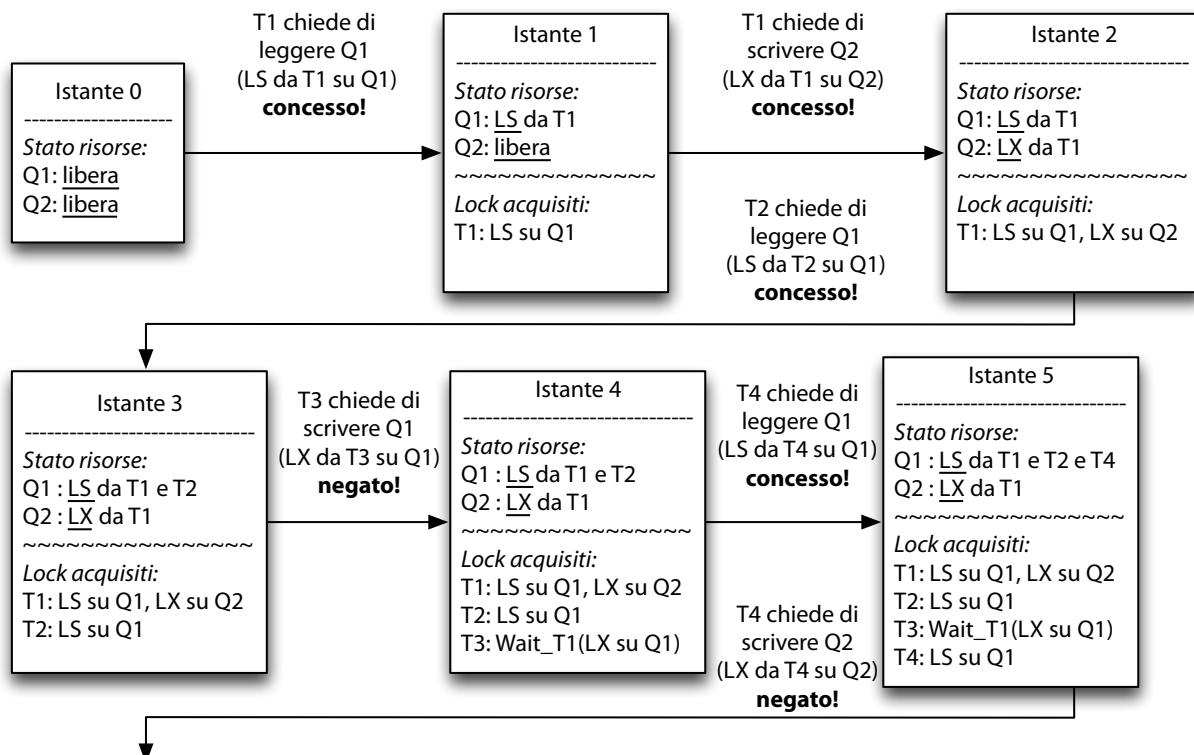
posseggono \ richiesta	LS	LX
LS	Concedi	Nega
LX	Nega	Nega

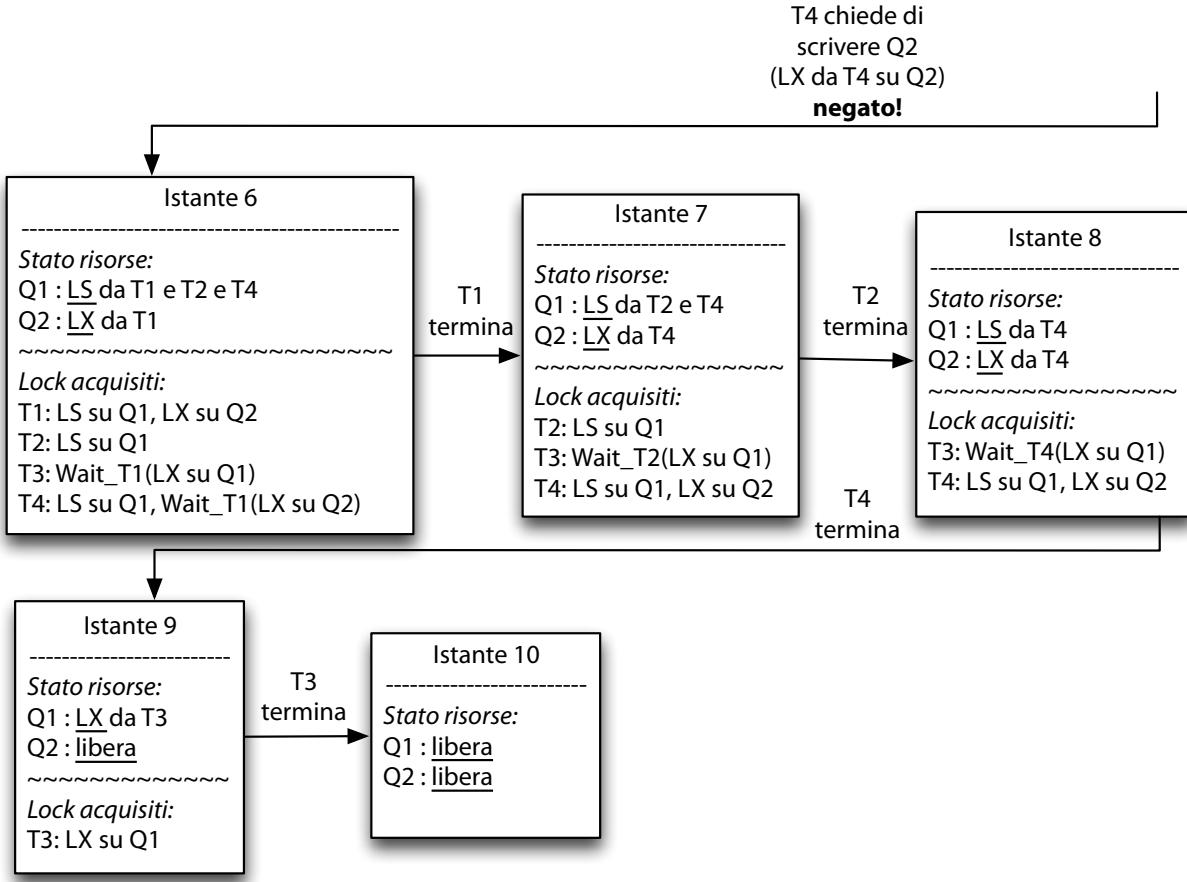
Quando il DBMS nega un lock, lascia in waiting la transazione che l'aveva richiesta.

