Basi di dati semplici (per davvero)

Rovesti Gabriel

Attenzione

Il file non ha alcuna pretesa di correttezza; di fatto, è una riscrittura attenta di appunti, slide, materiale sparso in rete, approfondimenti personali dettagliati al meglio delle mie capacità. Credo comunque che, per scopo didattico e di piacere di imparare (sì, io studio per quello e non solo per l'esame) questo file possa essere utile. Semplice si pone, per davvero ci prova.
Thank me sometimes, it won't kill you that much.

Gabriel

Sommario

Introduzione alle basi di dati e concetti base/Capitolo 1

Le *basi di dati* sono un insieme consistente, persistente, organizzato e condiviso dei dati utilizzati. Al di là di questo, si parla di dati, entità presente prima di ogni elaborazione e che devono essere interpretate sotto forma di informazioni. I dati stessi codificano le informazioni dando rappresentazioni più precise di informazione e conoscenza, magari per analisi o altri scopi.

Esse sono gestite dai *DBMS (Database Management System)* garantendo privacy, affidabilità, efficienza ed efficacia. In generale i database possono contenere molti dati, ammettiamo per esempio terabyte di dati o miliardi di record (*grande*), indipendente dalla singola esecuzione di un programma (*persistente*), dando appunto ai vari utenti accesso in vari modi alle porzioni della base di dati acceduta (*condivisa*).
Naturalmente si deve garantire ridondanza, informazioni ripetute e che devono essere mantenute in maniera sicura per evitare incoerenze. A tale scopo, attenzione al controllo della concorrenza, magari anche con utilizzo di strutture come lock o semafori, assieme a meccanismi specifici di autenticazione.

L'uso dei DBMS cerca di garantire *l'affidabilità*, resistendo a malfunzionamenti hardware/software, gestendo le *transazioni*, quindi un insieme di operazioni che modificano lo stato di una base di dati. Prendiamo l'esempio di trasferimento di fondi, classico esempio del prelievo/versamento; quindi, rimozione e/o aggiunta tra basi di dati, magari anche in contemporanea; qui si vede che l'ordine delle operazioni è fondamentale.

Naturalmente le risorse devono essere bastevoli tali che il sistema informativo stesso possa essere considerato efficiente. A noi interessa di più l'efficacia rispetto all'efficienza.
I dati sono organizzati normalmente secondo modelli di dati utilizzati per organizzare dati di interesse e quindi descriverne la dinamica, quindi con il modello relazionale, creando degli insiemi di record omogenei.

Ogni riga è una *tupla* (prodotto cartesiano di n domini, matematicamente parlando), combinando tra di loro tutti i possibili elementi; in altri termini è un singolo elemento di un database relazionale caratterizzato da uno o più attributi. Un esempio è:

Esempio di (porzione di) base di dati

Orario

Insegnamento	Docente	Aula	Ora
Analisi matem. I	Luigi Neri	N1	8:00
Basi di dati	Piero Rossi	N2	9:45
Chimica	Nicola Mori	N1	9:45
Fisica I	Mario Bruni	N1	11:45
Fisica II	Mario Bruni	N3	9:45
Sistemi inform.	Piero Rossi	N3	8:00

Insegnamento = String
Docente = String
Aula = {N1, N2, N3}
Ora = {0:01,0:02,…,23:58,23:59}
Orario ⊆ Insegnamento X Docente X Aula X Ora

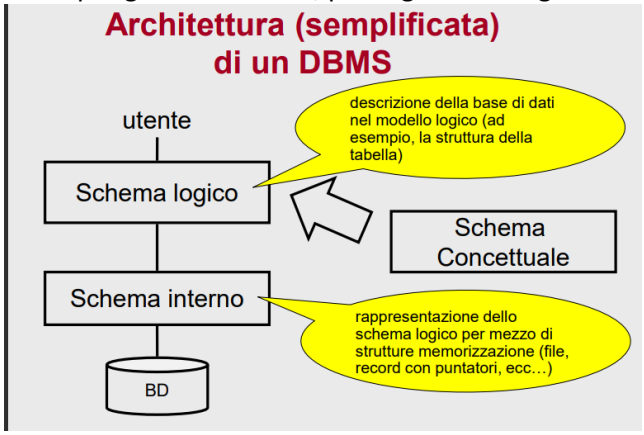
In ogni base di dati vi è lo *schema*, che descrive la struttura di un database (aspetto intensionale, quindi che non cambia nel tempo, es. intestazioni delle tabelle) e l'*istanza*, con i valori attualmente utilizzati e che possono cambiare anche molto rapidamente (ad es. il corpo di ciascuna tabella).

In generale avremo due tipi di modelli:

1) *logici*, adottati dai DBMS organizzando i dati a livello logico, sono utilizzati dai programmi e sono indipendenti dalle strutture fisiche. Possiamo per esempio enunciare modelli di tipo relazionale, reticolare, gerarchico, ad oggetto, XML, ecc.

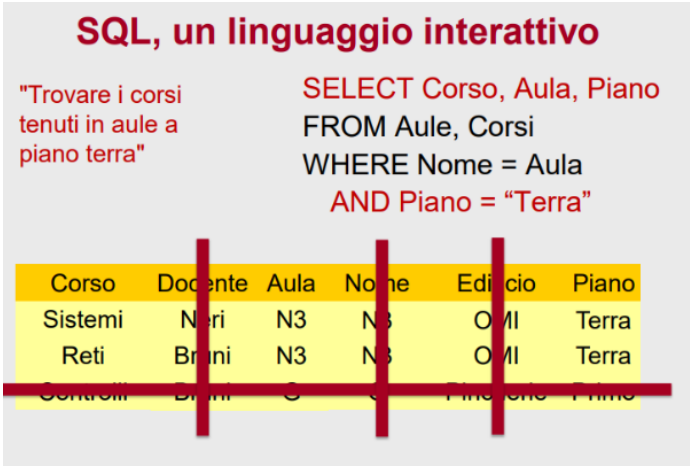
2) *concettuali*, rappresenta i dati in modo indipendente, cercando di descrivere i concetti reali e spesso usati nelle fasi preliminari di progettazione. Il modello in questo senso più diffuso è il cosiddetto *Entity-Relationship.*

A tale scopo definiamo così l'architettura, intendendo come utente ciò che viene creato dalla macchina finale per gestire tutto ciò; più in generale segue lo schema ANSI/SPARC (considerato standard).



A noi interessa soprattutto il livello *logico* piuttosto che quello *fisico*, indipendenti l'uno dall'altro. Si accenna anche lo scherma *esterno* (schema logico e interno visibile da altre entità). Se io volessi verificare una serie di informazioni, esso verrebbe fatto tramite una *vista*, vedendo una certa porzione di database in un certo momento con una query, ad esempio, e ciò viene fatto proprio per semplificarle e garantire privacy nel loro accesso.
Altro contributo all'efficacia delle basi di dati tramite linguaggi testuali interattivi (SQL), o anche comandi immersi in un linguaggio ospite (Pascal, Java, C, interagendo con SQL), tramite interfacce amichevoli (quindi non testuali).

Un esempio di interrogazione/query fatta per mezzo di SQL:



La nostra interazione grafica verrà fatta con POSTGRES. Nel caso pratico, anche cambiando sistemi di interazioni con le basi di dati, è possibile agire ugualmente sul sistema. La gestione è centralizzata e ha possibilità di standardizzazione purché si usino linguaggi simili al SQL standard. Questo funziona bene nelle grandi organizzazioni, ma in caso di pochi utenti vanno anche bene file semplici.
Le operazioni possibili su una base di dati avvengono solitamente tramite *DML (Data Manipulation Language)*, interrogando e/o aggiornando istanze delle basi di dati e *DDL (Data Definition Language)*, definendo schermi (logici, esterni, fisici) e altre operazioni generali (esempio, CREATE TABLE in SQL).

In generale i DBMS permettono ai dati di essere una risorsa comune, condivisa nell'organizzazione, centralizzata con possibilità di standardizzazione, riducendo inconsistenze/ridondanze e favorendo lo sviluppo indipendente delle applicazione, mantenendo i dati separati ma condivisi in un'organizzazione

Scritto da Gabriel

comune. Nel caso di sistemi informativi non particolarmente grandi non conviene adottare un DBMS, perché complesso e costoso; ciò è più consigliato nel caso di medio/grandi organizzazioni, purché si abbia una organizzazione concorrente, stabile e adeguata.

Modello relazionale/Capitolo 2

Il *modello relazionale* si basa sul concetto di *relazione*, quindi naturale rappresentazione per mezzo di tabelle. Abbiamo quindi n insiemi anche non distinti, descritti come $D_1 \ldots D_n$ da cui leghiamo i singoli elementi sotto forma di relazione matematica. Non viene violato il concetto di insieme, quindi nelle n-uple non ci sta un ordine prestabilito, le n-uple sono distinte e ogni n-upla è ordinata, cioè l'i-esimo valore proviene dall'i-esimo dominio; questo non è sempre vero, potendo adattare la struttura posizionale alle esigenze dei sistemi informativi.

Ogni dominio/colonnna è associato ad un nome unico (attributo) e, più in sintesi, riassumiamo questi fatti:
1. l'ordinamento tra le righe è irrilevante
2. l'ordinamento tra le colonne è irrilevante
3. le righe sono diverse fra loro
4. le intestazioni delle colonne sono diverse tra loro
5. i valori di ogni colonna sono definiti su domini omogenei

Ovviamente esistono dei legami tra i valori, in cui dati diversi vengono rappresentati per mezzo di domini di valori. Un esempio possibile è il collegamento per valore, con due colonne tra tabelle diverse che possiedono lo stesso valore e quindi hanno una correlazione logica. Essendo un dominio ciascuna tupla avrà una sua corrispondenza precisa e corrispondente.
Parliamo quindi di *relazione* ($R(A_1, \ldots, A_n)$) tra attributi, rappresentato sotto forma matematica, per poi raggruppare un insieme di relazione sotto forma di *schema di base di dati* ($R = \{R_1(X_1) \ldots, R_k(X_k)\}$)

Un esempio di istanza e tupla, concetti molto legati tra di loro, è il seguente:

Instanze / 1

Data la relazione $R(A_1, \ldots, A_n)$
Sia V_i il dominio dei valori dell'attributo A_i.

- A tupla r su R è una funzione
  $r: \{A_1, \ldots, A_n\} \rightarrow (V_1 \cup \ldots \cup V_n)$
  dove $r(A_i) \in V_i$.

Studente	Voto	Corso
3456	30	04

} Tupla

L'insieme di tuple o elementi di una tabella è chiamata *istanza di una relazione*, mentre l'insieme delle tuple cambierà nel tempo e con esse anche le relazioni tra le tuple, conseguentemente parliamo di *istanza di base di dati* descrivendo l'insieme di relazioni. Non è possibile che un attributo, già di per sé tupla, contenga più di una riga. Si nota che le informazioni possono essere duplicate, che rappresenta di per sé uno spreco di spazio ed è un rischio forte di incoerenza.

Ecco quindi che l'idea migliore può essere di spezzettare le informazioni tra più tabelle, collegando per esempio tra di loro i singoli valori. Un'altra idea è l'aggiunta di attributi ulteriori solidificando la struttura

(ad esempio, nel caso degli scontrini aggiungo il numero di riga oppure l'ora, giusto per dare un'idea). Non tutte le informazioni sono sempre subito disponibili; una soluzione può essere l'utilizzo di un valore nel dominio ("0", "99", stringa nulla, ecc.); tuttavia potrebbero esistere valori "non utilizzati" e quindi implementare controlli appositi, indebolendo il design di un database.

La soluzione, quindi, è l'utilizzo di un *valore nullo*, denotante l'assenza di un valore del dominio e di cui non ne fa parte; si possono anche aggiungere restrizioni eventualmente dicendo che non ci può essere un certo valore. Attenzione che i DBMS possono anche considerare valori nulli come valori sconosciuti, inesistenti oppure anche senza informazione.
Da questo punto di vista, *attenzione a non avere troppi valori nulli* (es. delle slide, tutti gli studenti devono avere una matricola, un voto non può essere dato ad uno studente sconosciuto senza corso, ecc.)

Oltre che *sintatticamente corretta* (cioè tutte le tuple sono diverse*)*, una istanza di una Base di dati può essere *semanticamente scorretta* (cioè dati impossibili per l'applicazione di interesse).
Per poter garantire entrambe queste proprietà, si usano i vincoli di integrità, funzioni booleane che restituiscono i valori vero/falso.
Abbiamo quindi due casi, quindi vincoli "supportati" dai DBMS nativamente (metti il caso in cui il DBMS "sappia già" che uno studente non può essere vuoto e che non esista il 27 e lode), rigettando tuple che violano i vincoli, oppure vincoli "non supportati" nativamente, in cui l'applicazione deve garantire il non inserimento di dati non conformi.

I vincoli che si riferiscono ad una sola relazione/tabella sono i *vincoli intrarelazionali*, oppure possono essere *interrelazionali*, coinvolgenti invece più tabelle. Altri esempi di vincoli sono i *vincoli di dominio*, che esprimono condizioni sui valori di ciascuna tupla, indipendentemente dalle altre, i *vincoli di tupla* che riguardano le singole righe/tuple (esempio banale la somma del netto e delle ritenute fiscali che deve ammontare al lordo), i *vincoli di chiave*, dove la chiave deve essere sempre univoca e con valori diversi su tutti gli altri attributi non chiave.

Non ci sono due tuple con lo stesso valore di chiave naturalmente (per esempio due matricole magari); esiste anche il concetto di *superchiave*, cioè se in una relazione non ci sono due tuple con gli stessi valori per tutti gli attributi dell'insieme considerato. Una superchiave identifica le tuple di una relazione; se tolgo un attributo, una superchiave rimane tale.

Una *chiave* è quindi una superchiave minimale (formata da un solo attributo) e una superchiave è chiave se non ha altre superchiavi al suo interno. Una chiave non può essere sottoinsieme di un'altra chiave e non esiste una chiave che coinvolga tutti gli attributi; ciò può accadere invece per la superchiave. *Ogni tabella ha sempre almeno una superchiave*, in quanto non ci devono essere attributi duplicati e l'insieme stesso degli attributi della tabella è considerabile superchiave; ovviamente, *una tabella deve avere anche almeno una chiave*.

Nell'esempio sotto, la chiave è il campo Matricola nel caso di studenti universitari, in quanto è superchiave ed è minimale. Analogamente, Nome/Cognome/Natricola è superchiave ed è anch'essa minimale.

Matricola	Cognome	Nome	Corso	Nascita
27655	Rossi	Mario	Ing Inf	5/12/98
78763	Rossi	Mario	Ing Inf	3/11/96
65432	Neri	Piero	Ing Mecc	10/7/99
87654	Neri	Mario	Ing Inf	3/11/96
67653	Rossi	Piero	Ing Mecc	5/12/98

Si nota quindi come i vincoli devono corrispondere alla realtà di interesse; in questo caso specifico, vista l'implementazione, un'altra possibile chiave e corretta è la coppia Cognome/Corso, chiave "per caso" perché solo in questo contesto funziona.

Scritto da Gabriel

Esercizio

Considerare le informazioni per la gestione dei prestiti di una **biblioteca personale.**

Il proprietario presta libri
- ai suoi amici che indica semplicemente attraverso i rispettivi **nomi** o soprannomi (così da evitare omonimie)
- fa riferimento ai libri attraverso i **titoli** (non possiede 2 libri con lo stesso titolo).

Quando presta un libro, prende nota della **data prevista** di restituzione.

1) Definire uno schema di relazione per rappresentare queste informazioni
2) Individuare opportuni domini per i vari attributi
3) Mostrare un'istanza in forma tabellare.
4) Indicare la chiave (o le chiavi) della relazione.

Soluzione:

Queste informazioni possono essere rappresentate da **una sola relazione** contente i prestiti, perché non ci sono altre informazioni su amici e libri oltre ai nomi e ai titoli.

Assunzione: Una persona non può prendere in presto lo stesso libro più di una volta

Un possibile schema è il seguente:

PRESTITO (Titolo, Nome, DataRestPres, DataRestEffe)

- Titolo: il titolo del libro
- Nome: il nome o il soprannome dell'amico
- DataRestPres: data di restituzione prevista del libro
- DataRestEffe: data di restituzione effettiva del libro (initialmente NULL)

Già dagli esempi sopra si nota che non esiste una soluzione univoca, in quanto deve corrispondere alla logica utilizzata; in particolare una soluzione va bene se non viola i vincoli stabiliti di dominio.

Le *chiavi* quindi identificano le tuple di una relazione e sono utilizzate per *referenziare le altre tabelle*. Diciamo anche che una *chiave* è tale *se non ci sono sottoinsiemi considerabili superchiave*. Tuttavia, le chiavi possono avere valori *nulli*. In presenza di valori nulli, naturalmente, la chiave non identifica tuple e quindi non realizza collegamenti con altre relazioni.

Una chiave che non ha valori nulli perché non ammessi è la *chiave primaria*. Si parla di *integrità referenziale* per correlare tra di loro i singoli valori con relazioni diverse mantenendo una coerenza tra questi e gli attributi hanno almeno una chiave primaria di seconda relazione, cosiddetta *chiave esterna*. Si intende quindi che questa integrità viene mantenuta tra più tabelle; infatti, riguarda la situazione dei vincoli interrelazionali.

È l'esempio della tabella *Infrazioni*, collegata univocamente con la tabella *Vigili* per mezzo della chiave esterna *Matricola*. Analogamente, un'infrazione è collegata ad un auto; da questo si ha il collegamento con una ipotetica tabella *Auto*, avendo come superchiave l'insieme *Stato/Numero*.

Un vincolo di integrità referenziale impone ai valori su X nella relazione R1 di comparire come valori nella chiave primaria di R2; tradotto significa che si collega ad un campo chiave primaria.
In qualche caso, si parla di chiave candidata. Ogni tabella può avere diverse chiavi candidate; per esempio, la tabella "Cliente" può avere due chiavi candidate. "CustomerId" come un singolo campo identificativo univoco oppure la combinazione di "Nome", "Cognome" ed "Email" avrebbe senso se ben contestualizzato. Tipicamente la scelta migliore utilizza un singolo campo come chiave, come ad esempio il caso del campo "CustomerId", in quanto consente di ottimizzare le prestazioni delle query.

Modello referenziale/Capitolo 2 e Algebra Relazionale

Un esempio di vincolo di integrità referenziale non rispettato è la corrispondenza di una chiave esterna su più tuple (nelle slide, si sfrutta il discorso appena descritto Infrazioni/Auto); ciò non può essere situazione valida. I vincoli quindi agiscono come meccanismi di gestione delle singole azioni, prevenendo violazioni.

Rappresentare per mezzo di una o più relazioni le informazioni contenute nell'**orario delle partenze giornaliero** di una stazione ferroviaria:

- numero del treno
- orario di partenza
- destinazione finale
- categoria
- fermate intermedie

Queste informazioni per tutti i treni in partenza.

Esercizio: Individuare tabelle e vincoli

Idea iniziale di implementazione è la creazione di una tabella
Legenda→ chiave esterna – chiave primaria

Treni:

Numero treno	*Destinazione*	*Categoria*	*Orario*	*Ferm. intermedie*
1740	Roma	ES	11	Ferrara
""	Firenze	""	11	""
""	Bologna	""	11	""

Problema: è tutto ridondante, si noti la ripetizione di tutti gli orari, ad esempio.
Soluzione: Attuo la *normalizzazione*, evitando di mantenere un'informazione ripetuta; separo tutto e creo tabelle separate per Treni e Fermate, ammettiamo.

Fermate:

Numero treno (anche parte della superchiave)	*Stazione*	*Orario fermata*
1740	Firenze	13:15
1740	Bologna	12:00

Treni:

Numero treno	*Destinazione*	*Categoria*	*Orario*	*Partenza*
1740	Roma	ES	11	Venezia

Altra idea: piuttosto che avre una tupla per ogni fermata intermedia posso supporre di avere il numero del treno, l'identificativo della fermata di andata/ritorno e gli orari delle fermate andata/ritorno.
La soluzione è buona concettualmente ma occuperebbe troppo spazio da un punta di vista pratico.
Altra possibile soluzione (slide):

Soluzione:

Ecco un possibile schema:

PARTENZE (Numero, Orario, Destinazione, Categoria)
FERMATE (Treno, Stazione, Orario)

PARTENZE rappresenta tutte le partenze della stazione

Il numero di fermate **cambia** per ogni treno, rendendo impossibile la rappresentazione delle fermate in PARTENZE (un numero fisso di attributi).

La **chiave** della relazione FERMATE è composta da due attributi, "Treno" e "Stazione", che indicano il numero di treno e le stazioni in cui si fermano.

È necessario introdurre un **vincolo di integrità referenziale** tra "Treno" in FERMATE e "Numero" in PARTENZE.

Il modello basato sui valori deve implementare dei meccanismi per prevenire violazioni e prevedere possibili valori nulli alle chiavi, in particolar modo sulle chiavi esterne.
Ad esempio, posso imporre come vincolo di tupla che un certo campo non sia nullo.

Supponendo di avere un'istanza di una base di dati e avendo una tupla, che succederebbe se provassi a togliere una tupla? Ciò potrebbe provocare una violazione.
Ci sono delle possibilità:

- *rifiutare* l'operazione in tronco;
- *generare un'eliminazione a cascata* (ad es. se cancello un treno, controllo tutte le tabelle relazionate cancellando a cascata tutte le fermate associate);
- *introdurre dei valori nulli nella tabella usata*, se possibile, indicando come soluzione ragionevole a livello logico che non utilizzerò o che quel campo è stato cancellato e non mi servirà.

Definire uno schema di base di dati per organizzare le informazioni di un'azienda che ha impiegati e filiali
Ogni **impiegato** avrà:

- codice fiscale
- cognome e nome
- data di nascita

Le **filiali** saranno caratterizzate con:

- codice
- sede
- direttore (è un impiegato)

Ogni impiegato lavora presso una sola filiale.

Soluzione:

Un esempio di base di dati per l'esercizio

Impiegati				
CF	Cognome	Nome	DataNascita	Filiale
RSS MRA 76E27 H501 Z	Rossi	Mario	27/05/1976	GT09
BRN GNN 90D03 F205 E	Bruni	Giovanni	03/04/1990	AB04
GLL BRN 64E04 F839 H	Gialli	Bruno	04/05/1964	GT09
NRE GNI 64L01 G273 Y	Neri	Gino	01/07/1964	AB04
RSS NNA 45R42 D969 X	Rossi	Anna	02/10/1945	PT67
RGI PNI 77M05 M082 B	Riga	Pino	05/08/1977	AB04

Filiali		
Codice	Sede	Direttore
AB04	Roma Tiburtina	NRE GNI 64L01 G273 Y
GT09	Roma Monteverde	RSS NNA 45R42 D969 X
PT67	Roma Eur	RSS MRA 76E27 H501 Z

Vincoli di integrità referenziale:

- "Filiale" della relazione IMPIEGATI → "Codice" di FILIALI
- "Direttore" della relazione FILIALI → "CF" di IMPIEGATI

Esercizio: Individuare le **chiavi** ed i **vincoli di integrità referenziale** che sussistono nella base di dati di cui sotto, e che è ragionevole assumere siano soddisfatti da tutte le basi di dati sullo stesso schema.
Individuare anche gli attributi sui quali possa essere sensato ammettere **valori nulli**.

PAZIENTI

Cod	Cognome	Nome
A102	Necchi	Luca
B372	Rossini	Piero
B543	Missoni	Nadia
B444	Missoni	Luigi
S555	Rossetti	Gino

MEDICI

Matr	Cognome	Nome	Reparto
203	Neri	Piero	A
574	Bisi	Mario	B
431	Bargio	Sergio	B
530	Belli	Nicola	C
405	Mizzi	Nicola	A
201	Monti	Mario	A

RICOVERI

Paziente	Inizio	Fine	Reparto
A102	2/05/94	9/05/94	A
A102	2/12/94	2/01/95	A
S555	1/11/94	3/12/94	B
B444	1/12/94	2/01/95	B
S555	5/10/94	1/11/94	A

REPARTI

Cod	Nome	Primario
A	Chirurgia	203
B	Medicina	574
C	Pediatria	530

Soluzione: CHIAVI

PAZIENTI

Cod	Cognome	Nome
A102	Necchi	Luca
B372	Rossini	Piero
B543	Missoni	Nadia
B444	Missoni	Luigi
S555	Rossetti	Gino

MEDICI

Matr	Cognome	Nome	Reparto
203	Neri	Piero	A
574	Bisi	Mario	B
431	Bargio	Sergio	B
530	Belli	Nicola	C
405	Mizzi	Nicola	A
201	Monti	Mario	A

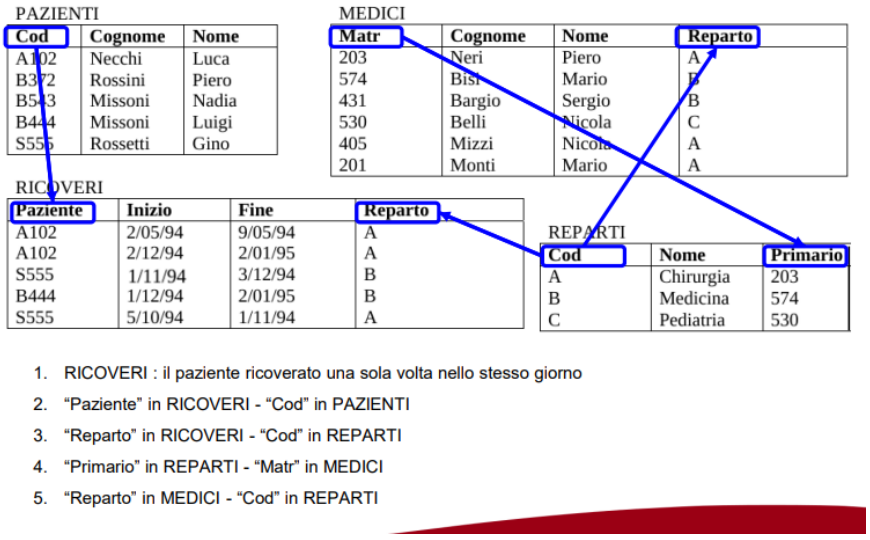
RICOVERI

Paziente	Inizio	Fine	Reparto
A102	2/05/94	9/05/94	A
A102	2/12/94	2/01/95	A
S555	1/11/94	3/12/94	B
B444	1/12/94	2/01/95	B
S555	5/10/94	1/11/94	A

REPARTI

Cod	Nome	Primario
A	Chirurgia	203
B	Medicina	574
C	Pediatria	530

Soluzione: Vincoli di integrità referenziale



1. RICOVERI : il paziente ricoverato una sola volta nello stesso giorno
2. "Paziente" in RICOVERI - "Cod" in PAZIENTI
3. "Reparto" in RICOVERI - "Cod" in REPARTI
4. "Primario" in REPARTI - "Matr" in MEDICI
5. "Reparto" in MEDICI - "Cod" in REPARTI

Per le interrogazioni, operazioni ed interrogazioni/query parte del DML si focalizza sulla *algebra relazionale*, usando un linguaggio dichiarativo come SQL. Le operazioni procedurali specificano la generazione del risultato e adotta un insieme di operatori che producono relazioni e possono essere composti. Le relazioni sono insiemi, i risultati debbono essere relazioni e ad essi è possibili applicare le operazioni di *unione, intersezione, differenza* solo se sono definite sugli stessi attributi.

