Basi di Dati 1

Marco Casu

Contents

1 Introduzione **4**

2 Il Modello Relazionale **6**
2.1 Notazione con Indice . . . 6
2.2 Rappresentazione come Funzioni . . . 7
2.3 Integrità dei Dati . . . 8
2.4 Le Chiavi . . . 9
2.4.1 La Chiave Esterna . . . 10
2.5 Le Dipendenze Funzionali . . . 10

3 Algebra relazionale **11**
3.0.1 Proiezione . . . 11
3.0.2 Selezione . . . 11
3.0.3 Unione . . . 12
3.0.4 Differenza . . . 12
3.0.5 Intersezione . . . 13
3.0.6 Prodotto Cartesiano . . . 13
3.0.7 Join . . . 14
3.0.8 Θ-Join . . . 14
3.0.9 Condizioni Negative . . . 14
3.1 Quantificazione Universale . . . 15
3.2 Esempi di Esercizi . . . 15
3.2.1 Esempio 1 . . . 15
3.2.2 Esempio 2 . . . 16

4 Teoria della Normalizzazione **17**
4.1 Definizioni Formali . . . 18
4.2 Gli Assiomi di Armstrong . . . 19
4.2.1 Chiusura di un Insieme di Attributi . . . 20
4.3 Teorema Fondamentale : $F^+ = F^A$ . . . 20
4.3.1 Dimostrazione . . . 20
4.4 La Terza Forma Normale . . . 22
4.4.1 Dipendenze Parziali e Transitive . . . 22
4.4.2 Decomposizione in più Schemi . . . 23
4.4.3 Forma Normale Boyce-Codd . . . 23
4.5 Calcolo per la Chiusura di X . . . 24
4.5.1 L'Algoritmo 1 (`Compute` X_F^+ `from` F) . . . 24
4.5.2 La Ricerca delle Chiavi . . . 26
4.6 Preservare le Dipendenze Funzionali . . . 27
4.6.1 Controllo dell'Unione . . . 28
4.6.2 L'Algoritmo 2 (`Compute` X_G^+ `from` F) . . . 29
4.6.3 Sommario del Procedimento Completo . . . 31
4.6.4 L'Algoritmo 3 (`Check if` $F \subseteq G^+$) . . . 31
4.7 Join Senza Perdita . . . 32
4.7.1 L'Algoritmo per il loseless JOIN (`check if` ρ `has a loseless JOIN`) 33
4.8 Trovare una Buona Decomposizione . . . 34
4.8.1 Copertura Minimale . . . 34

4.8.2 L'Algoritmo per la decomposizione (`find` ρ) 34

5 Organizzazione Fisica **36**
5.1 Specifiche Fisiche del Disco 36
5.2 Modelli di Organizzazione 37
5.2.1 Heap File Organization 38
5.2.2 Sequential File Organization 38
5.2.3 Random File Organization (Hashing) 38
5.2.4 Indexed Sequential File Organization 39
5.2.5 Tree-Structured Index 40

6 Controllo della Concorrenza **42**
6.1 Schedule e Transazioni 42
6.1.1 Errori sui Dati 43
6.2 Protocolli e Lock a 2 Stati 44
6.2.1 Definizione di Lock 44
6.2.2 Grafo di Serializzazione 46
6.2.3 Protocollo a Due Fasi 46
6.3 Lock a 3 Stati 47
6.3.1 Nuovo Grafo di Serializzazione 48
6.4 Deadlock e Livelock 48
6.4.1 Stallo delle Transazioni e Grafo di Attesa 49
6.4.2 Situazione di Starvation 49
6.5 Roll-back e Punti di Commit 49
6.5.1 protocollo a due fasi stretto 50

7 Complementi **51**
7.1 Formulario sull'Organizzazione Fisica 51
7.1.1 Hashing (Bucket) 51
7.1.2 B-Tree 51
7.1.3 ISAM 51

1 Introduzione

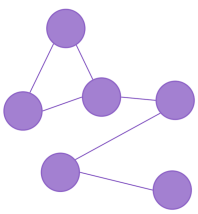
L'informazione memorizzata nei sistemi elettronici può essere di due tipi, **strutturata** e **non strutturata**. In questo corso ci occuperemo dell'informazion strutturata, ossia composta da oggetti matematici ben definiti. Un **sistema informativo** connette e contiene le informazioni, alla quale si può accedere da diversi componenti. Si prenda come esempio di sistema informativo l'archivio fisico dei documenti (ossia i dati) di una azienda, ad esso, possono accedere i vari reparti, come la sezione commercio o le risorse umane, ognuno ha a disposizione l'accesso ad un determinato sotto-insieme di dati (permessi). Prima di adoperare i sistemi informativi, ogni reparto aveva il suo personale archivio, ciò faceva si che molti dei dati fossero duplicati e presenti per più reparti, causando un elevata **ridondanza** di dati. Inoltre, due dati da 2 archivi diversi potrebbero dipendere tra loro. È quindi importante mantenere l'informazione **centralizzata**. I dati vanno organizzati, gestiti, e regolamentati da permessi di accesso a secondo dell'utente che vuole accedervi.

Un sistema informativo è composto dai seguenti componenti :

- Database (DB)
- Database Managment System (DBSM) - Ossia il software per il mantenimento dei dati
- Application Software
- Computer Hardware - La memoria nella quale è contenuto

È importante mantenere una visione astratta del modello di sistema informativo, che sia indipendente dall'hardware in uso. Fissiamo un modello che utilizzi tipi di dati che non possano variare nel tempo, strutturati in maniera completa, è necessario pensare ad un metodo formale per organizzare i dati. Vogliamo farlo in un ambiente condiviso, preoccupandoci quindi di quali e quanti dati vengono condivisi condivisi con i singoli utenti, che avranno permessi diversi e potranno accedere a dati diversi (ad esempio, uno studente può accedere ai suoi esami registrati, ma non a quelli degl altri). Inolte bisogna anche amministrare i metodi con cui si accede ai dati, preoccupandoci della concorrenza (accedere allo stesso dato nello stesso momento). I dati vengono organizzati in maniera omogenea, esiste un tipo di dato per descrivere un entità (ad esempio uno studente), e tutte le volte che si vuole immagazinare nel sistema uno studente, bisogna ricorrere allo stesso tipo di dato. L'informazione viene rappresentata da un aggregamento di più dati *grezzi*, ossia, rispettivamente la stringa "*Maurizio Ernesti*" e l'intero "3761523746", insieme rappresentano l'informazione di un professore (nome, cognome e numero di telefono).

Il modello è il modo in cui decidiamo di organizzare e collegare i dati, esistono modelli **logici**, indipendenti dalla struttura fisica, come il **modello relazionale**, ed esistono modelli **concettuali**, ossia rappresentazioni ancora più astratte indipendenti dal modello logico, i primi modelli sono stati introdotti negli anni 60.



un esempio è il modello *Mesh*, rappresentato con un grafo, dove i nodi sono i dati (i record), e gli archi le loro relazioni.

In questo modello i collegamenti sono esplicitati fisicamente (possiamo immaginare con dei puntatori), i modelli relazionali, diversamente, hanno relazioni rappresentate implicitamente dai valori stessi che contengono.

Studenti			
Matricola	Cognome	Nome	Compleanno
276545	Gialli	Lucia	25/11/1980
176515	Rossi	Mario	13/09/1982
348191	Verdi	Andrea	04/07/1981

Esami		
Studente	Voto	Corso
348191	25	01
176515	18	02
176515	27	02

La relazione tra gli esami e gli studenti (ogni studente ha n esami registrati) è data implicitamente dalla presenza del campo *Studente* nella tabella *Esami*, che equivale al campo *Matricola* della tabella *Studenti*, il modello relazionale è basato su oggetti, classi ed attributi.
Ad esempio, se volessi sapere che voto ha preso *Mario Rossi* al corso numero 02, mi basterebbe consultare la tabella *Esame*, e controllare il campo che ha la matricola equivalente a quella di *Mario Rossi*.
Lo schema logico descrive la presenza di tutte le entità con i loro rispettivi attributi, esistono poi gli schemi esterni, ossia sotto-insiemi degli schemi, destinati a determinati tipi di utenti che ne hanno l'accesso. Anche lo schema logico completo può essere schema esterno, il super-amministratore di un sistema informativo, come schema esterno, avrà l'intero schema logico, avendo accesso a tutte le entità.
Come già detto, lo schema logico rappresenta la struttura degli oggetti/entità, ed è invariata nel tempo, ma la sua istanza, ossia gli effettivi campi delle tabelle, possono variare, ed anche rapidamente. Per gestire gli schemi si utilizzano dei veri e propri linguaggi.

- **Data Definition Language (DDL)** - Per la definizione degli schemi logici ed altre operazioni generali.
- **Data Manipulation Language (DML)** - Per interrogare lo schema logico, leggerne i valori ed eventualmente modificarli.

È largamente utilizzato il linguaggio **SQL (Structured Query Language)**, che funge sia da DDL che da DML.
Un base di dati deve essere :

- Manipolabile
- Modificabile
- Centralizzata
- Il minimo ridondante
- Sicura

I dati molto spesso devono soddisfare certi vincoli (ad esempio, ogni studente ha una sola residenza) , tali vincoli son chiamati **dipendenze funzionali**, e possono riguardare anche il dominio di certi attributi (ad esempio, il voto verbalizzato di un esame deve essere maggiore o uguale a 18 e minore o uguale a 30). I dati devono essere protetti da accessi non autorizzati, è necessaria la dichiarazione di regole di accesso.
Definiamo adesso uno specifico tipo di operazione sulle basi di dati, ossia le **transizioni**, che non sono altr che sequenze ordinate di operazioni che vanno obbligatoriamente eseguite

insieme in sequenza, con il divieto assoluto che ne venga eseguita solo una parte. Facciamo un esempio, si dia il caso che su una base di dati bancaria, si vogliano trasferire 1000 euro dal conto *C1* al conto *C2*, le seguenti operazioni definite informalmente sono :

- Cercare il conto *C1*
- Sottrarne al bilancio 1000
- Cercare il conto *C2*
- Aggiungerne al bilancio 1000

Se per errore si eseguono solo i primi 2 passi, ci si ritroveranno 1000 euro persi, sottratti al primo conto, ma non addizionati al secondo. Una transizione va quindi completamente eseguita, se non dovesse essere così, l'intera sequenza di operazioni va abortita. Inoltre, è importante notare che data la concorrenzialità, in una base di dati potrebbe accadere di accedere allo stesso dato nello stesso momento, ciò potrebbe causare errori, è quindi importante evitare di lavorare contemporaneamente su uno stesso specifico campo.

2 Il Modello Relazionale

Il modello relazionale è basato sulla relazione intesa in senso matematico, una relazione non è altro che un insieme di tuple, tutte della stessa lunghezza, con elementi appartenenti a diversi domini. Il dominio è l'insieme dei possibili valori che gli elementi delle tuple possono assumere. Se prendiamo una lista di k domini, il prodotto cartesiano di tutti i k domini è l'insieme di tuple di lunghezza k.

$$D_1 \times D_2 \times D_3.... \times D_k = \{(v_1, v_2, v_3..., v_k) | v_1 \in D_1, v_2 \in D_2, v_3 \in D_3..., v_k \in D_k\} \quad (1)$$

Per esempio, per $k = 2$ consideriamo i seguenti domini $D_1 = \{White, Black\}$ e $D_2 = \{0, 1, 2\}$, si ha che :

$$D_1 \times D_2 = \{(White, 0), (White, 1), (White, 2), (Black, 0), (Black, 1), (Black, 2)\} \quad (2)$$

È una relazione di grado 2, perchè ogni tupla ha 2 coordinate.

Teorema 1 *Il grado di una relazione è equivalente al numero di elementi di ogni tupla appartenente a tale relazione.*

In tale prodotto cartesiano è possibile costruire 2^6 possibili relazioni, ossia sotto-insiemi del prodotto cartesiano.

Teorema 2 *Una relazione è un qualsiasi sottoinsieme del prodotto cartesiano.*

2.1 Notazione con Indice

Sia r una relazione di grado k, data t una tupla appartenente alla relazione r, ed i un intero da 1 a k, con $t[i]$ si intende l'elemento alla cooridnata i-esima della tupla t.

Tabella	
Black	0
White	0

Se prendiamo t come la prima riga della tabella, si avrà che $t[1] =$ "*Black*" e $t[2] = 0$.
Ricordando la notazione tabellare, in basi di dati le intestazioni delle colonne ed il loro dominio è denominato **attributo** (ad Esempio *Name : string*). In una tabella, due attributi distinti non possono avere lo stesso nome.

2.2 Rappresentazione come Funzioni

Sia R un oggetto definito come insieme di attributi, una tupla su R, ossia un istanza di tale oggetto, può essere visto come una funzione definita su R che associa ad ogni attributo A, un valore presente nel dominio di A. Considerando ciò, presa t una tupla di R, ed A uno dei suoi attributi, indichiamo con $t(A)$, il valore (ossia l'istanza) di quell'attributo preso dalla funzione t sulla variabile A.
Ad esempio, la relazione R ha la tupla $t_1 = (Paolo, Rossi, 2, 26.5)$, considerando t_1 come una funzione, essa associa ad ogni attributo A, un elemento del suo dominio :

$$f : (Nome, Cognome, Esami, Media) \rightarrow string \cup string \cup int \cup real \tag{3}$$

Da qui si ha $\quad t_1(Nome) = Paolo \quad t_1(Cognome) = Rossi \quad t_1(Media) = 26.5$
Lo schema logico, è il dominio di tale funzione, l'istanza sono le tuple, le istanze sono insiemi di tuple, ossia una relazione.

Teorema 3 *L'istanza è un sotto-insieme di tuple*

Ogni riga della tabella è una tupla distinta, ed ogni colonna corrisponde al dominio. Possiamo quindi rappresentare un oggetto della base di dati con la seguente notazione :

$$R(A_1, A_2, A_3..., A_k) \tag{4}$$

Qui R è una relazione dello schema, quindi uno schema di basi di dati non è altro che un insieme di relazioni $(R_1, R_2, R_3..., R_k)$ (invarianti nel tempo) per le quali, ognuna di esse possiede un istanza (variante nel tempo). Da qui in poi, utilizzeremo R per denominare le relazioni, ed r per le loro istanze.
Riprendendo la notazione con indice vista in precedenza 2.1, si può utilizzare puttosto che un indice intero, l'intestazione dell'attributo per il quale si voglia leggere l'istanza.

Luoghi		
Città	Regione	Popolozione
Roma	Lazio	3000000
Milano	Lombardia	1500000
Genova	Liguria	150000

Data t_1 la prima riga dell'istanza, si ha t_1[Città]="*Roma*". È possibile anche farlo con sottoinsiemi di attributi, ossia t_1[Regione,Popolazione]=("*Lazio*",3000000). Non è importante l'ordine degli attributi come argomenti, come "risultato" riceviamo una sotto-tupla della tupla t_1, detta *restrizione*. Abbiamo visto come le istanze delle tabelle non sono altro che insiemi di tuple, esistono tante possibili istanze quanto la cardinalità del prodotto cartesiano dei domini.
Può succedere in certi casi, che nell'istanza di una relazione, sia presente una riga in cui un attributo è **sconosciuto**, per rappresentare tale campo nelle basi di dati, si utilizzi il valore polimorfico[1] $NULL$, utilizzato per riempire gli spazi vuoti, ad esempio in una tabella contente

[1]Appartente a tutti i domini

i dati degli utenti iscritti ad un sito, è possibile che alcuni utenti abbiano omesso il numero di telefono, per loro il campo avrà valore $NULL$, ossia sconosciuto. Si ricordi che $NULL$ è diverso da 0.

Luoghi		
Matricola	Nome	Cellulare
1039	Luca	3475746371
4316	Giorgio	$NULL$
1499	Sandro	3857482845

2.3 Integrità dei Dati

La presenza di un valore $NULL$ può causare alcuni errori, vedremo come sono presenti alcuni attributi, le quali istanze devono per forza essere dichiarate e non sconosciute. Esistono però diversi tipi di errori, come dei campi identificativi duplicati o valori fuori dominio.

Studenti		
Matricola	Nome	Media
1039	Luca	28
4316	Giorgio	33
1039	Sandro	22

- Errore 1 - Nella prima e nella terza riga sono presenti due studenti con la stessa matricola. Il campo matricola identifica ogni singolo e distinto studente, e non può essere duplicato.
- Errore 2 - Uno studente ha come media dei voti 33, è impossibile dato che i voti sono compresi tra 18 e 30, è quindi un valore fuori dominio.

Tali errori vengono definiti problemi di **integrità dei dati**, per mantenere tale integrità è necessario che le istanze delle relazioni soddisfino delle determinate proprietà dette **vincoli**. Vedremo che esistono :

- Vincoli di chiave
- Vincoli di dominio
- Vincoli funzionali
- Vincoli di esistenza

Impiegati					
Codice Impiegato	Nome	Cognome	Ruolo	Assunzione	Dipartimento
01	Luca	Rossi	Analista	1785	01
02	Mario	Verdi	Amministratore	1980	02
02	Giorgio	Neri	Ricercatore	1985	05

Dipartimenti	
Numero	Nome
01	Managment
02	Amministrazione

Vediamo come nello schema logico appena mostrato ci sono diversi vincoli da definire che non sono rispettati. Ad esempio, va definito il vincolo di dominio per cui il valore "Assunzione"≥ 1980, ossia tale valore deve essere strettamente maggiore di una certa data (possibile data di nascita dell'azienda). Si noti che non si sta rispettando un vincolo di chiave, dato che il "Codice Impiegato" presente nella seconda e nella terza riga lo stesso valore, essendo esso l'attributo identificativo, non deve essere duplicato ("Codice Impiegato" $UNIQUE$). Un altro errore meno evidente, è che alla terza riga della tabella "Impiegati", è presente un campo dipartimento con codice 05, tale campo dovrebbe collegare quella riga con il suo rispettivo dipartimento presente su un altra tabella, ma notiamo che nella tabella "Dipartimenti", non è presente alcuna riga con "Numero" identificativo 05. Un altro vincolo noto è il vincolo di esistenza, che impone ad un certo attributo di non accettare valore $NULL$.