È grazie a queste scelte che il DBMS gestisce l'intero sistema di lock. Vediamo un esempio.

Date Q₁ e Q₂ come risorse, rappresentiamo in sequenza le varie richieste.

*Nota:* con Wait_T_n(X) si intende che la transazione attende che T_n abbia finito per poter eseguire l'azione X.





Questo sistema non è complesso ma così com'è non ci risolve nessun problema, infatti se lo applicassimo nella nostra  $S_1$  noteremmo che i lock funzionerebbero senza errore alcuno, infatti avremmo:

t	T ₁	T ₂
1	LS(A) <b>read(A)</b> UN(A)	
2	A := A - 50	
3		LS(A) <b>read(A)</b> UN(A)
4		temp = A · 0,1
5		A := A - temp
6		LX(A) <b>write(A)</b> UN(A)
7		LS(B) <b>read(B)</b> LS(B)

8	LX(A) <b>write(A)</b> UN(A)	
9	LS(B) <b>read(B)</b> LS(B)	
10	B := B + 50	
11	LX(B) <b>write(B)</b> UN(B)	
12		B := B + temp
13		LX(B) <b>write(B)</b> UN(B)

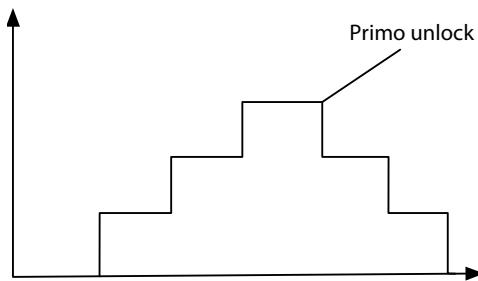
### 5.3) L'applicazione reale dei lock

I lock vengono quindi usati sotto certe condizioni.

- Le **transazioni** devono essere **ben formate** (ad ogni lock deve corrispondere un unlock)
- Le **storie** devono essere **legali** (un lock esclusivo deve essere presente in una unica transazione, ovvero ogni oggetto deve avere al più un lock esclusivo)
- Le **transazioni** devono avere il **protocollo a due fasi**.

#### Il protocollo a due fasi

Il protocollo a due fasi fa sì che una certa transazione possa chiedere lock a piacere, ma dopo il primo unlock non possa fare altro che rilasciare i suoi precedente. Perciò avremo un grafico del genere (sulle Y il conteggio dei lock, sulle X il tempo). Il protocollo a due fasi evita i rollback in cascata.



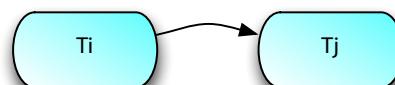
### 5.4) Il teorema di serializzabilità basato sul protocollo a due fasi

"Se la transazione T segue il protocollo a due fasi ed è ben formata, allora ogni sua storia legale è serializzabile."

Questo teorema si basa sulle definizioni del paragrafo 5.3.

Dimostriamo che il teorema è vero. Riprendiamo le anomalie e inseriamo i lock, per vedere quali situazioni avremmo se fossimo in una situazione anomala. A fianco il grafo ottenuto.

- 1) S: .....r_i(X).....UN_i(X).....LX_j(X).....w_j(X)....
- 2) S: .....w_i(X).....UN_i(X).....LS_j(X).....r_j(X).....
- 3) S: .....w_i(X).....UN_i(X).....LX_j(X).....w_j(X)....



Come vediamo il "pattern" è sempre lo stesso, quindi possiamo riflettere in maniera generale.

Chiamiamo *h* la posizione in cui avviene il generico UNLOCK all'interno di una storia e chiamiamo *k* il primo LOCK richiesto in seguito all'UNLOCK in posizione *h* (quindi le due parti in grassetto vengono rispettivamente chiamate *h* e *k*).

Definiamo poi una funzione **primoRilascio(T)** che ci dice la posizione della storia in cui avviene il primo UNLOCK (da parte di T). Quindi ogni transazione (ovvero ogni nodo) avrà una sua funzione **primoRilascio()** che ci dirà la posizione del primo UNLOCK di quella transazione. Non resta che trarre alcune conclusioni.

Sappiamo che **primoRilascio(T_i) ≤ h** poiché nella serie di UNLOCK possibili, potremmo avere che .....r_i(X).....UN_i(X).. sia il primo piuttosto che uno seguente.

Inoltre, dato che stiamo agendo sotto il protocollo a due fasi, abbiamo che **primoRilascio(T_j) > k** poiché nel protocollo a due fasi prima abbiamo tutti i LOCK e poi tutti gli UNLOCK, ed essendo *k* la posizione di una richiesta di LOCK, allora sarà sicuramente precedente (ovvero minore) del primo rilascio effettuato da T_j (UNLOCK).

Ma come vediamo da sopra, sappiamo che **h < k** e quindi **primoRilascio(T_i) < primoRilascio(T_j)** e dato che questo è vero ci garantisce la non presenza di cicli nel grafo.

Infatti, se avessimo dei cicli (supponiamo tre nodi collegati uno con l'altro), avremmo una situazione del genere: primoRilascio(T₁) < primoRilascio(T₂) < primoRilascio(T₃) < primoRilascio(T₁) che è chiaramente no-sense.