Ad esempio, *l'unione* considera l'aggregazione delle tuple, operazione che può anche aumentare il numero di tuple risultanti. Esempio di unione sensata ma impossibile: tabelle *Paternità* e *Maternità* in cui, avendo campi diversi, l'unione non è logicamente possibile, perché operando su campi semantici diversi. Se cambiassi il nome del campo *Padre* con *Madre* la cosa sarebbe possibile, in quanto si opera su campi omogenei (hanno lo stesso nome e quindi operano sugli stessi dati logicamente).

Idea migliore: *ridenominazione* del campo, il *Padre* diventa *Genitore*, analogamente anche la *Madre* diventa *Genitore*; in questo modo l'unione diventa possibile.
È possibile applicare l'operatore di ridenominazione su un solo attributo oppure anche su più attributi.
Quest'operazione è fondamentale per ottenere delle tuple omogenee quando non lo sono anche se il campo semantico di applicazione della query lo è; l'uso principale è:

- join della tabella con sé stessa o con altre tabelle e si esegue per salvare il risultato ed eliminare le ambiguità tra campi con stesso nome oppure tra gli unici campi diversi della relazione

Altra definizione è la *differenza*, restituendo tutte le tuple che non rispettano un vincolo (ad esempio nelle slide, tutti i laureati che non sono specialisti); è spesso usata negli esercizi (prendendo casi di massimo o di minimo, visto in esplicito a pagina 17). Con essa, se ho bisogno di un vincolo su qualche richiesta ("il massimo", "il minimo", "più di", "meno di") o in generale un certo vincolo sintattico che implica una parte di un tutto viene usata con alta frequenza.
In ogni caso aggiungo anche:

Ricerca del minimo/massimo

- Esempio dato uno schema R(A,B) trovare il minimo/massimo B in R
- Questo può essere fatto facendo un join di R su se stesa dopo aver rinominato tutti gli attributi nel seguente modo:

$$\pi_B(R) - \pi_B(R \bowtie_{B>B1} (\rho_{(A1,B1 \leftarrow A,B)}(R)))$$

- Nella seconda parte vengono trovati tutti quei valori che non sono il minimo. Per far questo viene fatto un join tra la relazione R e se stessa, con però tutti gli attributi rinominati. La condizione di join dice che ogni attributo B deve essere maggiore degli stessi attributi rinominati. In questo modo vengono tenute tutte le tuple **tranne quella in cui l' attributo B assume il valore minore**. Quindi per il principio di complementarietà sottraendo dall' insieme iniziale, l' insieme delle tuple dove B non è il minimo, otteniamo proprio il valore minimo che cercavamo.

Algebra relazionale: selezione, proiezioni, join e tipi di join

Come detto la volta scorsa, una volta ridenominati correttamente degli attributi è quindi possibile stabilire relazioni logiche sensate. Si può usare la *selezione (operatore σ*, dove la lettera qui è "sigma"*)* avendo come risultato lo stesso schema ed un sottoinsieme delle tuple (decomposizione orizzontale) che soddisfano una certa condizione, eliminando quindi quelle che non la soddisfano.
Si possono anche combinare le singole condizioni all'interno di una singola selezione, ad esempio:

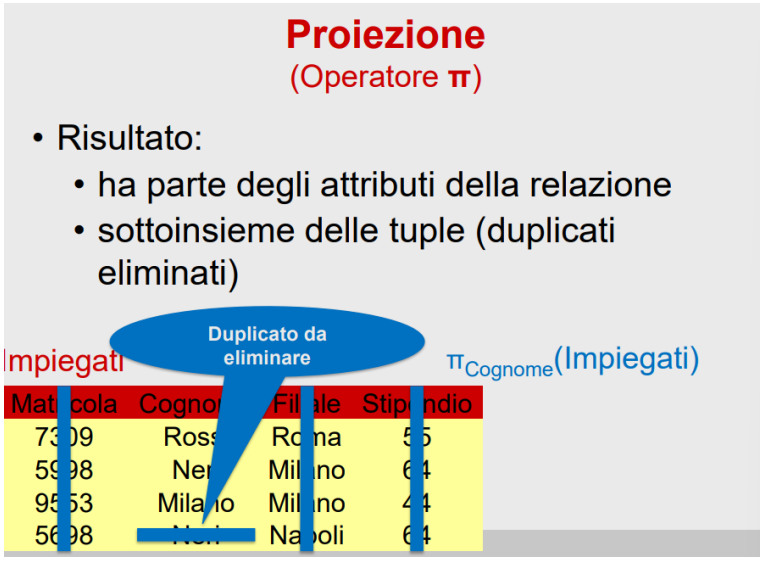
Esempio: Guadagnano più di 50 e lavorano a Milano

$\sigma_{\text{Stipendio} > 60 \text{ AND Filiale} = \text{'Milano'}}(\text{Impiegati})$

Impiegati

Matricola	Cognome	Filiale	Stipendio
5998	Neri	Milano	64

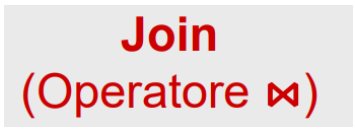
Altro operatore è quello di *proiezione (operatore π)*, quindi mantengo un sottoinsieme questa volta delle colonne, eliminandone alcune (decomposizione verticale). Possiede quindi parte degli attributi della relazione come sottoinsieme di tuple; quindi, non sono ammessi duplicati (eliminati senza un preciso criterio di ordine, essendo descritto in termini insiemistici, avviene come capita).



Una proiezione contiene quindi al più tante tuple quante l'operando, tuttavia può contenerne di meno. Se X, però, è superchiave, non posso avere due tuple con lo stesso valore (duplicati). Se facessi l'operazione di proiezione sulla superchiave e trovassi dei duplicati, naturalmente, essa non sarebbe superchiave. Anche se prendessi una parte della superchiave, comunque anch'essa superchiave (minimale se chiave invece), sarebbe garantita l'univocità.

Naturalmente possiamo combinare i due operatori. Fondamentale anche l'ordine con cui applico gli operatori, in quanto se invertiti potrebbe non essere più possibile una delle due operazioni (dipende dal contesto; esempio banale, se io volessi prendere Matricola/Cognome degli Impiegati come proiezione potrebbe non essermi possibile la selezione).

Un esempio di applicazione sia di selezione che di proiezione:



Operatore molto importante è quello di *join*, operatore binario su due relazioni A e B. Come risultato restituisce uno schema della relazione che è un'unione degli attributi degli operandi e come tuple il prodotto cartesiano le tuple A x B mantenendo quelle *con valori uguali su attributi uguali*.

A tuple uguali corrispondono tuple con la stessa logica: combinassi ad esempio le tuple con 1, andrei a prendere quella con il valore 1 sugli attributi comuni.

Join: Example

Voto

Numero	Voto
1	25
2	13
3	27
4	28

Candidati

Numero	Candidato
1	Mario Rossi
2	Nicola Russo
3	Mario Bianchi
4	Remo Neri

Voto ⋈ Candidati

Numero	Candidato	Voto
1	Mario Rossi	25
2	Nicola Russo	13
3	Mario Bianchi	27
4	Remo Neri	28

Più formalmente si definisce in questo modo:

- Date due relazioni $R_1(X_1)$, $R_2(X_2)$
- $R_1 \bowtie R_2$ è una relaz. su X_1X_2 (eq. $X_1 \cup X_2$)

$$\{ t \text{ su } X_1X_2 \mid t[X_1] \in R_1 \text{ e } t[X_2] \in R_2 \}$$

dove $t[X_1]$ indica la proiezione su X_1, cioè $\pi_{X_1}(R)$

Ogni singola tupla contribuisce ad un join nel caso del *join completo*; se parallelamente qualche tupla non contribuisce si ha invece un *join non completo*. Si noti che nel primo caso non ho perdita di informazioni mentre nel secondo caso potrebbe succedere e le tuple vengono "tagliate fuori" dal risultato.
Tecnicamente queste tuple vengono descritte come *dangling tuples*, dato che possono non contribuire a nessuna relazione esistente.
Esempi di join (completo e non completo):

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparto	Capo
A	Mori
B	Bruni

Impiegato	Reparto	Capo
Rossi	A	Mori
Neri	B	Bruni
Bianchi	B	Bruni

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparto	Capo
B	Mori
C	Bruni

Impiegato	Reparto	Capo
Neri	B	Bruni
Bianchi	B	Bruni

Può anche capitare un *join vuoto*, in cui si ha una sorta di prodotto cartesiano, quindi una combinazione di *n x m* tuple delle relazioni in atto. Nel caso sotto ad esempio non ci sono tutte le tuple corrispondenti e si genera la situazione del vuoto. Un join, in altri casi, potrebbe benissimo corrispondere invece ad un prodotto cartesiano delle tuple.

Un join vuoto

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparto	Capo
D	Mori
C	Bruni

Impiegato	Reparto	Capo

In generale nei casi di join, consideriamo che (avendo due relazioni generiche R1 ed R2):

- primo punto, notando che si ha almeno un attributo comune, il numero di tuple varia da 0 al prodotto cartesiano del numero delle tuple;
- secondo punto, se il join viene realizzato su una delle due relazioni, il sottoinsieme è <= a una delle due relazioni;
- terzo punto, in cui ogni tupla corrisponde ad una tupla dell'altra. Nel join ce ne saranno esattamente *n*, quindi la cardinalità di R1. Non avendo chiavi esterne, non è obbligatorio che tutte le tuple siano collegate tra di loro; se la tupla è nulla e non è chiave esterna non è nel risultato del join.

Cardinalità del join

- $R_1(A,B)$, $R_2(B,C)$
- In generale
  $$0 \leq |R_1 \bowtie R_2| \leq |R_1| \times |R_2|$$
- se B è chiave in R_2
  $$0 \leq |R_1 \bowtie R_2| \leq |R_1|$$
- se B è chiave in R_2 ed esiste vincolo di integrità referenziale fra B in R_1 e R_2:
  $$|R_1 \bowtie R_2| = |R_1|$$
  se l'attributo B in R1 non può assumere valore nullo

Il *join esterno/outer join* estende, con valori nulli, le tuple che verrebbero tagliate fuori da un *join interno/inner join* (quest'ultimo significa semplicemente collegare 2 tabelle). In particolare, l'outer join mantiene tutti i record anche se non trovasse un record corrispondente. Esso esiste in tre versioni:

- il *join sinistro/left* mantiene tutte le tuple del primo operando estendole con valori nulli, se necessario;
- il *join destro/right* mantiene tutte le tuple del secondo operando estendole con valori nulli, se necessario;
- il *join completo/full* mantiene tutte le tuple di entrambi gli operandi estendole con valori nulli, se necessario.

Esempio di join sinistro, destro e completo (in ordine)

Impiegati

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparti

Reparto	Capo
B	Mori
C	Bruni

Impiegati $\bowtie_{LEFT}$ Reparti

Impiegato	Reparto	Capo
Neri	B	Mori
Bianchi	B	Mori
Rossi	A	NULL

Impiegati

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparti

Reparto	Capo
B	Mori
C	Bruni

Impiegati $\bowtie_{RIGHT}$ Reparti

Impiegato	Reparto	Capo
Neri	B	Mori
Bianchi	B	Mori
NULL	C	Bruni

Impiegati

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparti

Reparto	Capo
B	Mori
C	Bruni

Impiegati $\bowtie_{FULL}$ Reparti

Impiegato	Reparto	Capo
Neri	B	Mori
Bianchi	B	Mori
Rossi	A	NULL
NULL	C	Bruni

Esiste inoltre il *semijoin*, operatore su due relazioni R1 ed R2, restituendo una relazione su *X1* con le tuple di *R1* che contribuiscono al join con *R2*.
Ad esempio, se io eseguissi il join tra Impiegati e Reparti per poi eseguire la proiezione, avrei ottenuto lo stesso risultato del semijoin.
Seguono esempi:

Semijoin come Proiezione e Join

Date due relazioni $R_1(A,B)$, $R_2(B,C)$

$R_1 \text{ SEMI}\bowtie R_2 = \pi_{A,B}(R_1 \bowtie R_2)$

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparti

Reparto	Capo
B	Mori
C	Bruni

Impiegati SEMI$\bowtie$ Reparti

Impiegato	Reparto
Neri	B
Bianchi	B

Il *natural join* (join naturale), considera come risultato la colonna con lo stesso nome rispetto a quello cercato e stessi tipi di dati presenti tra le tabelle su cui operare il join. Esso coincide con il prodotto cartesiano quando vi sono relazioni senza attributi in comune.
L'operazione viene chiamata *theta-join* (join condizionale) viene eseguita quando un prodotto cartesiano è seguito da selezione.
Essa ha questa sintassi: $R_1 \bowtie_{Condizione} R_2$
Nell'esempio il reparto viene rinominato in codice e poi facendo il join con i reparti, ottenendo Impiegato, Codice e Capo. Altro caso, facciamo la join tra Reparti e Selezione e poi eseguendo un theta join rinominando Codice e Reparti; il risultato qui è diverso, dato che ho un attributo in più.
Eseguendo il prodotto cartesiano e poi la selezione, ecco il theta join (theta come lettera perché storicamente indicava il confronto).
Morale: o faccio un join, oppure posso fare un join e poi una selezione.

Molto utile il seguente discorso:

1.3 Cardinalità Join

Definendo N la cardinalità del join:

1. se il join di r_1 e r_2 è completo allora $0 \leq N \leq max(|r_1|, |r_2|)$
2. se il join coinvolge una chiave di r_2, allora $0 \leq N \leq |r_1|$
3. se il join coinvolge una chiave di r_2 ed $\exists$ un vincolo di integrità referenziale tra un attributo di r_1 e la chiave di r_2, allora $N = |r_1|$

Usando l'operatore di uguaglianza nel theta-join, si applica un *equi-join.*
Esempi pratici (con relative soluzioni che si susseguono, incollati tutti perché il caro prof dice che vengono molto sbagliati agli esami. Si consideri inoltre che nell'esempio 3 viene volutamente fino alla parte 3 mostrata una soluzione sbagliata; quella corretta è nella parte 4).

Esempio 1: matricola, nome ed età degli impiegati che guadagnano più di 40

Impiegati

Matricola	Nome	Età	Stipendio
7309	Rossi	34	45
5998	Bianchi	37	38
9553	Neri	42	35
5698	Bruni	43	42
4076	Mori	45	50
8123	Lupi	46	60

Supervisione

Impiegato	Capo
7309	5698
5998	5698
9553	4076
5698	4076
4076	8123

$$\pi_{Matricola,\ Nome,\ Età}\ (\sigma_{Stipendio>40}(Impiegati))$$

Impiegati

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparti

Codice	Capo
A	Mori
B	Bruni

$Impiegati \bowtie_{Reparto=Codice} Reparti$

Impiegato	Reparto	Codice	Capo
Rossi	A	A	Mori
Neri	B	B	Bruni
Bianchi	B	B	Bruni

Esempio 2: Capi degli impiegati che guadagnano più di 40

Impiegati

Matricola	Nome	Età	Stipendio
7309	Rossi	34	45
5998	Bianchi	37	38
9553	Neri	42	35
5698	Bruni	43	42
4076	Mori	45	50
8123	Lupi	46	60

Supervisione

Impiegato	Capo
7309	5698
5998	5698
9553	4076
5698	4076
4076	8123

$$\pi_{Capo}\ (Supervisione \bowtie_{Impiegato=Matricola} (\sigma_{Stipendio>40}(Impiegati)))$$

Esempio 3: Matricola, nome e stipendio degli impiegati che guardagnano più dei capi / 1

Impiegati

Matricola	Nome	Età	Stipendio
7309	Rossi	34	45
5998	Bianchi	37	38
9553	Neri	42	35
5698	Bruni	43	42
4076	Mori	45	50
8123	Lupi	46	60

Impiegato	Capo
7309	5698
5998	5698
9553	4076
5698	4076
4076	8123

Supervisione

Matricola	Nome	Età	Stipendio	Impiegato	Capo
7309	Rossi	34	45	7309	5698
5998	Bianchi	37	38	5998	5698
9553	Neri	42	35	9553	4076
5698	Bruni	43	42	5698	4076
4076	Mori	45	50	4076	8123

$$\text{Supervisione} \bowtie_{\text{Impiegato=Matricola}} \text{Impiegati}$$

Esempio 3: Matricola, nome e stipendio degli impiegati che guardagnano più dei capi / 2

Impiegati

Matricola	Nome	Età	Stipendio
7309	Rossi	34	45
5998	Bianchi	37	38
9553	Neri	42	35
5698	Bruni	43	42
4076	Mori	45	50
8123	Lupi	46	60

Impiegato	Capo
7309	5698
5998	5698
9553	4076
5698	4076
4076	8123

Supervisione

$$\rho_{\text{MatrC,NomeC,StipC,EtàC} \leftarrow \text{ Matr,Nome,Stip,Età}}(\text{Impiegati})$$

MatrC	NomeC	EtàC	StipC
7309	Rossi	34	45
5998	Bianchi	37	38
9553	Neri	42	35
5698	Bruni	43	42
4076	Mori	45	50
8123	Lupi	46	60

Esempio 3: Matricola, nome e stipendio degli impiegati che guardagnano più dei capi / 3

Matricola	Nome	Età	Stipendio	Impiegato	Capo
7309	Rossi	34	45	7309	5698
5998	Bianchi	37	38	5998	5698
9553	Neri	42	35	9553	4076
5698	Bruni	43	42	5698	4076
4076	Mori	45	50	4076	8123

$$\text{Supervisione} \bowtie_{\text{Impiegato=Matricola}} \text{Impiegati}$$

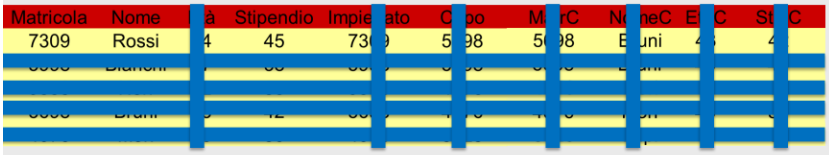
$$\rho_{\text{MatrC,NomeC,StipC,EtàC} \leftarrow \text{ Matr,Nome,Stip,Età}}(\text{Impiegati})$$

MatrC	NomeC	EtàC	StipC
7309	Rossi	34	45
5998	Bianchi	37	38
9553	Neri	42	35
5698	Bruni	43	42
4076	Mori	45	50
8123	Lupi	46	60

Matricola	Nome	Età	Stipendio	Impiegato	Capo	MatrC	NomeC	EtàC	StipC
7309	Rossi	34	45	7309	5698	5698	Bruni	43	42
5998	Bianchi	37	38	5998	5698	5698	Bruni	43	42
9553	Neri	42	35	9553	4076	4076	Mori	45	50
5698	Bruni	43	42	5698	4076	4076	Mori	45	50
4076	Mori	45	50	4076	8123	8123	Lupi	46	60

$$\rho_{\text{MatrC,NomeC,StipC,EtàC} \leftarrow \text{ Matr,Nome,Stip,Età}}(\text{Impiegati}) \bowtie_{\text{MatrC=Capo}} (\text{Supervisione} \bowtie_{\text{Impiegato=Matricola}} \text{Impiegati}))$$

Esempio 3: Matricola, nome e stipendio degli impiegati che guardagnano più dei capi / 4



$$\pi_{Matricola,Nome,Stipedio}(\sigma_{Stipendio>StipC}(\rho_{MatrC,NomeC,StipC,EtàC \leftarrow Matr,Nome,Stip,Età}(Impiegati) \bowtie_{MatrC=Capo} (Supervisione \bowtie_{Impiegato=Matricola} Impiegati)))$$

Esempio 4: Matricola dei capi i cui impiegati guardagnano tutti più di 40

Impiegati

Matricola	Nome	Età	Stipendio
7309	Rossi	34	45
5998	Bianchi	37	38
9553	Neri	42	35
5698	Bruni	43	42
4076	Mori	45	50
8123	Lupi	46	60

Supervisione

Impiegato	Capo
7309	5698
5998	5698
9553	4076
5698	4076
4076	8123

Implementato come «Restituisci tutti i capi, tranne quelli per cui c'è almeno un impiegato che guardagna < 40

$$\pi_{Capo}(Supervisione) - \pi_{Capo}(Supervisione \bowtie_{Impiegato=Matricola}(\sigma_{Stipendio\leq 40}(Impiegati)))$$

Nota importante: esiste un particolare tipo di join che unisce la tabella con sé stessa, chiamato *self-join*; qui non è enunciato in maniera esplicita ma spesso compare negli esercizi di algebra relazionale. Esso richiede almeno due tabelle, con la differenza che, anziché aggiungere una seconda tabella ad un'interrogazione, si aggiungerà una seconda istanza della stessa tabella. Esso permette di mantenere tutte quelle tuple in cui i valori degli attributi sono uguali alla loro controparte ridenominata.

Nota molto utile: ordine delle operazioni algebra relazionale.

Ordine delle operazioni

Esempio: selezione e proiezione

$$\Pi_{\text{matricola, cognome}}(\sigma_{\text{stipendio>62}}(\text{Impiegati}))$$

Ottengo il risultato:

Matricola	Cognome
5998	Neri
5698	Bianchi

e se scambiassi l'ordine di selezione e proiezione?

$$\sigma_{\text{stipendio>62}}(\Pi_{\text{matricola, cognome}}(\text{Impiegati}))$$

È un **errore**, poiché la tabella risultante dall'operazione di proiezione ha come schema gli attributi matricola e cognome, e su di esso non posso effettuare la selezione con predicato sull'attributo stipendio

Scritto da Gabriel

Nota di contorno utile: cardinalità dei join.

- **Cardinalità del Join:**

$$\max\{|r1|,|r2|\} \le |r1 \text{ join } r2| \le |r1| \times |r2|$$

- **Se Y contiene una chiave per R_2, allora**
  $|r_1 \text{ join } r_2| \le |r_1|$
- **Se vincolo di integrità referenziale fra $Z \subseteq Y$ e chiave K di R_2,**
  $|r_1 \text{ join } r_2| = |r_1|$
- **Join non completo: tuple dangling escluse dal risultato**

$r_1 \bowtie r_2$ contiene un numero di tuple compreso tra 0 e $|r_1| \times |r_2|$

Se il join è completo $\Rightarrow$ contiene almeno un numero di tuple pari al massimo tra $|r_1|$ e $|r_2|$

Se gli attributi comuni contengono una chiave per $r_1 \Rightarrow r_1 \bowtie r_2$ contiene al più $|r_2|$ tuple

Se gli attributi comuni contengono una chiave per $r_2 \Rightarrow r_1 \bowtie r_2$ contiene al più $|r_2|$ tuple

- È possibile che una tupla di una delle relazioni operande non faccia match con nessuna tupla dell'altra relazione; in tal caso tale tupla viene detta "dangling"
- Nel caso limite è quindi possibile che il risultato del join sia vuoto; all'altro estremo è possibile che ogni tupla di r_1 si combini con ogni tupla di r_2
- Ne segue che

  la cardinalità del join, $|r_1 \rhd\lhd r_2|$, è compresa tra 0 e $|r_1| * |r_2|$

- Se il join è eseguito su una superchiave di $R_1(X_1)$, allora ogni tupla di r_2 fa match con al massimo una tupla di r_1, quindi $|r_1 \rhd\lhd r_2| \le |r_2|$
- Se $X_1 \cap X_2$ è la chiave primaria di $R_1(X_1)$ e foreign key in $R_2(X_2)$ (e quindi c'è un vincolo di integrità referenziale) allora $|r_1 \rhd\lhd r_2| = |r_2|$

La cardinalità del risultato è soggetta alle regole seguenti:

- Se il join di r_1, r_2 è completo (i.e., ogni tupla di r_1 e di r_2 contribuisce ad almeno una tupla del risultato), allora il numero delle tuple finali sarà maggiore od uguale al massimo fra $|r_1|$ e $|r_2|$.
- Se $X_1 \cap X_2$ contiene una chiave di r_2, allora il numero delle tuple finali sarà minore od uguale a $|r_1|$.
- Se $X_1 \cap X_2$ è la chiave primaria di r_2 ed esiste un vincolo referenziale fra $X_1 \cap X_2$ in r_1 e tale chiave di r_2, allora il numero delle tuple finali sarà esattamente uguale a $|r_1|$.

Algebra relazionale: equivalenze, viste

Due espressioni sono *equivalenti* se producono lo stesso risultato per ogni valore o possibile istanza della base di dati, cercando di rendere il tutto meno pesante possibile dal punto di vista elaborativo. Questo è utile soprattutto se vogliamo passare da un'espressione ad un'altra.