Un altra importante distinzione da fare tra vincoli è di suddividerli in :

- **Intra-relazionali** - Vincoli definiti e da soddisfare all'interno della singola tabella (Ad esempio, un vincolo di dominio per il quale un valore deve essere sufficentemente grande).
- **Inter-relazionali** - Vincoli soddisfatti dai collegamenti di più tabelle (Un chiaro esempio è il sopra-citato errore sulla tabella del dipartimento).

2.4 Le Chiavi

In un istanza r di una relazione R, per ogni tupla è necessario che vi sia un attributo (o un insieme di attributi) X che la identifichi e distingua dalle altre tuple (che quindi non sia mai duplicato). Tale attributo/insieme di attributi è detto **chiave**, un chiaro esempio può essere il campo "Matricola" all'interno di un'ipotetica tabella studenti. Vediamo una definizione più formale.

Teorema 4
Punto 1 *- Per ogni istanza della relazione R, non esistono due tuple t_1, t_2 che hanno gli stessi valori per tutti i singoli attributi, preso un insieme di attributi X che funge da chiave, vale sempre $t_1[X] \neq t_2[X]$.*

$$sia\ X \in R(A_1, A_2..., A_k) | \forall t_1, t_2 \in r\ se\ t_1[X] = t_2[X] \implies t_1 = t_2 \tag{5}$$

Punto 2 *- Inoltre, non esistono sotto-insiemi di X che soddisfino la condizione sopra-citata.*

$$\forall X' \subseteq X, \exists t_1, t_2 \in r\ tale\ che\ t_1[X'] = t_2[X'] \wedge t_1 \neq t_2 \tag{6}$$

Approfondendo il punto 2, se esiste una relazione R con chiave $X = (A_1, A_2, A_3)$, è chiaro che, nella sua istanza r, non esisteranno due righe con gli stessi valori assegnati X, quindi, preso il sotto-insieme $X' = (A_1, A_2)$, se esistono due tuple t_1, t_2 tali che $t_1[X'] = t_2[X']$, per forza di cose esse non saranno la stessa tupla in quanto, per il punto 1, differiranno per l'attributo A_3. È importante per una relazione che abbia una chiave significativa basata sull'informazione che rappresenta (Una relazione che rappresenta degli studenti, non può avere come chiave il campo del nome, dato che potrebbero esserci 2 studenti con lo stesso nome, bensì si prediliga la matricola), si sceglie quindi una **chiave primaria**, ovviamente con vincolo di esistenza.

2.4.1 La Chiave Esterna

Può esistere inoltre un attributto nelle relazioni detto **chiave esterna** o **foreign key**, essa identifica all'interno di una relazione, un attributo associato ad un altra relazione (un altra tabella), identificandone la chiave primaria.

Studenti			
Matricola	Cognome	Nome	Compleanno
276545	Gialli	Lucia	25/11/1980
176515	Rossi	Mario	13/09/1982
348191	Verdi	Andrea	04/07/1981

Esami		
Studente	Voto	Corso
348191	25	01
176515	18	02
176515	27	02

Nella tabella "*Esami*", l'attributo "*Studente*" è una chiave esterna che identifica e collega tale tabella con la relazione "*Studenti*", tramite la sua chiave primaria "*Matricola*", Dato che ad ogni esame è associato uno studente che l'ha sostenuto. Il vincolo di integrità inter-relazionale precedentemente citato impone che per ogni valore presente su un attributo di una chiave esterna, esista il suo corrispettivo campo con tale valore nella relazione alla quale fa riferimento. Si ricordi che tale vincolo non è violato dalla presenza di un valore $NULL$.

Multe			
Codice	Data	Ufficiale	Targa
4312	01/12/1988	001	AA123AA
1351	12/04/1989	*NULL*	KA194AR
9572	10/10/1990	002	ND193MF

Ufficiali		
Codice	Nome	Cognome
001	Giancarlo	Pozzi
002	Sara	Tua
003	Nicola	Canti

Può quindi esistere un istanza dove la chiave esterna ha valore $NULL$, conseguentemente non avrà nessun riferimento nella relazione alla quale è collegata.

2.5 Le Dipendenze Funzionali

Come si possono definire facilmente i vincoli di integrità dei dati 2.3 visti nei paragrafi precedenti? Nelle basi di dati si utilizzano le note **dipendenze funzionali**, ossia un insieme di attributi che dipende da un altro insieme di attributi all'interno dello stesso schema. Tale definizione può sembrare poco esplicativa, ma è di vitale importanza in questo paragrafo che lo studente abbia ben saldo in mente il concetto di dipendenza funzionale, in quanto centrale nel corso di *Basi di Dati*. Formalmente, una dipendenza funzionale stabilisce un collegamento semantico tra due distinti insiemi di attributi X e Y appartenenti allo stesso schema. Si scrive:

$$X \to Y \tag{7}$$

e si legge : "X **determina** Y", stabilendone dei vincoli di integrità.

Teorema 5 *Sia r un istanza della relazione R, la dipendenza funzionale $X \to Y$ è soddisfatta se :*

- *Sia X che Y sono due sotto-insieme distinti di R.*
- *Le tuple di r che sono identiche per X, sono anche identiche per Y.*

$$\forall t_1, t_2 \in R \text{ se } t_1[X] = t_2[X] \implies t_1[Y] = t_2[Y] \tag{8}$$

3 Algebra relazionale

Tramite **l'algebra relazionale** possiamo fare interrogazioni alla nostra base di dati per ottenere informazioni su una porzione di essa, le *query* sono scritte in linguaggio SQL, e vengono tradotte nel linguaggio formale e procedurale dell'algera relazionale. L'algebra relazionale fornisce degli operatori che lavorano sulle istanze delle nostre relazioni, vi sono 4 tipi di operatori :

- Operatori di rimozione sulle singole relazioni
- Operatori insiemistici
- Operatori che combinano tuple da relazioni diverse
- Operatore di rinomina

Vediamo nel dettaglio tutti gli operatori che abbiamo a disposizione :

3.0.1 Proiezione

L'operatore di proiezione, indicato con π, esegue un *taglio verticale* su una tabella, selezionando un sotto-insieme degli attributi, creando una tabella come quella iniziale, ma con esclusivamente le istanze degli attributi selezionati.

$$\pi_{A_1,A_2,..A_k}(R)$$

Esempio :

Sia :

Customer		
Code	Name	Town
001	Giancarlo	Roma
002	Sara	Cagliari
003	Nicola	Roma

ho che $\pi_{Name}(Customer) =$

Roma
Cagliari

3.0.2 Selezione

La selezione, indicata con σ, esegue un *taglio orizzontale* sulla tabella, ossia, seleziona tutte le righe che soddisfano un determinato vincolo C, tale vincolo è un sepressione booleana della forma $A\Theta B$, dove $\Theta \in \{<, >, =, \leq, \geq\}$. La condizione ovviamente per esser valida, ha bisogno che A e B abbiano lo stesso dominio (non posso comparare un intero con una stringa).

$$\sigma_C(R)$$

Esempio :

Sia :

Customer		
Code	Name	Town
001	Giancarlo	Roma
002	Sara	Cagliari
003	Nicola	Roma

ho che :

$\sigma_{Town="Roma"}(Customer) =$

Code	Name	Town
001	Giancarlo	Roma
003	Nicola	Roma

Ovviamente posso utilizzare gli operatori logici per eseguire richieste multiple :

$\sigma_{Town=\text{"}Roma\text{"} \wedge Code=001}(Customer) =$

Code	Name	Town
001	Giancarlo	Roma

Vale la proprietà : $\sigma_C(\sigma_{C'}(R)) = \sigma_{C \wedge C'}(R)$

3.0.3 Unione

L'operazione di unione fra due relazioni, denominata con il simbolo $\cup$, come nel senso insiemistico, crea una nuova istanza della relazione, contenente le tuple da entrambe le relazioni. Non si possono unire due relazioni qualsiasi, è necessario che esse siano **union-compatibili**, ossia che abbiano lo stesso numero di attributi, e che gli attributi corrispondenti abbiano lo stesso dominio.

Siano :

Insegnanti	
Code	Name
001	Giancarlo
002	Sara
003	Nicola

Admin	
Code	Name
001	Luca
004	Andrea
005	Carlo

Si ha che :

$Insegnanti \cup Admin =$

Code	Name
001	Giancarlo
002	Sara
003	Nicola
004	Andrea
005	Carlo

Si noti come la riga | 001 | Luca | non appare nell'unione, dato che condivide la stessa chiave con la riga | 001 | Giancarlo | , quindi, quando si uniscono due relazioni, bisogna fare attenzione che due righe da esse non condividano la stessa chiave, altrimenti si avrà una perdita di informazioni. Si noti che, se una delle due tabelle avesse presentato un attributo di troppo, esso sarebbe potuto essere stato rimosso con una proiezione, rendendole comunque union-compatibili.

3.0.4 Differenza

Come per l'unione, è necessario che le due relazioni sulla quale voglio applicare la differenza siano union-compatibili. Si indica con il simbolo $-$, ed il risultato fra due relazioni, conterrà le tuple del primo operando, che non stanno anche nel secondo operando.

Siano :

Insegnanti	
Code	Name
001	Giancarlo
002	Sara

Admin	
Code	Name
002	Sara

Si ha che :

$Insegnanti - Admin =$

Code	Name
001	Giancarlo

La differenza non è commutativa! Infatti, in questo caso $Admin - Insegnanti = \emptyset$

3.0.5 Intersezione

Anch'essa richiede che i due operandi siano union-compatibili. Si indica con $\cap$, ed il risultato conterrà esclusivamente le tuple che fanno parte sia della prima che della seconda relazione. (si noti che $A \cap B = A - (A - B)$).

Siano :

Insegnanti	
Code	Name
001	Giancarlo
002	Sara

Admin	
Code	Name
002	Sara
004	Andrea

Si ha che :

$Insegnanti \cap Admin =$

Code	Name
002	Sara

Vediamo adesso gli operatori della terza categoria, che creano **relazioni multiple** combinando tabelle eterogenee.

3.0.6 Prodotto Cartesiano

Il prodotto cartesiano è indicato con il simbolo $\times$, e rappresenta tutte le possibili combinazioni di righe fra gli elementi del primo operando con gli elementi del secondo, tale operazione risulta parecchio dispendiosa, e potrebbe associare elementi che non hanno nessuna correlazione sensata, ad esempio :

Siano :

Customer	
Code	Name
001	Giancarlo
002	Sara

Order	
CO	Customer
AX00	001
AX01	002

Si ha che :

$Customer \times Order =$

Code	Name	CO	Customer
001	Giancarlo	AX00	001
001	Giancarlo	AX01	002
002	Sara	AX00	001
002	Sara	AX01	002

Logicamente però, stiamo associando a dei clienti, degli ordini che non gli appartengono (il campo *Customer* di *Order* è la foreign key che identifica il campo *Code* della tabella *Customer*). Quindi, se volessimo una tabella con ogni cliente e gli ordini ad esso associati, combineremo anche una selezione del tipo :

$\sigma_{Customer.Code=Order.Customer}(Customer \times Order) =$

Code	Name	CO	Customer
001	Giancarlo	AX00	001
002	Sara	AX01	002

È chiaro che, per tabelle con numerosi elementi, il prodotto cartesiano da computare risulta parecchio dispendioso, dato che, se R_1 ha n elementi, ed R_2 ha m elementi, il prodotto cartesiano $R_1 \times R_2$ avrà $n \cdot m$ elementi, e quando si opera su basi di dati con milioni di righe, tale operazione deve essere evitata quando possibile, per questo esiste un'operazione simile al prodotto cartesiano, che seleziona automaticamente le righe da combinare secondo parametri ben precisi.

3.0.7 Join

L'operatore join, indicato dal simbolo $\bowtie$, seleziona le tuple del prodotto cartesiano che soddisfano una precisa condizione C, ossia che gli attributi con lo stesso nome, debbano avere anche lo stesso valore, è quindi importante che si denominino in maniera precisa gli attributi di relazioni che si vogliono combinare con il join.

$$R_1 \bowtie R_2 = \sigma_C(R_1 \times R_2) \tag{9}$$

$$C = R_1.A_1 = R_2.A_1 \wedge R_1.A_2 = R_2.A_2 \wedge ... \wedge R_1.A_k = R_2.A_k \tag{10}$$

dove $A_1, A_2..., A_k$ sono gli attributi condivisi con lo stesso nome fra R_1 e R_2. La tabella risultante, non presenterà due volte gli attributi da R_1 e R_2 con lo stesso nome, ma li unirà in un attributo solo.
Se dovesse capitare che le due relazioni non hanno alcun attributo in comune, il risultato sarà il prodotto cartesiano.

Siano :

Customer	
Code	Name
001	Giancarlo
002	Sara
003	Lucia

Order		
CO	Code	Item
AX00	001	Glue
AX01	002	Bricks
AX02	001	Shoes

Si ha che :

$Customer \bowtie Order =$

Customer.Code	Name	CO	Item
001	Giancarlo	AX00	Glue
001	Giancarlo	AX02	Shoes
002	Sara	AX01	Bricks

3.0.8 Θ-Join

Il Θ-Join (Che si pronuncia "theta join"), seleziona le tuple risultanti dal prodotto cartesiano, che soddisfino la condizione $A\Theta B$ dove $\Theta \in \{<, >, =, \leq, ge\}$ ed A, B sono attributi rispettivamente della prima, e seconda relazione (il dominio di A deve essere lo stesso del dominio di B).

$$R_1 \bowtie_{A\Theta B} R_2 = \sigma_{A\Theta B}(R_1 \times R_2) \tag{11}$$

3.0.9 Condizioni Negative

È possibile utilizzare il simbolo $\neg C$ per intendere che si vogliono selezionare tutte le tuple che non soddisfino la condizione C.

Sia :

Customer		
Code	Name	Town
001	Giancarlo	Roma
002	Sara	Cagliari
003	Nicola	Roma
004	Gianluca	Venezia

ho che :

$\sigma_{\neg(Town="Roma")}(Customer) =$

002	Sara	Cagliari
004	Gianluca	Venezia

3.1 Quantificazione Universale

Per ora tutti gli operatori che abbiamo visto, implicano una quantificazione esistenziale, quando seleziono da una relazione le righe che soddisfano una condizione C, seleziona le righe anche se vi è un altro elemento che non soddisfa tale condizione.

Se seleziono tutti i clienti che hanno ordini con prezzo superiore a 100 euro, selezionerà tutti i clieni che hanno **almeno** un ordine superiore a 100 euro, non solo quelli che hanno **esclusivamente** ordini superiori a 100 euro.
Vogliamo scrivere query che implichino la quantificazione universale e non esistenziale, ossia, che tale condizione valga "**per ogni**", e ciò è equivalente a "**non c'e ne sono tali che**":

- I clienti che hanno solo ordini superiori a 100 euro = I clienti che non hanno ordini inferiori o uguali a 100 euro

Siano :

Customer	
Code	Name
001	Giancarlo
002	Sara
003	Lucia

Order		
CO	Code	Price
AX00	001	90
AX01	002	120
AX02	001	200

Vogliamo elencare nome e codice dei clienti che hanno fatto ordini esclusivamente con prezzo superiore a 100 euro, quindi, *Giancarlo* non sarà incluso, dato che ha fatto si un ordine da 200 euro, ma anche uno da 90 euro. La nostra query Q sarà :

$$Q = Customer - \pi_{Code,Name}(\sigma_{Price \leq 100}(Customer \bowtie Order)) \tag{12}$$

Quindi, prendere tutti coloro che hanno fatto solo ordini superiori a 100 euro, equivale a prendere il totale, e sottrarne coloro che hanno fatto almeno un ordine inferiore o uguale a 100 euro.