### 5.5) Correttivo del protocollo a due fasi: il protocollo a due fasi stretto / forte

Esaminiamo ora un caso che il protocollo a due fasi (così come visto da noi) non prevede:

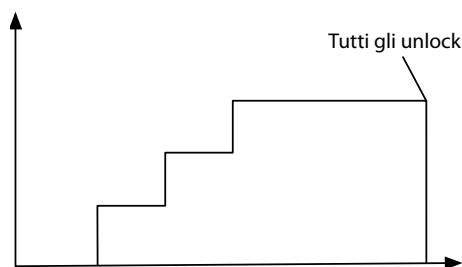
t	T ₁	T ₂
1	LX(X) <b>write(X)</b>	
2	LX(Y) <b>write(Y)</b>	
3	...	
4	UN(X)	
5	...	
6		LS(X) <b>read(X)</b> UN(X) <b>COMMIT</b>
7	...	
8	UN(Y)	
9	...	
10	<b>ABORT</b>	

Innanzi tutto notiamo che la storia è legale, è ben formata e viene rispettato il protocollo a due fasi (non c'è mai un LOCK dopo il primo UNLOCK in ogni transazione).

Ci aspettiamo quindi che tutto funzioni per il meglio, ma invece non è così.

Siamo infatti in una situazione di anomalia, più precisamente in un caso di lettura sporca, poiché ciò che legge T₂ è frutto delle write di T₁, la quale viene però abortita (e quindi il suo effetto dovrebbe essere "nullo").

Introduciamo quindi la nuova nozione di protocollo a due fasi **stretta**, dove una transazione rilascia tutti i lock solamente alla fine della transazione, ovvero dopo il commit.



V'è poi la versione **forte**, che è del tutto simile ma impone questo vincolo solo sui LX e non sui LS (sono solo i LX che permettono scritture e quindi potenziali "danni").

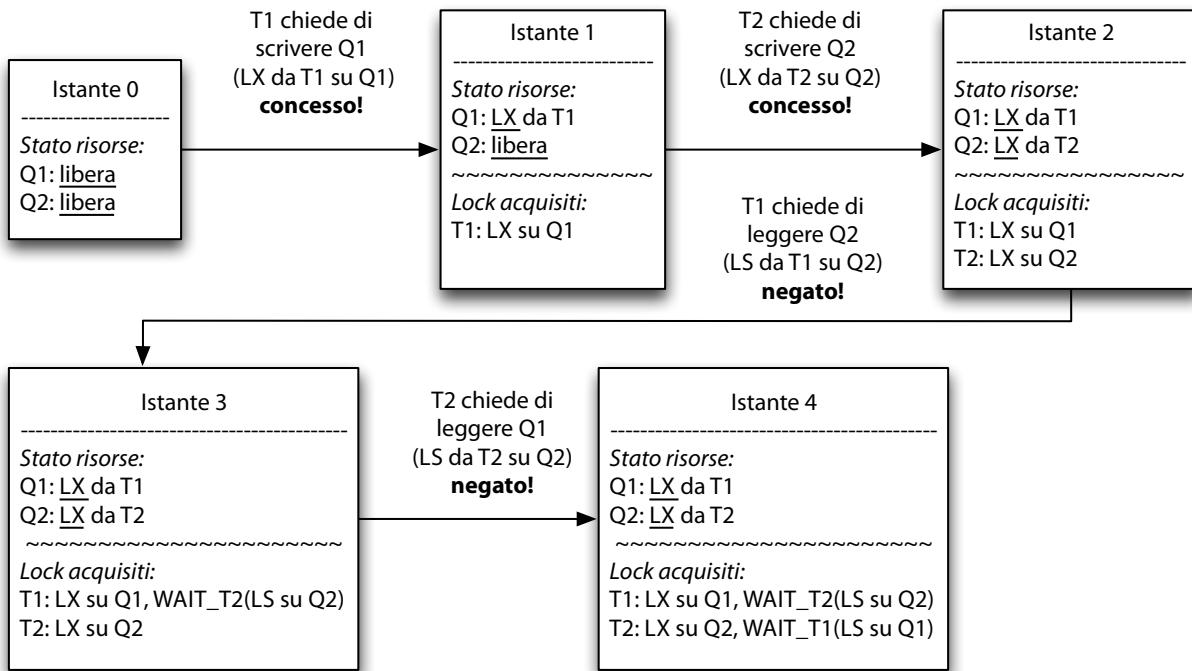
Usando questa versione:

Pro: garantisce la serializzazione come la versione base e risolve anche l'anomalia di lettura sporca.

Contro: grande perdita del parallelismo.

## 6) La gestione del deadlock

Può capitare che alcune transazioni rimangano "invischiate", cioè si attendano avvicendevolmente. Un caso banale:



Come si vede, mentre  $T_1$  è in WAIT per  $T_2$ ,  $T_2$  è in WAIT per  $T_1$ .

Vi sono tre tecniche risolutive, vediamole sommariamente.

### 6.1) Prima soluzione: grafo di attesa

Si genera un grafo dove ogni transazione è un nodo e ogni arco che va da  $T_i$  a  $T_j$  indica che  $T_i$  è in attesa che  $T_j$  liberi uno o più dei suoi lock.

Chiaramente se nel grafo c'è un **ciclo** significa che c'è una situazione di deadlock.

Il DBMS periodicamente verifica se c'è un ciclo, e se c'è forza l'ABORT di una delle varie T in deadlock. Nasce qui il problema dello scegliere una transazione rispetto ad un'altra, e vi sono diversi principi sui quali basarsi:

- ABORT di una T **casuale**
- ABORT della T che ha il LOCK di **più risorse**
- ABORT della T che è coinvolta in **più cicli**
- ABORT della T che ha il LOCK di **meno risorse** (il costo del ROLLBACK è minore).

C'è sempre il rischio di **starvation** ovvero che una transazione non venga mai eseguita (si pensi ad una transazione che viene ogni volta scelta per essere abortita). Questo problema è quindi risolto dall'**ageing**, ovvero ad ogni transazione viene data un "eta" che ha un peso nella scelta di quale transazione abortire.