Ad esempio: $X\,(Y + Z) = XY + XZ$

l'equivalenza è importante perché i DBMS cercano di eseguire espressioni equivalenti a quelle presenti, ma quelle meno costose.

Ad esempio sono equivalenti:

- Date due relazioni $R_1(X_1)$ e $R_2(X_2)$ con l'attributo $A \in X_2$
- Sono equivalenti:

$$\sigma_{A=10}(R_1 \bowtie R_2) \equiv R_1 \bowtie \sigma_{A=10}(R_2)$$

Il caso peggiore risulta essere il primo dei due elencati, il secondo migliore da un punto di vista computazionale:

- Consideriamo
  1. $\sigma_{A=10}(R_1 \bowtie R_2)$
  2. $R_1 \bowtie \sigma_{A=10}(R_2)$
- Assumiamo che le tuple di R_2 con A=10 è 10%
- Numero massimo di righe create (temporaneamente) nei due casi
  1. $|R_1| \times |R_2|$
  2. $|R_1| \times 0.1 \times |R_2|$

Mostriamo quindi alcune equivalenze (quindi appunto per ogni istanza producono lo stesso risultato). La sostanza è che le operazioni scelte sono più efficienti le une rispetto alle altre, ecco spiegato il significato di questa cosa.

- Data una relazione R(Z) dove $X,Y \subseteq Z$ e $X \cap Y = \emptyset$:
  1. $\sigma_{C1\ AND\ C2}(R) \equiv \sigma_{C1}\,\sigma_{C2}(R)$
  2. $\pi_X(\pi_{XY}(R)) \equiv \pi_X(R)$

In particolare, si parla di equivalenza perché possono essere fatte delle ottimizzazioni. Elenco le più importanti (che sono viste anche dalle slide):

- *atomizzazione delle selezioni*, dunque una congiunzione di selezioni può essere sostituita da una sequenza di selezioni atomiche;
- *idempotenza delle proiezioni*, dove una proiezione viene trasformata in una sequenza di queste;
- *anticipazione della selezione rispetto al join;*
- *anticipazione della proiezione rispetto al join;*

- *trasformazioni basate sulla corrispondenza tra operatori insiemistici e selezioni complesse* (quindi sostituire selezioni con AND/OR, operazioni di join, ecc.).

- Date due relazioni $R_1(X_1)$ e $R_2(X_2)$ con $Y_1 \subset X_1$, $Y_2 \subset X_2$:
  3. $\sigma_{C1}(R_1 \bowtie R_2) \equiv R_1 \bowtie \sigma_{C1}(R_2)$ se C1 condizione su X_2
  4. $\pi_{X_1Y_2}(R_1 \bowtie R_2) \equiv \pi_{X_1}(R_1 \bowtie \pi_{Y_2}(R_2))$ se X_2 - Y_2 non incoinvolti del join
- In linguaggio naturale:
  3. Selezione prima del join
  4. Anticipazione della proiezione su Y_2 se (X_2 - Y_2) non nel join e non nell'output

- Date due relazioni $R_1(X_1)$ e $R_2(X_2)$:
  5. $\sigma_C(R_1 \bowtie R_2) \equiv R_1 \bowtie_C R_2$
  6. $\sigma_C(R_1 \cup R_2) \equiv \sigma_C(R_1) \cup \sigma_C(R_2)$
  7. $\sigma_C(R_1 - R_2) \equiv \sigma_C(R_1) - \sigma_C(R_2)$
- In linguaggio naturale:
  5. Selezione su C allo stesso tempo del join
  6. Selezione prima dell'unione
  7. Selezione prima della differenza

Nel caso del punto 3, potrebbe non essere sempre valida la condizione, in quanto il fatto che C1 sia condizione su X2 limita il possibile punto di applicazione; quindi, potrei applicare una condizione generica che potrebbe non considerare determinati attributi (magari non esistenti).

Ciò si fa anche per ridurre lo scorrimento delle tuple in termini di tempo.
Esempio pratico per ottimizzazione di una query:

Esempio: Capi con impiegati con meno di 40 anni

Impiegati

Matricola	Nome	Età	Stipendio
7309	Rossi	34	45
5998	Bianchi	37	38
9553	Neri	42	35
5698	Bruni	43	42
4076	Mori	45	50
8123	Lupi	46	60

Supervisione

Impiegato	Capo
7309	5698
5998	5698
9553	4076
5698	4076
4076	8123

$\pi_{Capo}(\sigma_{(Impiegato=Matricola\ AND\ Età<40)}(Supervisione \bowtie Impiegati))$

Super-inefficiente!!

Questa operazione non è abbastanza efficiente, dato l'ordine delle operazioni, mettendo soprattutto il join per ultimo. Normalmente prima si ha la proiezione, poi la selezione e infine il join. Posso invece spezzare le singole operazioni, come avviene nel caso d'uso seguente:

Esempio: Capi con impiegati con meno di 40 anni / 2

$\pi_{Capo}(\sigma_{(Impiegato=Matricola\ AND\ Età<40)}(Supervisione \bowtie Impiegati))$

Regola 1: $\pi_{Capo}(\sigma_{Età<40}\ \sigma_{Impiegato=Matricola}(Supervisione \bowtie Impiegati))$

Regola 5: $\pi_{Capo}(\sigma_{Età<40}(Supervisione \bowtie_{Impiegato=Matricola} Impiegati))$

Regola 4: $\pi_{Capo}(\pi_{Matricola}(\sigma_{Età<40}(\text{~~Supervisione~~}_{Impiegati})) \bowtie_{Impiegato=Matricola} \text{~~Impiegati~~}_{Supervisione})$

I DBMS usano quindi regole di equivalenza per ottimizzare le interrogazioni, avendo quindi relazioni "intermedie" con il numero minimo di tuple ed attributi. I DBMS non eseguono veramente le interrogazioni come vengono formulate; gli stessi ottimizzano per noi la computazione.

Nell'algebra relazionale introduciamo di nuovo il concetto dei valori nulli, prendiamo per esempio la selezione come si vede a fianco.

Impiegati

Matricola	Cognome	Filiale	Età
~~7309~~	~~Rossi~~	~~Roma~~	~~32~~
5998	Neri	Milano	45
~~9553~~	~~Bruni~~	~~Milano~~	~~NULL~~

$\sigma_{Età > 40}(Impiegati)$

Una condizione può essere vera solo per valori non nulli, ma possono esistere valori nulli oppure non nulli; si noti che in SQL si hanno le condizioni *IS NULL* oppure *IS NOT NULL*, usate come possibile logica sulle query; se mi accorgessi che un campo è vuoto su una condizione che non lo impone, restituisco diretto falso, ad esempio. I valori nulli in un certo senso complicano la logica della creazione tabelle e formazione relazioni; si introduce una cosiddetta logica a tre valori, in cui un predicato diventa vero/falso/unknown (sconosciuto, stato di forse possiamo dire).

Per superare questi inconvenienti si cerca quindi di adattare l'utilizzo del valore nullo al contesto dato, cercando di usarlo a proprio vantaggio nelle query (con le condizioni SQL viste sopra).

Selezione con valori nulli: Soluzione

Impiegati

Matricola	Cognome	Filiale	Età
7309	Rossi	Roma	32
5998	Neri	Milano	45
9553	Bruni	Milano	NULL

$\sigma_{Età>30}(Persone) \cup \sigma_{Età\leq30}(Persone) \cup \sigma_{Età\ IS\ NULL}(Persone)$

=

$\sigma_{Età>30 \vee Età\leq30 \vee Età\ IS\ NULL}(Persone)$

=

Persone

Intendo quindi usare le ***viste***, dando rappresentazioni diverse per gli stessi dati e definendo relazioni per cui il contenuto sia funzione del contenuto di altre relazioni. In pratica vengono usate per semplificare le query, selezionando il contenuto di tuple e/o colonne e poi modificandole o agendo su di essere direttamente.
Segue a destra un piccolo esempio:

Viste: Esempio

Afferenza

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Direzione

Reparto	Capo
A	Mori
B	Bruni

una vista:
Supervisione =
$\pi_{\text{Impiegato, Capo}}$ (Afferenza ⋈ Direzione)

Si hanno due tipi di viste, ***materializzate*** e ***virtuali*** (quindi le viste classiche).
Quelle ***materializzate*** hanno il vantaggio di essere immediatamente disponibili per le interrogazioni, sorta di istantanee e risparmiano spazio; per contro sono ridondanti, appesantiscono gli aggiornamenti, sono salvate fisicamente su disco e raramente sono supportate dai DBMS.
Le relazioni ***virtuali*** (o viste) sono invece supportate da tutti i DBMS e un'interrogazione su una vista viene eseguita "ricalcolando" la vista in un certo momento, più lentamente.

Il seguente esempio dimostra l'applicazione di una interrogazione specifica "semplificata" dando all'utente ciò di cui ha bisogno in quel momento, contemporaneamente fornendo i dati richiesti:

Afferenza

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Direzione

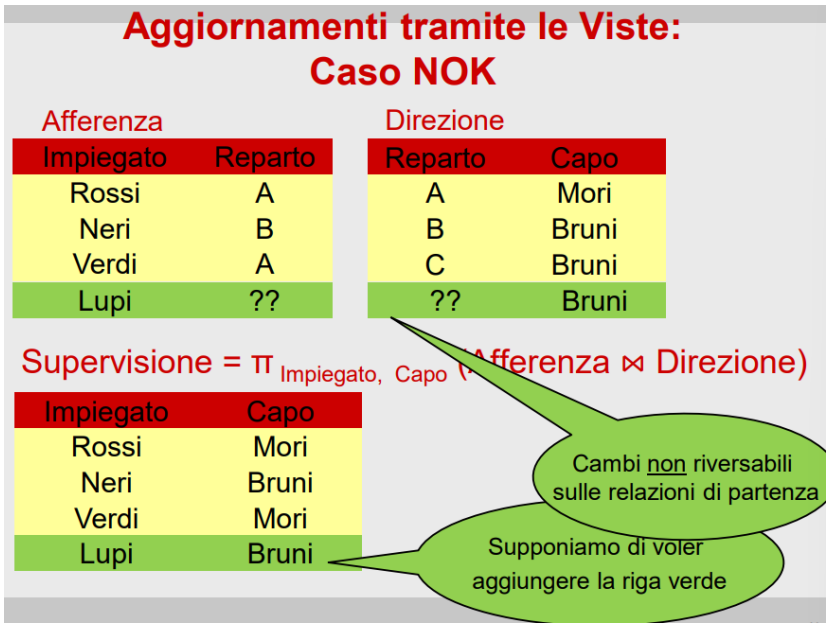
Reparto	Capo
A	Mori
B	Bruni

Supervisione = $\pi_{\text{Impiegato, Capo}}$ (Afferenza ⋈ Direzione)
$\sigma_{\text{Capo='Bruni'}}$ (Supervisione)

viene eseguita come
$\sigma_{\text{Capo='Bruni'}}(\pi_{\text{Impiegato, Capo}}$ (Afferenza ⋈ Direzione))

Grazie alle viste l'utente vede solo ciò che gli interessa e nel modo a cui interessa, inoltre vede solo ciò che è autorizzato a vedere. Come appena visto servono a semplificare la scrittura di query ed espressioni complessi, nonché sottoespressioni ripetute. Il loro utilizzo non condiziona l'efficienza delle interrogazioni.

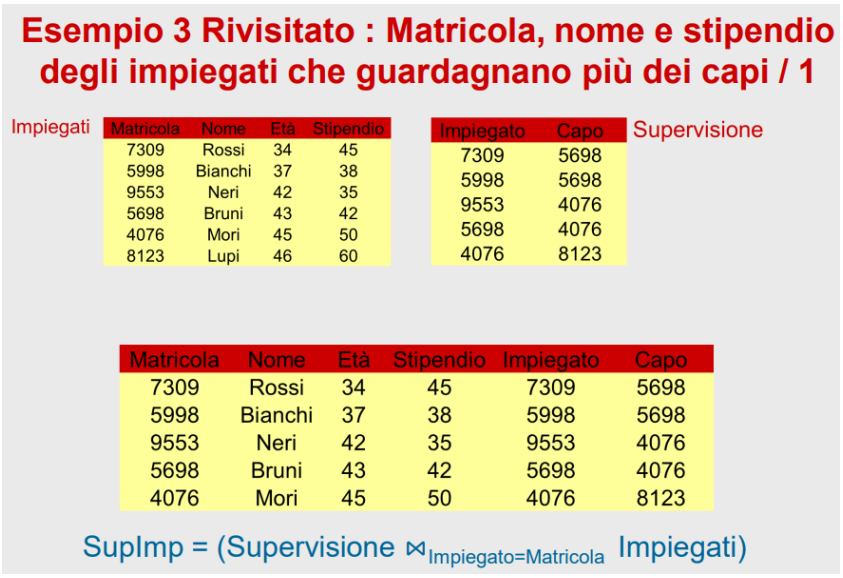
È possibile aggiornare un database tramite una vista se ottenibile tramite un join completo delle tabelle di partenza (essendo le viste una semplificazione delle query, qualora io volessi aggiornare, si richiede proprio almeno un'attinenza tra i campi presenti).
Tuttavia, la cosa non è sempre possibile, ad esempio con campi vuoti, come nell'esempio sotto. Si noti che si perdono per strada alcuni campi.



Quindi è utile cercare di modificare le relazioni di base affinché tramite la vista si riesca ad effettuare correttamente un aggiornamento. Ciò come visto è possibile in pochi casi, tra cui il join completo.

L'approccio usato in SQL ignora il natural join (introdotto questo solo nelle versioni più recenti del linguaggio)e riconosce attributi con lo stesso nome premettendo il nome della relazioni, usando invece le viste per ridenominare le relazioni.
Fondamentale dire una cosa semplice: una vista, all'atto pratico, è semplicemente una query ripetuta diverse volte o molto lunga scritta in maniera più corta, oppure una query ottimizzata usata per "compattare" la dicitura relazionale e conseguentemente la scrittura della query. Viene descritto come strumento semplificante per questa motivazione.

Si noti come nelle seguenti tabelle si applichi un join e una selezione direttamente, usando poi una *vista* per poter rinominare questa relazione, quindi *SupImp* e poi applicando selezione/proiezione.
Dunque:



Matricola	Nome	Età	Stipendio	Impiegato	Capo
7309	Rossi	34	45	7309	5698
5998	Bianchi	37	38	5998	5698
9553	Neri	42	35	9553	4076
5698	Bruni	43	42	5698	4076
4076	Mori	45	50	4076	8123

Impiegati

Matricola	Nome	Età	Stipendio
7309	Rossi	34	45
5998	Bianchi	37	38
9553	Neri	42	35
5698	Bruni	43	42
4076	Mori	45	50
8123	Lupi	46	60

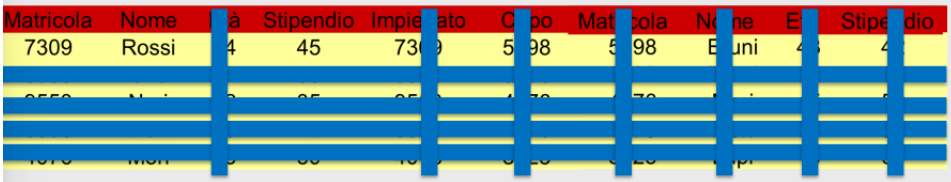
SupImp = (Supervisione $\bowtie_{Impiegato=Matricola}$ Impiegati)

SumImp | Impiegati

Matricola	Nome	Età	Stipendio	Impiegato	Capo	Matricola	Nome	Età	Stipendio
7309	Rossi	34	45	7309	5698	5698	Bruni	43	42
5998	Bianchi	37	38	5998	5698	5698	Bruni	43	42
9553	Neri	42	35	9553	4076	4076	Mori	45	50
5698	Bruni	43	42	5698	4076	4076	Mori	45	50
4076	Mori	45	50	4076	8123	8123	Lupi	46	60

SupImp $\bowtie_{Capo=Impiegati.Matricola}$ Impiegati

Esempio 3 Rivisitato : Matricola, nome e stipendio degli impiegati che guardagnano più dei capi / 3



$\pi_{SupImp.matricola,\ SupImp.Nome,\ SupImp.Stipendio}($
$\sigma_{SupImp.Stipendio>Impiegato.Stipendio}\ (SupImp \bowtie_{Capo=Impiegati.Matricola} Impiegati)$
$)$

Idiomi frequenti di interrogazione:

- Minimo e Massimo Assoluto

Dato lo schema relazionale $R(A, B)$, trovare il minimo/massimo in R. Si supponga di voler determinare il minimo B:

$$\pi_B(R) - \pi_B(R \bowtie_{B>B^1} (\rho A^1, B^1 \leftarrow A, B(R)))$$

Nella seconda parte vengono trovati tutti quei valori che non sono il minimo. Per fare ciò si deve joinare la relazione R con un'altra istanza di se stessa, con gli attributi ridenominati. La condizione del theta join indica che ogni attributo B deve essere maggiore degli stessi attributi ridenominati. In tal modo vengono mantenute tutte le tuple tranne quella in cui l'attributo B assume il valore minore.

- Minimo e Massimo Relativo

Dato lo schema relazionale $R(A, B)$, trovare per ogni A il minimo/massimo in R. Si supponga di voler determinare il massimo B in A:

$$\pi_{A,B}(R) - \pi_{A,B}(R \bowtie_{A=A^1 \wedge B<B^1} (\rho A^1, B^1 \leftarrow A, B(R)))$$

È molto simile al massimo assoluto. Il theta join in questo caso seleziona tutti i valori minimi di B per ogni attributo A.
Esempio con la relazione **ESAMI** in cui A sia Studente e B sia Voto:

$$S1 := \rho_{Studente^1, Voto^1 \leftarrow Studente, Voto}(\mathbf{ESAMI})$$
$$\pi_{Nome,Voto}(\mathbf{ESAMI}) - \pi_{Nome,Voto}(\mathbf{ESAMI} \bowtie_{Studente=Studente^1 \wedge Voto<Voto^1} (S1))$$

Seguono esercizi (si consideri che S1 ed S2 sono a tutti gli effetti due viste su cui andiamo ad effettuare operazioni).

- Si consideri la seguente base di dati dei diversi servizi sociali in città diverse di Italia:
  SERVIZI_SOCIALI (Citta, Servizio, Anno, Spesa)
  POSIZIONE (Citta, Regione, Abitanti)
  - Non è possibile avere due città con lo stesso nome.
  - Una tupla (Padova, Babysitting, 2019, 30000) in SERVIZI SOCIALI indica che Padova ha speso nel 2019 la cifra di 30000€ per il servizio sociale Babysitting.

Esercizio 1

1) Restituire le città che forniscono almeno due servizi sociali.

Riporto una possibile tabella di esempio (descrivo poi intuitivamente a parole la soluzione riportata sopra):

Città	*Servizio*	*Anno*	*Spesa*
Padova	B	2020	40K€
Padova	P	2019	50K€
Roma	B	2020	300K€

Città	*Servizio*
Padova	B
Padova	P
Roma	B

In pratica applico l'equijoin tra s1 città e s2 città e poi impongo che il servizio di s1 sia diverso da quello di s2.
Collegherei quindi la prima tupla della prima tabella con la seconda tupla della seconda tabella e la seconda tupla della prima tabella con la seconda tupla della seconda tabella. Applico poi la selezione.

S1=SERVIZI_SOCIALI
S2=SERVIZI_SOCIALI

$\pi_{S1.CITTA}$ (S1 $\bowtie_{S1.CITTA=S2.CITTA \text{ AND } S1.SERVIZIO<>S2.SERVIZIO}$ S2)

2) Restituire le città che forniscono esattamente un servizio sociale

S1=SERVIZI_SOCIALI
S2=SERVIZI_SOCIALI

$\pi_{S1.CITTA}(S1 \bowtie_{S1.CITTA=S2.CITTA \text{ AND } S1.SERVIZIO<>S2.SERVIZIO} S2)$

Esercitazione 1 – Algebra relazionale

Simboli: ⋈ σ ρ π

Considerare una relazione
R(A, B, C, D, E).

Indicare quali delle seguenti proiezioni hanno certamente lo stesso numero di ennuple di R:

Ragionamento semplice: basta che si consideri una proiezione che comprenda anche le chiavi B e C, quindi:

1. $\pi_{ABCD}(R)$ **SI**
2. $\pi_{AC}(R)$ **NO**
3. $\pi_{BC}(R)$ **SI**
4. $\pi_{C}(R)$ **NO**
5. $\pi_{CD}(R)$ **NO**

Considerare le relazioni
R_1(A, B, C) con cardinalità N_1
R_2(D, E, F) con cardinalità N_2
Assumere che sia definito un vincolo di integrità referenziale fra:
l'attributo C di R_1 e la chiave D di R_2

Indicare la cardinalità (K) di ciascuno dei seguenti join (specificare l'intervallo nel quale essa può variare):

1. $R_1 \bowtie_{A=D} R_2$
2. $R_1 \bowtie_{C=D} R_2$
3. $R_1 \bowtie_{A=F} R_2$
4. $R_1 \bowtie_{B=E} R_2$

1) Qui stiamo collegando due chiavi; quindi ci aspettiamo un numero di attributi che considera solo i valori utili di entrambe le relazioni.
Il collegamento utile qui è rappresentato da K_1

Detto ciò:

$$0 \leq |K_1| \leq \min(N_1, N_2)$$

2) Il collegamento viene fatto su K_2 da parte di un campo che ha un vincolo di integrità referenziale e nulla di più. Ci si aspetta un numero di tuple esattamente pari alla relazione che ha il vincolo di integrità referenziale con il campo chiave.
Detto ciò:

$$|K_2| = N_1$$

3) Qui colleghiamo un campo chiave con un campo non chiave; non si hanno perciò vincoli di integrità referenziale di mezzo e il numero di tuple varia da 0, al numero di tuple della prima relazione con campo chiave (R_1) fino al numero di tuple della seconda relazione (N_2)
Detto ciò:

$$0 \leq |K_3| \leq N_2$$

4) Qui colleghiamo due campi non chiave tra di loro e potenzialmente otteniamo il prodotto cartesiano, partendo sempre da un minimo di 0, un intermedio della prima relazione considerata ed un massimo pari appunto al natural join.
Detto ciò:

$$0 \leq |K_4| \leq N_1 \cdot N_2$$

Si assume nel contesto di questa esercitazione:

1. Il simbolo $\bowtie_{\mathbf{C}}$ denota il join di relazioni che "mette insieme" le tuple delle due relazioni che soddisfano la condizione **C**
2. Il simbolo $\bowtie$ senza condizione denota il join naturale, cioè join su valori uguali di attributi con lo stesso nome.
3. Si può usare il punto (".") per indicare l'attributo di una relazione (per esempio **R1.a** indica l'attributo **a** della relazione **R1**)

Note utili:

- Noi usiamo le "viste" come mezzo per evitare le ridenominazioni (p) previste in algebra relazionale classica, mezzo utile per eventualmente applicare delle selezioni e/o operare sulla stessa tabella facilmente usando equi-join
- Noi usiamo, sempre al fine di accorciare le query relazionali e, allo stesso tempo, evitare ridenominazioni ottimizzando, operazioni con il theta—join, che infatti applica una selezione al suo interno (quindi una condizione). Graficamente è come segue:

Algebra relazionale con join naturale

$$DIPENDENTI\_DI\_SIENA = \pi_{NomeDip}(\ DIPENDENTI \bowtie \rho_{CittaResidenza \leftarrow CodiceCitta}(\sigma_{NomeCitta='Siena'}(CITTA)))$$

$$interrogazione\ \pi_{NomeDip}(\ DIPENDENTI) - DIPENDENTI\_DI\_SIENA$$

Algebra relazionale con theta- join

$$DIPENDENTI\_DI\_SIENA = \pi_{NomeDip}(\ DIPENDENTI \bowtie_{CittaResidenza=CodiceCitta}(\sigma_{NomeCitta='Siena'}(CITTA)))$$

$$interrogazione\ \pi_{NomeDip}(\ DIPENDENTI) - DIPENDENTI\_DI\_SIENA$$

- Non è indispensabile mettere nell'equi-join il nome del campo di collegamento se il nome è uguale; serve solo se è diverso (si veda anche dall'immagine, se fosse Citta=Citta sarebbe inutile, ma in questo contesto abbiamo CittaResidenza = CodiceCitta, quindi si rende necessaria tale distinzione)
- Si ricorda che l'ordine delle operazioni è: Proiezione/Selezione/Join

Si consideri lo schema di base di dati che contiene le seguenti relazioni:
FORNITORI (fid: integer, fnome: String, indirizzo: String)
PEZZI(pid: integer, pnome: String, colore: String)
CATALOGO (fid: integer, pid: integer, costo: real)

1. Trovare i nomi dei fornitori che forniscono pezzi rossi
2. Trovare i fid dei fornitori che forniscono pezzi rossi o pezzi verdi
3. Trovare i fid dei fornitori che forniscono pezzi rossi o si trovano a via Cavour
4. Trovare i fid dei fornitori che forniscono sia pezzi rossi che pezzi verdi
5. Trovare coppie di fid tali che il fornitore con il primo fid applica per alcuni tipi di pezzo un prezzo maggiore di quello del fornitore con il secondo fid.
6. Trovare i pid dei tipi di pezzi forniti da almeno due diversi fornitori
7. Trovare i pid del tipo di pezzo più costoso fornito da "UniPd" (assumendo che sia uno)

1) $\pi_{fnome}(FORNITORI \bowtie(\sigma_{colore="Rosso"}(PEZZI) \bowtie CATALOGO))$
2) $\pi_{fid}((\sigma_{colore="Rosso"}(PEZZI)) \bowtie CATALOGO) \cup \pi_{fid}((\sigma_{colore="Verde"}(PEZZI)) \bowtie CATALOGO)$
3) $\pi_{fid}((\sigma\ colore="Rosso"(PEZZI) \bowtie CATALOGO)) \cup \pi_{fid}(\sigma\ indirizzo="Via\ Cavour"(FORNITORI))$
4) $\pi_{fid}(FORNITORI \bowtie(\sigma_{colore="Rosso"}(PEZZI) \bowtie CATALOGO)) \cap \pi_{fid}(FORNITORI \bowtie(\sigma_{colore="Verdi"}(PEZZI) \bowtie CATALOGO))$
5) R1 = Catalogo R2 = Catalogo
$\pi_{R1.fid,\ R2.fid}(R1 \bowtie_{R1.pid=R2.pid \wedge R1.costo > R2.costo} R2)$
6) R1 = Catalogo R2 = Catalogo
$\pi_{R1.pid}(R1 \bowtie_{R1.pid=R2.pid \wedge R1.fid \neq R2.fid} R2)$
7) R1 = Catalogo $\bowtie(\sigma_{fnome="UniPd"}$ Fornitori) R2 = Catalogo $\bowtie(\sigma_{fnome="UniPd"}$ Fornitori)
$\pi_{R1.pid}(R1 \bowtie_{R1.fid=R2.fid \wedge R1.costo < R2.costo} R2)$

AR) Si consideri lo schema relazionale composto dalle seguenti relazioni:
IMPIEGATO (Matricola, Cognome, Stipendio, Dipartimento)
DIPARTIMENTO (Codice, Nome, Sede, Direttore)

Con i seguenti vincoli di integrità referenziale:
- tra Dipartimento della relazione IMPIEGATO e Codice della relazione DIPARTIMENTO
- tra Direttore della relazione DIPARTIMENTO e Matricola della relazione IMPIEGATO.