$Q =$

Code	Name
002	Sara

3.2 Esempi di Esercizi

3.2.1 Esempio 1

Ci sono situazioni in cui dobbiamo combinare una relazione con se stessa per ottenere paia di tuple della stessa tabella, vediamo un esempio, si osservino le seguenti relazioni :

Impiegati			
Name	Cod	Salario	CodSup
Rossi	C1	100	C3
Pirlo	C2	200	C3
Bianchi	C3	500	NULL
Verdi	C4	200	C2
Neri	C5	150	C1
Tosi	C6	100	C1

Dove *CodSup* identifica il *Cod* della riga che ogni impiegato ha come supervisore. Ad esempio, Rossi ha come supervisore Bianchi, e Neri ha come supervisore Rossi. Vogliamo trovare tutti gli impiegati che hanno lo stipendio superiore o uguale al loro supervisore. Per procedere possiamo combinare tramite il prodotto cartesiano la tabella con se stessa, prima però rinominando ogni attributo aggiungendoci una C davanti, per identificare una tabella che è copia dell'altra:

$$ImpiegatiC = \rho_{Name,Cod,Salario,CodSup \to CName,CCod,CSalario,CCodSup}(Impiegati) \tag{13}$$

Adesso procediamo col combinare tali tabelle, però selezionando esclusivamente le tuple che hanno il *CodSup* ed il *CCod* identici, in modo che avremo una lista dei dipendenti con a destra il loro superiore.

$$\sigma_{CodSup=CCod}(Impiegati \times ImpiegatiC) \tag{14}$$

Name	Cod	Salario	CodSup	CName	CCod	CSalario	CCodSup
Rossi	C1	100	C3	Bianchi	C3	500	NULL
Pirlo	C2	200	C3	Bianchi	C3	500	NULL
Verdi	C4	200	C2	Pirlo	C2	200	C3
Neri	C5	150	C1	Rossi	C1	100	C3
Tosi	C6	100	C1	Rossi	C1	100	C3

Fatto ciò adesso, ci basta selezionare quelli che hanno il salario superiore o uguale a quello del proprio supervisore, ossia $Salario \geq CSalario$:

$$\sigma_{(CodSup=CCod) \wedge (Salario \geq CSalario)}(Impiegati \times ImpiegatiC) \tag{15}$$

Name	Cod	Salario	CodSup	CName	CCod	CSalario	CCodSup
Verdi	C4	200	C2	Pirlo	C2	200	C3
Neri	C5	150	C1	Rossi	C1	100	C3
Tosi	C6	100	C1	Rossi	C1	100	C3

3.2.2 Esempio 2

Vediamo adesso un esempio di un altro tipo, si consideri sempre la stessa relazione di prima :

Imp			
Name	Cod	Salario	CodSup
Rossi	C1	100	C3
Pirlo	C2	200	C3
Bianchi	C3	500	NULL
Verdi	C4	200	C2
Neri	C5	150	C1
Tosi	C6	100	C1

Adesso vogliamo trovare l'impiegato che ha il salario più alto, ma come possiamo fare? Procederemo con il comparare ogni impiegato con tutti gli altri, seleziondo esclusivamente quelli che hanno lo stipendio inferiore all'impiegato con la quale sono stati comparati (selezioniamo esclusivamente il codice impiegato):

$$Imp2 = Imp \tag{16}$$

$$\pi_{Imp.cod}(Imp \bowtie_{Imp.Salario < Imp2.Salario} Imp2) \tag{17}$$

Ottengo la tabella :

Impiegati.Cod
C1
C2
C4
C5
C6

Che rappresenta tutti gli impiegati che hanno qualcuno con lo stipendio superiore al loro. Quindi, chi non è presente in questa tabella, sarà l'impiegato con lo stipendio più alto. Prendiamo allora tutti gli impiegati e sottraiamo ad essi la nostra tabella.

$$\pi_{Cod}(Imp) - \pi_{Imp.cod}(Imp \bowtie_{Imp.Salario<Imp2.Salario} Imp2) =$$

$$= \begin{array}{|c|}\hline C1 \\ C2 \\ C3 \\ C4 \\ C5 \\ C6 \\ \hline \end{array} - \begin{array}{|c|}\hline C1 \\ C2 \\ C4 \\ C5 \\ C6 \\ \hline \end{array} = \boxed{C3}$$

Quindi l'impiegato di codice $C3$, ossia | Bianchi | C3 | 500 | NULL | è colui con lo stipendio più alto.

4 Teoria della Normalizzazione

L'obbiettivo è capire come creare uno schema in maniera corretta, immaginiamo di dover creare una base di dati per memorizzare le informazioni relative agli studenti di un corso di laurea triennale, ed i relativi esami sostenuti.

Matricola	SurN	Name	BirthD	City	Prov	ExCode	ExName	Doc	Date	Grade
01	Rossi	Mario	...	Roma	Roma	10	Physics	Pippo	...	28
02	Bianchi	Paolo	...	Tolfa	Roma	10	Physics	Pippo	...	26
01	Rossi	Mario	...	Roma	Roma	20	Chemistry	Pluto	...	27

È estremamente sbagliato salvare tutti i dati in una sola relazione, per ogni esame, devo salvare ogni volta tutti i dati di uno studente, creando ridondanza e spreco di memoria, inoltre se uno studente non ha sostenuto alcun esame, non apparirà nell'archivio, ed un esame che non è stato sostenuto da nessuno risulterà inesistente. È corretto suddividere tale relazione in 3 diverse tabelle :

Studenti					
Matricola	SurN	Name	BirthD	City	Prov
01	Rossi	Mario	...	Roma	Roma
02	Bianchi	Paolo	...	Tolfa	Roma
01	Rossi	Mario	...	Roma	Roma

Corso		
ExCode	ExName	Doc
10	Physics	Pippo
20	Chemistry	Pluto

Esami			
Matricola	ExCode	Data	Grade
01	10	...	28
01	20	...	27
02	10	...	26

4.1 Definizioni Formali

Uno **schema relazionale**, che si denota con R è un insieme di attributi del tipo:

$$R = \{A_1, A_2, A_3..., A_k\}$$

Solitamente un sotto insieme di attributi di R viene indicato con le lettre X oppure Y, e l'unione si può denotare $XY \equiv X \cup Y$. Data una relazione R, una **tupla** r è una funzione che associa ad ogni attributo, un valore appartenente al suo dominio :

$$R = \{nome, cognome\}\ r[nome] = marco, r[cognome] = rossi$$

Preso $X \subset R$, diciamo che due tuple r_1, r_2 **coincidono** su X se :

$$\forall A \in X, r_1[A] = r_2[A]$$

Corso		
Nome	Cognome	Voto
Paolo	Rossi	29
Mario	Rossi	29

Le due tuple coincidono su {Cognome,Voto}

Un **istanza** su uno schema relazionale R è l'insieme di tutte le tuple su R. Una **dipendenza funzionale** su R è una coppia ordinata X, Y di sotto-insiemi di R, ossia di attributi di R, che si denota con $X \to Y$, ed indica che :

$$\text{per ogni coppia di tuple } r_1, r_2, \text{ vale che } r_1[X] = r_2[X] \implies r_1[Y] = r_2[Y]$$

E si dice X *determina* Y, dove X è detto determinante ed Y dipendente. Le dipendenze funzionali esprimono dei *vincoli*. Uno schema R può avere un **insieme di dipendenze funzionali** denotato con F :

$$F = \{A \to B, D \to B..., G \to H\}$$

se un istanza soddisfa ogni dipendenza di F, ossia tutte le dipendenze funzionali, è detta istanza **legale**. Le dipendenze godono di una relazione di *transitività*, ossia, se $A \to B$ e $B \to C$, allora $A \to C$, non è però necessario aggiungere quest'ultima all'insieme F, dato che è implicita, e vogliamo mantenere F il più piccolo possibile. Ci sono quindi dipendenze funzionali che sono soddisfatte per ogni istanza della relazione, che però non è necessario includere in F, un altro esempio :

$$\text{se } taxCode \to name, surname$$
$$\text{è ovvio che } taxCode \to name \text{ e } taxCode \to surname$$

Ma queste due ultime non saranno incluse in F. Tali dipendenze sono dette **banali**, ad esempio, se $Y \subset X$, se $A \to X$ ovviamente $A \to Y$. Il numero delle dipendenze banali è *esponenziale* :

$$X \to Y \iff \forall A \in Y, X \to A$$

Tutte le dipendenze banali che non è necessario includere in F, si trovano in un insieme più grande detto **chiusura di** F, che si denota con F^+, contenente tutte le dipendenze funzionali soddisfatte da un'istanza, è chiaro che $F \subseteq F^+$.
Passiamo alla definizione di **chiave**. Un sotto-insieme di attributi K è detto chiave se :

- $K \to R \in F^+$
- $\nexists K' \subset K | K' \to R \in F^+$

Detto in maniera meno formale, non esisteranno due tuple di una relazione con gli stessi valori per le chiavi. Nei linguaggi come *SQL*, fra gli attributi della chiave se ne definisce uno in particolare detto **chiave primaria**,che non può assumere valore `NULL`.

4.2 Gli Assiomi di Armstrong

Dato uno schema R, un insieme di dipendenze funzionali F, ed un istanza legale r, se tale istanza soddisfa una dipendenza si scrive $r\ SAT\ X \to Y$, abbiamo poi definito la chiusura di F come :

$$F^+ = \{X \to Y | \forall r \text{ legale } r\ SAT\ X \to Y\}$$

Se voglio verificare che K sia una chiave, non mi è possibile eseguire manualmente il controllo su tutto F^+, che abbiamo visto essere di dimensioni esponenziali rispetto ad F, inoltre ci è impossibile controllare ogni istanza, dato che esse sono potenzialmente infinite. Definiamo adesso un nuovo insieme, a partire da F, che ci permette di esaminare lo schema relazionale più facilmente.

F^A è un insieme di dipendenze funzionali su R, ottenuto dai seguenti *assiomi di Armstrong* :

- Se $X \to Y \in F$ allora $X \to Y \in F^A$
- Se $Y \subseteq X \in R$ allora $X \to Y \in F^A$ (**Riflessività**)
- Se $X \to Y \in F^A$ allora $XZ \to YZ \in F^A\ \forall Z \in R$ (**Aumento**)
- Se $X \to Y \in F^A$ e Se $Y \to Z \in F^A$ allora Se $X \to Z \in F^A$ (**Transitività**)

Vediamo alcune proprietà che seguno dagli assiomi :

- $X \to Y \in F^A \land X \to Z \in F^A \implies X \to YZ \in F^A$ (Unione)
- $X \to Y \in F^A \land Z \subseteq Y \implies X \to Z \in F^A$ (Decomposizione)
- $X \to Y \in F^A \land WY \to Z \in F^A \implies XW \to Z \in F^A$ (Pseudotransitività)

Dimostrazioni delle proprietà : (Unione) Se ho $X \to Y \in F^A$, applicando *l'aumento*, utilizzando X, ottengo $X \to XY \in F^A$, Avendo poi $X \to Z \in F^A$, ed applicando *l'aumento* con Y, si ottiene $XY \to YZ \in F^A$. Ottengo quindi :

$$\begin{cases} X \to XY \in F^A \\ XY \to YZ \in F^A \end{cases} \implies \boxed{\text{per transitività}} \implies X \to YZ \in F^A \tag{18}$$

(Decomposizione) Se $Z \subseteq Y$, allora per *riflessività*, si ha $Y \to Z \in F^A$, dal momento che $X \to Y \in F^A$, si ha :

$$\begin{cases} X \to Y \in F^A \\ Y \to Z \in F^A \end{cases} \implies \boxed{\text{per transitività}} \implies X \to Z \in F^A \tag{19}$$

(Pseudotransitività) Se $X \to Y \in F^A$. applicando *l'aumento* con W si ottiene $WX \to WY \in F^A$, e ne consegue che :

$$\begin{cases} WX \to WY \in F^A \\ WY \to Z \in F^A \end{cases} \implies \boxed{\text{per transitività}} \implies WX \to Z \in F^A \tag{20}$$

Osservazione : Siano $A_1, A_2..., A_n$ un insieme di attributi, se $\forall i, X \to A_i \in F^A$, per unione, $X \to A_1A_2...A_n \in F^A$, analogamente, data tale condizione, per decomposizione : $\forall i, X \to A_i \in F^A$, è chiaro che : decomposizione $\iff$ unione .

4.2.1 Chiusura di un Insieme di Attributi

Sia R uno schema relazionale, ed F l'insieme delle sue dipendenze funzionali. Preso un insieme di attributi $X \subseteq R$, definiamo come **la chiusura di** X su F, che denotiamo con X_F^+, l'insieme di tutti gli attributi *determinati* da X dopo aver applicato gli assiomi di Armstrong.

$$X_F^+ = \{A | X \to A \in F^A\}$$

Banalmente, per riflessività, $X \subseteq X_F^+$.

Vediamo un importante *Lemma* : Sia R uno schema relazionale, ed F l'insieme delle sue dipendenze funzionali, vale che : $X \to Y \in F^A \iff Y \subseteq X_F^+$.
Dimostrazione : Sia $Y = A_1, A_2..., A_n$ un insieme di attributi, se $Y \subseteq X_F^+$, allora, per unione, $\forall i,\ X \to A_i \in F^A$. D'altro canto, se $X \to Y \in F^A$, per decomposizione, $\forall i,\ X \to A_i \in F^A$, quindi $\forall i,\ A_i \in F^A$, ne consegue che $Y \subseteq X_F^+$.

4.3 Teorema Fondamentale : $F^+ = F^A$

Segue uno dei teoremi più importanti del corso :

$$F^+ = F^A$$

> Partendo da F, applicando ricorsivamente gli assiomi di Armstrong, si ottiene la chiusura di F.

4.3.1 Dimostrazione

Procederemo col dimostrare tale teorema per doppia inclusione, dimostrando quindi la veridicità di $F^+ \subseteq F^A$ e $F^A \subseteq F^+$.
$\boxed{F^A \subseteq F^+}$: Si procede per induzione, come parametro da incrementare, si adopererà il numero di volte in cui applico un determinato assioma di Armstrong. Ossia, dimostreremo che, se una dipendenza è in F^+, è perchè si è applicato i volte un assioma di Armstrong su F. Si consideri quindi la seguente notazione : F_i^A è l'insieme delle dipendenze funzionali ottenute applicando i volte un assioma di Armstrong. Ne consegue ovviamente che :

$$F_0^A \subseteq F_1^A \subseteq F_2^A \subseteq ...F_n^A \subseteq F^A$$

Lo scopo è quindi dimostrare, che applicando tali assiomi un numero di volte finito, si ottiene F^+.

Caso Base $(i = 0)$: $F_0^A = F \subseteq F^+$

Passo Induttivo : La tesi è, che $F_i^A \subseteq F^+ \implies F_{i+1}^A \subseteq F^+$. Distinguiamo i 3 casi di applicazione dei 3 assiomi.

- *Caso 1* $X \to Y$ è stato ottenuto in F_{i+1}^A applicando la riflessività su $Y \subseteq X$. Se r è un istanza di R, e t_1, t_2 due tuple tali che $t_1[X] = t_2[X]$, ovviamente vale che $t_1[Y] = t_2[Y]$, quindi $X \to Y \in F^+$.

- *Caso 2* $X \to Y$ è stato ottenuto in F_{i+1}^A applicando l'aumento su una dipendenza funzionale $V \to W \in F_i^A$. Per ipotesi induttiva, $V \to W \in F^+$, vorrebbe dire che $X = VZ$ e $Y = WZ$ per qualche $Z \subseteq R$. Se r è un istanza legale, e t_1, t_2 due tuple tali che $t_1[X] = t_2[X]$, ne consegue che $t_1[V] = t_2[V]$ e $t_1[Z] = t_2[Z]$, ma per ipotesi induttiva, essendo $V \to W \in F^A$, ne consegue che $t_1[W] = t_2[W]$, e da ciò ne consegue $t_1[Z] = t_2[Z]$, quindi $t_1[Y] = t_2[Y]$, ne consegue $X \to Y \in F^+$.

- *Caso 3* $X \to Y$ è stato ottenuto in F_{i+1}^A applicando la transitività da due dipendenze $X \to Z, Z \to Y \in F_i^A$, ottanute ricorsivamente applicando gli assiomi i volte. Se r è un istanza legale, e t_1, t_2 due tuple tali che $t_1[X] = t_2[X]$, ne consegue che $t_1[Z] = t_2[Z]$, ma per ipotesi induttiva allora $t_1[Y] = t_2[Y]$, quindi $X \to Y \in F^+$.

$\boxed{F^+ \subseteq F^A}$: Procediamo con una dimostrazione per assurdo. Supponiamo che esiste una dipendenza funzionale $X \to Y \in F^+$, tale che $X \to Y \notin F^A$. Utilizzeremo una particolare istanza r di R che ci aiuterà a giungere ad una contraddizione, tale istanza è :

X_F^+			$R - X_F^+$		
A1	A2	A3	B1	B2	B3
1	1	1	1	1	1
1	1	1	0	0	0

Tale istanza ha solamente due tuple, in rosso sono indicati gli attributi che appartengono ad X_F^+, in blu, gli attributi restanti di R che non appartengono ad X_F^+. Tali due tuple, sono identiche sugli attributi di X_F^+, ma differiscono sui restanti attributi. Dimostriamo adesso due fatti :

- r **è un istanza legale** : Assumiamo che non lo sia, vuol dire che esiste almeno una dipendenza $V \to W \in F$, quindi anche in F^A, che non è soddisfatta da r, vuol dire che vi sono due tuple che sono uguali su V e differenti su W, questo implicherebbe che $V \subseteq X_F^+$ e $W \cap (R - X_F^+) \neq \emptyset$, dal momento che $V \subseteq X_F^+$, per lemma è vero che $X \to V \in F^A$, per assioma di transitività, essendo che $V \to W \in F$, si ottiene che $X \to W \in F^A$, per lemma quindi $W \subseteq X_F^+$, ma ciò è una contraddizione, dato che abbiamo detto che $W \cap (R - X_F^+) \neq \emptyset$, quindi r è legale.

- **se** $X \to Y \in F^+$**, è impossibile che** $X \to Y \notin F^A$: Supponiamo che ciò sia vero, sappiamo che r è un istanza legale, quindi se le dipendenze in F^+ sono soddisfatte da ogni istanza legale, allora r soddisfa $X \to Y$. Sappiamo che $X \subseteq X_F^+$, dato che su r ci sono 2 tuple identiche su X, vi devono essere anche su Y, quindi $Y \subseteq X_F^+$, per lemma, ne consegue che $X \to Y \in F^A$, ma abbiamo detto inizialmente che $X \to Y \notin F^A$, quindi si ha una contraddizione.

Quindi, è necessariamente vero che $X \to Y \in F^+ \implies X \to Y \in F^A$. Quindi $F^A = F^+$. ■

4.4 La Terza Forma Normale

All'inizio di questo paragrafo 4, si è parlato di alcune caratteristiche che una base di dati deve avere per essere il più possibile "corretta", in modo da evitare la ridondanza o altre anomalie durante le interrogazioni. Formalizziamo l'insieme delle caratteristiche in quella che si chiama **terza forma normale (3NF)**. Ricordiamo prima una definizione :

> Una dipendenza $X \to A$ è detta *non banale* se $A \notin X$.

Definiamo adesso rigorosamente :

> Una base di dati è in *Terza Forma Normale* se tutte le dipendenze non banali sono della forma $K \to X$, dove si verifica *almeno una* condizione fra :
>
> - K contiene una chiave (è superchiave).
> - X è attributo di una chiave/è contenuto in una chiave (è primo).

Si può verificare che una base di dati sia in 3NF controllando solo le dipendenze non banali ottenute decomponendo F.