Questa soluzione è molto costosa.

### 6.2) Seconda soluzione: Timeout

Ogni transazione ha un "counter" che tiene il tempo di quanto la transazione sia stata in WAIT. Quando raggiunge un certo valore, si fa per ipotesi che la transazione sia coinvolta in un deadlock, e quindi viene abortita.

La scelta del tempo di timeout è discriminante: un tempo troppo basso porterebbe all'ABORT di transazioni non in deadlock, ma un tempo troppo alto manterebbe le transazioni in deadlock più a lungo del necessario.

### 6.3) Terza soluzione: Timestamp

La soluzione che viene utilizzata realmente nei DBMS si basa sul timestamp.

Tale soluzione è preventiva al deadlock e questo è un vantaggio non da poco.

Poniamoci in due situazioni differenti (che sono le uniche possibili).

- 1)  $T_i$  è la prima transazione a partire (timestamp più vecchio) ed accumula i suoi lock.  $T_j$  (timestamp più recente) richiede dei lock che possiede al momento  $T_i$ :  $T_j$  si mette quindi in waiting (*è quello che abbiamo sempre fatto finora*).
- 2)  $T_i$  è la prima transazione a partire (timestamp più vecchio).  $T_j$  (timestamp più recente) richiede dei lock che gli vengono concessi.  $T_i$  richiede quindi dei lock che possiede  $T_j$ :  $T_j$  subisce un ABORT, libera quindi i suoi LOCK affinché  $T_i$  possa accaparrarseli.

Questo sistema, che potremmo simpaticamente definire "basato sul nonnismo" ha molti vantaggi poiché risolve anche il problema della starvation, infatti  $T_j$  sarà messa in waiting e poi verrà normalmente eseguita appena  $T_i$  avrà completato le sue operazioni.

## 7) Riflessioni finali

### 7.1) Granularità del lock

Abbiamo sempre scritto  $r(X)$ , ma cosa è  $X$ ? La granularità del lock ce lo dice, ovvero può essere una tupla, piuttosto che un attributo di una tupla (molto costoso) oppure può avere granularità multipla.

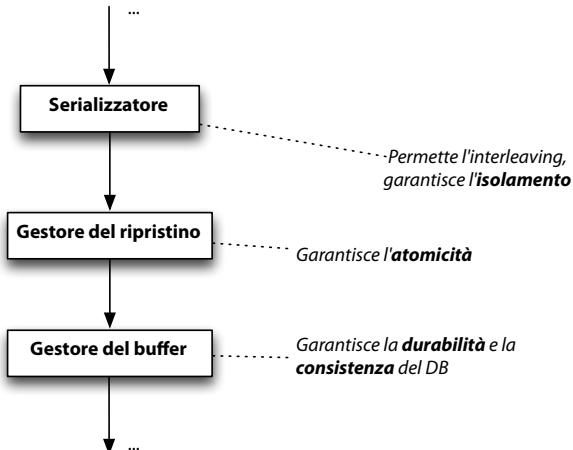
Se ad esempio abbiamo un'operazione effettuata su un indice allora usiamo una granularità di pagina, se invece stiamo eseguendo transazioni di applicazioni esterne allora una buona granularità è la tupla.

Notiamo che non esiste solo la tecnica dei lock, esiste anche un'altra tecnica basata sul timestamp (che non ha nulla a che fare con quello usato nel deadlock).

## Parte II: Crash Recovery

### 1) Contestualizzazione del crash recovery

Riprendiamo una parte schema del DBMS:



Come vediamo, l'elemento preposto al ripristino è il **gestore del ripristino**.

Ricordiamo che il buffer è soggetto ad una politica, **steal/no-steal**.

Ma c'è anche un'altra politica da considerarsi, ovvero la **flush/no-flush**.

La politica di **steal** prevede che quando non c'è spazio all'interno del buffer per una pagina desiderata da una transazione, ne venga killata una (e venga quindi ricoppiata sulla periferica se modificata) al suo posto venga messa la pagina desiderata.

La politica di **no-steal** prevede che quando non c'è spazio all'interno del buffer per una pagina desiderata da una transazione, tale transazione venga messa in WAIT.

La politica di **flush** prevede che quando una transazione viene completata, tutte le pagine coinvolte dalla transazione (ovvero modificate) vengano ricopiate in memoria secondaria.

La politica di **no-flush** prevede che quando una transazione viene completata, tutte le pagine coinvolte dalla transazione (ovvero modificate) vengono lasciate lì dove sono, e non le ricopia. Questa tecnica risparmia accessi in memoria se quelle pagine dovessero essere riutilizzate in seguito.

Date queste quattro politiche e dato il fatto che il serializzatore ed il gestore del ripristino lavorano "a braccetto" è facile intuire che ci siano quattro approcci differenti al problema per tutte le combinazioni di politiche adottabili.

Si noti che un crash di sistema può essere inteso come **crash della periferica** (l'hard disk si fonde ad esempio) oppure come **crash del sistema centrale** (va via la luce ad esempio). Per il momento poniamoci nel secondo caso.

### 2) Le quattro tipologie di crash recovery (crash del sistema)

Vediamo ora separatamente ogni caso (accoppiate di politiche). Ma prima un po' di introduzione.

#### 2.1) Il file di log

Si noti che è necessario l'utilizzo di un **file di log** che tenga traccia dei vari cambiamenti (quindi solo delle write). Il file di log è una raccolta sequenziale delle attività di modifica sul database.

Il log contiene diverse operazioni, fra cui:

- **<T_i, start>** (la transazione inizia)
- **<T_i, X, BS(X), AS(X)>** (X è l'oggetto che viene modificato, possiamo pensarla come un RID. BS è il Before State ovvero il contenuto di X prima della modifica, mentre l'AS è l'After State ovvero il contenuto di X in seguito ad una modifica. A seconda delle politiche potremmo non avere BS, AS o entrambe.)
- **<T_i, ABORT>** (la transazione T_i è abortita)
- **<T_i, COMMIT>** (la transazione T_i è in commit)

Il file di log usa inoltre i checkpoint, ma si rimanda la spiegazione al paragrafo 2.3, dopo aver visto una recovery.