Scrivere nel foglio delle risposte le espressioni in algebra relazionale per le seguenti interrogazioni:

a) Trovare i cognomi degli impiegati che NON sono direttori di dipartimento (se un impiegato non direttore di dipartimento ha lo stesso cognome di un direttore di dipartimento qualsiasi, allora tale cognome non deve comparire)

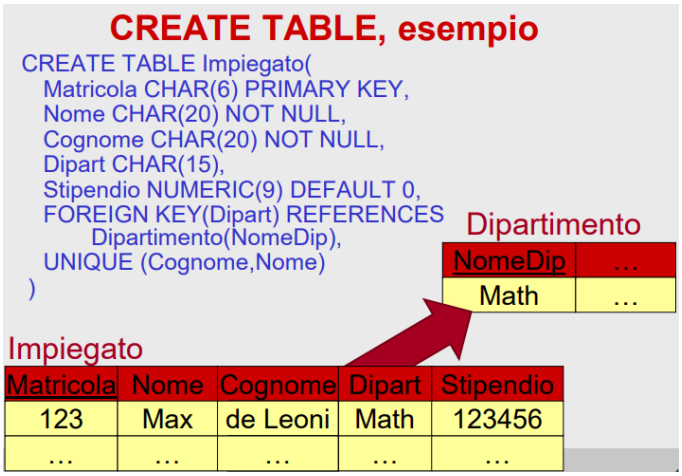
$\pi_{Cognome}(IMPIEGATO) - \pi_{Cognome}(DIPARTIMENTO \bowtie_{Direttore=Matricola} IMPIEGATO)$

b) Trovare i nomi dei dipartimenti in cui lavorano impiegati che guadagnano più di 60.000 euro

$\pi_{Nome}(\sigma_{Stipendio>60000}$ (IMPIEGATO $\bowtie_{Dipartimento=Codice}$ DIPARTIMENTO))

SQL (Structured Query Language): domini, operazioni, join, operatori aggregati

Descriviamo ora il linguaggio SQL, normalmente utilizzato all'interno di basi di dati con schema relazionale, in particolare descrivendo le istruzioni DDL e DML.
Un esempio concreto è l'istruzione ***CREATE TABLE,*** che definisce uno scherma di relazione, ne crea un'istanza vuota e ne specifica attributi, domini e vincoli. Qui ne si riporta subito un esempio:



Ogni attributo ha un ***dominio*** di definizione, tale che ognuno possa fare parte di *domini elementari* (predefiniti) e/o *domini definiti dall'utente* (semplici e riutilizzabili).
Alcuni esempi di domini elementari (ad esempio vi può essere l'array di byte per eventuale memorizzazione di file o immagini, oppure anche i domini *blob/clob*, per memorizzare oggetti binari/carattere di grandi dimensioni):

- Stringhe di lunghezza X:
  - Fissa: char(X)
  - Approssimati: varchar(X)
- Tipi Numerici: integer, smallint, float, …
- Tipi Numerici esatti con X cifre intere (e Y decimali): numeric(X,Y)
- Data, ora, data+ora: date, time, timestamp
- Boolean

Essi possono essere creati e includendo particolari vincoli rendendo un dominio riutilizzabile, ma con parametri definiti, eventuali vincoli e valori di default, con l'istruzione *CREATE DOMAIN* come segue:

```
CREATE DOMAIN Voto
  AS SMALLINT DEFAULT NULL
  CHECK ( value >=18 AND value <= 30 )
```

Qui posso impostare per esempio un limite con dei vincoli, ad esempio *varying* che dice che la dimensione del mio dominio può variare di una certa quantità specificata.
Seguono poi i vincoli intrarelazionali, ad esempio:

- *NOT NULL,* il valore nullo non è ammesso come possibile valore di attributo;
- *UNIQUE*, impone che i valori dell'attributo siano una (super)chiave, quindi che righe differenti non possano avere gli stessi valori;
- *PRIMARY KEY*, definisce la chiave primaria;
- *UNIQUE + NOT NULL*, chiave (non primaria)

- *CHECK*, vincoli generici definiti.

Nel caso della definizione di attributo vengono inseriti UNIQUE e PRIMARY KEY in linea con l'attributo, se forma da solo la chiave, oppure come possibile elemento separato, nel caso coinvolga più attributi.
Attenzione alle chiavi su più attributi, non si ha la stessa cosa tra i due casi riportati con i due diversi colori.

```
Nome            CHAR(20) NOT NULL,
Cognome  CHAR(20) NOT NULL,
UNIQUE (Cognome,Nome),

Nome            CHAR(20) NOT NULL UNIQUE,
Cognome  CHAR(20) NOT NULL UNIQUE,
```

Non sono la stessa cosa:

- Caso sopra: (Cognome,Nome) è chiave
  = Impossibile avere due tuple con lo stesso cognome e lo stesso nome
- Caso sotto: Cognome è chiave + Nome è chiave
  = Impossibile avere due tuple con lo stesso cognome
  = Impossibile avere due tuple con lo stesso nome

Possono essere definiti vincoli di integrità relazionale in casi interrelazionali, come *REFERENCES* (creazione di un legame tra i valori di una tabella slave e quelli di una tabella master, cioè esterna) oppure *FOREIGN KEY* (vincolo che coinvolge più attributi, sull'altra tabella mi aspetterò uno UNIQUE o PRIMARY KEY).
È possibile definire politiche di reazione alla violazione.
Per esempio (vincoli violati quando cerco di modificare una tupla oppure una di queste è referenziata in altre relazioni):

Infrazioni

Codice	Data	Vigile	Stato	Numero
34321	1/2/15	3987	I	CC953MS
53524	4/3/15	3295	I	FV077XM
64521	5/4/16	3295	F	AB234ZK
73321	5/2/18	9345	F	AB234ZK

Auto

Stato	Numero	Cognome	Nome
I	CC953MS	Rossi	Mario
I	FV077XM	Rossi	Mario
F	AB234ZK	Neri	Luca

Vigili

Matricola	Cognome	Nome
3987	Rossi	Luca
3295	Neri	Piero
9345	Neri	Mario
7543	Mori	Gino

```
CREATE TABLE Infrazioni(
   Codice CHAR(5) PRIMARY KEY,
   Data DATE NOT NULL,
   Vigile INTEGER NOT NULL REFERENCES Vigili(Matricola),
   Stato VARCHAR(2),
   Numero VARCHAR(8) ,
   FOREIGN KEY(Stato, Numero) REFERENCES Auto(Stato, Numero),
   CHECK(Data > '01/01/2020')
)
```

Seguono le politiche di reazione per le singole operazioni, in questo caso per la *DELETE.*
Le politiche di reazione impostabili possono essere fatte a cascata (*cascade*), assegnazione del valore nullo al posto del valore cancellato (*set null*), setting del valore di default al posto del valore cancellato (*set default*), la cancellazione non viene consentita (*no action*).

Passiamo ora al caso *UPDATE.*
Le politiche di reazione impostabili possono essere fatte come propagazione di valore a cascata in altre tabelle (*cascade*), assegnazione del valore nullo al posto del valore modificato nella tabella (*set null*), setting del valore di default al posto del valore modificato nella tabella esterna (*set default*), la modifica non viene consentita (*no action*).
Di default la politica che si segue sia per *DELETE* che per *UPDATE* è la *cascade.*

Piccolo esempio in questo caso:

```
CREATE TABLE Infrazioni(
   Codice CHAR(5) PRIMARY KEY,
   Data DATE NOT NULL,
   Vigile INTEGER NOT NULL REFERENCES Vigili(Matricola)
          on update cascade
          on delete no action,
   …
)
```

Tipicamente questi strumenti vengono implementati in modalità grafica, tramite schema della base di dati poi tradotto internamente in SQL.
Vi sono una serie di operazioni sulle tuple, classico inserimento (*INSERT),* eliminazione (*DELETE*), modifica (*UPDATE*). Esempio di inserimento qui a destra (importante l'ordine e la corrispondenza degli attributi tra le singole operazioni):

```
INSERT INTO Tabella [ ( Attributi ) ]
      VALUES( Valori )

oppure

INSERT INTO Tabella [ ( Attributi )]
       SELECT ...
```

Le liste quindi devono avere lo stesso numero di elementi; se la lista di attributi è omessa, si fa riferimento a tutti gli attributi della relazione. Se invece la lista non contiene tutti gli attributi, per gli altri viene inserito il valore nullo/default. Quindi appunto l'ordine degli attributi è significativo.
L'eliminazione delle tuple avviene appunto con *DELETE*; meglio mettere una condizione altrimenti cancella tutti i contenuti di una certa tabella (perché di default considera la condizione WHERE true quindi attenzione).

```
DELETE FROM Persone
WHERE Eta < 35

DELETE FROM Paternita
WHERE Figlio NOT in (SELECT Nome
                     FROM Persone)

DELETE FROM Paternita
```

Segue la modifica delle tuple, una o più, con l'istruzione *UPDATE*, con le successive istruzioni *SET* di impostazione. Ad esempio:

```
UPDATE NomeTabella
SET  Attributo = < Espressione |
                   SELECT … |
                   NULL |
                   DEFAULT >
      [ WHERE Condizione ]
```

```
UPDATE Persone SET Reddito = 45
  WHERE Nome = 'Piero'

UPDATE Persone
  SET Reddito = Reddito * 1.1
  WHERE Eta < 30
```

Altra istruzione utile è la *SELECT*, che permette di ricavare attributi *da* (*FROM*) alcune tabelle *dove (WHERE)* si setta una condizione. Gli attributi considerati dalla SELECT sono detti *target list*.
Importante inoltre che gli attributi di SELECT sono quelli su cui si fa la proiezione, mentre quelli su cui si esegue la WHERE sono quelli su cui si opera la selezione.

Nome e reddito delle persone con meno di trenta anni

$\pi_{Nome,\ Reddito}(\sigma_{Eta<30}(Persone))$

```
SELECT Nome, Reddito
FROM Persone
WHERE Eta < 30
```

```
SELECT Nome, Reddito
FROM Persone
WHERE Eta < 30

SELECT P.Nome as Nome,
       P.Reddito as Reddito
FROM Persone as P
WHERE P.Eta < 30
```

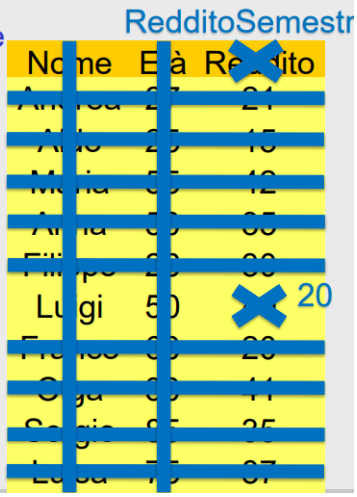
Nel caso a destra la tabella viene rinominata con l'uso di *AS* e su essa viene eseguita una proiezione. In questo caso a volte per velocizzare si può anche evitare di scrivere AS e la tabella assume un alias comunque. Ad esempio: Persone AS P o Persone P è uguale concettualmente.
La selezione senza proiezioni, quindi il "*seleziona tutto*", avviene tramite l'operatore *star (*)*.

```
SELECT *
FROM Persone
WHERE Eta < 30
```

Vediamo un esempio concreto (si segnala che l'algebra relazionale indica opera abuso di notazione, in quanto non è così che propriamente si realizza, dice il prof).

```
SELECT Reddito/2 as
RedditoSemestrale
FROM Persone
WHERE Nome = 'Luigi'
```

$\rho_{RedditoSemestrale \leftarrow Reddito}(\pi_{Reddito}(\sigma_{Nome="Luigi"}(Persone))/2)$



Le condizioni complesse invece combinano e formano molteplici condizioni, come si vede qui a destra:

```
SELECT *
FROM Persone
WHERE Reddito > 25
    and (Eta < 30 or Eta > 60)
```

$\sigma_{(Eta < 30 \text{ or } Eta > 60) \text{ and } Reddito > 25}(Persone)$

Si ha poi la condizione *LIKE* che analizza il contenuto parziale di una stringa, ritrovando delle somiglianze o delle sottostringhe. Esso utilizza la *percentuale*, sostituendo un insieme di caratteri di una stringa oppure *l'underscore*, che sostituisce il singolo carattere della stringa.

```
SELECT *
FROM Persone
WHERE Nome like 'A_d%'
```

Gli stessi valori nulli vengono gestiti normalmente con *IS NULL*.
Attenzione alle differenze tra SQL ed Algebra Relazionale. Se volessi fare in modo di prendere i valori duplicati come avviene nell'algebra relazionale si usa la parola *DISTINCT* (altrimenti restituirei tutte le tuple di un certo tipo avendo anche duplicati).

Scritto da Gabriel

La combinazione della *SELECT* più relazioni nella *FROM* permettono di realizzare join e prodotti cartesiani.

Esempio:
Supponiamo due relazioni R1(A1,A2) e R2(A1,A3)
La query $\pi_{R1.A1,A3}(\sigma_{R1.A1>R2.A1}(R1 \bowtie R2))$ è

```
SELECT R1.A1, A3
FROM   R1, R2
WHERE  R1.A1 > R2.A1
```

La ridenominazione poi si esegue con *AS*, sulla base di precedenti/successive operazioni tra tabelle/relazioni, con le stesse funzionalità della classica ridenominazione, come accennato prima.

Supponiamo una relazione R1(A1,A2)
R2=R1
$\rho_{B1 \leftarrow R1.A1, B2 \leftarrow R1.A2}(\pi_{R1.A1,R1.A2}(\sigma_{R1.A1>R2.A1}(R1 \bowtie R2)))$

diventa:

```
SELECT R1.A1 AS B1, R1.A2 AS B2
FROM  R1, R1 AS R2
WHERE  R1.A1 > R2.A1
```

Le interrogazioni vengono comunque ottimizzate per conto nostro da parte dei DBMS; importante per noi non è tanto l'efficienza, quanto piuttosto la chiarezza delle interrogazioni da parte nostra, aggiungendo tutti i vincoli del caso. Se ad esempio volessi *"I padri di persone che hanno come reddito più di 20"*:

$\pi_{Padre}(\sigma_{Reddito>20 \wedge Figlio=Nome}(paternita \bowtie persone))$

Maternità

Madre	Figlio
Luisa	Maria
Luisa	Luigi
Anna	Olga
Anna	Filippo
Maria	Andrea
Maria	Aldo

Paternità

Padre	Figlio
Sergio	Franco
Luigi	Olga
Luigi	Filippo
Franco	Andrea
Franco	Aldo

Persone

Nome	Età	Reddito
Andrea	27	21
Aldo	25	15
Maria	55	42
Anna	50	35
Filippo	26	30
Luigi	50	40
Franco	60	20
Olga	30	41
Sergio	85	35
Luisa	75	87

```
SELECT DISTINCT Padre
FROM Persone, Paternita
WHERE Figlio = Nome and Reddito > 20
```

Si noti la scrittura del JOIN all'interno della WHERE; è un altro modo di scrivere la join. In algebra relazionale avremo scritto proprio (proiezione padre, join paternita, persone con figlio=nome, selezione reddito > 20).
Possiamo inoltre seguire una serie di relazioni con una serie di ridenominazioni e vincoli.
Ad esempio, nel caso *"Il nome delle persone che guadagnano più dei rispettivi padri"*:

```
SELECT DISTINCT Paternita.Figlio
FROM Persone AS P1, Persone AS P2, Paternita
WHERE Figlio = P1.Nome AND Padre = P2.Nome
                AND P2.Reddito < P1.Reddito
```

Parliamo poi dei tipi di join, ad esempio vediamo i join espliciti:

```
SELECT Madre, Paternita.Figlio, Padre
FROM Maternita join Paternita on
      Paternita.Figlio = Maternita.Figlio
```

L'operazione di join, volendo, può essere scritta in maniera più compatta direttamente dentro il *FROM*, mettendolo quindi dentro il *WHERE* (basta una semplice uguaglianza). Per convenzione possiamo adottare il fatto di non usare le espressioni che non riguardano il *JOIN* dentro il *WHERE*.
Similmente possiamo usare gli *outer* join, nel caso d'uso il left join (se ce ne sta uno solo dei due, si metterà *NULL*; questo è molto utile nelle query che richedono "se manca il valore nella tabella X, mettere NULL").

```
SELECT Padre, Paternita.Figlio, Madre
FROM Paternita left join Maternita
      on Paternita.Figlio = Maternita.Figlio

SELECT Padre, Paternita.Figlio, Madre
FROM Paternita left outer join Maternita
      on Paternita.Figlio = Maternita.Figlio
```

La differenza tra "left join" ed "inner join", non ci sarebbe l'inclusione di alcune tuple nel risultato, in questo caso solo una.

Se invece eseguissi un "full outer join", quindi mettendo sia il left che right join all'interno dell'operazione, si avrebbe a tutti gli effetti l'inclusione di tutte le tuple.

Padre	Pat.Figlio	Mat.Figlio	Madre
Sergio	Franco	*NULL*	*NULL*
Luigi	Olga	Olga	Anna
Luigi	Filippo	Filippo	Anna
Franco	Andrea	Andrea	Maria
Franco	Aldo	Aldo	Maria
NULL	*NULL*	Maria	Luisa
NULL	*NULL*	Luigi	Luisa

Righe aggiunte perché «full join»

Risultato: Differenza tra "left join" e "inner join"

Maternità

Madre	Figlio
Luisa	Maria
Luisa	Luigi
Anna	Olga
Anna	Filippo
Maria	Andrea
Maria	Aldo

Paternità

Padre	Figlio
Sergio	Franco
Luigi	Olga
Luigi	Filippo
Franco	Andrea
Franco	Aldo

```
SELECT Padre, Paternita.Figlio, Madre
FROM Paternita left join Maternita
      on Paternita.Figlio =
Maternita.Figlio
```

Padre	Pat.Figlio	Madre
Sergio	Franco	*NULL*
Luigi	Olga	Anna
Luigi	Filippo	Anna
Franco	Andrea	Maria
Franco	Aldo	Maria

Se «Join» senza «Left Join» (conosciuto anche come «Inner Join»), la prima riga sarebbe esclusa dal risultato

Lo stesso risultato può essere ordinato tramite *ORDER BY,* quindi seguendo l'ordine di un attributo.
Questo viola il principio di non ordine delle tuple; normalmente infatti non si ha alcuna garanzia del loro ordine (es. classico ordine alfabetico).
Posso impostare anche l'ordine decrescente di ordinamento, aggiungendo *DESC* dopo aver messo il nome dell'attributo per il quale effettuare l'ordinamento (oppure esplicitare l'ordine crescente con *ASC*, comunque policy di default).

```
SELECT Nome, Reddito
FROM Persone
WHERE ETA < 31
ORDER BY Nome
```

Nome	Reddito
Aldo	15
Andrea	21
Filippo	30

Posso invece applicare *operatori aggregati*, avendo espressioni e valori a partire da insiemi di tuple, ad esempio:

- conteggio (*COUNT)*
- minimo *(MIN)*
- massimo (*MAX)*
- media (*AVG*)

- somma (*SUM)*

Ad esempio, il numero di figli di Franco:

```
SELECT count(*) as
  NumFigliDiFranco
FROM Paternita
WHERE Padre = 'Franco'
```

Si usa anche la variante *COUNT DISTINCT*, prendendo quindi le tuple senza duplicati (scritto come sotto COUNT(DISTINCT _campo_), eseguito sul reddito in questo caso e ne prende 2 (escludendo quelle due con 21 in quanto hanno stesso reddito).

```
SELECT count(*) FROM persone                    4

SELECT count(distinct reddito) FROM persone     2
```

Persone

Nome	Età	Reddito
Andrea	27	21
Aldo	25	35
Maria	55	21
Anna	50	35

Prendiamo poi la media dei redditi dei figli di Franco:
SELECT avg(reddito) FROM persone join paternita on nome=figlio WHERE padre='Franco'
Le operazioni aggregate ignorano I valori NULL; ad esempio nel COUNT, come si vede anche sotto, nonostante un valore nullo, normalmente (senza vincoli) conta comunque tutte le tuple.

```
SELECT count(*) FROM persone                    4

SELECT count(reddito) FROM persone              3

SELECT count(distinct reddito) FROM persone     2
```

Persone

Nome	Età	Reddito
Andrea	27	21
Aldo	25	NULL
Maria	55	21
Anna	50	35

Attenzione che le query devono essere scritte in modo logicamente e sintatticamente chiaro, come nel caso:

```
SELECT nome, max(reddito)
FROM persone
```

sulla tabella →

Persone

Nome	Età	Reddito
Andrea	27	30
Aldo	25	NULL
Maria	55	36
Anna	50	36

SQL: Operatori di raggruppamento e query nidificate

Come detto, abbiamo il raggruppamento degli operatori con GROUP BY, permettendo di fare gruppi per min/max/count, ecc. Il GROUP BY considera come attributi di raggruppamento quelli su cui sto facendo il SELECT; altrimenti, come capita in PostgreSQL darebbe errore se non facessi così.
La semantica delle interrogazioni è quindi

1) l'esecuzione della query senza la GROUP BY
2) raggruppamento delle tuple
3) reintroduzione/applicazione dell'operatore a ciascun gruppo.

Ulteriore filtro per considerare un particolare gruppo togliendone altri è la *HAVING*, che funziona come il *WHERE* ma per funzioni aggregate.
Ad esempio:

I padri con figli i cui reddito medic
maggiore di 25; mostrare padre e
reddito medio dei figli

```
SELECT padre, avg(reddito)
FROM persone, paternita
WHERE  nome=figlio
GROUP by padre
HAVING avg(reddito) > 25
```

Persone

Nome	Età	Reddit
Andrea	27	21
Aldo	25	15
Maria	55	42
Anna	50	35
Filippo	26	30

Paternità

Padre	Figlio
Sergio	Anna
Luigi	Maria
Luigi	Filippo
Franco	Andrea
Franco	Aldo

Prima di tutto eseguo tutta la parte prima dell'HAVING (forma il primo gruppo logico), poi imponendo una condizione dell'HAVING e prendendo il sottogruppo.
Attenzione all'ordine di applicazione di questi operatori; prima viene applicato il WHERE, poi raggruppa con GROUP BY ed infine usa HAVING. *Non ha senso di solito usare HAVING senza un GROUP BY.*

Nel caso riportato sotto esistono 4 gruppi, uno per ogni singolo valore di A.

A	B
1	11
2	11
3	null
4	null

Reintroduce il count, applicata sulla tupla di ogni gruppo.
A questo punto divide i gruppi in 2, tra quelli con valore B che è NULL e poi gli altri che non sono NULL.