Esempio : Si consideri la seguente base di dati :

$$R = \{A, B, C, D\}\ F = \{A \to B, B \to CD\}$$

Sappiamo che la chiave è A, infatti si nota che $A \to R \in F^+$ (più avanti definiremo come trovare in maniera algoritmica la chiave di uno schema relazionale). Controlliamo che R sia in 3NF, decomponendo F, otteniamo :

- $A \to B$ (che era già in F)
- $B \to C$
- $B \to D$

Per queste ultime due, B non è una superchiave, e sia C che D non sono elementi che appartengono ad una chiave, quindi tale schema *non* è in terza forma normale.

4.4.1 Dipendenze Parziali e Transitive

Sia R uno schema relazionale ed F il suo insieme delle dipendenze funzionali, si ha che :

- $X \to A \in F^+ | A \notin X$ è una **dipendenza parziale** se A non è primo ed X invece è contenuto nella chiave, si dice che A **dipende parzialmente** dalla chiave.
- $X \to A \in F^+ | A \notin X$ è una **dipendenza transitiva** se A non è primo e per ogni chiave K su R, si ha che X non è propriamente contenuto in K e $K - X = \emptyset$. Si dice che A **dipende transitivamente** da K(chiave) se $K \to X \in F^+$ e $X \to A \in F^+$ dove X non è chiave ed A non è parte della chiave.

Dato uno schema R, è in terza forma normale se e solo se **non** presenta *Dipendenze Parziali* o *Transitive* (analogamente, non presenta attributi che dipendono parzialmente o transitivamente da una chiave).

4.4.2 Decomposizione in più Schemi

Vogliamo saper creare delle basi di dati che siano in terza forma normale, ossia che, ognuna delle sue relazioni sia in 3NF. Avendo uno schema che viola tale forma, è sempre possibile *decomporlo*, creando più relazioni che a loro volta non violino la 3NF.

> **Attenzione!** non tutte le decomposizioni in 3NF di uno schema che inizialmente non era in 3NF sono necessariamente corrette, esistono diverse scomposizioni in 3NF, ma alcune violano le dipendenze funzionali che vi erano in origine.

Vediamo un *Esempio*, ho un insieme di dipendenze $F = \{A \to B, B \to C\}$, dove A è la chiave. tale insieme viola la terza forma normale ($B \to C$ è una dipendenza transitiva), vediamo due ipotesi di scomposizione :

$$(1) \begin{cases} R1 = \{A, B\} \text{ con } F = \{A \to B\} \\ R2 = \{B, C\} \text{ con } F = \{B \to C\} \end{cases}$$

$$(2) \begin{cases} R1 = \{A, B\} \text{ con } F = \{A \to B\} \\ R2 = \{A, C\} \text{ con } F = \{A \to C\} \end{cases}$$

Presa una qualsiasi istanza legale della relazione originale, se venisse decomposta in (1) o (2), il `JOIN` delle due relazioni scomposte dovrebbe creare una nuova istanza che sia ancora legale come quella originale, ciò non è soddisfatto dalla scomposizione (2), vediamo un esempio :

R1	
A	B
a1	b1
a2	b2

$\bowtie$

R2	
A	C
a1	c1
a2	c2

$=$

R1⋈R2		
A	B	C
a1	b1	c1
a2	b1	c2

Anche se sia $R1$ che $R2$ sono in 3NF, il risultato del loro `JOIN` non soddisfa la dipendenza funzionale $B \to C$.
Si noti come, anche se prendessimo in considerazione la decomposizione (1), essa soddisfa le dipendenze funzionali iniziali, ma non andrebbe comunque bene perché il `JOIN` genererebbe delle tuple che non vi erano in origine (perdita di informazioni).

In conclusione, la decomposizione di uno schema non in 3NF in più schemi in 3NF deve :

- Preservare le dipendenze funzionali dello schema originale.
- Poter ricostruire lo schema originale tramite un `JOIN` con ogni istanza legale senza aggiungere informazioni aggiuntive.

4.4.3 Forma Normale Boyce-Codd

La terza forma normale non è la forma normale più restrittiva, ce ne sono altre, inclusa la seguente :

> Una relazione è in **forma normale Boyce-Codd**, se, per ogni singola dipendenza funzionale, il determinante è una superchiave (o ovviamente, chiave).

$$\text{Sia } K \text{ la chiave : } \forall A \to B \in F^+, K \subseteq A$$

Una relazione che è in forma normale Boyce-Codd è anche in terza forma normale, non è vero però il contrario, se una relazione è in terza forma normale, non è necessariamente in forma normale Boyce-Codd. Non è sempre possibile decomporre uno schema in sotto-schemi in forma normale Boyce-Codd, è però sempre possibile farlo in sotto-schemi in terza forma normale, per questo, considereremo quest'ultima durante il corso.

4.5 Calcolo per la Chiusura di X

Nel capitolo precedente 4.2.1 abbiamo dato la definizione dell'insieme X^+, ossia la chiusura di X (Un insieme di attributi dello schema relazionale), l'insieme contenente tutti gli attributi A tali che $X \to A \in F^A$. Vedremo adesso un *algoritmo* in grado di calcolare, a partire da un attributi, la sua chiusura, ciò risulterà utile anche in funzione di stabilire quali sono le chiavi di una relazione.

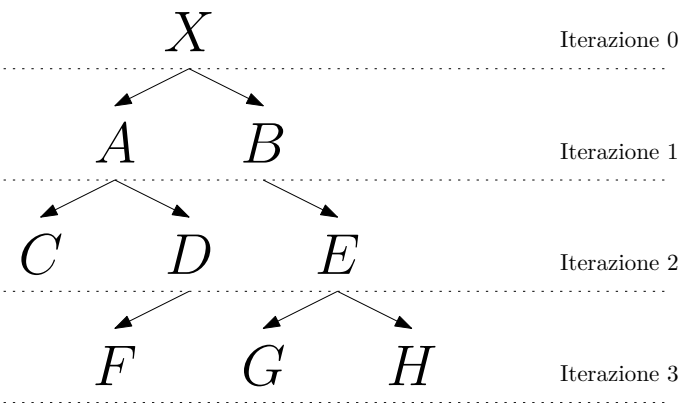
4.5.1 L'Algoritmo 1 (`Compute` X_F^+ `from` F)

`Input` : Lo schema R, l'insieme delle dipendenze funzionali F, ed $X \subseteq R$.
`Output` : Denotato con Z_f, sarà la chiusura di X.

```
begin {
    Z = X
    S = {A|∃Y → V ∈ F, Y ⊆ Z ∧ A ∈ V}
    while S ⊈ Z {
        Z = Z ∪ S
        S = {A|∃Y → V ∈ F, Y ⊆ Z ∧ A ∈ V}
    }
return Z
}
```

Ad ogni passo iterativo, ad S aggiungiamo tutti gli attributi determinati da X, direttamente o per transitività, si può immaginare un albero di "determinanza" che ad ogni passo cresce verso il basso.

In questo esempio, X determina A e B, quindi all'iterazione 1, A e B verranno inclusi in Z, per transitività, gli attributi C e D determinati da A, ed E determinato da B, sono anche determinati da X, quindi all'iterazione 2 verranno inclusi in Z, stesso procedimento per l'iterazione 3.



Denoteremo con $Z_0, Z_1, Z_2 \ldots, Z_n$ la sequenza dei valori che assume Z, dove Z_i è il valore di Z all'i-esima iterazione, stessa cosa per la variabile S con la sequenza $S_0, S_1, S_2 \ldots, S_n$.
L'algoritmo termina quando S sarà incluso in Z, finché l'algoritmo continuerà l'esecuzione, è chiaro che $Z \cup S \neq Z$. Per ogni iterazione, sarà vero che $Z_{i+1} = Z_i \cup S_i$, ciò implica

ovviamente che $Z_0 \subset Z_1 \subset Z_2 \cdots \subset Z_n$, quindi la sequenza dei valori di Z, è strettamente crescente ad ogni iterazione, ed è ovviamente "limitata" da R, è ovvio che $\exists f < \infty | Z_f = Z_{f+1} \implies S_f \subset Z_f \implies$ l'algoritmo, ad un certo punto termina. Abbiamo quindi denotato con Z_f il valore che assume Z al termine dell'algoritmo.

Teorema : $Z_f = X^+$

Dimostrazione : Facciamo una dimostrazione per doppia inclusione, partendo da $\boxed{Z_f \subseteq X^+}$: Utilizzeremo l'indice i delle iterazioni per una dimostrazione per induzione, ho il *caso base* : $Z_0 \subseteq X^+$, so che $Z_0 = \{X\}$, e la chiusura di X sicuramente contiene X, quindi $Z_0 \subseteq X^+$ è verificato, passiamo al *passo induttivo* : la nostra ipotesi è che $Z_i \subseteq X^+$ e vogliamo dimostrare che $Z_{i+1} \subseteq X^+$, prendo un qualsiasi $A \in Z_{i+1}$, sapendo che $Z_{i+1} = Z_i \cup S_i$, l'attributo A o appartiene a Z_i oppure a S_i, se A dovesse appartenere a Z_i, avremmo già risolto e dimostrato il passo induttivo, se invece A dovesse appartenere ad S_i, vuol dire che $\exists Y \to V \in F$ tale che $Y \subseteq Z_i \wedge A \in V$. Utilizzo l'ipotesi che $Z_i \subseteq X^+$, quindi se $Y \subseteq Z_i \implies Y \subseteq X^+$, e ciò significa che $X \to Y \in F^A$, e sapendo che $Y \to V \in F^A$, per transitività ottengo che $X \to V \in F^A$, ricordando che $A \in V$, questo implica che $X \to A \in F^A$, ne concludiamo che A, che è un qualsiasi elemento di Z_{i+1} (in questo caso di S_i), è anche in X^+, quindi la prima inclusione è verificata.
$\boxed{X^+ \subseteq Z_f}$: Similmente alla dimostrazione del capitolo 4.3.1, ci serviremo di una particolare istanza r, composta da due tuple, in cui a sinistra compaiono tutti gli attributi contenuti in Z_f, e a destra quelli contenuti nel complementare.

$r :=$		Z_f				$R - Z_f$			
	t_1	1	1	...	1	1	1	...	1
	t_2	1	1	...	1	0	0	...	0

Dimostreremo che r sia legale, prendo una qualsiasi dipendenza funzionale $Y \to V \in F$, $t_1[Y] = t_2[Y]$, so per certo che $Y \subseteq Z_f$, dato che, per come è strutturata l'istanza, non ci può essere un attributo in Z_f che determina un attributo in $R - Z_f$ (Ciò è corretto in quanto, se un attributo fosse determinato da un elemento in Z_f (che sarebbe la chiusura di X), ciò significherebbe che tale attributo è determinato da X, quindi dovrebbe esso stesso essere in Z_f), quindi anche $t_1[V] = t_2[V]$, ed r è legale.
Sia $A \in X^+$ un qualsiasi attributo della chiusura di X, allora $X \to A \in F^A$, ed r soddisfa tale dipendenza, sappiamo che $X = Z_0 \subset Z_f \implies t_1[X] = t_2[X] \implies t_1[A] = t_2[A] \implies A \in Z_f$.

$$\begin{cases} Z_f \subseteq X^+ \\ X^+ \subseteq Z_f \end{cases} \implies Z_f = X^+ \qquad \blacksquare$$

Esempio : Si consideri il seguente schema relazionale : $R = (A, B, C, D, E, H)$ con il seguente insieme di dipendenze funzionali : $F = (AB \to CD, EH \to D, D \to H)$, si calcoli AB^+.

- iterazione 0 - $Z_0 = \{AB\}$, $S_0 = \{CD\}$, noto che $S_0 \nsubseteq Z_0$, quindi continuo.
- iterazione 1 - $Z_1 = \{AB, CD\}$, $S_1 = \{CD, H\}$, noto che $S_1 \nsubseteq Z_1$, quindi continuo.
- iterazione 2 - $Z_2 = \{AB, CD, H\}$, $S_2 = \{CD, H\}$, noto che $S_2 \subseteq Z_2$, quindi mi fermo.

`Output` : $Z_2 = Z_f = AB^+ = \{AB, CD, H\}$

4.5.2 La Ricerca delle Chiavi

Sappiamo che, se un insieme di attributi X determina tutto lo schema relazionale R, allora tale insieme è una *chiave*. Se volessimo capire quali sono le chiavi di uno schema, potremmo quindi calcolare la chiusura di tutti i possibili sotto-insiemi di R, e compararli ad R stesso per vedere se sono equivalenti.

$$\forall X \in \mathcal{P}(R): \quad \texttt{if (Algoritmo(}\ X\ \texttt{)==}\ R\ \texttt{) \{ return Algoritmo(}\ X\ \texttt{) \}}$$

Sappiamo però che, tutti i sotto insiemi di R, sono in numero $2^{|R|}$, e per ognuno di questi si dovrà eseguire l'algoritmo per calcolarne la chiusura 4.5.1, è chiaro che questo tipo di problema, necessiti di un calcolo di una complessità *non polinomiale*, è quindi, per quanto corretto, impensabile di poter trovare la chiave di una relazione usando la "forza bruta".

Osservazione : Se un insieme di attributi non compare mai come determinato in F, allora è parte della chiave. Con tale osservazione, si è può ridurre in maniera notevole il numero di attributi candidati ad essere chiave.

Esempio : $R = (A, B, C, D, E, H)$ $F = (AB \to CD, C \to E, AB \to E, ABC \to D)$ Si vuole trovare la chiave di tale schema relazionale. Si facciano due osservazioni, l'attributo H, non compare in nessuna dipendenza funzionale, inoltre, gli attributi A e B, non compaiono mai come determinati in F, ma solo come determinanti. Si consideri quindi la chiusura di ABH. `Algoritmo(` ABH `)` $= ABH^+ = (A, B, H, C, D, E) = R$, quindi, per definizione 4.1 , se non esiste nessun sottoinsieme di ABH che determina R, tale ABH è chiave. Ci si rende conto dopo pochi calcoli che ABH è chiave.

Osservazione : Gli attributi che compaiono esclusivamente come determinati, non sono chiavi. È consigliabile, iniziare a controllare le chiusure dei sottoinsiemi di cardinalità più alta, se essi non determinano R, non c'è bisogno di controllarne i sotto insiemi

Remark : Durante la prova d'esame, è necessario giustificare tutti i passaggi e le considerazioni che portano il candidato ad escludere un certo insieme di attributi durante la ricerca della chiave.

Osservazione : Le chiavi di uno schema relazionale possono essere molteplici, è quindi opportuno cominciare controllando tutti gli attributi che non compaiono mai come determinati. Se la chiusura di un insieme X è uguale ad R, bisogna verificare che non lo siano anche i suoi sottoinsiemi.

Esempio : $R = (A, B, C, D, E, G, H)$ $F = (AB \to D, G \to A, G \to B, H \to E, H \to G, D \to H)$ Si vogliono trovare le 4 chiavi. Si cominci dagli attributi che determinano altri ma non sono determinati da nessuno, i candidati sono AB, G, D e H. Noto che C non compare mai in nessuna dipendenza funzionale, quindi sarà parte di tutte le chiavi. Comincio allora con ABC, trovo che `Algoritmo(` ABC `)` $= R$, controllo poi i sottoinsiemi e vedo che `Algoritmo(` AC `)` $= (AB, BC^+)$ e che $BC^+ = (BC)$, quindi ABC è una chiave. Ne rimangono 3, gli altri 3 candidati erano G, D e H, utilizzo l'algoritmo e trovo che $GC^+ = DC^+ = HC^+ = R$, sono quindi tutte e 3 le chiavi, senza necessità di controllare i sottoinsiemi (dato che C da sola non può essere chiave, ma deve essere contenuto in ogni chiave).

Dato uno schema R, se viene trovata una chiave, spesso ci si chiede se tale chiave sia l'unica, è quindi opportuno fare un **test di unicità**, dato che, per controllare se uno schema sia in terza forma normale, necessitiamo di sapere quali siano tutte le chiavi.

Esiste un modo semplice per determinare se la chiave di uno schema R è unica :

$$\text{Sia } X = \bigcap_{V \to W \in F} R - (W - V)$$

Una volta aver trovato tale X, se $X^+ = R$, allora X è l'unica chiave di R, altrimenti, esistono più chiavi, ed X non è una superchiave di R.

Esempio : $R = (A, B, C, D, E)$ $F = (AB \to C, AC \to B, D \to E)$, voglio capire se la chiave sia unica, applico quindi il calcolo appena visto :

$$X = (ABCDE - (C - AB) \cap ABCDE - (B - AC) \cap ABCDE - (E - D)) =$$
$$= (ABCDE - C \cap ABCDE - B \cap ABCDE - E) = (ABDE \cap ACDE \cap ABCD) = AD$$

calcolo $X^+ = AD^+ = AD \neq R$, quindi ci sono più chiavi.

4.6 Preservare le Dipendenze Funzionali

Nel capitolo 4.4.2 abbiamo visto come, uno schema che non è in terza forma normale, può essere "decomposto" in più schemi, che non compromettano l'integrità della base di dati, ma che in più, rispettino la 3NF. Una decomposizione di R, è una famiglia di insiemi $R_1, R_2 \ldots, R_n$ tali che $\bigcup_{i=1}^{n} R_i = R$ (la decomposizione impone che i diversi sotto-schemi non siano disgiunti, non deve essere una partizione).

Una volta decomposto uno schema, ad ogni sotto-schema sarà assegnato un nuovo insieme di dipendenze funzionali. Decomporre una tabella (e la sua istanza), equivale ad eseguire una proiezione 3.0.1, diamo un importante definizione.

Definizione : Due insiemi di dipendenze funzionali F e G, si dicono *equivalenti*, e si denota $F \equiv G$, se la loro chiusura è identica, quindi, determinano lo stesso insieme di istanze legali.

$$F \equiv G \iff F^+ = G^+$$

Quando si decompone uno schema, occorre verificare che le dipendenze funzionali siano preservate, quindi, che il nuovo insieme di dipendenze sia equivalente a quello originale. Ovviamente, F^+ e G^+ sono troppo grandi, e controllarli nella loro interezza richiederebbe un tempo non polinomiale, risulta utile però il seguente lemma.