## 2.2) UNDO / REDO

Vediamo due piccoli algoritmi che verranno usati dagli algoritmi per il crash recovery (se ne capirà meglio dopo l'uso):

### UNDO( $T_1, \dots, T_k$ )

L'UNDO( $T_1, \dots, T_k$ ) è una operazione che riporta il database allo stato che aveva prima che le transizioni  $T_1, \dots, T_k$  venissero eseguite. In sostanza impongo il BS di ogni quadrupla che coinvolge  $T_1, \dots, T_k$ . L'UNDO è utilizzato in seguito ad una richiesta di ROLLBACK.

```

LER := ∅
* Scansione del file di log in avanti *
Per ogni ( $T_i, X, BS(X), AS(X)$ )  $\wedge T_i \in \{T_1, \dots, T_k\}$  {
    if(  $X \notin LER$  ) {
        FORCE su periferica (X, BS(X));
        LER := LER + X;
    }
}

```

In parole, creo una lista di elementi già rifatti (LER) inizialmente vuoto. Dopodiché, scandisco il file di log dall'alto verso il basso e per ogni quadrupla incontrata laddove  $T_i$  sia tra le  $T_1, \dots, T_k$  di cui si vuole fare la UNDO, allora se  $X$  non appartiene ancora agli elementi già rifatti LER si forza il vecchio stato di  $X$  su periferica (si sovrascrive) e poi si aggiorna LER.

Questo algoritmo è molto ottimizzato poiché partendo dall'inizio del file di log recupera ogni  $X$  al primo cambiamento effettuato dalla  $T_i, \dots, T_k$ , e quindi ristabilisce direttamente lo stato prima che tutta la serie di azioni compiute da  $T_i, \dots, T_k$  avvenissero.

### REDO( $T_1, \dots, T_k$ )

IL REDO( $T_1, \dots, T_k$ ) è una operazione che riporta il database allo stato che aveva in seguito all'esecuzione delle transizioni  $T_1, \dots, T_k$ . In sostanza impongo l'AS di ogni quadrupla che coinvolge  $T_1, \dots, T_k$ .

```

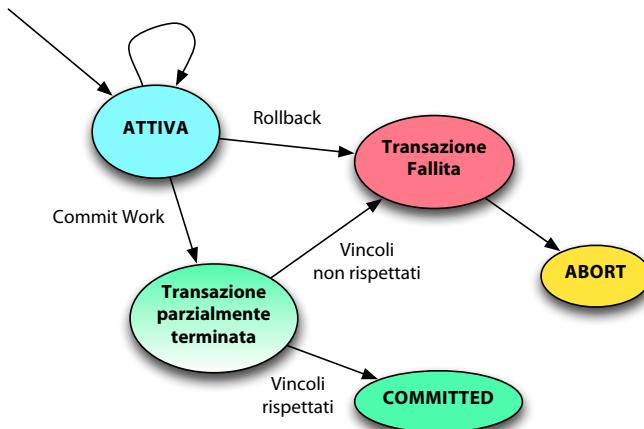
LER := ∅
* Scansione del file di log a ritroso*
Per ogni ( $T_i, X, BS(X), AS(X)$ )  $\wedge T_i \in \{T_1, \dots, T_k\}$  {
    if(  $X \notin LER$  ) {
        FORCE su periferica (X, AS(X));
        LER := LER + X;
    }
}

```

In parole, creo una lista di elementi già rifatti (LER) inizialmente vuoto. Dopodiché, scandisco il file di log dal basso verso l'alto e per ogni quadrupla incontrata laddove  $T_i$  sia tra le  $T_1, \dots, T_k$  di cui si vuole fare la REDO, allora se  $X$  non appartiene ancora agli elementi già rifatti LER si forza il nuovo stato di  $X$  su periferica (si sovrascrive) e poi si aggiorna LER.

Questo algoritmo è molto ottimizzato poiché partendo dalla fine del file di log recupera ogni  $X$  all'ultimo cambiamento effettuato dalla  $T_i, \dots, T_k$ , e quindi ristabilisce direttamente lo stato finale senza dover ripetere tutta la serie di azioni compiute da  $T_i, \dots, T_k$ .

Prima di proseguire, per poter seguire attentamente i vari casi, si riporta l'automa a stati finiti che rappresenta la vita di una transazione generica.



### 2.3) Caso uno: steal / no-flush

Vediamo prima come viene usato il file di log in questo caso, e poi vediamo il funzionamento della crash recovery.

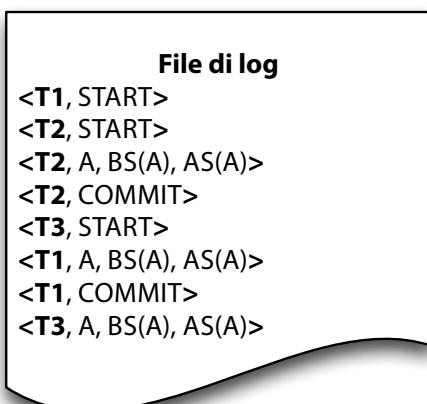
#### Uso del file di log

- * Si scrive sul file di log **<Ti, start>**
- * Ogni scrittura in colonna sul file di log la quadrupla **<Ti, X, BS(X), AS(X)>**
- * Alla fine della transazione vi sono solamente due casi possibili:
  - ❖ Se per qualche motivo  $T_i$  dovesse essere fallita, allora dovremmo fare un ROLLBACK. Ma dato che la politica adottata è **steal**, qualcuna delle pagine modificate da  $T_i$  potrebbero essere già state ricopiate in memoria secondaria. Allora effettua una **UNDO(Ti)** e poi scrivo sul file di log **<Ti, ABORT>**.
  - ✓ Se  $T_i$  chiede il commit, entro nello stato di **parzialmente terminato** e devo verificare che i vincoli di integrità referenziale siano rispettati, e quindi:
    - ✓ Se così è, allora si effettua una **FORCE LOG** (si intende che il file di LOG viene ricopiato in memoria secondaria, per sicurezza!) e **<Ti, COMMIT>** viene aggiunto al file di log.
    - ❖ Se non lo è,  $T_i$  è fallita (si torna quindi allo stato sopra e si eseguono le operazioni indicate)

#### Sistema del crash recovery

Useremo questa crash recovery come "modello" da cui partire, poiché negli altri casi avremo un comportamento simile. Inoltre, useremo questo primo esempio per spiegare il concetto di **checkpoint** all'interno del file di log.