Successivamente applica la selezione sulla base del criterio di ricerca su A.
L'esempio sotto riporta il caso completo appena descritto completamente:

```
SELECT B, count (*)
FROM R
GROUP by B
```

B	
11	2
null	2

```
SELECT A, count (*)
FROM R
GROUP by A
```

A	
1	1
2	1
3	1
4	1

```
SELECT A, count (B)
FROM R
GROUP by A
```

A	
1	1
2	1
3	0
4	0

Segue poi l'unione, che non viene eseguita solo con il SELECT, ma serve proprio un costrutto esplicito, cioè *UNION*, che elimina anche i duplicati (*a meno che non si usi "all"*). Li elimina anche dalla proiezione.
Esempio:

```
SELECT A,  B
FROM R
union
SELECT A , B
FROM S
```

```
SELECT A,  B
FROM R
union all
SELECT A , B
FROM S
```

Posizionalmente, quindi, avremo inoltre sulla query seguente, i nomi degli attributi della prima relazione in poi. Quindi qui prenderebbe prima il Padre, poi la Madre:

```
SELECT padre, figlio
FROM paternita
union
SELECT madre, figlio
FROM maternita
```

Maternità

Madre	Figlio
Luisa	Maria
Luisa	Luigi
Anna	Olga
Anna	Filippo
Maria	Andrea
Maria	Aldo

Paternità

Padre	Figlio
Sergio	Franco
Luigi	Olga
Luigi	Filippo
Franco	Andrea
Franco	Aldo

In merito quindi alla selezione, comunque vengono ordinati in base alla prima relazione e non cambia nulla; questo principio rimane tale anche per la ridenominazione.
Possiamo inoltre introdurre *EXCEPT*, quindi la differenza e *INTERSECT*, quindi l'operazione di intersezione; quest'ultima è zucchero sintattico, perché l'uso di INTERSECT è già coperto dalla WHERE e nella pratica non è poi utilizzatissimo. Cosa più interessante è la nidificazione delle interrogazioni, innestandole una dentro l'altra con l'uso di parentesi tonde:

Interrogazioni Nidificate

Nome e reddito del padre di Franco

```
SELECT Nome, Reddito
FROM Persone, Paternita
WHERE Nome = Padre
and Figlio = 'Franco'
```

Maternità

Madre	Figlio
Luisa	Maria
Luisa	Luigi
Anna	Olga
Anna	Filippo
Maria	Andrea
Maria	Aldo

Paternità

Padre	Figlio
Sergio	Franco
Luigi	Olga
Luigi	Filippo
Franco	Andrea
Franco	Aldo

Persone

Nome	Età	Reddito
Andrea	27	21
Aldo	25	15
Maria	55	42
Anna	50	35
Filippo	26	30
Luigi	50	40
Franco	60	20
Olga	30	41
Sergio	85	35
Luisa	75	87

```
SELECT Nome, Reddito
FROM Persone
WHERE Nome =    ( SELECT Padre
                  FROM Paternita
                  WHERE Figlio = 'Franco')
```

Interrogazione all'interno di un'altra

Sempre qui si consideri l'esempio delle "*persone con il reddito superiore alla media*":

```
SELECT *
FROM Persone
WHERE Reddito >=    (SELECT avg(Reddito)
                     FROM Persone)
```

Un altro esempio composto, in cui appare l'operatore ulteriore "*IN*", che permette una sottoselezione di ulteriori tuple all'interno di una lista e inoltre come ulteriore interrogazione all'interno di un'altra. Dunque, possiamo avere una cosa di questo tipo:

Nome e reddito dei padri di persone che guadagnano più di 20

```
SELECT distinct P.Nome, P.Reddito
FROM
Persone P, Paternita, Persone F
WHERE P.Nome = Padre and
        Figlio = F.Nome and F.Reddito > 20
```

Paternità

Padre	Figlio
Sergio	Franco
Luigi	Olga
Luigi	Filippo
Franco	Andrea
Franco	Aldo

Persone

Nome	Età	Reddito
Anna	50	35
Filippo	26	30
Luigi	50	40
Franco	60	20
Olga	30	41
Sergio	85	35
Luisa	75	87

Altro possibile operatore è *ANY*, con una selezione tra una delle qualsiasi tuple all'interno di un certo sottoinsieme. Ragiona con il principio dell' "*almeno uno che soddisfi una certa condizione*".

```
SELECT Nome, Reddito
FROM Persone
WHERE Nome in (SELECT Padre FROM Paternita, Persone
                WHERE Figlio = Nome and Reddito > 20)
```

Scritto da Gabriel

Similmente, possiamo avere l'operatore ALL, quindi prendendo invece tutte le tuple che soddisfano o non soddisfano una certa condizione di qualche tipo. Ultima keyword in questo senso importante è *EXISTS*, dove vengono estratte dal database le tuple/colonne che rispettano una certa condizione o l'esistenza di un certo record nelle subqueries. *Attenzione: l'uso di EXISTS prevede nella sottoquery, normalmente, la selezione totale (quindi di tutti gli attributi, cioè usando (*); se si cerca un attributo specifico, può bastare EXISTS (nome_colonna))*.

Estrarre le persone omonime

```
SELECT *
FROM Persona AS P
WHERE EXISTS    (SELECT *
                 FROM PERSONA Q
                 WHERE Q.Nominativo=P.Nominativo
                 AND Q.CodFisc≠P.CodFisc)
```

Questa è possibilmente da evitare, date le condizioni di ricerca mischiando tra loro la ricerca di tuple interne ed esterne. Senza EXISTS:

Estrarre le persone omonime

```
SELECT P.*
FROM Persona AS P, Persona AS Q
WHERE P.Nominativo = Q.Nominativo AND
            P.CodFisc <> Q.CodFisc
```

Similmente, per contro si ha la variante *NOT EXISTS*, che sarà naturalmente il complemento di quanto descritto. *Attenzione: l'uso di NOT EXISTS prevede nella sottoquery, normalmente, la selezione totale (quindi di tutti gli attributi, cioè usando (*); se si cerca un attributo specifico, può bastare NOT EXISTS (nome_colonna))*. Ad esempio:

```
SELECT *
FROM Persona AS P
WHERE NOT EXISTS    (SELECT *
                     FROM PERSONA Q
                     WHERE Q.Nominativo=P.Nominativo
                     AND Q.CodFisc≠P.CodFisc)
```

Similmente senza EXISTS abbiamo questa situazione:

```
SELECT P.*
FROM Persona AS P
WHERE P.Nominativo NOT IN
                (SELECT Nominativo
                FROM Persona AS Q
                WHERE Q.Nominativo = P.Nominativo
                        AND Q.CF <> P.CF)
```

Attenzione che nelle query annidate deve essere ben chiara la visibilità dei dati (cioè basta scrivere bene la sottoquery). Ad esempio questo, dove le tuple rappresentate non sono interne tra di loro.

```
SELECT *
FROM Impiegato
WHERE Dipart in (SELECT Nome
                 FROM Dipartimento D1
                 WHERE Nome = 'Produzione') or
      Dipart in (SELECT Nome
                 FROM Dipartimento D2
                 WHERE D2.Citta = D1.Citta)
```

ERRORE DEL LIBRO SEGNALATO DAL PROF:
Se eseguo una SELECT su un attributo su cui eseguo anche il GROUP BY, il libro dice che funziona sempre l'aggregazione degli attributi che sono nel GROUP BY; tuttavia nel caso di PostgreSQL non funziona.

Seguono alcuni esercizi di riepilogo fatto sulle seguenti tabelle:

Persone

Nome	Reddito	Età	Sesso
Mario	15	80	M
Carlo	25	24	M
Giuseppe	30	45	M
Maria	76	43	F
Gianni	60	50	M
Francesca	18	26	F
Paola	45	60	F
Marco	80	35	M
Antonio	15	86	M

Genitori

Figlio	Genitore
Paola	Mario
Marco	Paola
Carlo	Gianni
Carlo	Maria
Francesca	Giuseppe
Marco	Giuseppe
Gianni	Antonio

1) Trovare l'elenco ordinato dei genitori in cui almeno un figlio guadagna più di 20 milioni

```
SELECT DISTINCT Genitore
FROM Genitori
WHERE Figlio
IN (SELECT Nome FROM Persone WHERE Reddito > 20)
ORDER BY Genitore
```

oppure

```
SELECT DISTINCT Genitore
FROM Genitori
JOIN Persone ON Nome=Figlio
WHERE Reddito > 20 ORDER BY Genitore
```

2) Trovare i nonni di ogni persona

```
SELECT G1.Figlio as Nipote, G2.Genitore As Nonno
FROM GEN G1 JOIN GEN G2
ON G1.Genitore=G2.Figlio
```

3) Trovare il numero di figli per genitore

```
SELECT Genitore, COUNT (*) AS Num_figli
```

Scritto da Gabriel

```
FROM Genitori
GROUP BY Genitore
```

4) Per ogni genitore trovare la somma del reddito di tutti i figli

```
SELECT Genitore, SUM Reddito AS Reddito_figli
FROM Genitori JOIN Figli ON Nome=Figlio
GROUP BY Genitore
HAVING Sum(Reddito)
```

oppure

```
SELECT Genitore, SUM(Reddito)
FROM Genitori, Persone
GROUP BY Genitore
WHERE Nome=Figlio
```

5) Trovare il reddito medio per genere

```
SELECT Sesso, AVG(Reddito) AS Sum_reddito
FROM Persone
GROUP BY Sesso
```

6) Trovare la donna che guadagna di più

```
SELECT Nome, MAX(Reddito) AS Max_reddito
FROM Persone
WHERE Sesso=”F”
```

oppure

```
SELECT *
FROM Persone
WHERE Sesso=”F” AND Reddito = SELECT (MAX(Reddito) FROM Persone WHERE Sesso=”F”))
```

Esercitazione 2: SQL

Fa parte di uno degli esercizi del libro (presenti su Mega nella cartella BasidiDati_5Ed), libro di riferimento di questo anno di corso

Dato lo schema seguente, formulare le query SQL:

ESECUZIONE (CodiceReg, TitoloCanz, Anno)
AUTORE (Nome,TitoloCanzone)
CANTANTE (NomeCantante, CodiceReg)

- Autori e cantanti puri, cioè autori che non hanno mai registrato una canzone e cantanti che non hanno mai scritto una canzone.

```
SELECT Nome FROM Autore
FROM Autore
```

Scritto da Gabriel

```
WHERE Nome NOT IN (SELECT Nome_Cantante FROM Cantante
UNION
SELECT NomeCantante FROM Cantanti
WHERE NomeCantante NOT IN (SELECT Nome FROM Autore)
```

- I cantanti che hanno solamente cantato canzoni che sono anche cantate da altri cantanti (la rifaremo, ha generato non poche controversie in classe)

```
SELECT C.NomeCantante
FROM Cantante C, Esecuzione E, Cantante C2, Esecuzione E2
WHERE C.CodReg = E.CodReg AND C2.CodReg = E2.CodReg
WHERE C.Nome < > C2.Nome AND C.Titolo = C2.Titolo
```

- Per ogni canzone scritta da "Mogol" restituire il numero di esecuzioni registrate per ogni anno
  Nota: Nel Count va bene anche lo (*)

```
SELECT Anno, COUNT(TitoloCanz) As Numero_canzoni
FROM Autore JOIN Esecuzione
ON Autore.TitoloCanz=Esecuzione.TitoloCanz
WHERE Autore="Mogol"
GROUP BY Anno, TitoloCanz
```

Note utili SQL:

- L'ordine di esecuzione delle query è quello previsto a lato:
- Normalmente, anche in Postgre, le stringhe sono indicati ad apici singoli ' ' e non a virgolette " "

- `FROM/JOIN` and all the `ON` conditions
- `WHERE`
- `GROUP BY`
- `HAVING`

Concetti avanzati di SQL

In generale nelle tuple, approfondiamo il concetto del CHECK, per approfondire la specifica di vincoli fra tuple della stessa/diversa relazione nel formato CHECK(Condizione). La buona scrittura dei vincoli in CHECK è utile per scrivere dati compatti e ben leggibili.

Ad esempio, in questo caso, controlliamo lo stipendio del dipendente, che deve essere minore di quello del capo.
L'operazione descritta contrasta la logica generale dei DBMS, non supportando mai come si vede:

```
CREATE TABLE Imp
(
Matricola INTEGER PRIMARY KEY,
Nome CHARACTER(20),
Eta INTEGER CHECK (Eta > 16 AND Eta<100),
Stipendio INTEGER CHECK (Stipendio > 0) ,
Capo INTEGER,
CHECK (Stipendio <= (SELECT Stipendio
                     FROM Imp J
                     WHERE Capo = J.Matricola) )
)
```

Non supportato da quasi tutti i DBMS

In generale quindi ci riferiamo solo alla tupla che sto inserendo; il CHECK innestato, quindi non rappresenta in generale una buona soluzione.
Meglio fare così (esempio funzionante, assumendo che lo stipendio sia lordo):

```
CREATE TABLE Imp
(
      Matricola INTEGER PRIMARY KEY,
      Nome CHARACTER(20),
      Eta INTEGER CHECK (Eta > 16 AND Eta<100),
      Stipendio INTEGER CHECK (Stipendio > 0) ,
      Capo INTEGER,
      Ritenute INTEGER,
      Netto INTEGER,
      CHECK (Stipendio =Netto+Ritenute )
)
```

Similmente posso definire *asserzioni*, cioè specifici vincoli a livello di schema/database piuttosto che ad una singola tupla, quindi riguardando più tabelle esprimendo vincoli altrimenti non definibili. Esso non è del tutto supportato, attualmente. Se le condizioni non vengono soddisfatte, l'intera tabella o database potrebbe non funzionare. Nei DB moderni non si usano; per eliminarle si può usare la DROP.

```
CREATE ASSERTION NomeAss
  CHECK ( Condizione )

CREATE ASSERTION AlmenoUnImpiegato
    CHECK (1 <= (       SELECT COUNT(*)
                  FROM Impiegato ))
```

Ritorniamo alle *viste*, create esplicitamente con *CREATE VIEW*, eventualmente aggiornandole con INSERT/UPDATE/DELETE, vengono utilizzate per semplificare le query e si trattano come fossero tabelle, una volta eseguita la query e rinominata come vista. Un esempio concreto:

```
CREATE VIEW ImpiegatiNonCapo(Nome, Eta, Stipendio) AS
    SELECT Nome, Eta, Stipendio
    FROM Impiegato
    WHERE Nome NOT IN
              Matricola   (SELECT Capo FROM Impiegato)
```

Le viste poi vengono usate nelle interrogazioni come relazioni di base, quindi come tabelle in pratica, come nell'esempio sottostante. Da un punto di vista computazionale e pratico, importante comunque non abusarne.

```
  SELECT * FROM ImpiegatiNonCapo

equivale a (e viene eseguita come)

    SELECT Nome, Eta, Stipendio
    FROM Impiegato
    WHERE Nome NOT IN
                    (SELECT Capo FROM Impiegato)
```

Nel caso invece di *aggiornamenti tra viste*, essi normalmente sono ammessi solo su viste definite su una relazione, perché altrimenti il DBMS dovrebbe verificare al volo ogni singola volta che ci sono queste situazioni se tutti i vincoli sono rispettati sulle tabelle. Si mette la clausola come definito sotto, non permettendo ad una tupla di uscire da una vista.

```
CREATE VIEW ImpiegatiNonCapoPoveri as
    SELECT *
    FROM ImpiegatiNonCapo
    WHERE Stipendio < 50
    WITH CHECK OPTION
```

- CHECK OPTION permette modifiche, ma solo a condizione che la tupla continui ad appartenere alla vista (non posso modificare lo stipendio portandolo a 60)

Normalmente una modifica sulle tuple si ripercuote su tutte quelle della definizione (cascaded). Interessante notare l'uso di *CHECK OPTION,* che definisce il livello di controllo a seguito di inserimento od aggiornamento di una vista.

```
UPDATE ImpiegatiNonCapoPoveri
    SET Stipendio = 60
    WHERE Nome = 'Paola'
```

Voglio quindi evitare diritti su dati che non appartengono ad un certo gruppo di utenti; questo è il senso logico dell'opzione presentata. Non sarebbe ammesso eseguire una modifica come segue se non definita esplicitamente dalla keyword presentata.

Le viste si pongono le scopo di formulare interrogazioni altrimenti inesprimibili con il normale linguaggio SQL, inoltre definendo delle query semplici, leggibili e utili nel caso anche di nidificazione multipla.
Di fianco l'esempio visto degli *"autori (o uno solo) con la canzone con più esecuzioni":*

```
SELECT *
FROM Autore
WHERE TitoloCanz
IN
(
        SELECT TitoloCanz
        FROM ESECUZIONE
        GROUP BY TitoloCanz
        HAVING COUNT(*) >
                SELECT MAX(CONTA) FROM
                        (SELECT COUNT(*) AS CONTA
                        FROM ESECUZIONE
                        GROUP BY TitoloCanzone)
                                AS CONTEGGIO
)
```

Basterebbe solo l'uguale, piuttosto che max

Con le viste, quindi, diventa:

```
CREATE VIEW ESECUZIONE_X_CANZONE(Titolo,NumEsecuzioni) AS
SELECT TitoloCanz,COUNT(*)
FROM ESECUZIONE
GROUP BY TitoloCanz

SELECT *
FROM Autore
WHERE TitoloCanz
IN
(
        SELECT Titolo
        FROM ESECUZIONE_X_CANZONE
        WHERE NumEsecuzioni=
                (SELECT MAX(NumEsecuzioni)
                 FROM ESECUZIONE_X_CANZONE)
)
```

Introduciamo altre funzioni condizionali, come ad esempio *Coalesce*, che restituisce il primo valore non nullo in una serie di espressioni.

- Esempio: coalesce(NULL,'A','B','Ignoto') restituisce 'A'
- Esempio: *Estrarre i nomi, e l'età degli impiegati. Scrivere 0 se non si conosce l'età (eta=null)*

```
select Nome, coalesce(Età,0)
from Impiegato
```

- Esempio: *Estrarre i nomi, e l'età degli impiegati. Scrivere NULL se l'età = 0*

```
select Nome, nullif(Età,0)
from Impiegato
```

Similmente, abbiamo anche *nullif* in cui, data una certa costante, se l'espressione risultasse uguale a questa metterebbe NULL. Rappresenta l'opposto della precedente.

Un costrutto simile allo switch-case del C, è la funzione *Case*, mettendo anche when/else/end:

- Esempio: *Estrarre i nomi e la classe di età degli impiegati (<30 giovani, >60 anziani, altrimenti media)*

```
select Nome, case
        when (Età<30) then 'Giovane'
        when (Età>60) then 'Anziani'
        else 'Medio'
        end
from Impiegato
```

Esistono le *funzioni scalari* agiscono con particolari vincoli o logiche sui vari dati:

- Temporali:
  current_date, extract(year from …)
- Manipolazione stringhe:
  char_length, lower
- Conversioni:
  cast

In SQL è possibile specificare chi (utente), come (lettura, scrittura), dove (tabelle, viste, attributi…) utilizzare la base di dati, dando quindi determinati privilegi o proprietà.
Un privilegio è caratterizzato da una risorsa a cui si riferisce, utente che concede/riceve il privilegio, azione permessa, trasmissibilità del privilegio.

I tipi di privilegi offerti da SQL sono:

INSERT: permette di inserire nuovi oggetti (tuple)
UPDATE: permette di modificare il contenuto
DELETE: permette di eliminare oggetti
SELECT: permette di leggere la risorsa
REFERENCES: permette la definizione di vincoli di integrità referenziale verso la risorsa (può limitare la possibilità di modificare la risorsa)
USAGE: permette l'utilizzo in una definizione (per esempio, di un dominio)

La gestione delle autorizzazioni deve nascondere gli elementi a cui un utente non può accedere. Per esempio, se volessi sapere se un impiegato è un capo, non posso sfruttare operazioni esterne a quelle che ho per capire la logica delle basi di dati, violandone la privacy (ad esempio, se non potessi inserire nuovi capi ma volessi sapere, avendo i permessi di selezione, se un certo impiegato sia un capo, eseguo le sole operazioni di selezioni e capisco "logicamente" se l'impiegato sia capo o meno). Quindi l'utente lo stesso messaggio sia nel caso la tabella non esistesse, sia se esistesse ma l'utente non è autorizzato.

Attraverso una vista, quindi, definiamo condizioni di selezione oppure attribuiamo autorizzazioni, piuttosto che sulla relazione di base. Segue un esempio di concessione/autorizzazione di diritti (prima occorre creare l'utente, poi agire su di esso assegnando poteri e per revocare autorizzazioni si usa il comando REVOKE GRANT, con sintassi similare alla concessione di autorizzazioni):

```
CREATE VIEW ImpiegatiNonCapo(Nome, Eta, Stipendio) AS
    SELECT Nome, Eta, Stipendio
    FROM Impiegato
    WHERE Nome NOT IN (SELECT Capo FROM Impiegato)
```

Concedere a tutti i privilegi di leggere a *ImpiegatiNonCapo*:

```
GRANT SELECT ON ImpiegatiNonCapo TO PUBLIC;
```

Dare a «Manuel» tutti i privilegi su *ImpiegatiNonCapo* :

```
GRANT ALL PRIVILEGES ON ImpiegatiNonCapo TO Manuel;
```

Dare a «Max» la possibilità di aggiungere *Impiegato*:

```
GRANT INSERT PRIVILEGES ON Impiegato TO Max;
```

Seguono esercizi conclusivi alla lezione (realizzati con le viste):

STADIO(Nome, Citta, Capacità)
INCONTRO(NomeStadio, Data, Ora, Squadra1, Squadra2)
NAZIONALE(Squadra, Continente, Categoria)

- Estrarre la città in cui si trova lo stadio dove la nazionale italiana gioca più partite

```
CREATE VIEW Pi(Nome, Num_partite) AS
SELECT NomeStadio, COUNT(*)
FROM Incontro I
WHERE Squadra1 = 'Italia' OR Squadra2= 'Italia'
GROUP BY NomeStadio

SELECT Citta
FROM Stadio
WHERE Nome IN(SELECT Nome FROM Pi
                WHERE Num_partite = SELECT MAX(Num_partite) FROM Pi
```

VIAGGIO(Paese, Costo, CodViaggio)
CLIENTE(CF, Nome, Cittadinanza)
EFFETTUA(CF, CodViaggio, Data)

- Trovare la cittadinanza che investe il maggior costo complessivo per i viaggi

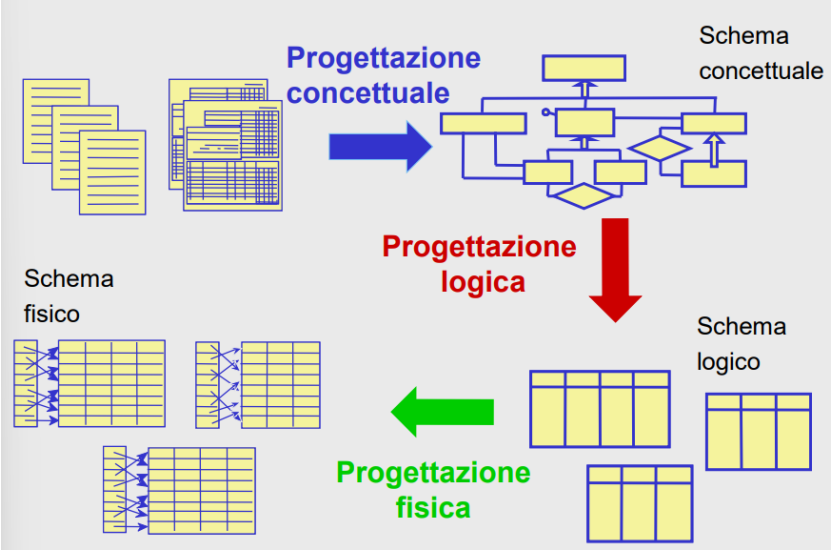
```
CREATE VIEW Costi_x_cittadinanza AS
SELECT Cittadinanza, Sum(Costo) As Somma
FROM Cliente C, Effettua E, Viaggio V
WHERE C.CF=E.CF AND V.CodViaggio=E.CodViaggio
GROUP BY Cittadinanza

SELECT Cittadinanza
FROM Costo_x_cittadinanza
WHERE Cittadinanza = (SELECT Max(Somma) FROM Costo_x_cittadinanza)
```

Metodologie e modelli per il progetto di una base di dati: concetti base (relazioni, cardinalità, attributi, modello E/R)

Parte di vita del ciclo di Sistemi Informativi compone la sequenzializzazione delle attività per sviluppo ed uso di sistemi informativi, svolte da analisti, utenti o altro, secondo una serie di sessioni capendo le funzionalità offerte.
L'idea di realizzazione é la seguente, designando le singole attività (a noi interessa raccolta e analisi dei requisiti e la progettazione):

- *studio di fattibilità*, definendo costi e priorità
- *raccolta e analisi dei requisiti*, studiando le proprietà del sistema
- *progettazione* di funzionalità e dati manipolati
- *realizzazione* del sistema
- *validazione e collaudo*, verifica delle funzionalità del sistema
- funzionamento, il sistema diventa operativo per l'utente

A questo punto distinguiamo la base di dati pensata concettualmente (*progettazione concettuale)*, che non guarda come la base di dati viene implementata e solo a livello di analisi perché non guarda la vera implementazione), traducendolo logicamente in relazioni per il DBMS in uso (*progettazione logica*), implementando realmente in una base di dati in termini di tabelle un certo sistema (*schema fisico*).