Lemma : Sia R uno schema, ed F e G due insiemi di dipendenze funzionali, vale che :

$$F^+ \subseteq G^+ \iff F \subseteq G^+$$

Dimostrazione : Se la prima implicazione è scontata, la seconda, ossia $F \subseteq G^+ \implies F^+ \subseteq G^+$, risulta meno banale ed assai efficace. L'ipotesi, è che $F \subseteq G^+$, sia $f = X \to Y$ una qualsiasi dipendenza funzionale in F^+. So che $F^+ = F^A$, quindi, se f è in F^A, ciò vuol dire, che esiste un insieme di dipendenze funzionali $\{f_1, f_2 \ldots, f_n\} \subseteq F$, tale che, applicando gli assiomi di Armstrong 4.2 su di esse, si ottiene f.

$$\{f_1, f_2 \ldots, f_n\} \xrightarrow{\text{Armstrong}} f$$

L'ipotesi però, è che $F \subseteq G^+$, ma $G^+ = G^A$, quindi $F \subseteq G^A$, questo vuol dire che, se un elemento è in G^A, esiste un insieme di dipendenze funzionali, tali che applicando gli assiomi di Armstrong su di esse, si ottiene quell'elemento. Si osservi che, $\{f_1, f_2 \ldots, f_n\} \subseteq F \subseteq G^A$, quindi, $\forall f_i \in \{f_1, f_2 \ldots, f_n\}, \exists \{g_1^i, g_2^i \ldots, g_k^i\} \subseteq G$ tale che $\{g_1^i, g_2^i \ldots, g_k^i\} \xrightarrow{\text{Armstrong}} f_i$.

$$\begin{array}{c} \{g_1^1, g_2^1 \ldots, g_k^1\} \\ \cdots \\ \cdots \\ \{g_1^n, g_2^n \ldots, g_k^n\} \end{array} \begin{array}{c} \xrightarrow{\text{Armstrong}} \\ \\ \\ \xrightarrow{\text{Armstrong}} \end{array} \left\{ \begin{array}{c} f_1 \\ \cdot \\ \cdot \\ f_n \end{array} \right\} \xrightarrow{\text{Armstrong}} f \in F^+$$

Questo significa che, applicando gli assiomi di armstrong a partire da elementi in G, ottengo degli elementi in F^+, ma se $F \subseteq G^+$, so che ogni elemento di F^+ si ottiene applicando gli assiomi su F, ed F si ottiene applicando gli assiomi su G, quindi $F^+ \subseteq G^+$. ■

Dato tale lemma, si giunge alla **conclusione fondamentale** che :

$$F \equiv G \iff F \subseteq G^+ \wedge G \subseteq F^+$$

Adesso che sappiamo come controllare l'equivalenza di due dipendenze funzionali, possiamo enunciare il metodo con cui, si controlla se una scomposizione, preserva l'insieme delle dipendenze funzionali.

4.6.1 Controllo dell'Unione

Sia R uno schema relazionale, F l'insieme delle dipendenze funzionali, e $\rho = \{R_1, R_2 \ldots, R_n\}$ una sua decomposizione. Definiamo per ogni elemento di ρ, un insieme di dipendenze funzionali correlato ad F, definito in tal modo :

$$\pi_{R_i}(F) = \{X \to Y \in F^+ | \{X, Y\} \subseteq R_i\}$$

Quindi, per ogni sotto-schema, è associato l'insieme $\pi_{R_i}(F)$, che contiene le dipendenze funzionali che sono in F^+, se come determinante e determinato appaiono esclusivamente elementi che siano in tale sotto-schema.

Diremo che, la decomposizione ρ **preserva le dipendenze funzionali** in F, se e solo se :

$$G = \bigcup_{i=1}^{n} \pi_{R_i}(F) \equiv F$$

Per controllare che $G \equiv F$, devo controllare solamente che $F \subseteq G^+$, non è necessario controllare che $G \subseteq F^+$, dato che, per come è definito G, è l'unione sottoinsiemi di F^+, quindi, che G sia contenuto nella chiusura di F è ovvio. Per controllare che $F \subseteq G^+$, bisogna prendere una qualsiasi dipendenza $X \to Y \in F$, e controllare che sia in G^+.

Osservazione : Se $X \to Y \in G^+$, allora $Y \subseteq X_G^+$, è quindi necessario controllare ogni dipendenza di F, calcolare la chiusura del determinante rispetto a G, e controllare che il determinato sia in tale chiusura, se per uno solo di questi, non vale ciò, allora F non è equivalente a G.

Problema : L'osservazione appena fatta, enuncia che ci basta controllare la chiusura di un insieme di attributi rispetto a G per verificare l'equivalenza, il problema, è che G deriva da F^+, quindi è troppo grande per essere calcolato esplicitamente, e non può essere applicato l'algoritmo visto nel capitolo 4.5.1. Vedremo quindi, un nuovo algoritmo, capace di calcolare la chiusura di un insieme di attributi X rispetto a $G = \bigcup_{i=1}^{n} \pi_{R_i}(F)$ partendo da F.

4.6.2 L'Algoritmo 2 (`Compute` X_G^+ `from` F)

`Input` : Lo schema R, l'insieme delle dipendenze funzionali F, una decomposizione $\rho = \{R_1, R_2 \ldots, R_k\}$ ed $X \subseteq R$.

`Output` : Denotato con Z_f, sarà la chiusura di X rispetto a $G = \bigcup_{i=1}^{n} \pi_{R_i}(F)$.

```
begin {
    Z = X
    S' = ∅
    for  i = 1  to  k{
        S' = S' ∪ (Z ∩ R_i)^+_F ∩ R_i
    }
    while  S' ⊈ Z {
        Z = Z ∪ S'
        for  i = 1  to  k{
            S' = S' ∪ (Z ∩ R_i)^+_F ∩ R_i
        }
    }
    return  Z
}
```

Teorema : L'algoritmo calcola correttamente la chiusura di X rispetto a G, ossia $Z_f = X_G^+$.

Dimostrazione : Si procede per doppia inclusione, partendo col dimostrare $\boxed{Z_f \subseteq X_G^+}$: Si dimostra per induzione, *Caso Base* : $Z_0 = X \subseteq X_G^+$. *Passo induttivo* : L'ipotesi è che, $Z_i \subseteq X_G^+$, voglio dimostrare che se ciò è vero, è vero anche che $Z_{i+1} \subseteq X_G^+$. Considero un qualsiasi $A \in Z_{i+1}$, per come è definito Z_{i+1}, ciò significa che $A \in Z_i$ oppure $A \in S_i'$. Il primo caso dimostra di per se la tesi, si consideri quindi il caso in cui $A \in S_i'$. Ricordo che $S_i' = \bigcup_{j=1}^{k} (Z_i \cap R_j)_F^+ \cap R_j$, allora, $\exists j \in \{1, 2 \ldots, k\} | A \in (Z_i \cap R_j)_F^+ \cap R_j$, ciò implica che $A \in R_j \wedge A \in (Z_i \cap R_j)_F^+$, ma questo'ultimo implicherebbe che $Z_i \cap R_j \to A \in F^A$. So che $Z_i \cap R_j \subseteq R_j$, e che $A \in R_j$, allora $Z_i \cap R_j \to A \in \pi_{R_j}(F) \subseteq G$, ma ricordiamoci che per ipotesi $Z_i \subseteq X_G^+$, allora, anche il suo sotto insieme $Z_i \cap R_j$ è contenuto in X_G^+, ciò implica che $X \to (Z_i \cap R_j) \in G^A$, per transitività, ho che $X \to A \in G^A \implies A \in X_G^+$ come volevasi dimostrare. Adesso rimane da dimostrare che $\boxed{X_G^+ \subseteq Z_f}$: Tale dimostrazione risulta più ostica e richiede più ragionamento, si invita il lettore ad essere particolarmente attento. Prima di tutto, occorre un osservazione :

Osservazione Fondamentale : Siano A e B due insiemi di attributi ed F un insieme di dipendenze funzionali su di essi. se $A \subseteq B$, allora sicuramente $A_F^+ \subseteq B_F^+$.

Tornando a noi, vogliamo dimostrare che $X_G^+ \subseteq Z_f$, ma per come è stato costruito Z_f, sappiamo sicuramente che $X \subseteq Z_f$, allora, per l'osservazione, sicuramente $X_G^+ \subseteq (Z_f)_G^+$. Se riuscissimo a dimostrare che $Z_f = (Z_f)_G^+$, vuol dire che avremmo dimostrato che $X_G^+ \subseteq Z_f$. Dimostriamo quindi che $Z_f = (Z_f)_G^+$, possiamo utilizzare l'algoritmo 4.5.1, che calcola la chiusura di un insieme di attributi rispetto un insieme di dipendenze funzionali, quindi passiamo come input Z_f e ne calcoliamo la chiusura rispetto a G, se l'output dell'algoritmo sarà uguale all'input, allora sarà dimostrato.

Durante il corso di questa dimostrazione, si osservi l'algoritmo 4.5.1.

Notiamo che, nella seconda riga dell'algoritmo, si inizia costruendo un insieme S definito in tal modo :

$$S = \{A | \exists Y \to V \in G, Y \subseteq Z_f \wedge A \in V\} \tag{21}$$

Prendiamo quindi A, che è un qualsiasi elemento dell'insieme S costruito nella seconda riga di codice. Notiamo che $A \in V$, ed $Y \to V \in G$, per la regola di decomposizione, possiamo affermare che $Y \to A \in G$, adesso, ricordiamo come è stato costruito G :

$$G = \bigcup_{i=1}^{n} \pi_{R_i}(F) \equiv F$$

G è l'unione delle dipendenze funzionali nelle proiezioni, quindi, $Y \to A$ appartiene ad una delle proiezioni :

$$\exists j \in \{1, 2 \ldots, n\} | Y \to V \in \pi_{R_j}(F)$$

ma una proiezione, è definita in tal modo :

$$\pi_{R_i}(F) = \{X \to Y \in F^+ | \{X, Y\} \subseteq R_i\}$$

Quindi, se $Y \to V \in \pi_{R_j}(F)$, allora $Y \in R_j$ e $V \in R_j \implies A \in R_j$.

Passo cruciale 1! $A \in R_j$

Abbiamo quindi visto che $Y \to V \in \pi_{R_j}(F)$, quindi $Y \subseteq R_j$. Sapendo che $Y \subseteq Z_f$, possiamo dire che $Y \subseteq R_j \cap Z_f$, ma $R_j \cap Z_f \subseteq (R_j \cap Z_f)_F^+$, quindi $Y \subseteq (R_j \cap Z_f)_F^+$, ne consegue che $R_j \cap Z_f \to Y \in F^+$, essendo che $Y \to V \in F^+$, per transitività si ha $R_j \cap Z_f \to V \in F^+$, ed essendo che $A \in V$, per decomposizione ho che $R_j \cap Z_f \to A \in F^+ \implies A \in (R_j \cap Z_f)_F^+$.

Passo cruciale 2! $A \in (R_j \cap Z_f)_F^+$

A questo punto vedo che :

$$\begin{cases} A \in (Z_f \cap R_j)_F^+ \\ A \in R_j \end{cases} \implies A \in (Z_f \cap R_j)_F^+ \cap R_j \tag{22}$$

Adesso, noi sappiamo che $A \in (Z_f \cap R_j)_F^+ \cap R_j$, si osservi ora, il secondo algoritmo 4.6.2, quello che è stato usato per costruire il nostro Z_f. Si noti che, ad ogni passo iterativo di quell'algoritmo, Z_f viene ricostruito, a partire da un altro insieme S', definito nel seguente modo :

$$(Z \cap R_i)_F^+ \cap R_i \text{ osservazione : tale } Z \text{ è contenuto nel nostro } Z_f \tag{23}$$

Abbiamo detto però, che $A \in (Z_f \cap R_j)_F^+ \cap R_j$, quindi, tale A, sarà in S' ad un certo passo iterativo del secondo algoritmo 4.6.2! Essendo che Z_f è l'unione di tutti gli insiemi S' ricostruiti iteramente nel secondo algoritmo 4.6.2, ciò, significa che $A \in Z_f$.

In *conclusione*, abbiamo dimostrato che, eseguendo il primo algoritmo 4.5.1 su Z_f, alla seconda riga di codice viene costruito un insieme S, i quali elementi sono già tutti in Z_f, questo significa, che l'algoritmo **non** eseguirà il `while`, che richiede come condizione che $S \nsubseteq Z_f$, terminando immediatamente, e restituendo come output, lo stesso Z_f dato in input. Essendo che quello l'algoritmo calcola la chiusura di un insieme di attributi, in conclusione, possiamo affermare che, la chiusura di Z_f, rispetto a G, è identica a Z_f : $Z_f = (Z_f)_G^+$.

Ricostruendo il tutto a partire dall'inizio, si ha che :

$$\begin{cases} X \subseteq Z_f \implies X_G^+ \subseteq (Z_f)_G^+ \\ Z_f = (Z_f)_G^+ \\ Z_f \subseteq X_G^+ \text{ (dimostrato)} \end{cases} \implies \begin{cases} X_G^+ \subseteq Z_f \\ Z_f \subseteq X_G^+ \end{cases} \implies Z_f = X_G^+ \quad \blacksquare$$

4.6.3 Sommario del Procedimento Completo

Ricapitoliamo adesso ciò che è necessario per controllare se una decomposizione preserva l'insieme delle dipendenze funzionali. Siano :

- $R :=$ Schema Relazionale
- $\rho = (R_1, R_2 \ldots, R_n) :=$ Decomposizione di R
- $F :=$ Dipendenze funzionali su R

Come prima cosa si costruisce un nuovo insieme :

$$G = \bigcup_{i=1}^{n} \pi_{R_i}(F)$$

Bisogna controllare che $G \equiv F$, per fare ciò, abbiamo osservato, e dimostrato, che è necessario controllare che $F \subseteq G^+$. **Osservazione** : Abbiamo visto che, per controllare che $F \subseteq G^+$, basta selezionare ogni $X \to Y \in F$, calcolare X_G^+ (con il secondo algoritmo 4.6.2), e controllare che $Y \in X_G^+$, ciò deve valere per ogni dipendenza di F. Detto ciò, possiamo riassumere il tutto con un ultimo algoritmo.

4.6.4 L'Algoritmo 3 (`Check if` $F \subseteq G^+$)

`Input` : Lo schema R, due insiemi di dipendenze funzionali F e G su R.
`Output` : Un valore booleano `success`, esso varrà `true` se $F \subseteq G^+$, altrimenti `false`.

Chiameremo `Compute` X_G^+ il secondo algoritmo 4.6.2, dove X è l'insieme di attributi in Input, e G l'insieme di dipendenze funzionali sul quale calcolare la chiusura.

```
begin {
    success=true
    for each X → Y ∈ F{
        T = Compute X_G^+
        if Y ⊈ T {
            success=false
            break
        }
    }
    return success
}
```

4.7 Join Senza Perdita

Nel capitolo 4.4.2, abbiamo visto come una decomposizione che soffisfi la terza forma normale, non è comunque una buona decomposizione se, eseguendo un `JOIN` su un istanza legale, si generano tuple che non erano presenti nell'istanza originale prima della decomposizione.

Vogliamo quindi creare una decomposizione che ci permetta di non perdere informazioni, ossia che ci permetta di ottenere un cosiddetto `JOIN` senza perdita. Sia r un istanza *legale* di uno schema R, e $\rho = \{R_1, R_2 \ldots, R_k\}$ una decomposizione di R, è definita la seguente istanza :

$$m_\rho(r) = \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \cdots \bowtie \pi_{R_k}(r)$$

Si ha un `JOIN` senza perdita se e solo se, *per ogni* istanza legale r di R, vale che $m_\rho(r) = r$.

Teorema : Sono vere le seguenti proprietà :

1. $r \subseteq m_\rho(r)$
2. $\pi_{R_i}(m_\rho(r)) = \pi_{R_i}(r)$
3. $m_\rho(m_\rho(r)) = m_\rho(r)$

Dimostrazione : 1 : Sia t una tupla di r, Se $t \subset r$ allora $t[R_i] \in \pi_{R_i}(r)$, si ha che :
$t \in \{t[R_1]\} \bowtie \{t[R_2]\} \cdots \bowtie \{t[R_k]\}$, ma $\forall i \in \{1, 2 \ldots, k\}$ si ha che $\{t[R_i]\} \subseteq \pi_{R_k}(r)$, quindi :
$t \in \{t[R_1]\} \bowtie \{t[R_2]\} \cdots \bowtie \{t[R_k]\} \subseteq \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \cdots \bowtie \pi_{R_k}(r) = m_\rho(r) \implies r \subseteq m_\rho(r)$ □
2 : Dimostro per doppia inclusione, dalla proprietà 1, deriva $\pi_{R_i}(r) \subseteq \pi_{R_i}(m_\rho(r))$, bisogna ora dimostrare $\pi_{R_i}(m_\rho(r)) \subseteq \pi_{R_i}(r)$. Si consideri una tupla $t^i \in \pi_{R_i}(r)$, allora
$\exists t \in m_\rho(r) | t[R_i] = t^i \implies t_1, t_2 \ldots, t_k \in r | \forall j \in \{1, 2 \ldots, k\} t_j[R_j] = t[R_j] \implies t^i = t[R_i] = t_i[R_i] \in \pi_{R_i}(r) \implies \pi_{R_i}(m_\rho(r)) \subseteq \pi_{R_i}(r) \implies \pi_{R_i}(m_\rho(r)) = \pi_{R_i}(r)$. □
3 : So che :

$$m_\rho(m_\rho(r)) = \pi_{R_1}(m_\rho(r)) \bowtie \pi_{R_2}(m_\rho(r)) \cdots \bowtie \pi_{R_k}(m_\rho(r))$$

Ma per la proprietà 2 si ha che $\pi_{R_i}(m_\rho(r)) = \pi_{R_i}(r)$, quindi :

$$m_\rho(m_\rho(r)) = \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \cdots \bowtie \pi_{R_k}(r) = m_\rho(r) \ \square$$

Le proprietà 1,2 e 3 sono state dimostrate. ■

Per verificare che ρ abbia un `JOIN` senza perdita, non possiamo controllare ogni istanza r dato che sono potenzialmente infinite, esiste però un algoritmo adatto a tale compito, che utilizza l'insieme di dipendenze funzionali F.