Consideriamo questo esempio di file di log (dopo l'ultima operazione riportata immaginiamo di avere il crash):



* Creiamo le due liste:

- **LC** (lista che contiene le transazioni in commit al crash)
- **LA** (lista che contiene le transazioni attive al crash)

In questo caso  $LC = \{ T_1, T_2 \}$   $LA = \{ T_3 \}$

* Eseguiamo poi le operazioni:

- **UNDO(LA)**
- **REDO(LC)**

Perché queste due operazioni?

Tramite la UNDO riportiamo tutti i BS delle vecchie transazioni (Before State) che non sono terminate in maniera completa, ovvero che non hanno avuto il commit.

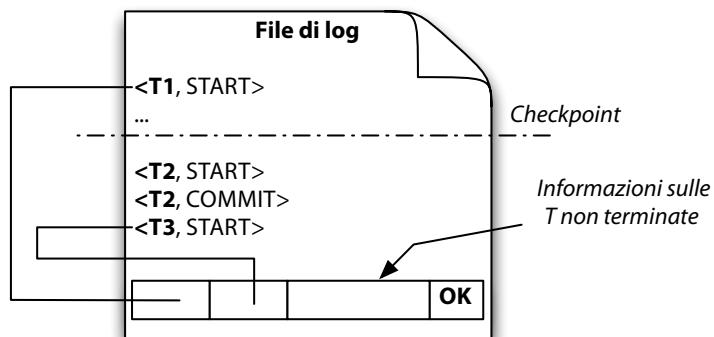
Tramite la REDO invece ci assicuriamo che le transazioni in commit vengano effettivamente riportate sul DB (ovvero venga copiato l'AS (After State) su DB), questo poiché adottando una politica no-flush non siamo certi del fatto che siano state ricopiate le pagine coinvolte.

Alla fine abbiamo un risultato che è paragonabile all'esecuzione di T₂T₁ (serializzato) mentre T₃ dovrà essere ripetuta dall'inizio poiché non è stata eseguita del tutto.

Notiamo un enorme **problema di efficienza**. Infatti, le operazioni di UNDO e REDO sarebbero mastodontiche se pensiamo che un file di log può contenere anche centinaia di migliaia di righe. Per questo si adotta la tecnica del **checkpoint**.

#### 2.4) Il checkpoint nei file di log

Solamente ora che abbiamo visto un primo esempio di recovery possiamo finalmente parlare del **checkpoint**. Il checkpoint è un modo empirico per gestire i costi di recovery che altrimenti sarebbero proibitivi. Ogni 15/20 minuti viene aggiunto al file di log un record particolare. Durante questa operazione l'intero evolversi del sistema si blocca (non vengono accettate nuove transazioni e quelle in corso non proseguono) per poter ottenere in pratica una "fotografia". Il record sopracitato ci darà informazioni importanti, ma vediamolo meglio in maniera visiva:

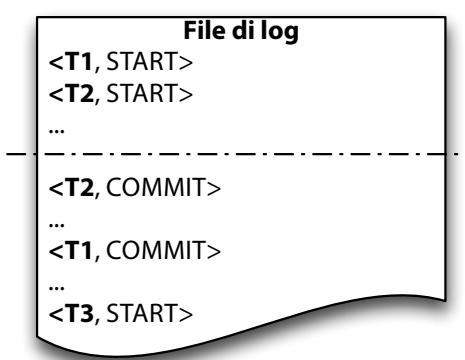


Il record che vediamo a fondo file contiene tutti i puntatori alle transazioni non ancora terminate più altre informazioni. Le azioni compiute sono due:

- 1) **FORCE** sul DB di tutte le transazioni in commit avvenute prima dell'ultimo checkpoint.
- 2) Delle transazioni non in commit vengono conservati i puntatori e altre informazioni.

La recovery usando il checkpoint non cambia quasi di nulla, la differenza sta nella creazioni delle liste. Infatti,

- **LC** (lista che contiene le transazioni in commit al crash dopo il checkpoint)
- **LA** (lista che contiene le transazioni non attive dopo il checkpoint)



Quindi, per esempio, qui abbiamo:  
 $LC = \{ T_1, T_2 \}$   
 $LA = \{ T_3 \}$

Il grande vantaggio è che le liste sono **più corte** e quindi le REDO e UNDO sono più leggere.

## 2.5) Caso due: no-steal / no-flush

Vediamo prima come viene usato il file di log in questo caso, e poi vediamo il funzionamento della crash recovery.