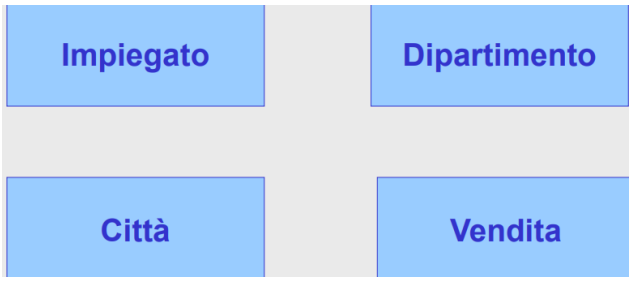


Se iniziassimo dallo schema concettuale avremmo problemi perché non sapremmo da dove iniziare, perdendoci nei dettagli implementativi, pensando da subito a come correlare le varie tabelle, con chiavi e quant'altro.
Dal punto di vista dei *modelli concettuali*:

- ragionare sulla realtà di interesse, indipendentemente dagli aspetti realizzativi
- rappresentazione di classi di oggetti di interesse e correlazioni tra gli stessi
- efficaci rappresentazioni grafiche

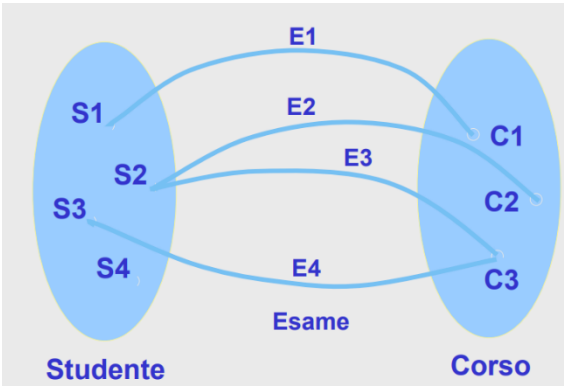
Naturalmente il modello utilizzato è *Entity-Relationship (Entità-Relazione)*, partendo da *entità*, definite come classi di oggetti con proprietà comuni e con esistenza "autonoma" (esempi: studente, corsi, impiegato, ordine, ecc.), istanziabili attraverso modello.
La stessa denominazione delle entità generalmente è al singolare (meglio "studente" al posto di "studenti"). Mettiamo e colleghiamo logicamente oggetti con le *relationships* (tradotte in italiano con relazioni, ma meglio legami), collegando concetti strutturalmente connessi, come:

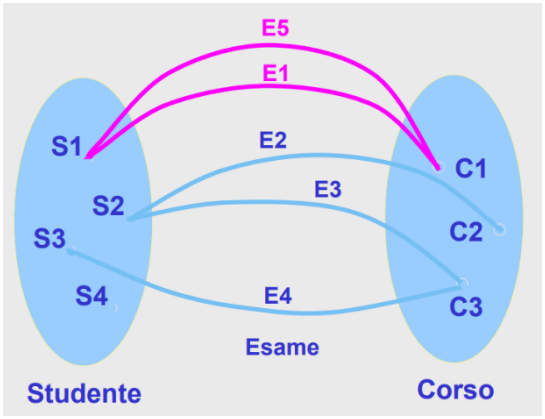
In ogni base di dati vi è lo *schema*, invariato nel tempo che descrive la struttura (intensionale) del DB e l'*istanza*, con i valori attuali e che cambiano molto rapidamente nel tempo. Ogni tupla è un oggetto (o istanza) di una certa entità. Noi ci focalizziamo a livello astratto con le entità, come:



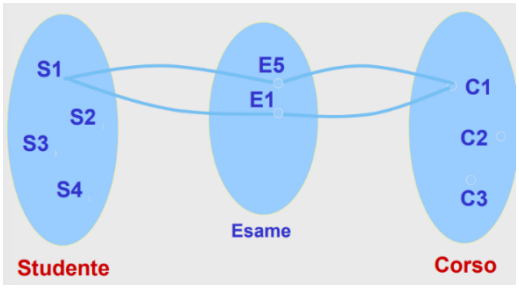
Dato che si parla di un prodotto presentato al pubblico, occorre dare *nomi espressivi* e adottare *opportune convenzioni* (ad esempio, chiamare al singolare le entità).
Le *relazioni* sono rappresentate con un rombo, le *entità* con dei *rettangoli*.
Anche le relazioni solitamente devono avere nomi espressivi e convenzioni apposite (sostantivi invece che verbi per definirle).
Invece le occorrenze nelle relazioni sono:

- binarie (una singola occorrenza per ogni entità coinvolta)
- n-aria (una occorrenza di ogni entità coinvolta)
- nessuna occorrenza ripetuta (coppie/tuple)

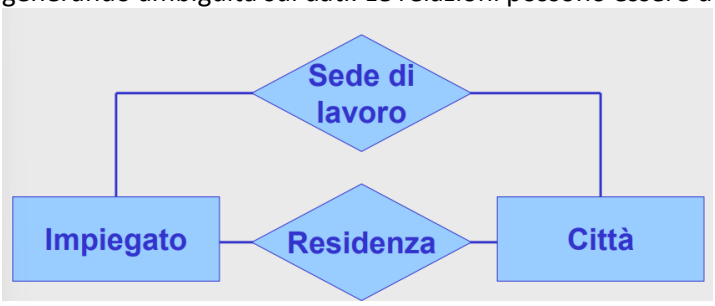




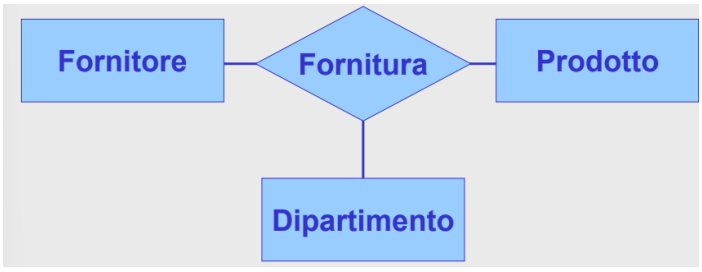
Attenzione anche alla progettazione; per esempio possiamo avere uno studente che può dare un esame una volta sola, come si vede sotto.

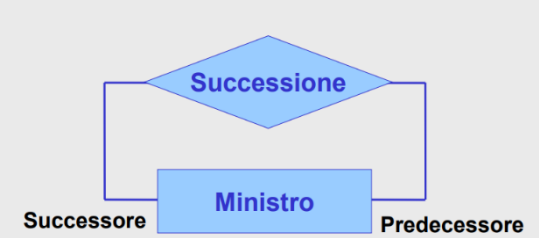


Andiamo quindi a "promuovere" le relazioni, aggiungendo una relazione intermedia e quindi non generando ambiguità sui dati. Le relazioni possono essere definite doppie sulle stesse entità come:



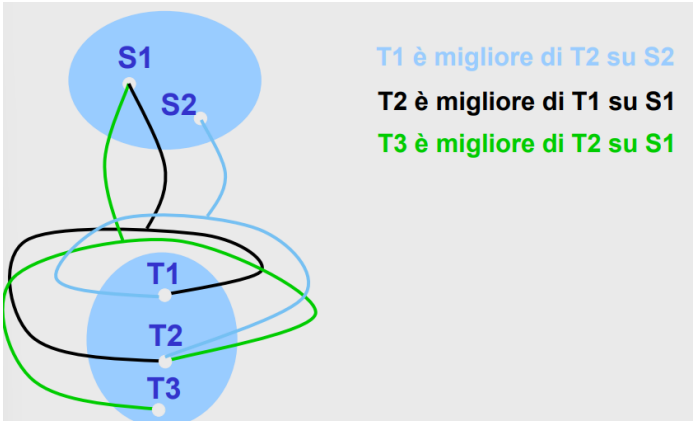
le relazioni n-arie concettualmente esistono ma potrebbero essere eccessivamente complicate, quindi tendenzialmente ridotte a binarie. La scelta nell'utilizzo di modellazione dipende da quello che vogliamo modellare, come:

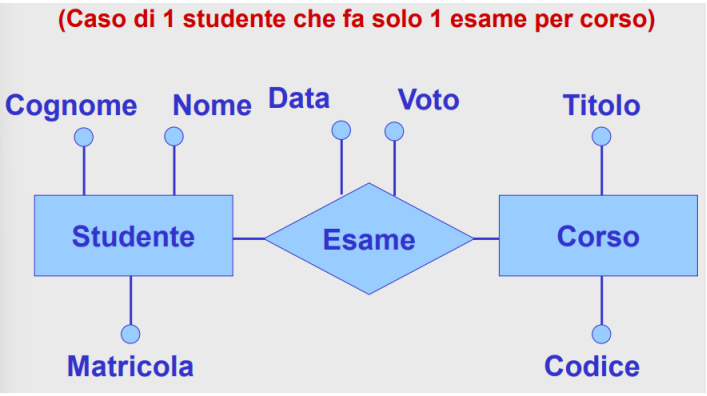




Tutto ciò è rappresentabile graficamente da archi multipli per descrivere le occorrenze.
Similmente posso avere relazioni ricorsive, come ad esempio il Padre che contemporaneamente è anche Figlio.
Gli archi di rappresentazione possono essere circolari per rappresentare il doppio collegamento.

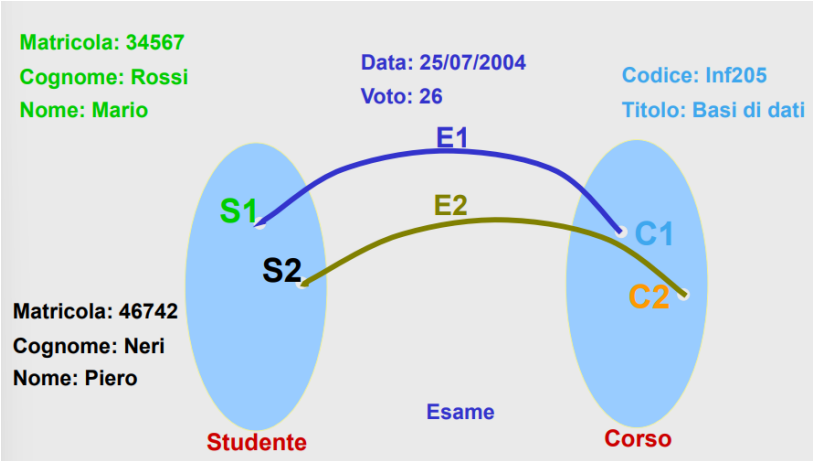
Similmente alle relazioni binarie esistono quelle ternarie, ovviamente con tre archi nei collegamenti e anch'esse possono essere ricorsive.
Vediamo anche che le occorrenze sono similari:





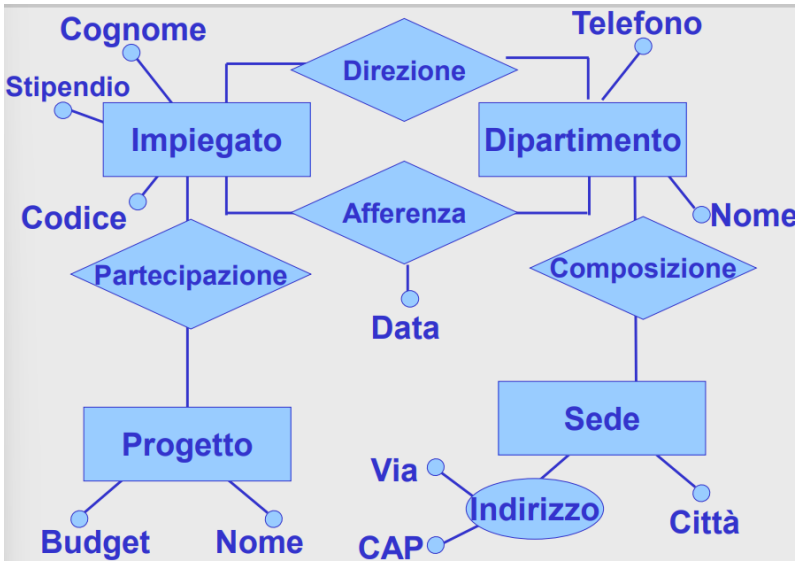
Il concetto di *attributo* rappresenta una proprietà elementare di un'entità o di una relazione, ai fini di interesse dell'applicazione che sto usando. Associo ad ogni occorrenza di entità/relazione un valore appartenente al *dominio*.
Vediamo la rappresentazione grafica di fianco (con più pallini nel caso di attributi composti):

Informazioni intermedie devono essere relazionalmente sensate. Concretamente, come si vede qui, se mettessi data e voto nella tabella Studente, ognuno avrebbe lo stesso voto e data per tutti i corsi, che non avrebbe senso.
Funzionalmente quindi:

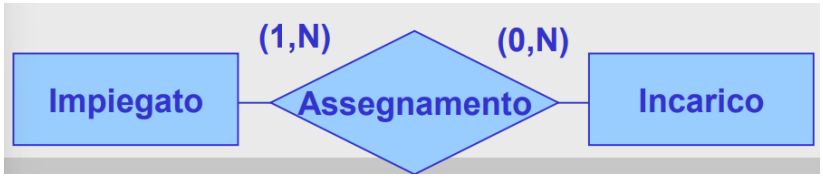


É possibile anche definire attributi composti, raggruppando caratteristiche descrittive di entità e relazioni che presentano le singole affinità.

Un esempio esteso può essere (afferenza significa appartenenza):

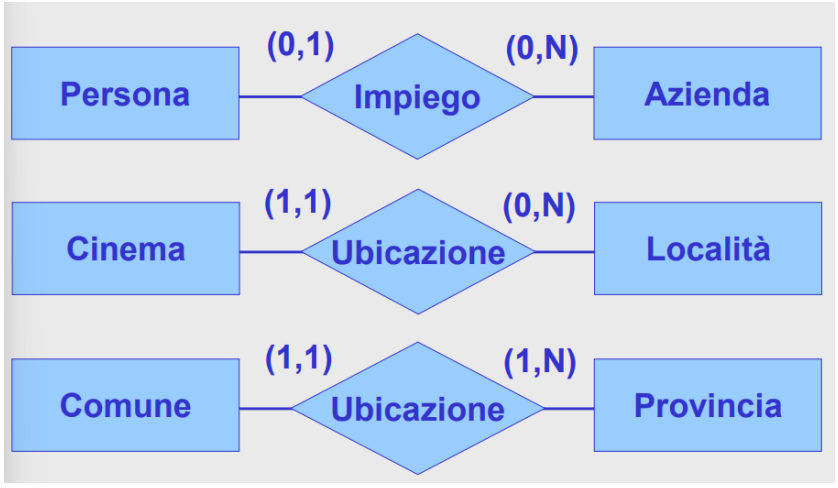


Definiamo inoltre le *cardinalità*, che possono essere "di relazione", rappresentando una coppia di valori per la prima tra num. minimo di occorrenze (a) e num. massimo di occorrenze (b), quindi coppia (a,b) e "di attributo". Un esempio concreto di cardinalità, definendo "0" come partecipazione *opzionale*, "1" per partecipazione *obbligatoria*, ponendo 1 per minima ed N per la massima (N non pone limiti e può essere un numero qualsiasi):

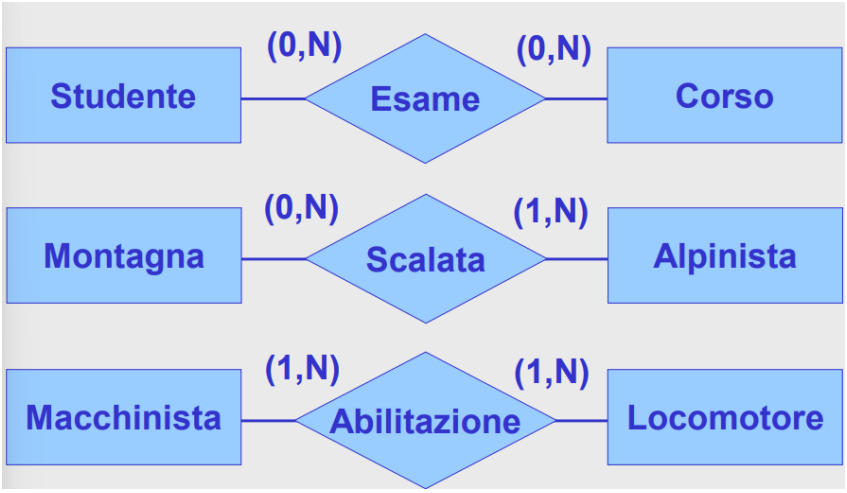


Proseguiamo sui tipi di relazioni, descrivendo nell'ordine:

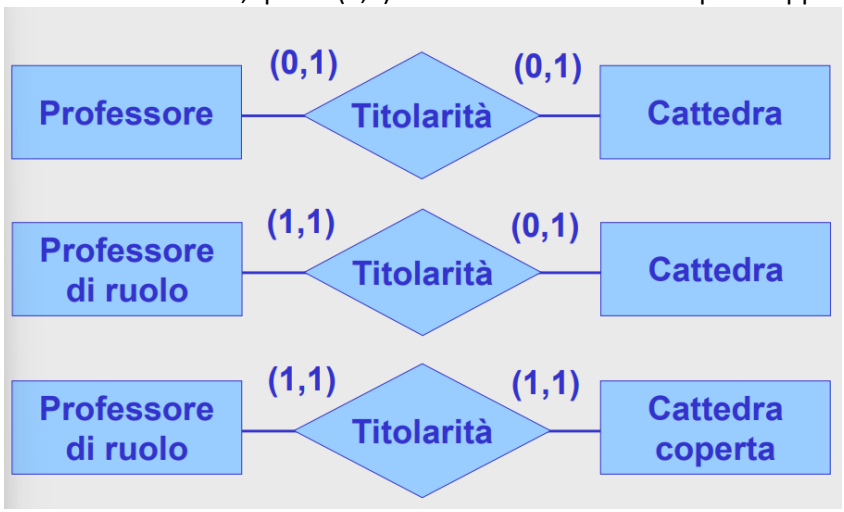
- uno a molti (1, N)



- molti a molti (N,N), generalmente da un minimo di 0 ad un massimo di N:



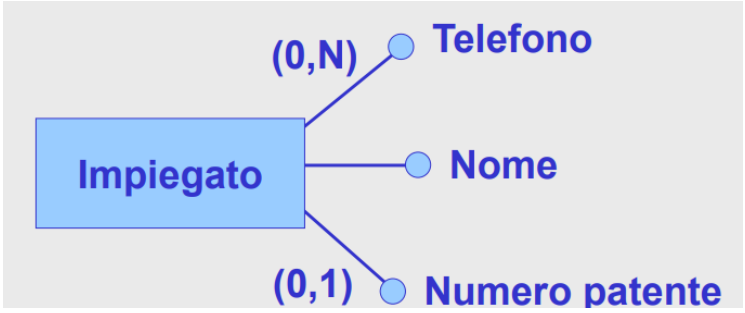
- a uno a uno, quindi (1,1) da almeno una delle due parti rappresentante univocità:



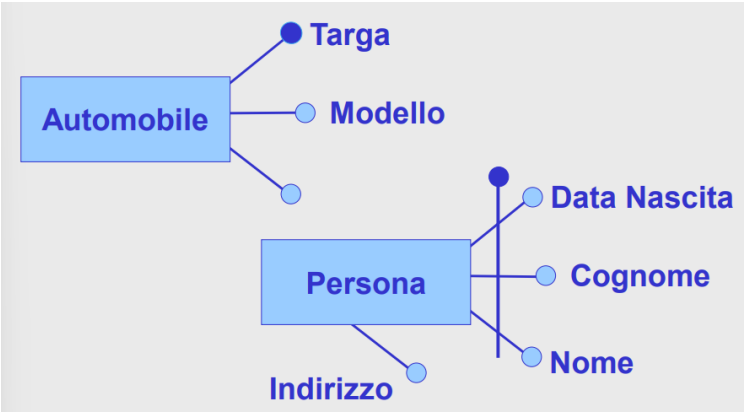
Attenzione poi ai versi nelle relazioni: nell'esempio sotto, dovrei vedere una persona impiegata in un'azienda singola, con un'azienda che possiene N persone con quell'impiego.
<u>In questo specifico caso</u>, non è vero che una persona è impiegata a più aziende, o un'azienda data a 0/1 persona.

É possibile poi associare delle cardinalità anche agli attributi per

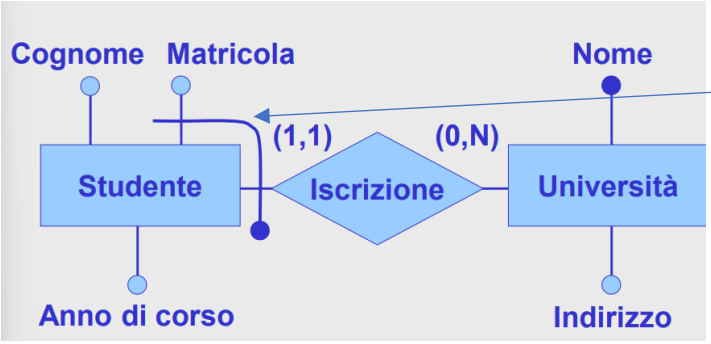
- indicare opzionalità (informazioni incomplete)
- indicare attributi multivalore



Se (come per l'attributo Nome) non si scrive nulla, l'esatta cardinalità è 1-1.
Per l'identificazione univoca di occorrenze di un'entità (similmente alla chiave), si hanno identificatori costituiti da entità esterne (*identificatore esterno*) oppure possibili attributi dell'entità (*identificatore interno*).
Qui a destra esempio di identificatori interni:

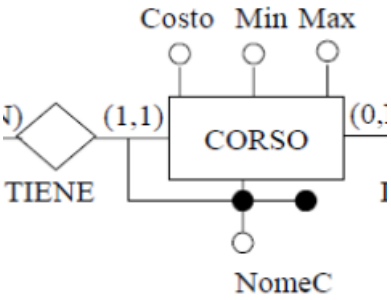


Segue qui sotto l'esempio di *identificatore esterno*, dove uno studente presenta una singola iscrizione verso N università, qui l'id. esterno si pone come la tabella Iscrizione. Si consideri inoltre il pallino blu che rappresenta sia chiave esterna (esempio sotto) che superchiave (esempio qui sopra con Persona). Normalmente rappresenta la seconda.



Attenzione:

Questo indicato dalla freccia riferisce che la matricola è unica nell'ambito di quella specifica iscrizione all'università.
Il segno, quindi, indica che l'attributo chiave viene usato per identificare esternamente attributi di relazione, assieme ad altri attributi chiave



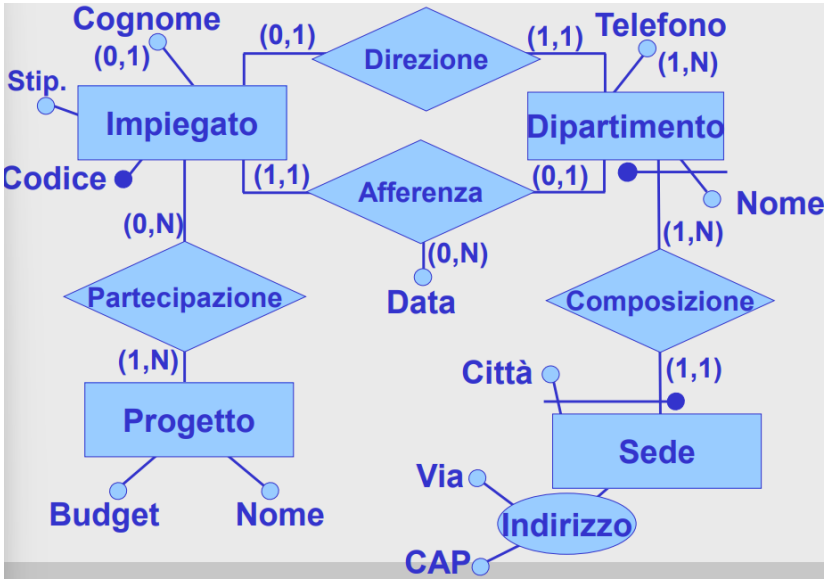
Qui a lato:
Rappresentazione alternativa di "unico all'interno di" da un punto di vista grafico.

Altro esempio:
Il casello dell'autostrada; possono esserci più caselli con lo stesso nome nelle varie autostrade e quindi si usa il doppio collegamento (casello autostrada, codice) per risalire a quello corretto.

Nelle modellazioni questo è definito da "unico all'interno di", come in questo caso, lo studente la cui matricola ha senso nel contesto dell'università.
La sua chiave, pertanto, sarà formata da Matricola e Nome_Università, quindi doppia.

Esercizio

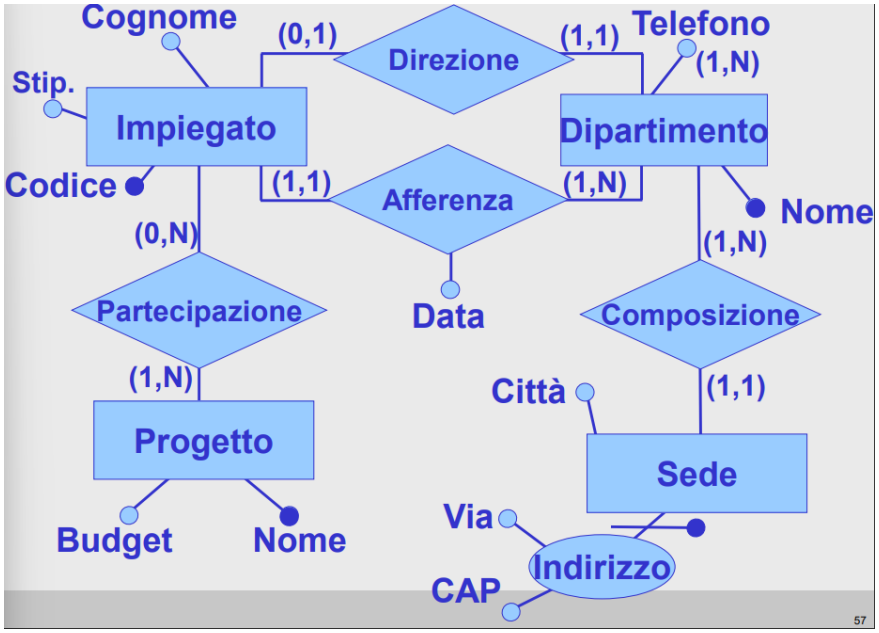
Trova gli errori nel diagramma alla prossima slide



Errori possibili:

- (1,N) su Composizione; si risolve mettendo (1,1), perché un dipartimento potrebbe avere più sedi e dunque cambiare il dominio di dati (mettendo (1,N) tra Composizione e Sede). Un'altra idea è l'introduzione di un campo Id all'interno di Dipartimento per identificare la sede (usata dal prof).
- Progetto non ha un identificatore; di fatto i progetti hanno sempre un nome che li identifica e basta nome come identificativo
- Anche Cognome ha poco senso mantenere (0,1) come cardinalità; si mette (1,1)
- (1,N) tra Afferenza e Dipartimento, perché un impiegato è presente in più Dipartimenti e si considera Id e non Nome come campo primario
- (1, N) tra Impiegato ed Afferenza perché un impiegatìo presenzia con più date, in base alla propria presenza in più dipartimenti
  Per risolvere il problema bisogna far diventare Afferenza entità piuttosto che relazione; questo si fa perché si vuole memorizzare una persona che va via da un dipartimento e ritorna successivamente
  Quindi si elimina (0,N) tra Afferenza e Data

Una possibile soluzione finale è:



Caso particolare da segnalare: relazioni (1,1) da entrambe le parti.
Link di riferimento e tradotto: https://vertabelo.com/blog/one-to-one-relationship-in-database/

Vediamo alcuni esempi reali di relazioni uno-a-uno:

1. Paese - capitale: Ogni paese ha esattamente una capitale. Ogni capitale è la capitale di esattamente un paese.
2. Persona - le loro impronte digitali: Ogni persona ha un set unico di impronte digitali. Ogni serie di impronte digitali identifica esattamente una persona.
3. E-mail - account utente: per molti siti Web, un indirizzo e-mail è associato esattamente a un account utente e ogni account utente è identificato dal suo indirizzo e-mail.
4. Coniuge - coniuge: in un matrimonio monogamo, ogni persona ha esattamente un coniuge.
5. Profilo utente - impostazioni utente: un utente dispone di un set di impostazioni utente. Un set di impostazioni utente è associato esattamente a un utente.