4.7.1 L'Algoritmo per il loseless JOIN (`check if` ρ `has a loseless JOIN`)

`Input` : Lo schema R, un insiemi di dipendenze funzionali F su R, una decomposizione $\rho = \{R_1, R_2 \ldots, R_k\}$.
`Output` : Un istanza r che dirà se ρ ha o non ha un `JOIN` senza perdita.

```
begin {
    r = Construct()
        if ∃t1, t2 ∈ r|t1[X] = t2[X] ∧ t1[Y] ≠ t2[Y] {
            for all Aj ∈ Y {
                if t1[Aj] = a {
                t2[Aj] = a
                } else {
                t1[Aj] = t2[Aj]
                }
            }
        }
        until (r ha una riga con tutte a) ∨ (r non è cambiata)
    }
    return r
}
```

`Construct()` costruisce un istanza r come segue : L'istanza ha come colonne gli attributi di R, e ad ogni riga, è associata un elemento R_i di ρ. Ad ogni posizione dell'i-esima riga e della j-esima colonna, si inserisce a se, l'attributo della j-esima colonna è contenuto nell'elemento R_i di ρ, altrimenti b_i.

L'istanza costruita all'inizio può essere vista come un istanza di R, non necessariamente legale, l'algoritmo poi opererà per renderla tale.

Osservazione : L'algoritmo non modificherà mai gli elementi dell'istanza uguali ad a, bensì lavorerà esclusivamente sugli elementi b_i.

Ciò ci garantisce che il numero delle a, con l'andare avanti delle iterazioni è strettamente crescente, ciò garantisce la terminazione dell'algoritmo, in quanto, se non dovesse modificare nulla terminerebbe, diversamente, se dovesse continure ad aggiungere a, prima o poi creerebbe una riga con tutte a, terminando. Ciò garantisce anche che l'istanza finale sia legale, per motivi analoghi a quelli appena considerati.

Teorema : Una decomposizione ρ di R ha un `JOIN` senza perdita, se e solo se, l'istanza r restituita dall'algoritmo `check if` ρ `has a loseless JOIN` ha una riga con tutte a.

Dimostrazione : Verrà dimostrato un solo verso della doppia implicazione, si assuma quindi che ρ abbia un `JOIN` senza perdita, vale quindi che $m_\rho(r) = r$ per ogni istanza legale. Considero adesso r^i, ossia l'istanza costruita dall'algoritmo alla i-esima iterazione, ed r^f come l'istanza finale che verrà restituita. Considero $t_i^0 \in r^0$ una tupla appartenente ad r appena costruita, restringo la tupla ad R_i ed ho che $t_i^0[R_i] = (a, a \ldots, a)$ contiene tutte a. Essendo che le a non vengono mai modificate, si ha che $t_i^0[R_i] = t_i^f[R_i]$. Denoto con t_a, una tupla intera,

non ristretta a nulla, contenente solo a. Ho che :

$$t_a \in \{t_1^f[R_1]\} \bowtie \{t_2^f[R_2]\} \cdots \bowtie \{t_k^f[R_k]\} \subseteq m_\rho(r^f)$$

Ma per ipotesi, $m_\rho(r) = r$, quindi la tupla con tutte a è contenuta nell'istanza finale r^f. ■

4.8 Trovare una Buona Decomposizione

4.8.1 Copertura Minimale

Dato uno schema, è sempre possibile trovare una buona decomposizione, che sia in 3NF, preservi l'insieme di dipendenze funzionali, ed abbia un `JOIN` senza perdita. Una buona decomposizione per uno schema, non è unica, ce ne sono molteplici, vedremo poi un algoritmo, che sarà in grado di calcolarle tutte, restituendone una.

Definizione di copertura minimale : Dato un insieme di dipendenze funzionali F su uno schema R, si dice copertura minimale di F, un insieme di dipendenze funzionali G equivalente ad F, con le seguenti proprietà :

1. $\forall X \to Y \in G$, Y è un attributo singolo.
2. $\forall X \to A \in G$, non esiste un sottoinsieme $X' \subset X | G \equiv G - \{X \to A\} \cup \{X' \to A\}$, ciò minimizza i determinanti.
3. Non esista alcuna dipendenza $X \to A \in G | G \equiv G - \{X \to A\}$, quindi ogni dipendenza in G è necessaria e non ci sono ridondanze.

Un insieme F ha più coperture minimali, e ad ognuna di esse è associata una buona decomposizione.

Calcolo per la Copertura Minimale
Si può trovare una copertura minimale di un insieme F seguendo 3 fasi :

1. *Fase 1* : Si minimizzano tutti i determinati tramite la decomposizione, ad esempio, $X \to AB$ diventa $X \to A, X \to B$.
2. *Fase 2* : Per ogni dipendenza $X \to A \in F$, controllo tutti i sottoinsiemi $X' \subset X$, e verifico che $X' \to A \in F^+$, se ciò è vero, posso sostituire $X \to A$ con $X' \to A$, controllando poi i sottoinsiemi di X'.
3. Per ogni dipendenza $X \to A \in F$, controllo che $X \to A \in (F \backslash \{X \to A\})^+$, se si, allora è ridondante, e può essere eliminata, basta controllare.

Le operazioni fatte su F per trovare la copertura minimale, fan si che non si modifichi la sua chiusura.

4.8.2 L'Algoritmo per la decomposizione (`find` ρ)

Vediamo adesso l'algoritmo in grado di computare una decomposizione, partendo dalla chiusura minimale.

Piuttosto che una buona decomposizione nella sua totalità, l'algoritmo troverà prima una decomposizione che preservi F ed abbia tutti i sottoschemi in terza forma normale, ma che non abbia necessariamente un `JOIN` senza perdita.
`Input` : Lo schema R, un insiemi di dipendenze funzionali F che sia una copertura minimale.
`Output` : Una decomposizione ρ di R che preservi F e sia in 3NF.

```
begin {
    S = ∅
    ρ = ∅
    for each A ∈ R|∀X → Y ∈ F, A ∉ X ∧ A ≠ Y { // A non è in nessuna dip. funz.
        S = S ∪ A
    }
    if S ≠ ∅{
        R = R − S
        ρ = ρ ∪ {S}
    }
    if ∃X → A ∈ F|X ∪ A == R{
        ρ = ρ ∪ {R}
    } else {
        for each X → A ∈ F{
            ρ = ρ ∪ {XA}
        }
    return ρ
}
```

Teorema : La decomposizione ρ restituita dall'algoritmo, è in terza forma normale, e preserva l'insieme di dipendenze funzionali F.

Dimostrazione : `ρ preserva F` : Si considerino le proiezioni di F sugli schemi già viste nel capitolo 4.6.1, ossia $G = \bigcup_{i=1}^{k} \pi_{R_i}(F)$, dato che per ogni dipendenza funzionale $X \to A \in F$ abbiamo che $XA \in \rho$ (è uno dei sottoschemi), abbiamo che questa dipendenza di F sarà anche in G, e per definizione sappiamo che $F \equiv G$.
`ρ è in 3NF` : Bisogna suddividere la dimostrazione in 3 casi :

1. $S \in \rho$ - ogni attributo in S è parte della chiave, quindi ρ rispetta la 3NF.
2. $R \in \rho$ - C'è una dipendenza funzionale che coinvolge tutto R, è una dipendenza del tipo $\{R - A\} \to A$, dove non vi è un sottoinsieme proprio di $\{R - A\}$ che determini A. Una chiave, è quindi $\{R - A\}$. Sia $Y \to B$ una qualsiasi altra dipendenza funzionale in F, si ha che :
   (a) $B = A$ - Allora $Y = \{R - A\}$ è chiave e lo schema è in 3NF.
   (b) $B \neq A$ Allora $B \in \{R - A\}$, è quindi un attributo primo e lo schema è in 3NF.
3. $XA \in \rho$ - Essendo F una copertura minimale, non esiste $X' \to A \in F | X' \subset X$, quindi X è una chiave di XA. Sia $Y \to B$ una qualsiasi altra dipendenza funzionale in F, tale che $YB \subseteq XA$, se $B = A$, essendo F copertura minimale, $Y = X$, ed Y è superchiave. Se $B \neq A$, allora $B \in X$, quindi è primo, e lo schema è in 3NF.

Può succedere che si verifichino i casi 1,2 - 1,3 - 3. In ogni situazione, lo schema è in 3NF. ■

Una volta calcolato il risultato, rimane solamente da capire se ρ ha un `JOIN` senza perdita.

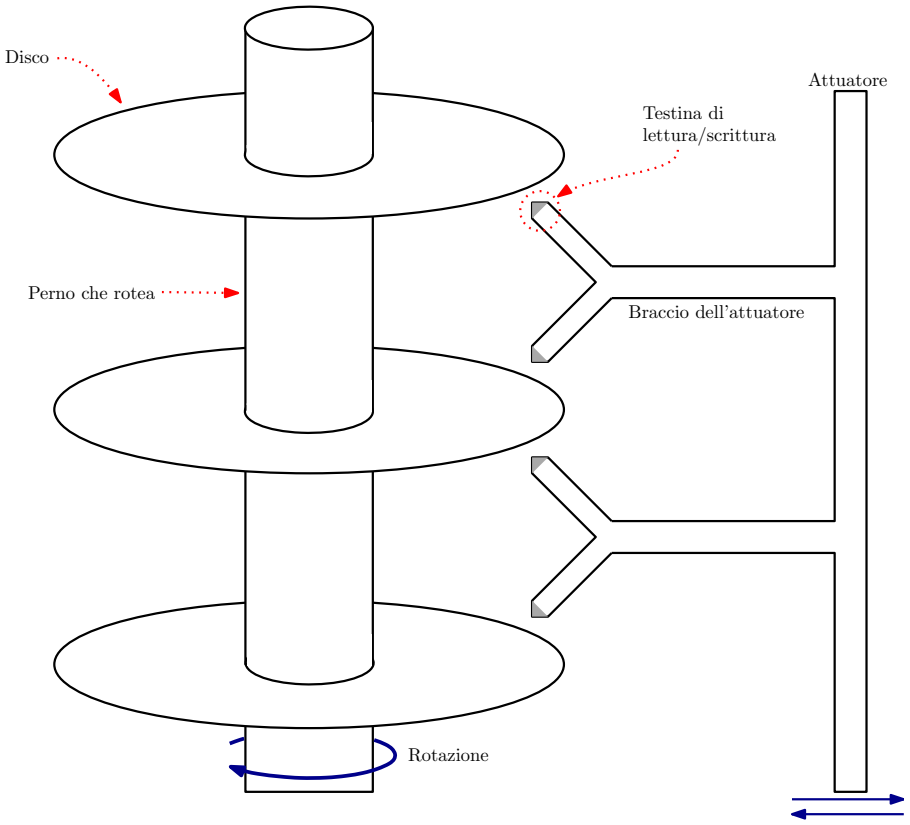
Proposizione : Sia ρ l'output dell'algoritmo per la decomposizione, se in ρ è presente un sottoschema che contiene una chiave di R, allora ρ ha un `JOIN` senza perdita.

In caso non dovesse esserci, sarebbe necessario aggiungere una chiave di R come sottoschema a ρ dopo aver utilizzato l'algoritmo, e si otterrebbe una buona decomposizione, che preveda un `JOIN` senza perdita, preservi F e sia in terza forma normale.

5 Organizzazione Fisica

5.1 Specifiche Fisiche del Disco

Un DataBase Managment System è implementato su un supporto fisico, che richiede un unità di gestione della memoria, ed un unità di gestione dell'accesso concorrenziale ai dati da parte di diversi utenti. I dati sono contenuti in delle memorie, tutt'oggi, nonostante possa sembrare obsoleto, la maggiorparte dei database utilizzano ancora degli hard disk piuttosto che dei dischi a stato solido. Vediamo nel dettaglio il funzionamento di un hard disk, ed i tempi di delay fisici ad esso correlati, per poi implementare dei sistemi di ricerca e scrittura che sfruttino al meglio i tempi del supporto fisico.



Vi è un perno che rotea, sulla quale poggiano dei dischi, essi sono divisi in settori (cerchi concentrici) contenenti dei blocchi di memoria di dimensione fissa. Vi è un *attuatore*, che si muove avanti ed indietro, composto da bracci che contengono a loro volta delle testine.

L'attuatore, muovendosi avanti ed indietro seleziona il settore del disco da leggere, il perno, roteando, selezionerà il blocco di memoria che la testina posta sul braccio (di dimensioni identiche al blocco) leggerà.

Leggere un blocco di memoria richiederà quindi 3 diversi intervalli di tempo :

1. **seek time** - Il tempo di movimento dell'attuatore in cui posiziona la testina sul giusto settore.
2. **rotation delay** - Il tempo di rotazione del disco per far selezionare alla testina il blocco corretto.
3. **transfer time** - Il tempo impiegato dalla testina per leggere il blocco, dipende dalle dimensioni di esso, e dalla densità delle particelle magnetiche.

Il tempo totale per leggere un dato, detto **service time**, è la somma di questi 3 delay elencati.

Si può organizzare un DBSM in modo che gli accessi alla memoria siano sequenziali oppure randomici, ciò conferisce una notevole distinzione in termini di tempi, gli accessi sequenziali, prevedono che il disco debba solo roteare, in quanto i dati sono organizzati in blocchi consecutivi. Se l'accesso fosse randomico, ogni lettura richiederebbe che l'attuatore si riposizioni nel settore esatto.

seek time:= $seek$ rotation delay:= rot dimensione dei blocchi:= BL transfer time:= TR
tempo stimato di lettura/scrittura con accessi randomici = T_{rba}
tempo stimato di lettura/scrittura con accessi sequenziali = T_{sba}

$$T_{rba} = seek + \frac{rot}{2} + \frac{BL}{TR} \qquad T_{sba} = \frac{rot}{2} + \frac{BL}{TR}$$

Ad esempio :

seek time medio	8.9 ms
velocità di rotazione	7200 rotazioni al minuto
velocità di trasferimtento (TR)	150 $MegaByte$ al secondo
dimensioni del blocco (BL)	4096 (bytes)

$ROT = \frac{60 \cdot 1000}{7200} = 8.33ms$
$\frac{ROT}{2} = 4.167ms$
$T_{rba} = 8.9 + 4.167 + 0.026 = 13.093ms$
$T_{sba} = 4.167 + 0.026 = 4.193ms$

5.2 Modelli di Organizzazione

Bisogna adesso tradurre il modello logico in un modello fisico e considerare diversi tipi di organizzazione. Una base di dati, consiste in un insieme di file, è quindi un insieme dei blocchi prima citati (solitamente, di $4KB$), dentro i blocchi vi sono i *record*, che non sono altro che le tuple/righe viste nei capitoli precedenti (un record però, può anche contenere dei puntatori), un record, consiste in una serie di *campi*, ossia gli attributi della tupla. Vediamo adesso, alcuni dei diversi modelli di organizzazione.

5.2.1 Heap File Organization

È il modello più semplice di organizzazione, ogni nuovo record, viene inserito alla fine del file, e non vi sono relazioni fra gli attributi del record e la loro posizione fisica, l'unico modo per cercare un file, è fare una *ricerca lineare*, scorrendo eventualmente, tutti i blocchi. Su un file contenente n blocchi, in media, si dovranno scorrere $n/2$ blocchi prima di trovare quello ricercato. I blocchi, vengono sempre cercati ed identificati tramite il campo della loro chiave primaria, detta anche *chiave di ricerca*. La ricerca in questo modello ha costo computazionale $O(n)$.

5.2.2 Sequential File Organization

I record vengono ordinati in base all'ordine (crescente o decrescente) della loro chiave di ricerca, permettendo l'accesso sequenziale alla memoria se si ricercano i file in maniera lineare. Essendo le chiavi di ricerca ordinate, piùttosto che ricercare in maniera lineare, è possibile eseguire una ricerca binaria tramite accessi randomici, permettendo di abbassare il costo computazionale a $O(\log_2(n))$.

Esempio : Supponiamo di dover ricercare un blocco. Vi sono 1500 blocchi, è possibile fare ricerca lineare con accessi sequenziali o ricerca binaria con accessi randomici :

$$\frac{1500}{2} \cdot T_{sba} = 750 \cdot T_{sba}$$
$$\log_2(1500) \cdot T_{rba} \simeq 11 \cdot T_{rba}$$

La ricerca binaria risulta molto più efficiente. Il problema di questa organizzazione, è l'inserimento di nuovi record, in quanto richiede lo spostamento e la riorganizzazione di tutti i record in modo da mantenere l'ordine delle chiavi, risulta quindi estremamente più lento aggiornare la base di dati, piuttosto che nel modello Heap File.

5.2.3 Random File Organization (Hashing)

Vi è una relazione diretta fra la chiave di ricerca di un record e la sua posizione fisica, un record può essere acceduto con uno o pochi accessi. Si utilizza una funzione *hash*, che definisce una mappa che mette in relazione le chiavi di ricerca con gli indirizzi fisici in memoria.