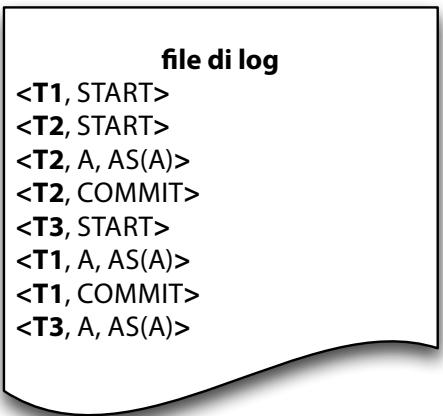
### Uso del file di log

- * Si scrive sul file di log **<Ti, start>**
- * Ogni scrittura in colonna sul file di log la tripla **<Ti, X, AS(X)>** (il BS risulta inutile poiché se  $T_i$  risultasse fallita, essendo in politica no-steal tutte le modifiche si troverebbero sul buffer, senza rischio di aver intaccato il DB). Pertanto non useremo la UNDO, e quindi non ci serve il BS.
- * Alla fine della transazione vi sono solamente due casi possibili:
  - ❖ Se per qualche motivo  $T_i$  dovesse essere fallita, essendo in politica no-steal e no-flush nessuna delle pagine del buffer è stata mai ricopiata in memoria stabile quindi è sufficiente rilasciare il buffer sulle pagine che  $T_i$  ha modificato! Dopodiché scrivo sul file di log **<Ti, ABORT>**.
  - ✓ Se  $T_i$  chiede il commit, entro nello stato di parzialmente terminato e devo quindi verificare che i vincoli di integrità referenziale siano rispettati, e quindi:
    - ✓ Se così è, allora si effettua una **FORCE LOG** (si intende che il file di LOG viene ricopiatato in memoria secondaria, per sicurezza!) e **<Ti, COMMIT>** viene aggiunto al file di log.
    - ❖ Se non lo è,  $T_i$  è fallita (si torna quindi allo stato sopra e si eseguono le operazioni indicate)

### Sistema del crash recovery

Si ricorda che il file di log utilizza checkpoint (non disegnato per semplicità)

Consideriamo questo esempio di file di log (dopo l'ultima operazione riportata immaginiamo di avere il crash):



* Creiamo le due liste:

- **LC** (lista che contiene le transazioni in commit al crash)
- **LA** (lista che contiene le transazioni attive al crash)

In questo caso  $LC = \{ T_1, T_2 \}$   $LA = \{ T_3 \}$

* Eseguiamo poi le operazioni:

- **REDO(LC)**

Come detto sopra le modifiche effettuate dalle transazioni in LA non sono state spostate dal buffer alla memoria, quindi semplicemente svuotiamo il buffer e non ci curiamo di queste transazioni. Non sappiamo niente però delle transazioni in commit, potrebbero essere state ricopiate (non certamente, poiché siamo in politica no-flush). Comunque, per certificare la consistenza è necessario l'uso di una REDO che copri in memoria stabile tutte le operazioni che avevamo dato in commit.

Si noti che questa politica ha un grosso difetto, il buffer si riempie in fretta (non si svuota mai!).

## 2.6) Caso tre: steal / flush

Vediamo prima come viene usato il file di log in questo caso, e poi vediamo il funzionamento della crash recovery.

### Uso del file di log

- * Si scrive sul file di log **<Ti, start>**
- * Ogni scrittura incolonna sul file di log la tripla **<Ti, X, BS(X)>** (l'AS risulta inutile poiché se  $T_i$  risultasse in commit, essendo in politica flush tutte le modifiche verrebbero ricopiate dal buffer alla memoria secondaria, senza rischio di perdere tali cambiamenti. Pertanto non useremo la REDO, e quindi non ci serve l'AS).
- * Alla fine della transazione vi sono solamente due casi possibili:
  - ❖ Se per qualche motivo  $T_i$  dovesse essere fallita, allora dovremmo fare un ROLLBACK. Ma dato che la politica adottata è steal, qualcuna delle pagine modificate da  $T_i$  potrebbero essere già state ricopiate in memoria secondaria. Allora effettuo una **UNDO(Ti)** e poi scrivo sul file di log **<Ti, ABORT>**.
  - ✓ Se  $T_i$  chiede il commit, entro nello stato di parzialmente terminato e devo quindi verificare che i vincoli di integrità referenziale siano rispettati, e quindi:
    - ✓ Se così è, allora è necessario implementare il sistema della flush. Si effettua una **FORCE LOG** (si intende che il file di LOG viene ricopiato in memoria secondaria, per sicurezza!), una **FORCE pagine usate da Ti su DB** e **<Ti, COMMIT>** viene aggiunto al file di log.
    - ❖ Se non lo è,  $T_i$  è fallita (si torna quindi allo stato sopra e si eseguono le operazioni indicate)

### Sistema del crash recovery

Si ricorda che il file di log utilizza checkpoint (non disegnato per semplicità)

Consideriamo questo esempio di file di log (dopo l'ultima operazione riportata immaginiamo di avere il crash):

<b>file di log</b>
<b>&lt;T1, START&gt;</b>
<b>&lt;T2, START&gt;</b>
<b>&lt;T2, A, BS(A)&gt;</b>
<b>&lt;T2, COMMIT&gt;</b>
<b>&lt;T3, START&gt;</b>
<b>&lt;T1, A, BS(A)&gt;</b>
<b>&lt;T1, COMMIT&gt;</b>
<b>&lt;T3, A, BS(A)&gt;</b>

* Creiamo le due liste:

- **LC** (lista che contiene le transazioni in commit al crash)
- **LA** (lista che contiene le transazioni attive al crash)

In questo caso  $LC = \{ T_1, T_2 \}$   $LA = \{ T_3 \}$

* Eseguiamo poi le operazioni:

- **UNDO(LA)**

Come detto sopra le modifiche effettuate dalle transazioni in LC sono state certamente portate in memoria per via dell'utilizzo di una politica flush. Ma dato che adottiamo anche una politica steal, alcune pagine del buffer potrebbero essere state ricopiate in memoria secondaria, quindi adottiamo la solita UNDO(LA).

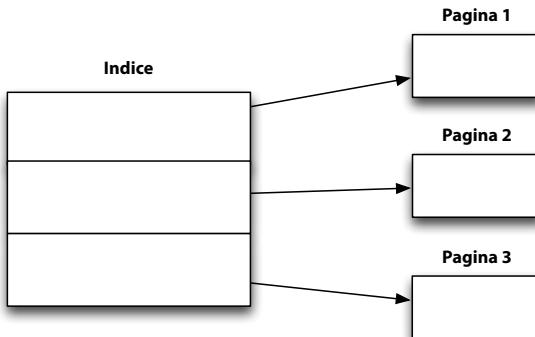
## 2.7) Caso quattro: no-steal / flush

Questo caso è molto banale: non si adopera il file di log. Sarebbe inutile poiché ogni transazione in commit è salvata sul DB mentre ogni azione attiva sta lavorando solo sul buffer: in caso di crash recovery sarà sufficiente svuotare il buffer.