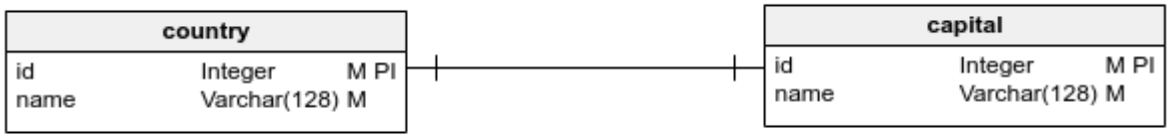
Per chiarezza, confrontiamo questi esempi con relazioni che non sono uno da uno:

6. Paese - città: Ogni città è esattamente in un paese, ma la maggior parte dei paesi ha molte città.
7. Genitore - figlio: ogni bambino ha due genitori, ma ogni genitore può avere molti figli.
8. Dipendente - manager: Ogni dipendente ha esattamente un supervisore o manager immediato, ma ogni manager di solito supervisiona molti dipendenti.

Disegnare una relazione uno-a-uno in un diagramma ER

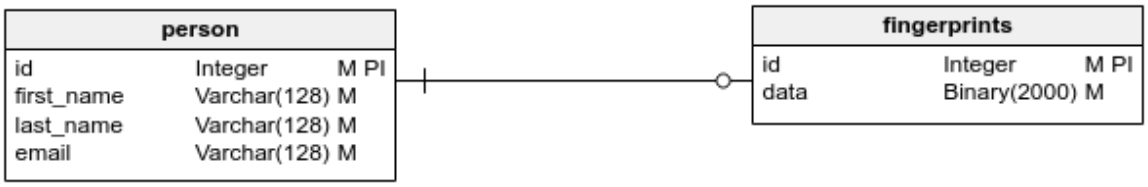
Una relazione uno-a-uno in un diagramma ER è indicata, come tutte le relazioni, con una linea che collega le due entità. La cardinalità "uno" è indicata con una singola linea retta. (La cardinalità "molti" è indicata con il simbolo del piede di gallina.)

La relazione uno-a-uno tra paese e capitale può essere indicata in questo modo:



Le linee rette perpendicolari significano "obbligatorio". Questo diagramma mostra che è obbligatorio per una capitale avere un paese ed è obbligatorio per un paese avere una capitale.

Un'altra possibilità è che uno o entrambi i lati della relazione siano facoltativi. Un lato opzionale è indicato con un cerchio aperto. Questo diagramma dice che esiste una relazione uno-a-uno tra una persona e le sue impronte digitali. Una persona è obbligatoria (le impronte digitali devono essere assegnate a una persona), ma le impronte digitali sono facoltative (una persona potrebbe non avere impronte digitali assegnate nella banca dati).



Relazioni uno-a-uno in un database fisico

Esistono diversi modi per implementare una relazione uno-a-uno in un database fisico.

Chiave primaria come chiave esterna
Un modo per implementare una relazione uno-a-uno in un database consiste nell'utilizzare la stessa chiave primaria in entrambe le tabelle. Le righe con lo stesso valore nella chiave primaria sono correlate. In questo esempio, la Francia è un paese con l'id 1 e la sua capitale è nella tabella capitale sotto id 1.

paese

Id	nome
1	Francia
2	Germania
3	Spagna

capitale

Tecnicamente, una delle chiavi primarie deve essere contrassegnata come chiave esterna, come in questo modello di dati:

La chiave primaria nella tabella maiuscola è anche una chiave esterna che fa riferimento alla colonna ID nel paese della tabella. Poiché capital.id è una chiave primaria, ogni valore nella colonna è univoco, quindi la capitale può fare riferimento al massimo a un paese. Deve anche fare riferimento a un paese: è una chiave primaria, quindi non può essere lasciata vuota.

Chiave esterna aggiuntiva con vincolo univoco

Un altro modo per implementare una relazione uno-a-uno in un database consiste nell'aggiungere una nuova colonna e renderla una chiave esterna.

In questo esempio, aggiungiamo la colonna country_id nella tabella maiuscola. La capitale con id 1, Madrid, è associata al paese 3, la Spagna.

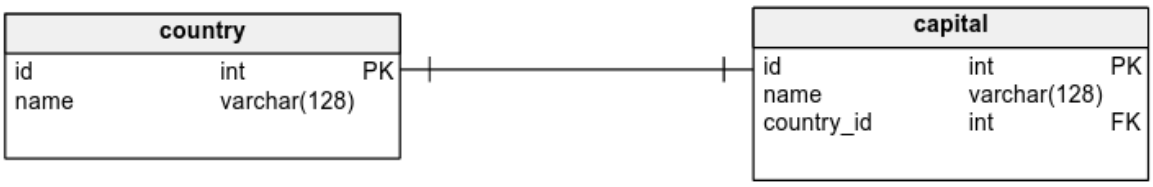
paese

Id	nome
1	Francia
2	Germania
3	Spagna

capitale

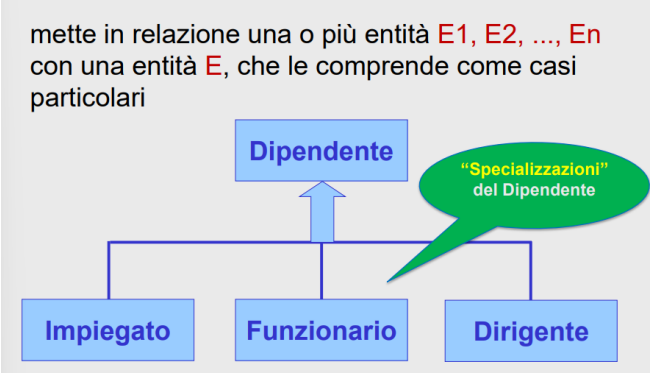
Id	nome	country_id
1	Madrid	3
2	Berlino	2
3	Parigi	1

Tecnicamente, la colonna country_id dovrebbe essere una chiave esterna che fa riferimento alla colonna id nel paese della tabella. Poiché si desidera che ogni capitale sia associata esattamente a un paese, è necessario rendere unica la colonna della chiave esterna country_id.



Metodologie e modelli per il progetto di una base di dati: generalizzazione e tipi, esercizi

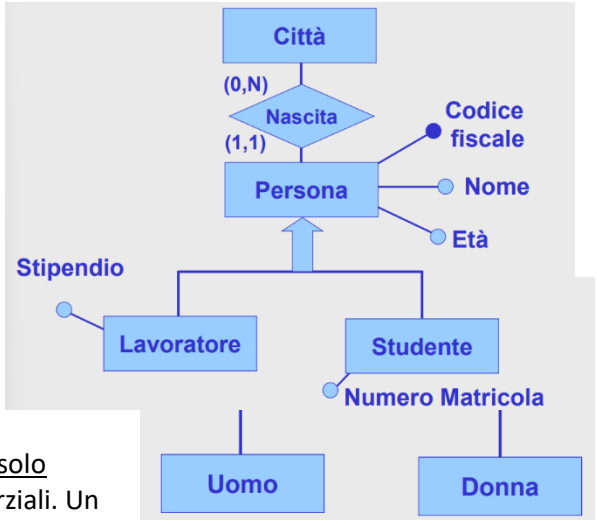
Parliamo della *generalizzazione*, che mette in relazione più entità secondo una logica gerarchica oppure di appartenenza logica. *E* quindi può essere generalizzazione di *E1, E2, … En* (queste ultime invece sono definite come *specializzazioni*). Esempio pratico:



Se si ha una generalizzazione E di E1, E2, En

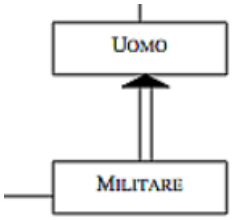
1) ogni proprietà di E è anche di E1, E2, … En
2) ogni occorrenza di E1, E2, .. En è anche occorrenza di E

A livello pratico significa che le varie entità ereditano una serie di caratteristiche logiche; per esempio Lavoratore possiede come informazioni Città/Nascita/Codice Fiscale, essendo una Persona.

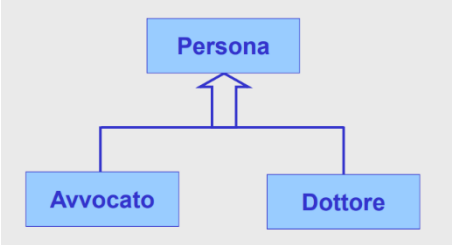


Di solito infatti parliamo di *generalizzazione totale*, quindi ogni occorrenza dell'entità genitore è anche di una delle entità figlie; altrimenti viene definita *generalizzazione parziale*.
Viene definita la generalizzazione *esclusiva*, se ogni occorrenza dell'entità genitore è occorrenza al più in uno dei figli, *sovrapposta* altrimenti. Noi consideriamo solo generalizzazioni esclusive e distinguiamo fra totali e parziali. Un esempio di generalizzazione totale (con freccia piena, significa che Persona è composta sia da Uomo che da Donna).

Nota di contorno: le generalizzazioni parziali possono avere anche (da un punto di vista grafico), la freccia piena solo nella parte superiore e vuote nella parte inferiore, come visibile da immagine apposita.



Altra nota: in alcuni casi (sempre non visti nel nostro corso), si indica nello schema ER la cardinalità della generalizzazione totale come (t, e), che significa "totale, esclusiva".

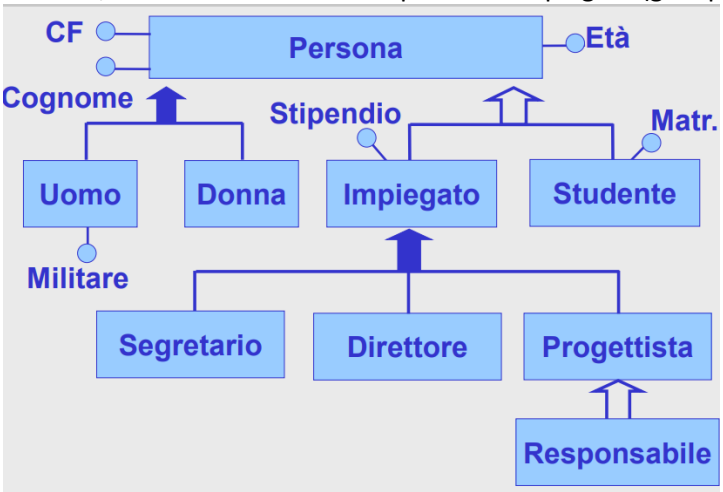


La generalizzazione parziale indica che ci sono occorrenze dell'entità padre che non sono di nessun figlio (freccia vuota, significa che una Persona può essere Avvocato o Dottore, senza essere per forza entrambi):

Esercizio

- Le persone hanno CF, cognome ed età; inoltre, ci sono persone speciali:
  - Gli uomini anche la posizione militare;
  - Gli impiegati hanno lo stipendio e possono essere: *segretari*, *direttori* o *progettisti* (un progettista può essere anche responsabile di progetto);
  - Gli studenti (che non possono essere impiegati) un numero di matricola;
  - Alcune persone non sono né impiegati né studenti

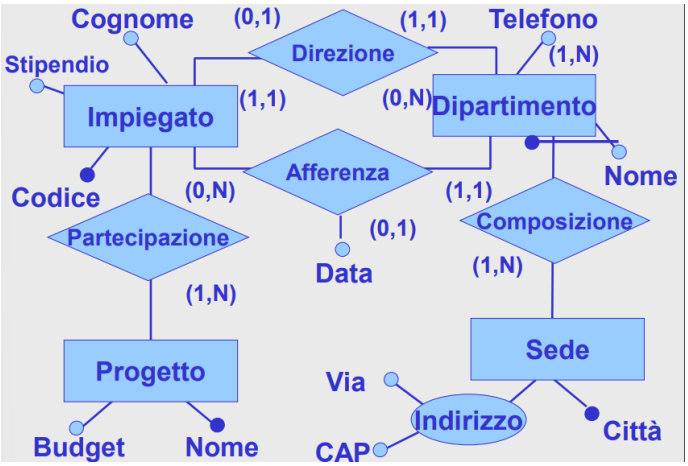
Il caso pratico è: il progettista è anche responsabile.
Segretario, Direttore e Progettista fanno parte di Impiegato. Sia Studente che Impiegato fanno parte di Persona, ma uno Studente non è per forza Impiegato (gen. parziale). Segue quindi l'idea realizzativa:



Tipicamente agli schemi concettuali vengono associati:

- dizionario dei dati spiegando entità/relazioni (specificando entità, breve descrizione, componenti ed attributi)
- vincoli non esprimibili sull'ER (ad esempio, in una base di dati riferita all'università, uno studente fa parte del secondo anno se ha raggiunto i 20 CFU di soglia minima oppure un laureando che deve aver superato tutti gli esami dei primi due anni, con al massimo 3 rimanenti, per andare in stage). Altro esempio utile: premi di anzianità per i dipendenti dopo un certo tempo. Sono tutte cose che nello schema ER non verranno mai rappresentate.

Ad esempio, allo schema della soluzione presentata nella scorsa lezione (leggermente diversa):



Il dizionario dei dati è composto da entità:

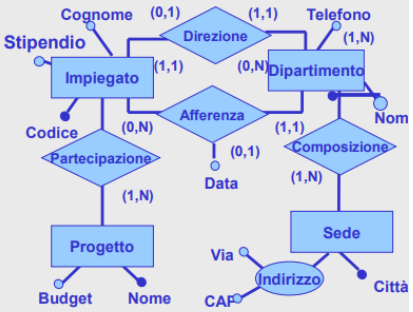
Entità	Descrizione	Attributi	Identificatore
Impiegato	Dipendente dell'azienda	Codice, Cognome, Stipendio	Codice
Progetto	Progetti aziendali	Nome, Budget	Nome
Dipartimento	Struttura aziendale	Nome, Telefono	Nome, Sede
Sede	Sede dell'azienda	Città, Indirizzo	Città

e da relazioni:

Relazioni	Descrizione	Componenti	Attributi
Direzione	Direzione di un dipartimento	Impiegato, Dipartimento	
Afferenza	Afferenza a un dipartimento	Impiegato, Dipartimento	Data
Partecipazione	Partecipazione a un progetto	Impiegato, Progetto	
Composizione	Composizione dell'azienda	Dipartimento, Sede	

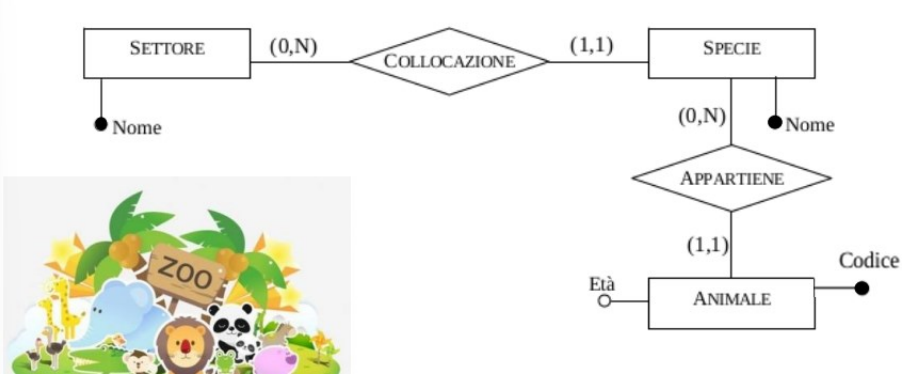
enunciando poi esplicitamente i vincoli non esprimibili in ER:

1. Se Impiegato I dirige Dipartimento D, allora I afferisce a D
2. Se Impiegato I dirige Dipartimento D, allora I ha uno stipendio più alto di tutti gli altri afferenti a D
3. Non ci sono due dipartimenti con lo stesso nome

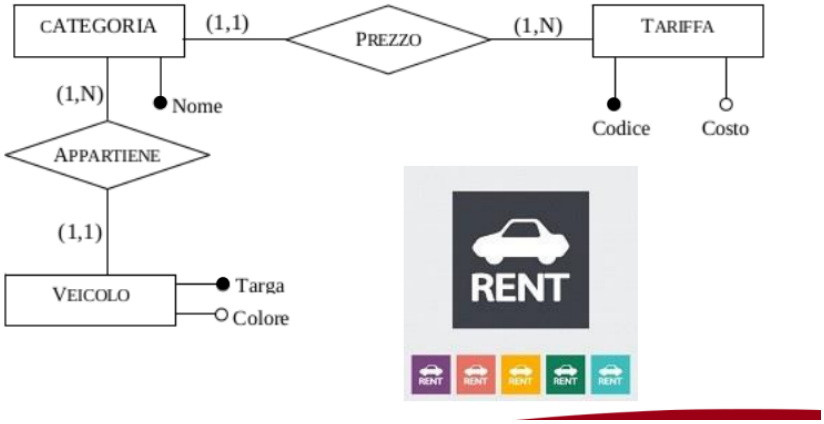


Rappresentare le seguenti realtà utilizzando i costrutti del modello Entità-Relazione e introducendo solo le informazioni specificate.

1) In un giardino zoologico ci sono degli animali appartenenti a una specie e aventi una certa età; ogni specie è localizzata in un settore (avente un nome) dello zoo.

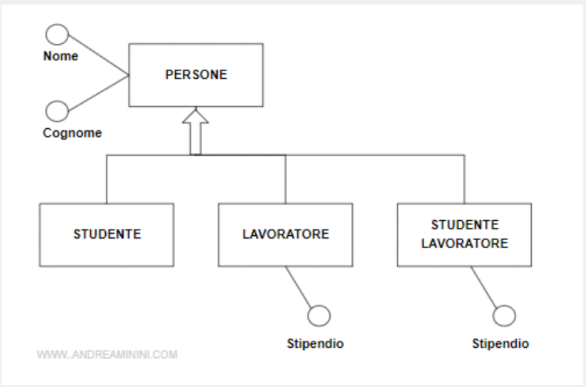


2) Una agenzia di noleggio di autovetture ha un parco macchine ognuna delle quali ha una targa, un colore e fa parte di una categoria; per ogni categoria c'è una tariffa di noleggio.



Nota: esistono anche le generalizzazioni sovrapposte.

Nota. Le generalizzazioni sovrapposte possono essere trasformate in esclusive aggiungendo una nuova entità per le intersezioni. La presenza delle generalizzazioni esclusive rende più chiara la rappresentazione concettuale della base dati. Ad esempio, aggiungere l'entità Studente-Lavoratore.



Dopo questa modifica il risultato è comunque una generalizzazione parziale, perché non include altre situazioni (es. Disoccupato). Tuttavia, non è più una generalizzazione sovrapposta ma esclusiva, perché le occorrenze possono appartenere al più a una entità figlia.

Scritto da Gabriel

Progettazione concettuale

Ci concentriamo quindi su una serie di attività interconnesse per acquisizione ed analisi dei requisiti, costruendo lo schema di realtà in essere ed il glossario di riferimento.
Le possibili fonti possono essere:

- gli utenti/committenti, attraverso interviste o documentazione apposita;
- documentazione esistente, prendendo come riferimento leggi, regolamenti e realizzazioni preesistenti
- modulistica di riferimento

L'attività di *raccolta ed analisi dei requisiti* non è standardizzabile, essendo attività non facile e dipendente dal contesto applicativo; capita anche l'*acquisizione per interviste*, parlando con utenti che conoscono il sistema ma anche con chi gestisce a livello più alto, con visione meno dettagliata ma capendo le esigenze di chi richiede l'applicazione in oggetto.
Naturalmente si controlla che le informazioni siano coerenti e che il modello abbia un giusto grado di astrazione (generalizzazione in base al contesto informatico d'uso), focalizzandosi sugli aspetti essenziali.
L'insieme di regole utile è il seguente:

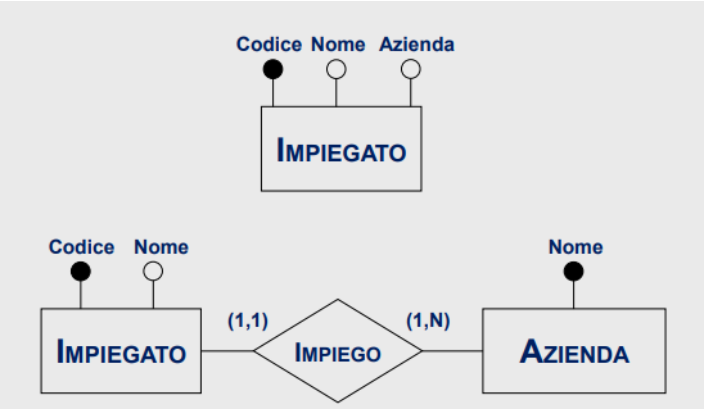
- scegliere livello corretto di astrazione
- standardizzazione delle frasi usate, coerente tra utente e committente
- suddividere le frasi articolate, mantenendo tutto il più semplice possibile
- separare le frasi sui dati da quelle delle funzioni
- costruire un glossario dei termini usati
- individuare omonimi e sinonimi
- rendere esplicito il rapporto e riferimento tra termini, intesi da parte delle due parti
- riorganizzazione delle frasi per concetti

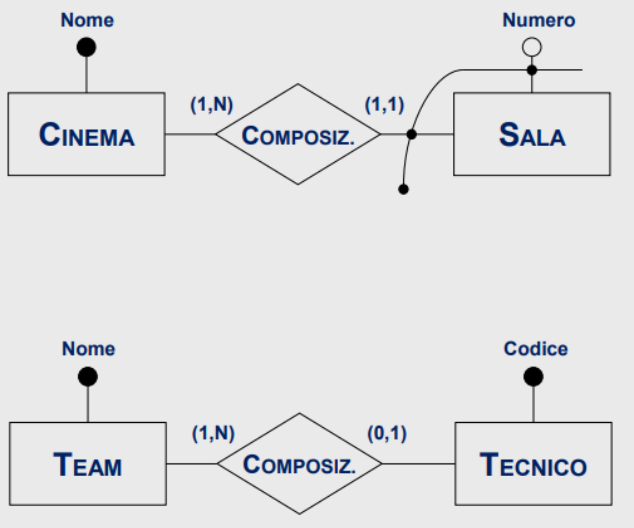
A livello E-R si caratterizzano:

- entità, con proprietà significative e oggetti con esistenza autonoma;
- attributo, entità semplice senza proprietà;
- relationship, correlazione di due o più concetti;
- generalizzazione, quindi distinguere entità logicamente simili definendo le giuste differenze.

Le soluzioni progettuali comuni seguono i *design pattern*, comunemente utilizzati per classi di problemi.

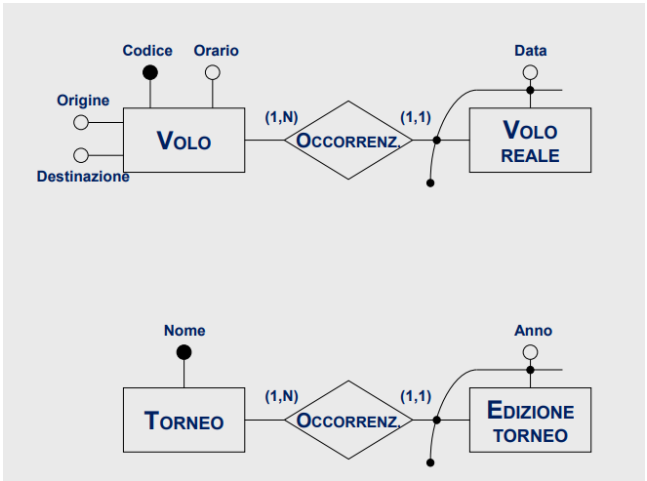
Partiamo dalla *reificazione di attributo di entità* (reificare significa considerare concetti e categorie come fossero oggetti concreti e, in particolare, reificare una relazione significa trasformarla in entità e introdurre chiavi esterne). Questo si riferisce alle relazioni n-arie, creando normalmente una nuova classe per costruire una nuova entità, ad esempio:

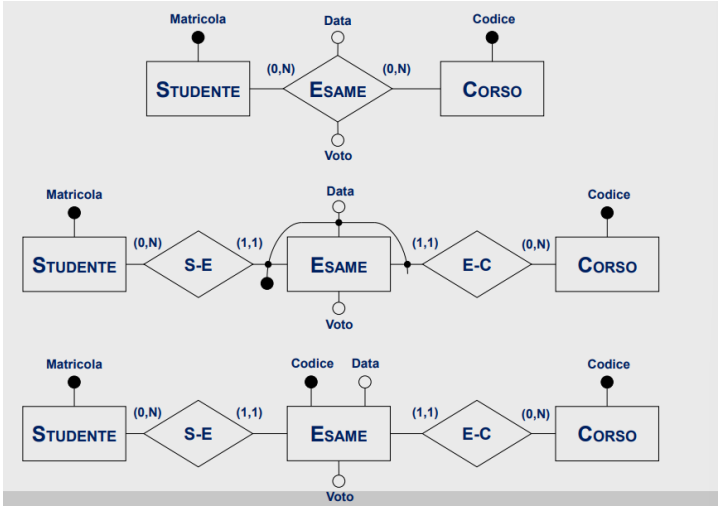




Similmente si parla di *part-of*, con entità che hanno senso se contestualizzate ad altre similari (caso d'esempio è la Sala che non esiste da sola senza considerare il Cinema; il suo campo chiave, quindi, dovrà anche considerare la chiave di Cinema).

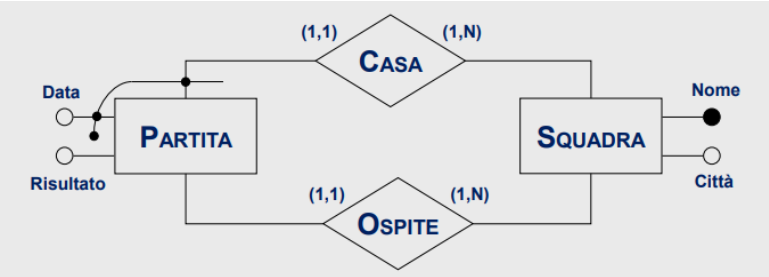
Altro pattern utilizzato è *istance-of*, per esempio i voli rappresentati come tratte e il vero e proprio volo con una certa data. Si intende quindi una creazione più o meno specifica di un'idea, per "caratteristiche" quindi.
Concettualmente non possiamo esprimerli come generalizzazione, in nessuna delle forme possibili.