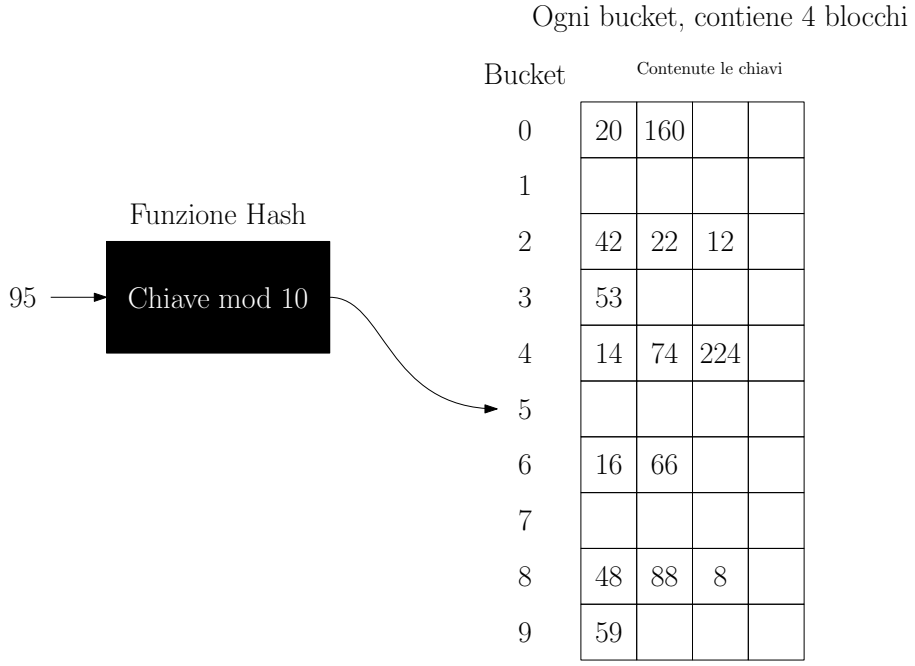
La funzione hash, non garantisce una mappa biunivoca fra chiavi ed indirizzi, diverse chiavi di ricerca potrebbero essere mappate sullo stesso indirizzo (collisione), che in questo caso chiameremo *bucket*, quindi, soprattutto se vi sono parecchi record memorizzati, sarà più probabile che in ogni bucket rientreranno più record. L'algoritmo di hashing, deve garantire una distribuzione uniforme.

Alcuni dei più popolari algoritmi di hashing, prevedono l'utilizzo della divisione con resto, dove ad ogni chiave di ricerca, si assegna l'indirizzo uguale alla chiave modulo ad un numero (solitamente primo), leggermente maggiore al numero di indirizzi disponibili.

$$\text{Address}(Key_i) = Key_i \mod M$$

I record che finiranno negli stessi bucket, saranno per definizione della funzione hash, vicini fra loro, si accedere in maniera randomica al bucket, per poi ricercare in maniera lineare con accessi sequenziali, il record corretto che si trova in quel bucket. La dimensione fissa dei

bucket è arbitraria, dei bucket di dimensione maggiore, implicano una minore probabilità di collisione, ma i bucket che contengono più record saranno più fitti, e bisognerà impiegare più tempo nella ricerca.



5.2.4 Indexed Sequential File Organization

L'organizzazione di tipo Random è efficiente nel cercare record individuale in base alla chiave di ricerca, l'organizzazione sequenziale è efficiente nel ricercare più record ordinati fra loro, questo tipo di organizzazione Indexed Sequential, riconosce entrambi i concetti, combina i fila organizzati in maniera sequenziale con più *indici*.

I file sono divisi in intervalli o partizioni, ogni intervallo è rappresentato da una *index entry*, che contiene la chiave di ricerca del primo record nell'intervalo, ed il puntatore alla posizione fisica del primo record dell'intervallo. Il puntatore, può puntare ad un blocco, o ad un record (combinazione dell'indirizzo del blocco + codice identificativo del record). Gli indici sono sequenziali, e sono ordinati in base alla chiave di ricerca contenuta nell'entry.

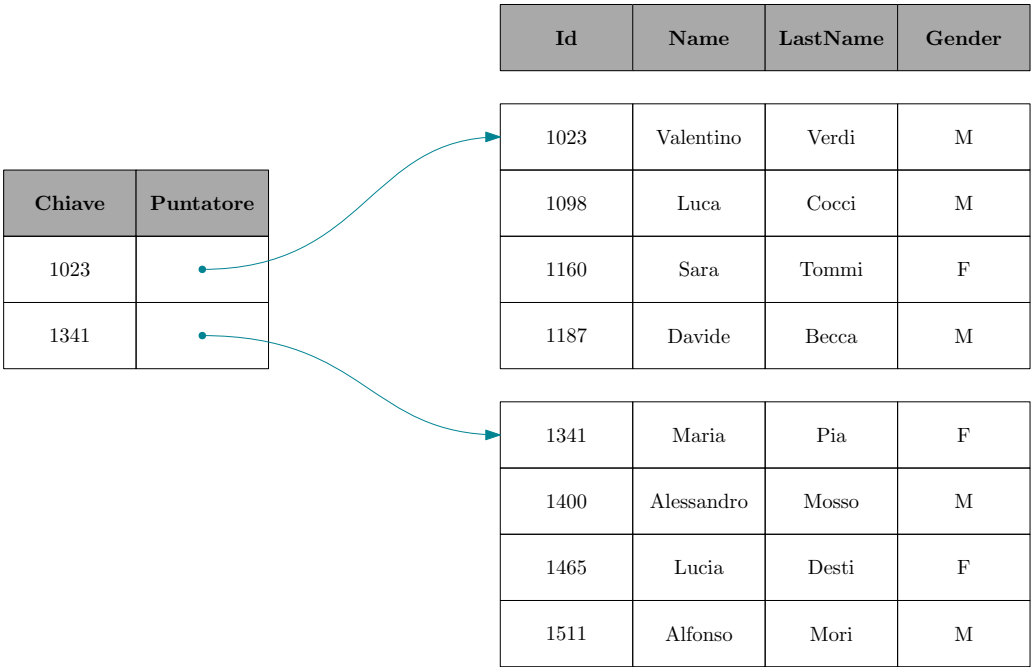
La lista degli indici può essere organizzata in due modi :

- **Indici densi** - Vi è una index entry per ogni possibile chiave di ricerca.
- **indici sparsi** - Vi sono index entry, solo per alcuni record, più record son divisi in gruppi, e gli indici si riferiscono quindi a diversi gruppi.

Ogni intervallo corrisponde ad un blocco di memoria sul disco. La ricerca di un record può avvenire sia in maniera lineare, binaria, oppure basata sugli indici :
n = numero dei blocchi, ni = numero di blocchi per indice, si ha che $ni < n$.

Ricerca lineare	$O(n)$
Ricerca binaria	$O(\log_2(n))$
Ricerca sugli indici (TR)	$O(\log_2(ni))$

Esempio di index sparsi :



Chiave	Puntatore
1023	
1341	

Id	Name	LastName	Gender
1023	Valentino	Verdi	M
1098	Luca	Cocci	M
1160	Sara	Tommi	F
1187	Davide	Becca	M
1341	Maria	Pia	F
1400	Alessandro	Mosso	M
1465	Lucia	Desti	F
1511	Alfonso	Mori	M

5.2.5 Tree-Structured Index

Sappiamo che un albero è un tipo di grafo privo di cicli, con un nodo radice ed i vari nodi che si sviluppano in una gerarchia "parentale" fino a giungere alle foglie. Chiamiamo un albero *bilanciato* o *B-Tree*, se tutti i suoi nodi fogli si trovano allo stesso livello, ci occuperemo di considerare un organizzazione dei record che si rifà al concetto di albero m-ario bilanciato, implementato fisicamente con nodi e puntatori.

Un albero m-ario, è una generalizzazione degli alberi binari, che ci permette di ordinare in un certo modo i nostri record in base alle chiavi, per poter eseguire una ricerca m-aria, decisamente più efficiente della ricerca binaria.

Ogni nodo, contiene $n-1$ chiavi ed n puntatori, son ordinati in memoria in modo da alternarsi, vi è prima un puntatore, e poi una chiave, fino all'ultimo elemento che è un puntatore. Ogni puntatore è associato alla chiave che lo segue, e le chiavi sono ordinate in maniera crescente.

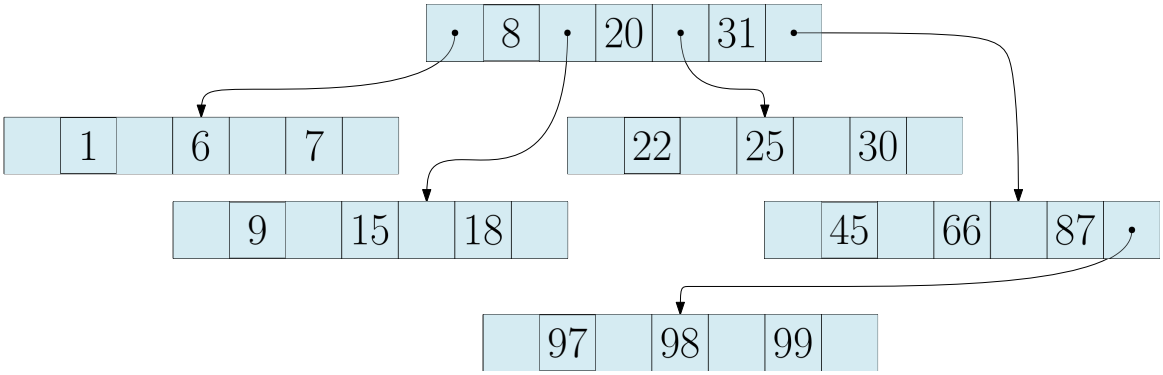
P_0	K_0	P_1	K_1	$\cdots$ $\cdots$	P_{n-1}	K_{n-1}	P_n

P_i è il puntatore associato alla chiave K_i

Ogni nodo quindi, ha n puntatori, che faranno riferimento agli altri nodi.

L'ordinamento è strutturato nel modo seguente : Ogni chiave all'interno di un nodo, è sempre maggiore della chiave alla sua sinistra, son quindi ordinate in maniera crescente $K_i < K_{i+1}$, ogni puntatore P_i associato alla chiave K_i, punta ad un nodo, in cui tutte le chiavi sono

minori della chiave K_i, garantendo quindi un ordinamento in tutto l'albero. Il puntatore alla fine del nodo, seguente alla chiave K_{n-1}, che non è associato a nessuna chiave, punterà ad un nodo, in cui la prima chiave è maggiore della chiave K_{n-1}, quindi per transitività, di tutte le altre chiavi del nodo "puntante". Si veda il seguente esempio :



Ogni nodo, equivale ad un blocco di memoria, quindi chiameremo m il numero massimo di puntatori che può avere ogni nodo, che è dato appunto dalle dimensioni fisiche del blocco, m è detta la *larghezza* dell'albero. Per garantire l'efficienza nella ricerca, si vogliono imporre delle restrizioni per far si che l'albero sia sempre bilanciato.

1. Ogni nodo deve avere al massimo m figli.
2. Ogni nodo (escluse foglie e radice) deve avere al minimo $m/2$ figli (ogni nodo deve essere riempito almeno fino a metà).
3. Il nodo radice deve avere almeno 2 figli.
4. Tutte le foglie sono allo stesso livello (l'immagine di sopra viola tale vincolo).

Nella rappresentazione precedente, è stato omesso che ogni nodo contiene per ogni chiave, un ulteriore puntatore ai dati del record identificato da quella chiave, essi son però immagazinati da un altra parte, e non all'interno dell'albero.

La ricerca avviene in maniera ricorsiva partendo dalla radice, ricercando la chiave desiderata. Se essa non è nel nodo corrente, si cercherà nel nodo selezionato dal puntatore associato alla chiave minore del nodo, più grande del valore ricercato. Se il valore ricercato è maggiore a tutte le chiavi del nodo, si andrà a cercare nel nodo selezionato dal puntatore finale più a destra.

La ricerca sarà molto efficiente, nel caso peggiore si controllano tanti nodi quanto è l'altezza dell'albero.

$$N = \text{Numero di nodi nell'albero}$$
$$m = \text{numero massimo di figli che un nodo può avere}$$
$$d = m/2 = \text{numero minimo di figli che un nodo può avere}$$

- Caso migliore (altezza minimale) : $h_{min} = \log_m(N+1) - 1$
- Caso peggiore (altezza massimale) : $h_{max} = \log_d(\frac{N+1}{2})$

6 Controllo della Concorrenza

Nel DBSM, più utenti possono accedere allo stesso file, sappiamo che la CPU esegue un solo processo alla volta, quindi più utente che faranno richiesta nello stesso momento saranno gestiti in maniera concorrente, ciò può essere radice di svariati problemi, è quindi necessario che l'accesso concorrenziale sia controllato.

6.1 Schedule e Transazioni

Una **transazione** è una sequenza di istruzioni ordinate da eseguire in maniera atomica, ciò vuol dire che le istruzioni non possono essere eseguite in modo parziale, o si completa l'intera transazione, o essa non viene eseguita.

Le transazioni devono essere:

- **Consistenti**, ossia devono avvenire su un istanza legale della base di dati, e dopo essere operate, la base di dati deve trovarsi ancora in uno stato legale.
- **Isolate** ed indipendenti dalle altre transazioni.
- **Durabili**, nel senso che gli effetti di esse devono essere tangibili e non devono andare persi.

Definiamo con il nome di **Schedule**, un certo insieme T di transazioni, le cui singole istruzioni sono ordinate in un certo modo, con il vincolo che, esse mantengano l'ordinamento per singola istruzione. Ad esempio, uno schedule delle due transazioni T_1 e T_2 può vedere le istruzioni di esse ordinate in un qualsiasi modo, a patto che le istruzioni di T_1 e T_2 siano ordinate fra loro rispetto l'ordinamento originale.

$$T_1 = \{o_1, o_2, o_3, o_4\}$$

$$T_2 = \{i_1, i_2, i_3, i_4\}$$

$$schedule(T_1, T_2) = \{o_1, i_1, i_2, o_2, i_3, o_3, i_4, o_4\} \text{ valido}$$

$$schedule(T_1, T_2) = \{o_1, i_2, i_3, o_2, i_1, o_3, i_4, o_4\} \text{ non valido}$$

Gli schedule **seriali** sono ottenuti *permutando* le transazioni, senza interfogliarle fra loro. Un insieme di n transazioni può comporre $n!$ schedule seriali, ad esempio, i due possibili schedule seriali delle transazioni T_1 e T_2 sono :

$$schedule(T_1, T_2) = \{i_1, i_2, i_3, i_4, o_1, o_2, o_3, o_4\}$$

$$schedule(T_1, T_2) = \{o_1, o_2, o_3, o_4, i_1, i_2, i_3, i_4\}$$

Gli schedule seriali sono importanti quando si parlerà di definire un corretto interfogliamento fra transazioni. Si noti come uno schedule seriale, non vede una vera e propria concorrenza, dato che le transazioni sono semplicemente eseguite in maniera sequenziale. Lo scopo principale sarà definire degli interfogliamenti fra transazioni di uno schedule, in modo che abbiano gli stessi effetti di un possibile schedule seriale.

6.1.1 Errori sui Dati

Quando si interfogliano più transazioni in uno schedule, è possibile che esse leggano o scrivano sullo stesso dato/ sulla stessa istanza, vediamo quali sono i 3 tipi di errori che possono causarsi da uno schedule non corretto.

Considereremo diversi schedule basati sulle seguenti transazioni T_1 e T_2 e T_2':

T_1	T_2
read(X)	
X=X-N	
	read(X)
	X=X+M
write(X)	
read(Y)	
	write(X)
Y=Y+N	
write(Y)	

Questo tipo di problema è noto come **aggiornamento perso**, dato che le operazioni fatte da T_1 sull'attributo X, non vengono considerate nelle operazioni fatte da T_2, che a sua volta modificherà X, e lo salverà scrivendo il dato, dopo la scrittura fatta da T_1, facendo si che quest'ultima non venga considerata.

T_1	T_2
read(X)	
X=X-N	
write(X)	
	read(X)
	X=X+M
read(Y)	
ERRORE	
	write(X)

Questo problema è detto **dato sporco**, in quando l'operazione fatta da T_1 fallisce, comportando il *roll back* delle operazioni, ma T_2 ha ormai letto il dato scritto da T_2, aggiornando di conseguenza X, facendo si che T_1 abbia comportato delle modifiche dovute ad una sua esecuzione parziale e non totale.

T_1	T_2'
	S=0
read(X)	
X=X-N	
write(X)	read(X)
	S=S+X
	read(Y)
	S=S+Y
read(Y)	
Y=Y+N	
write(Y)	

La transazione T_2' deve calcolare la somma di X ed Y, ma questi ultimi due, vengono aggiornati in T_1 in maniera concorrente, l'interfogliamento errato compromette il valore della variabile S, causanto un **aggregato non corretto**.

Siamo portati a dire che questi tre schedule sono **non corretti**, perchè gli effetti che hanno sulla base di dati, non sono equivalenti agli effetti che avrebbe uno schedule seriale. Voglio assicurarmi che uno schedule interfogliato sia equivalente ad uno schedule seriale, se ciò è vero, diremo che lo schedule è **serializzabile**.

Come possiamo però definire due schedule equivalenti o meno? È necessaria una definizione : Diremo che due valori (ossia gli attributi modificati dalle transazioni) sono **uguali**, se e solo se sono il risultato della stessa sequenza di operazioni.

Diremo che due schedule di un insieme di transazioni sono **equivalenti**, se e solo se, per ogni dato da essi modificato, producono valori *uguali*. Definiremo quindi uno schedule *corretto* o *serializzabile*, se esso è equivalente ad uno qualsiasi degli schedule seriali.

6.2 Protocolli e Lock a 2 Stati

Piuttosto che controllare se un certo schedule sia serializzabile (in quanto risulta essere un operazione dispendiosa), è possibile introdurre dei protocolli per anticipare la possibile creazione di schedule non corretti. Creeremo gli schedule seguendo dei protocolli che garantiscano la serializzabilità.

Definiamo con **item**, l'unità il cui accesso concorrente è controllato. Negli esempi precedenti, le transazioni operavano su attributi come X o Y, le transazioni adesso opereranno sugli item, la cui dimensione è variabile, possono essere singoli attributi, tuple o anche l'intera base di dati. Le dimensioni degli item sono definite in base all'uso. Se le dimensioni sono grandi, la gestione della concorrenza è più efficiente, una dimensione piccola invece aumenta il livello di concorrenza, appesantendo però il carico computazionale sul sistema.