Se ci dovesse essere il crash mentre si ricopiano le pagine coinvolte da una transazione in commit, i programmatore si sono messi al sicuro usando piccoli file di log durante l'operazione.

### 3) Crash recovery con pagine ombra (DB multimediali)

Giusto per sfizio, guardiamo questa tecnica che è implementata su database che trattano file multimediali di grandi dimensioni. Abbiamo un indice in memoria stabile che punta alle pagine usate da una certa transazione.

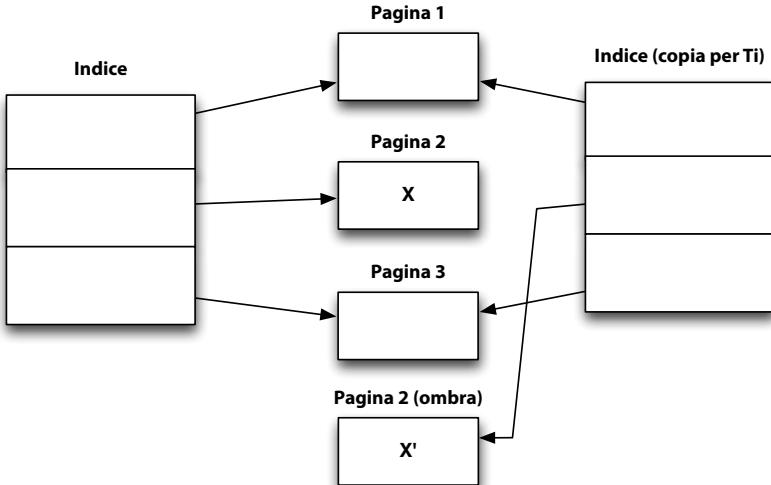


Quando una transazione  $T_i$  necessita di quelle pagine, gli viene fornita una copia dell'indice.

Quando  $T_i$  chiede un LX su una pagina ( $X$ ) (quindi avremo una granularità di lock pari a livello delle pagine) quella pagina viene contrassegnata e **ricopiata in memoria principale**.

Dopodiché viene aggiornato sull'indice posseduto da  $T_i$  il puntatore alla pagina ( $X$  nel disegno) alla pagina copia ( $X'$ ).

Nessuno potrà modificare la pagina 2 poiché è in LX a  $T_i$ . Una volta che  $T_i$  avrà dato il commit, l'indice copia di  $T_i$  verrà dapprima **ricopiato in memoria stabile** (dove c'è anche l'altro) e poi verrà **sostituito** all'indice attuale.



Questo procedimento è dal punto di vista dell'idea molto interessante, poiché, in concomitanza della serializzazione fa sì che le modifiche vengano portate su memoria stabile in tutta sicurezza. Si noti che l'indice viene prima copiato e poi sostituito poiché in caso di crash abbiamo ancora le due versioni e possiamo recuperare tutti i dati.

#### 4) Dump Restore (crash della periferica)

Come detto sopra, un crash può essere derivante da un crollo totale del sistema oppure dal crollo di una periferica. Una delle moltissime tecniche adottate per ovviare alla seconda tipologia di crash è il **dump restore**. Questa tecnica si basa su un log completo di tutte le transazioni.

Intorno alle due di notte, in un momento di bassa attività, il DBMS ferma tutte le transazioni e ricopia il file di log in memoria stabile.

In caso di crollo della periferica, oltre all'ovvia sostituzione sarà necessario ripescare il file di log (detto quindi dump) ed effettuare una mastodontica **REDO(LC)** che conterà moltissime transazioni!

Infatti, questa operazione può durare anche ore.

#### 5) Rivisitazione della tabella iniziale

Nel capitolo 1, paragrafo 6 abbiamo visto una tabella che mostrava i vantaggi dell'uso di un database rispetto all'archiviazione di un file. Ora, con tutte le informazioni acquisite possiamo rileggerla aggiungendo quali sono i sistemi adottati per garantire i famosi dieci vantaggi.

Archivio	DBMS	Vantaggi	Come?
Applicazioni ed archivi generano ridondanza	Dati condivisi	<b>Integrazione</b>	<i>Schema logico</i>
Stesse informazioni aggiornate in tempi differenti	Aggiornamento unico e quindi simultaneo	<b>Consistenza</b>	<i>Normalizzazione</i>
Ogni applicazione deve garantire l'integrità	Controllo centralizzato dell'integrità	<b>Integrità</b>	<i>Vincoli interni e vincoli globali</i>
Variazioni di tipo/formato devono essere riportate in ogni applicazione	Descrizione centralizzata: solo i programmi interessati subiscono variazioni	<b>Indipendenza logica</b>	<i>Architettura ANSI / SPARC (le viste di SQL)</i>
Modalità di accesso ed organizzazione fisica	Accesso indipendente	<b>Indipendenza fisica</b>	<i>Gli indici</i>
Sinonimia nel riferire gli stessi dati	Solo nomi definiti nello schema (unico)	<b>Standardizzazione</b>	<i>Schema concettuale</i>
Accesso solo via applicativo	Accesso eventualmente interattivo	<b>Facilità d'uso</b>	<i>Il linguaggio SQL</i>
Sicurezza gestita dalle applicazioni	Meccanismi unici di sicurezza	<b>Sicurezza</b>	<i>GRANT tramite SQL</i>
Procedure di protezione da malfunzionamenti	Meccanismi specifici di recupero	<b>Affidabilità</b>	<i>Crash recovery</i>
Uso esclusivo da ogni applicazione	Accesso simultaneo	<b>Concorrenza</b>	<i>Lock a due fasi</i>