6.2.1 Definizione di Lock

Il significato euristico di una **lock**, è quello di essere un *privilegio* di accesso ad un singolo *item* X. L'implementazione avviene tramite una variabile booleana, è quindi necessario un singolo bit. Ad un certo item X, è associata una lock, che può appunto assumere 2 stati, lo stato *locked* (bloccato) ed *unlocked* (disponibile).

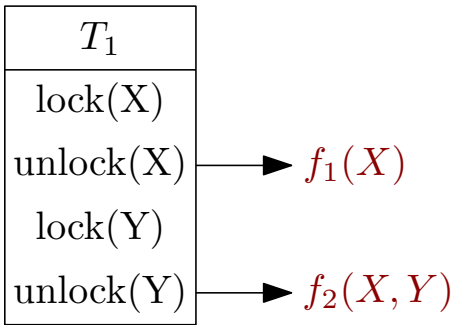
Quando una transazione vuole fare un operazione su un certo item X, dovrà richiederne la lock, se essa è disponibile, la transazione bloccherà l'item X, rendendolo locked, in modo tale che le altre transazioni non possano accedervi, de facto, se una transazione vuol accedere ad un item locked, dovrà attendere che la transazione che ha in uso quell'item lo rilasci.

Utilizzando le lock, evitiamo che due transazioni agiscano contemporaneamente sullo stesso dato cercando di modificarlo, le transazioni hanno ha disposizione le funzioni `lock(X)` e `unlock(X)`, ogni volta che una transazione vuole leggere o scrivere su un item, ne deve richiedere l'accesso con il metodo `lock`, per poi rilasciarlo con il metodo `unlock` dopo che ha terminato con l'utilizzo.

Definiremo uno schedule **legale**, se ogni sua transazione effettua un lock prima di leggere/scrivere su un item, per poi rilasciare ogni lock ottenuto. L'utilizzo di tale protocollo *risolve* totalmente il problema dell'aggiornamento perso.

Definiamo adesso il template nella quale rientrerà ogni transazione. Una transazione generica, non fa altro che leggere, operare, ed eventualmente scrivere su un certo insieme di item. Abbiamo detto che una transazione appartenente ad uno schedule legale deve chiedere la lock

di un item prima di operarci sopra, e rilasciarla dopo aver terminato. Approssimiamo quindi ogni transazione ad un insieme di coppie `lock` - `unlock`, che lavorano su un item. Ad ogni coppia sarà quindi associata in modo univoco una funzione f_i, che avrà come parametri, tutti gli item letti (quindi sulla quale si è richiesto un lock) dalla transazione in precedenza.



Utilizzeremo questo template per dire che due schedule sono equivalenti se le funzioni che danno i valori finali per ciascun item sono uguali. Vediamo un esempio, consideriamo le due transazioni :

T_1	
lock(X)	
unlock(X)	$f_1(X)$
lock(Y)	
unlock(Y)	$f_2(X, Y)$

T_2	
lock(Y)	
unlock(Y)	$f_3(Y)$
lock(X)	
unlock(X)	$f_4(X, Y)$

Consideriamo ora il seguente schedule, X_0 è il valore iniziale di X ed Y_0 è il valore iniziale di Y. Si hanno le relative formule per i valori finali di X ed Y :

T_1	T_2	
lock(X)		
unlock(X)		$X = f_1(X_0)$
	lock(Y)	
	unlock(Y)	$Y = f_3(Y_0)$
lock(Y)		
unlock(Y)		$Y = f_2(X_0, f_3(Y_0))$
	lock(X)	
	unlock(X)	$X = f_4(f_1(X_0), Y_0)$

In questo schedule, il valore finale di X è prodotto dalla formula $f_4(f_1(X_0), Y_0)$, ed il valore finale di Y è prodotto dalla formula $f_2(X_0, f_3(Y_0))$. Questo schedule sarà quindi serializzabile e corretto se e solo se esiste uno schedule seriale equivalente, ossia, le cui formule finali per X ed Y sono identiche a queste due appena presentate. Con una breve verifica, si nota che le formule per X ed Y prodotte dai due schedule seriali sono differenti, questo schedule è quindi non corretto. Basta che le formule siano diverse per un solo item per concludere che gli schedule non sono equivalenti.

Detto ciò, possiamo affermare che uno schedule è equivalente ad uno schedule seriale se, per ogni item in cui le varie transazioni fanno un lock, segue un ordine identico a quello di uno schedule seriale. È possibile, fatte tali premesse, costruire un metodo algoritmico per testare la serializzabilità di uno schedule.

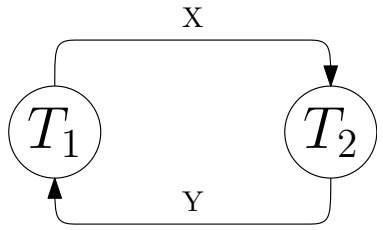
6.2.2 Grafo di Serializzazione

Dato uno schedule S di n transazioni, costruiamo un grafo G nel seguente modo :

- I nodi di G sono le transazioni $T_1, T_2 \ldots, T_n$ della schedule S.
- Ci sarà un arco $T_i \rightarrow_X T_j$ con etichetta X se la transazione T_i esegue un unlock sull'item X, e la transazione T_j esegue il successivo lock su X.

T_1	T_2
lock(X)	
unlock(X)	
	lock(Y)
	unlock(Y)
lock(Y)	
unlock(Y)	
	lock(X)
	unlock(X)

ha grafo :



Teorema : Se il grafo G associato allo schedule S ha un ciclo, allora S *non* è serializzabile. Se G non ha un ciclo, applicando *l'ordinamento topologico* su G, si ottiene lo schedule seriale S' equivalente ad S. Uno schedule è serializzabile se e solo se il suo grafo di serializzazione è aciclico.

Il cosiddetto **ordinamento topologico**, è un operazione ricorsiva da fare su un grafo aciclico, nel quale si rimuove un nodo che non ha archi entranti, insieme ai suoi archi nascenti. Un grafo può ammettere più ordinamenti topologici.

6.2.3 Protocollo a Due Fasi

Diciamo che una transazione è *a due fasi* se prima effettua tutte le operazioni di lock, poi tutte le operazioni di unlock. Il *protocollo a due fasi* impone a tutte le transazioni di uno schedule di essere, appunto, a due fasi.

Teorema : Se ogni transazione di uno schedule S è a due fasi, allora S è serializzabile.

Dimostrazione : Poniamo per assurdo che S abbia solo transizioni a due fasi e che non sia serializzabile. Quest'ultima condizione implicherebbe che nel grafo c'è un ciclo. Ciò implica che c'è una transizione T_j che esegue un operazione di unlock su un attributo (ha un arco uscente), ma poi ri-esegue un operazione di lock su un altro attributo (arco entrante), dopo che un altra transazione vi ha eseguito un operazione di unlock, ciò va in contraddizione con l'ipotesi che ogni transazione di S sia a due fasi. ■

In conclusione, possiamo dire che se tutte le transazioni di uno schedule sono a due fasi esso è sicuramente serializzabile, possono però esistere schedule che non hanno le transazioni a due fasi, che siano comunque serializzabili.

6.3 Lock a 3 Stati

Possiamo aumentare il grado di concorrenza facendo alcune considerazioni. Ora come ora, il privilegio di accesso per un item viene richiesto senza considerare quali operazioni faremo su quell'item. Una transazione, può voler accedere ad un item in sola lettura, senza necessariamente modificarlo, introduciamo quindi un nuovo tipo di lock, detto *a 3 stati*, in cui un certo item può essere *wlocked* (locked in scrittura), *rlocked* (locked in lettura), o unlocked.

Le transazioni hanno a disposizione ora, al posto del metodo `lock`, i metodi `rlock` e `wlock`. Il metodo `rlock(X)` viene richiesto quando si ha intenzione di leggere ma non modificare l'item X, in questo modo, tutte le altre transazioni, potranno accedere ad X per leggerlo, ma *non* per modificarlo. Il metodo `wlock(X)` viene richiesto da una transazione che vuole modificare X, se un item è in questa modalità, le altre transazioni non potranno accedervi ne in lettura ne in scrittura. Entrambi gli stati vengono rilasciati con il metodo `unlock(X)`.

Valore lock su X	T richiede un :	Risultato
unlocked	`rlock(X)`	T ottiene il lock in lettura ed X diventa rlocked.
rlocked	`rlock(X)`	T ottiene il lock in lettura
wlocked	`rlock(X)`	T attende il rilascio
unlocked	`wlock(X)`	T ottiene il lock in scrittura ed X diventa wlocked.
rlocked	`wlock(X)`	T ottiene il lock in lettura
wlocked	`wlock(X)`	T ottiene il lock in lettura

La concorrenza è maggiore in quanto più transazioni adesso possono accedere contemporaneamente in lettura ad un item. Adesso il template di uno schedule è quello di una sequenza di operazione `rlock`, `wlock` e `unlock`.

- Sia un `rlock(X)` che un `wlock(X)` implicano la lettura di X.
- Ogni `unlock(X)` associato ad una `wlock(X)` implica la scrittura di X.

T_1
rlock(X)
unlock(X)
wlock(Y)
unlock(Y)

$f_1(X, Y)$

In questo modo l'insieme degli item scritti da una transazione è sottoinsieme degli item letti. In questo nuovo modello, il nuovo valore di un item è calcolato da una funzione associata univocamente ad ogni coppia `wlock` - `unlock`, che ha come parametri tutti gli item precedentemente letti (sulla quale si è eseguito un `rlock` o `wlock`).

Dato il nuovo modello, due schedule si dicono **equivalenti** se :

- Producono lo stesso valore finale per ogni item su cui viene effettuato un `wlock`.
- Ogni operazione `rlock(X)` legge lo stesso valore di X nei due schedule.

La definizione di serializzabilità rimane invariata.

6.3.1 Nuovo Grafo di Serializzazione

In questo nuovo modello, il grafo G viene costruito in tal modo :

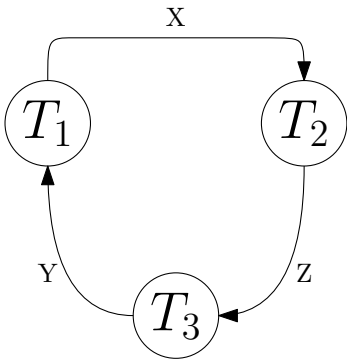
- I nodi di G sono le transazioni $T_1, T_2 \ldots, T_n$ della schedule S.
- Ci sarà un arco $T_i \rightarrow_X T_j$ con etichetta X se in S se si verifica una delle due seguenti :
  - T_i esegue un `rlock(X)` o `wlock(X)` e T_j è la transazione che esegue la successiva `wlock(X)`.
  - T_i esegue un `wlock(X)`, e T_j esegue la successiva `rlock(X)` prima che un altra transazione esegua una `wlock(X)`.

Dato il nuovo grafo, vale comunque il teorema visto nel caso del lock a due fasi, per cui uno schedule è serializzabile se e solo se il suo grafo di serializzazione è aciclico.

La definizione di transazione a due fasi rimane simile, è verificata se una transazione se nessuna operazione di lock (rlock o wlock) segue una operazione di unlock. La presenza di sole transazioni a due fasi implica ancora la serializzabilità di uno schedule.

T_1	T_2	T_3
rlock(X)		
unlock(X)		
	wlock(X)	
	unlock(X)	
		rlock(Y)
		unlock(Y)
	rlock(Z)	
	unlock(Z)	
		wlock(Z)
		unlock(Z)
wlock(Y)		
unlock(Y)		

ha grafo :



non è serializzabile

6.4 Deadlock e Livelock

L'introduzione dei meccanismi di sincronizzazione come i lock è necessaria per poter stabilire dei protocolli, volti a gestire in maniera corretta l'accesso concorrente su uno stesso item, nonostante essi prevengano gli errori tipici precedentemente elencati, possono comportare ulteriori errori, dovuti proprio alla loro natura intrinseca di "bloccare" le risorse.

6.4.1 Stallo delle Transazioni e Grafo di Attesa

Con il termine **deadlock**, si definisce una situazione di *stallo* in cui possono ricadere le transazioni, che comporta il blocco totale dell'esecuzione di una schedule. Si è in una situazione di deadlock quando, ogni transazione di una schedule in esecuzione è in attesa di un item per cui un altra transazione mantiene la lock.

Se ogni transazione è in attesa di un item per rilasciare il proprio, lo stallo sarà definitivo e l'esecuzione sarà interrotta. Esistono sia dei meccanisimi di *risoluzione* di deadlock che di *prevenzione*, è possibile sviluppare un metodo che verifichi la presenza o no di un deadlock in una schedule.

Si costruisce il cosiddetto *grafo di attesa* G_A per una schedule S, definito nel seguente modo :

- I nodi di G_A sono le transazioni $T_1, T_2 \ldots, T_n$ della schedule S.
- Ci sarà un arco $T_i \to T_j$ se la transazione T_i è in attesa di ottenere un lock su un item bloccato e mantenuto da T_j.

Dalla definizione del grafo, è intuibile come la presenza di un ciclo è segnale che si è in una situazione di deadlock. Per ovviare a tale problema, per una delle transazioni presente nello stallo deve avvenire un **roll-back** (si approfondirà in seguito), ossia un operazione che prevede l'interruzione della transazione, il ripristino dei suoi effetti sulla base di dati, ed il rilascio di tutti i lock da essa mantenuti.

Piuttosto che prevenire un deadlock, è più opportuno considerare degli approcci *preventivi*, uno di questi, prevede l'ordinamento degli item disponibili alla lettura/scrittura, per poi imporre alle transazioni di richiedere i lock su tali item seguendo l'ordine stabilito. Tale protocollo impedisce la creazione di cicli nel grafo di attesa.

6.4.2 Situazione di Starvation

Indichiamo con **livelock**, la condizione in cui può ricadere una transazione, in cui essa attende indefinitamente il permesso di accesso ad un item senza che esso venga mai rilasciato, entrando in una cosiddetta condizione di starvation.

Tale problema può essere facilmente risolto, implementando la coda di attesa delle transazioni per una lock con una politica *First-In-First-Out*, oppure adoperando una *coda prioritaria*, aumentando il grado di priorità di una transazione in base al tempo trascorso nella coda, facendo ciò, tutte le transazioni prima o poi potranno acquisire la lock richiesta.

6.5 Roll-back e Punti di Commit

Abbiamo accennato già all'operazione di *abort* di una transazione, essa può avvenire in diversi casi :

1. La transazione esegue un operazione non corretta (ad esempio, divisione per zero).
2. Viene rilevato un deadlock.
3. Si verifica un malfunzionamento hardware o software.

Possiamo considerare i cosiddetti *punti di commit*, un protocollo volto alla prevenzione, essi prevengono il causarsi delle situazioni (1) e (2), in modo che non ci sia bisogno di eseguire roll-back.

Definiamo con punto di commit, un punto in cui una transazione ha ottenuto tutti i lock che gli sono necessari, ed ha effettuato i calcoli in locale. Ridefiniamo quindi con **dati sporchi**, i dati scritti da una transazione sulla base di dati prima che essa abbia raggiunto un punto di commit.

Un roll-back può essere costoso da applicare, in quanto abortendo una transazione T, bisogna ripristinare la base di dati nello stato in cui era prima che T operasse, e bisogna annullare gli effetti delle transazioni che hanno utilizzato i dati "sporcati" da T, creando un cossiddetto *effetto a cascata*.

6.5.1 protocollo a due fasi stretto

Per ovviare al problema dei dati sporchi occorre considerare delle regole più restrittive rispetto a quelle del protocollo a due fasi, introduciamo il **protocollo a due fasi stretto**, una transazione soddisfa tale protocollo se :

- Non scrive sulla base di dati fino a quando non ha raggiunto il suo punto di commit.
- Non rilascia un lock finchè non ha finito di scrivere sulla base di dati.

7 Complementi

7.1 Formulario sull'Organizzazione Fisica

7.1.1 Hashing (Bucket)

- $Puntatori \times Blocco = \lfloor \frac{BlockSize}{PointerSize} \rfloor$
- $BlocchiDellaBucketDirectory = \lceil \frac{NumBucket}{Puntatore \times Blocco} \rceil$
- $Blocchi \times Bucket = \lceil \frac{Record \times Bucket}{Record \times Blocco} \rceil$
- $Record \times Blocco = \lfloor \frac{BlockSize - PointerSize}{RecordSize} \rfloor$
- $Record \times Bucket = \lceil \frac{NumRecord}{NumBucket} \rceil$
- $AvgTime = \lceil \frac{Blocchi \times Bucket}{2} \rceil$

7.1.2 B-Tree

- $Record \times Blocco = \lfloor \frac{BlockSize}{RecordSize} \rfloor$

Se lo volessi pieno al minimo : $Record \times Blocco = \lceil \frac{{}^{BlockSize}/_{2}}{RecordSize} \rceil$

- $NumBlocchiMainFile = \lceil \frac{NumRecord}{Record \times Blocco} \rceil$
- $Key \times Blocco = \lfloor \frac{BlockSize - PointerSize}{PointerSize + KeySize} \rfloor$

Se lo volessi pieno al minimo $Key \times Blocco = \lceil \frac{{}^{BlockSize}/_{2} - PointerSize}{PointerSize + KeySize} \rceil$

- I blocchi totali sono la somma dei blocchi in ogni livello.
- Il tempo di accesso necessario è l'altezza dell'albero.

7.1.3 ISAM

- $Record \times BloccoIndex = \lfloor \frac{BlockSize}{PointerSize + KeySize} \rfloor$
- $Record \times Blocco = \lfloor \frac{BlockSize}{RecordSize} \rfloor$
- $TotBlockMain = \lceil \frac{NumRecord}{Record \times Blocco} \rceil$
- $TotBlockIndex = \lceil \frac{TotBlockMain}{Record \times BloccoIndex} \rceil$
- $TempoAccesso = \log_2(TotBlockIndex) + 1$