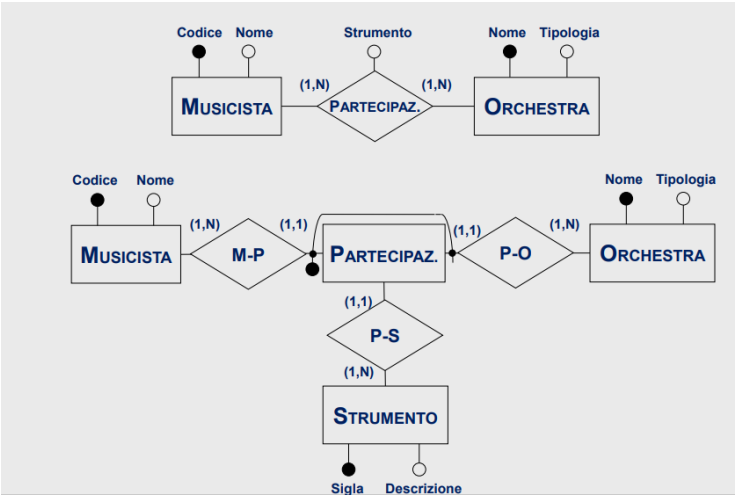
Similmente si ha una *reificazione* nel caso particolare *di relazione binaria*, avendo quindi una differenziazione nel caso in cui ci siano più oggetti correlati alla stessa relazione (esempio concreto, gli studenti che possono fare lo stesso esame più di una volta per i vari corsi, facendoli nella stessa data; l'idea della terza parte di immagine è sbagliata, si usa la seconda riga):

Attenzione: si eviti di usare i codici come campi chiave, causano solo casini per i motivi detti sopra (cit. De Leoni).

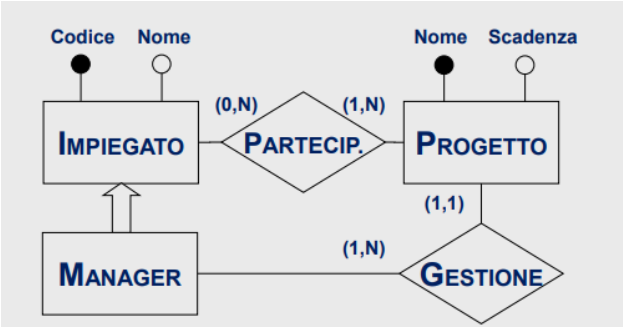


Similmente si ha una *reificazione di relazione ricorsiva*, nel caso per esempio delle partite effettuate in casa o in trasferta:

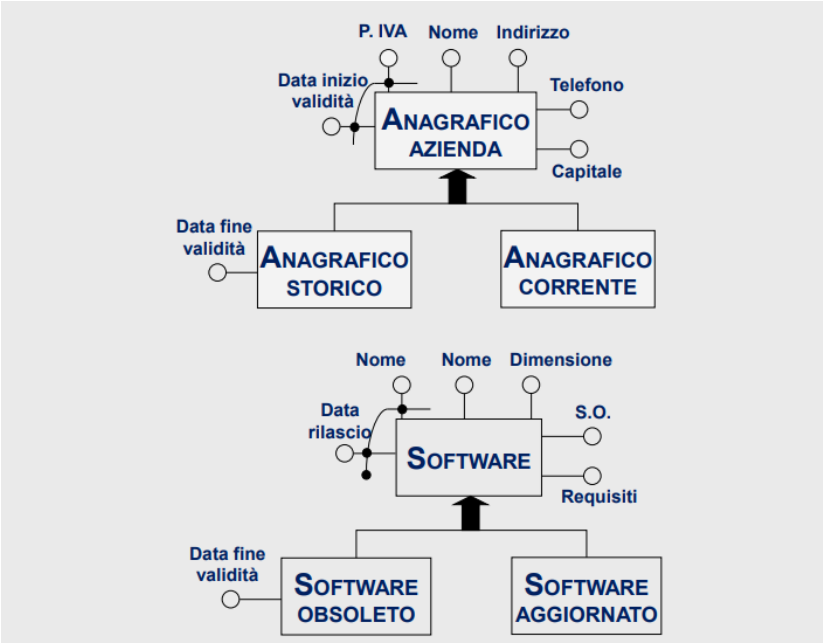
Altro caso utile è la *reificazione di attributo di relazione*, in cui il musicista ha una partecipazione ad un'unica orchestra, avendo che ogni partecipazione può avere strumento diverso:



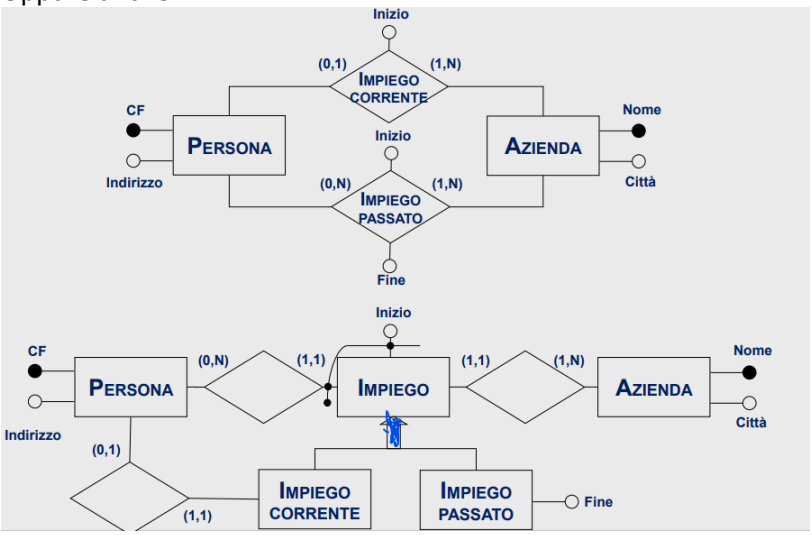
Caso particolare: l'impiegato manager che partecipa a più progetti (come Impiegato) partecipando alla gestione di uno singolo (come Manager):



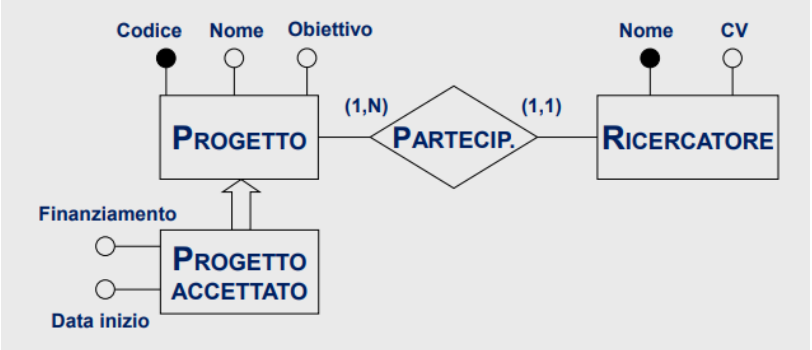
Similmente avendo più occorrenze a livello cronologico di un certo concetto logico si ha la *storicizzazione del concetto*, rappresentendo la gestione delle versioni attuali/future, con generalizzazione totale delle entità relazionalmente collegate:



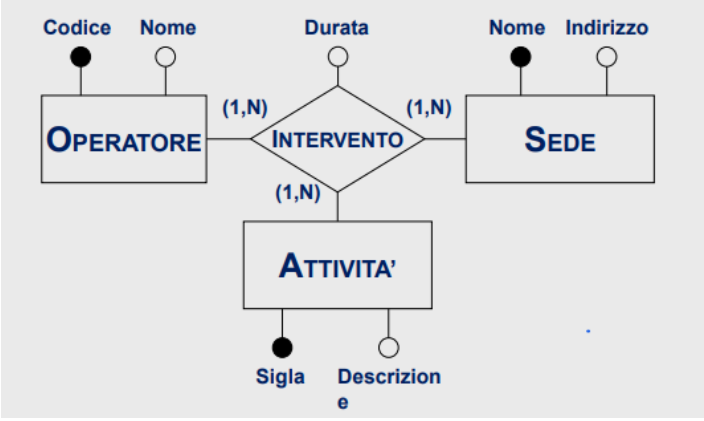
Oppure anche:



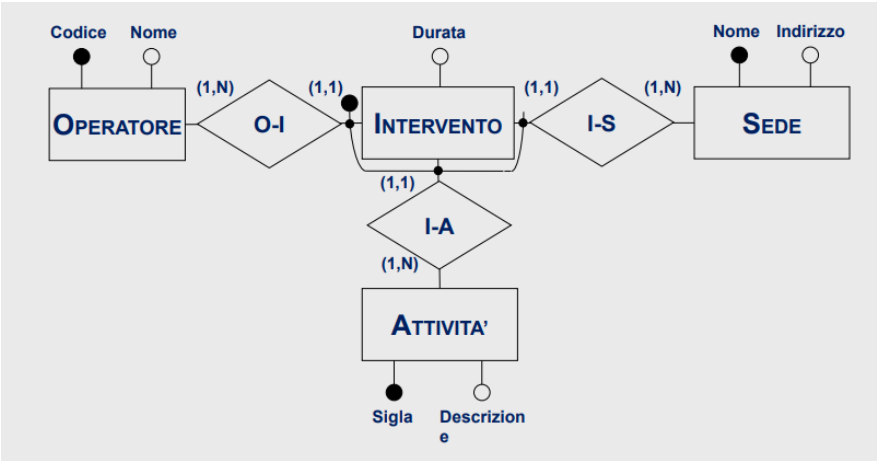
Similmente si ha l'*evoluzione di concetto*, per esempio nel corso del tempo si vuole descrivere un finanziamento e l'accertamento di un progetto certificato (evitando se possibile relazioni ternarie):



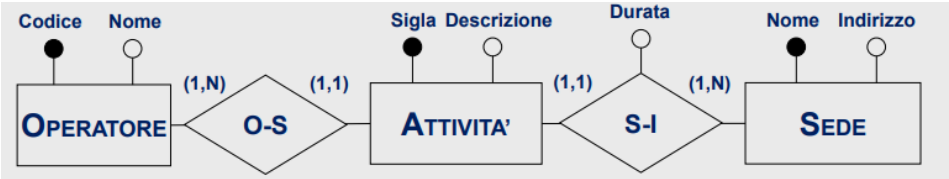
Come detto, ecco le *relazioni ternarie*, che complicano la gestione e ottimizzazione del progetto DB:



se possibile invece reificandola (*reificazione di relazione ternaria*), esprimendola come entità:

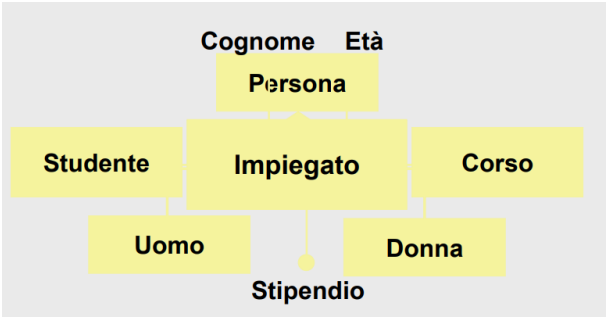


oppure:

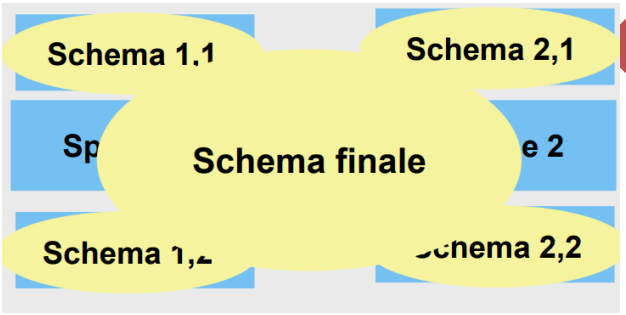


Le *strategie di creazione* quindi sono tre:

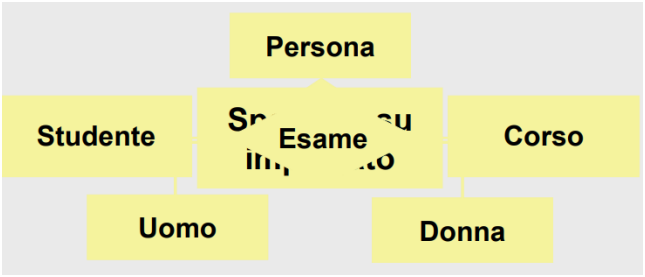
- *top down*, partendo dalle specifiche iniziali, si crea lo schema iniziale (creazione delle entità iniziali), allargando ad altre entità importanti (schema intermedio), arrivando allo schema finale (conclusione delle attività). Per esempio, Studente che fa un Esame per un corso, oppure Persona, generalizzando per Uomo e Donna. É quindi coerente, avendo meno ridondanza senza preoccuparsi dell'implementazione, offrendo visibilità completa di effetti e modifiche sul database.



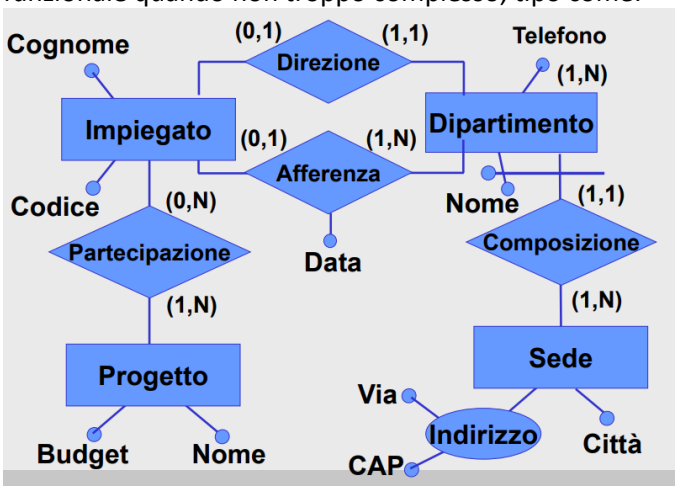
- *bottom up,* partendo con la creazione di schemi dagli attributi alle entità, poi mettendole insieme. Ottiene rapidamente cose funzionanti grazie alla visione ad alto livello di tabelle/relazioni/viste, ma spendere più tempo per requisiti che possono essere migliorati.



con successive raffinazioni:



Successivamente si ha anche la *inside-out*, partendo da un punto ed estendendo il diagramma logico, funzionale quando non troppo complesso, tipo come:



Nella pratica si segue una *strategia mista*, individuando i concetti principali, poi decomponendoli e magari raffinandoli nel corso del tempo con l'ausilio delle varie pratiche appena descritte.

Per descrivere la *qualità di uno schema concettuale* valuta:

- la *correttezza*, attenzione ad errori sintattici (aggiunta generalizzazioni tra relazioni ed identificatori di relazione) oppure semantici (con possibile violazione della specifiche). Quindi attenzione al diagramma ER;
- la *completezza*, traducendo tutti i concetti parte del diagramma ER;
- la *leggibilità*, creando concetti comprensibili sia da tecnici che non tecnici, disegnando uno schema in maniera chiara (senza intersezioni di linee/sovrapposizioni), partendo dai concetti chiave al centro e poi mettendo entità genitori sopra le figlie;
- la *minimalità*, legato alla leggibilità, evitando generalizzazioni non necessarie oppure entità senza attributi o entità non utili alla progettazione, quindi superflue.

Una possibile metodologia può essere:

- analisi dei requisiti, eliminando ambiguità, creando glossario utile e raggruppando requisiti in insiemi omogenei;
- passo base (schema scheletro, con concetti rilevanti), *decomponendo* i requisiti dallo schema scheletro e *integrando* vari sottoschemi;

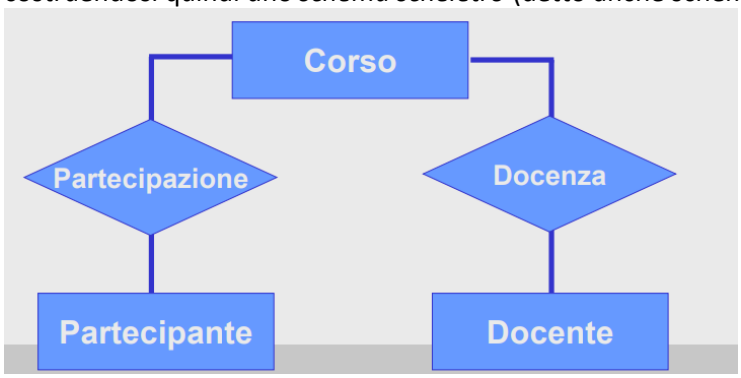
- passo iterativo (da ripetere finché non soddisfatto), raffinando i concetti base e aggiungendone di specifici;
- analisi di qualità, analizzando lo schema presente e modificandolo.

Esempio: società di formazione, partendo da una frase generica di richiesta (realizzato in maniera completa, con approccio misto):

Frasi di carattere generale

Si vuole realizzare una base di dati per una società che eroga corsi, di cui vogliamo rappresentare i dati dei partecipanti ai corsi e dei docenti.

costruendosi quindi uno *schema scheletro* (detto anche schema skeleton):

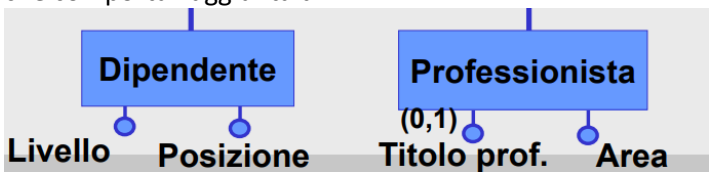


specificando il tutto con apposite frasi:

Frasi relative a tipi specifici di partecipanti

Per i partecipanti che sono liberi professionisti, rappresentiamo l'area di interesse e, se lo possiedono, il titolo professionale. Per i partecipanti che sono dipendenti, rappresentiamo invece il loro livello e la posizione ricoperta.

che comporta l'aggiunta di:



discernendo quindi in fasi successive requisiti ed entità utili (non tutto serve, es. 5000 come numero non dice niente alla progettazione) o anche cose non chiare inizialmente:

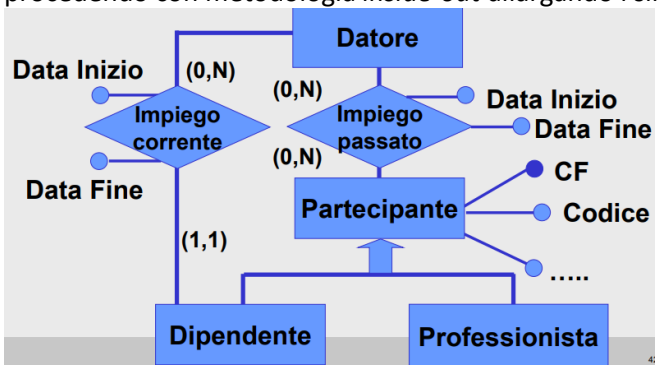
Frasi relative ai partecipanti

Per i partecipanti (circa 5000), identificati da un codice, rappresentiamo il codice fiscale, il cognome, l'età, il sesso, la città di nascita, i nomi dei loro attuali datori di lavoro e di quelli precedenti (insieme alle date di inizio e fine rapporto)

Frasi relative ai datori di lavoro

Relativamente ai datori di lavoro presenti e passati dei partecipanti, rappresentiamo il nome, l'indirizzo e il numero di telefono.

procedendo con metodologia *inside-out* allargando i singoli pezzi:

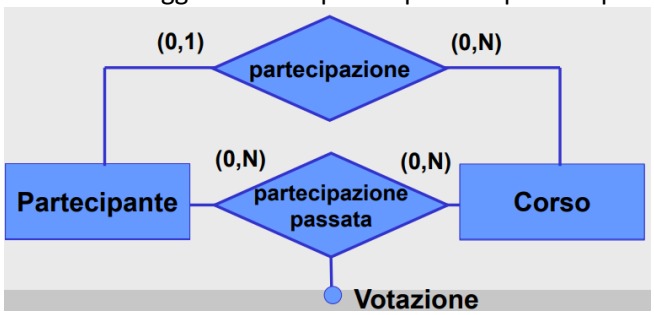


poi si procede alla storicizzazione ulteriore dei partecipanti:

Frasi relative ai partecipanti (2)

Per i partecipanti, si vuole mantenere le informazioni sulle edizioni dei corsi che stanno attualmente frequentando e quelli che hanno frequentato nel passato, con la relativa votazione finale in decimi.

definendo l'aggiunta della partecipazione passata per distinzione concettuale/logica:

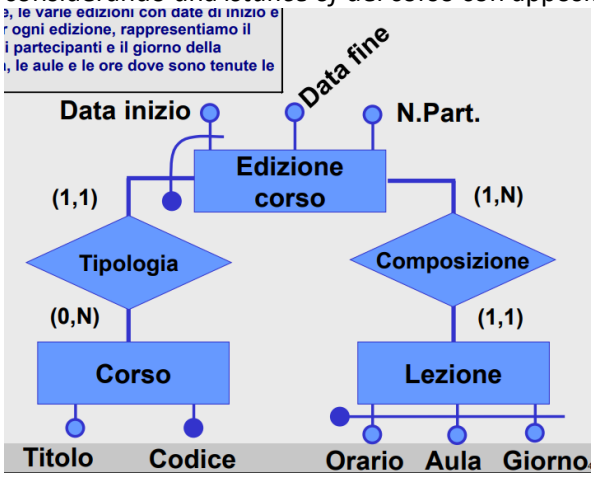


proseguendo con la modellazione che considera l'informazione dei corsi tenuti nelle rispettive edizioni per partecipanti e periodo di partecipazione:

Frasi relative ai corsi

Per i corsi (circa 200), rappresentiamo il titolo e il codice, le varie edizioni con date di inizio e fine e, per ogni edizione, rappresentiamo il numero di partecipanti e il giorno della settimana, le aule e le ore dove sono tenute le lezioni.

considerando una *istance of* del corso con appositá relazione parte dell'identificatore:



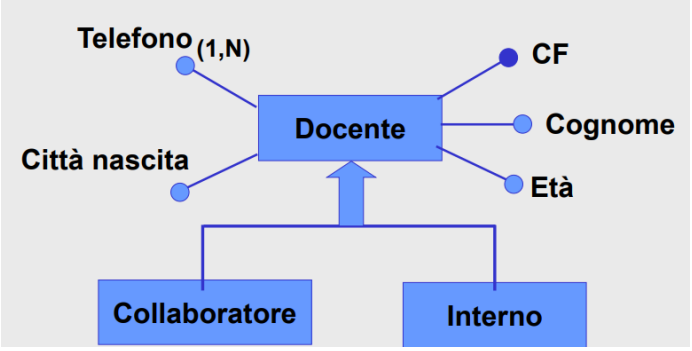
Scritto da Gabriel

inserendo successivamente le informazioni per i docenti:

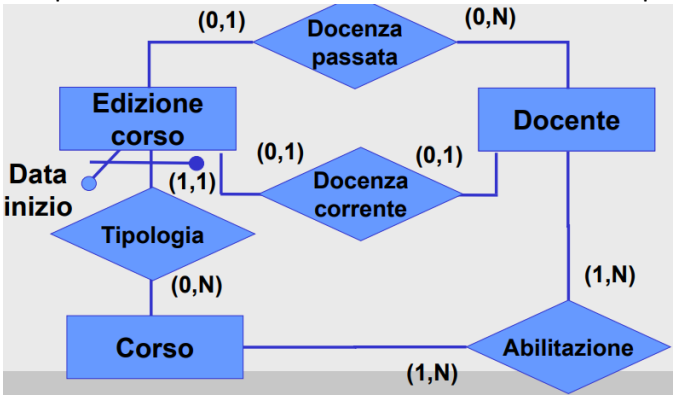
Frasi relative ai docenti

Per i docenti (circa 300), rappresentiamo il cognome, l'età, la città di nascita, tutti i numeri di telefono. I docenti possono essere dipendenti interni della società di formazione o collaboratori esterni.

e si distingue tra i due tipi di docenti con aggiunta di attributi:



completando tutte le informazioni relative a docenze passate/attuali ed edizioni dei corsi:



Esercitazione 3: Progettazione concettuale

La serra **PianteBellissime** vuole dotarsi di un sistema informativo per gestire la manutenzione e la vendita delle piante. La serra è specializzato in piante di alto fusto, ma vende anche piante fiorite.

Per la piante al alto fusto, si vogliono memorizzare i singoli esemplari in magazzino. Ogni esemplare è contraddistinto da un codice individuale, la specie, il nome botanico, l'età in anni, il clima ed il tipo di terriccio ideali.

Per le piante fiorite, non è di intesse memorizzare i singoli esemplari, ma solo le quantità a disposizione in forma aggregata. Di conseguenza, I tipi di piante fiorite non hanno un codice individuale, sono raggruppate in insiemi identificati tramite il nome botanico; di ogni insieme sono noti il periodo e la durata della fioritura, il numero di esemplari in magazzino, nonché il costo del singolo esemplare (uguale per tutti gli esemplari). [1]

Le piante di alto fusto, se non vendute, devono essere sottoposte ad un trattamento annuale di svasamento con terricci diversi a seconda del tipo. A tale scopo, occorre associare ogni specie di pianta ad uno o più terricci.

La serra ha una clientela di due tipi: singoli individui che acquistano al dettaglio e negozianti che acquistano all'ingrosso. Soltanto per questi ultimi è necessario conoscere nome, cognome, codice fiscale e partita iva.

Di ogni vendita interessa memorizzare la data, l'identificativo, le piante acquistate. Ogni vendita contiene uno o più tipi di piante fiorite e piante ad alto fusto. Per ogni vendita, ogni tipo di pianta fiorita è venduto in una certa quantità; viceversa, ogni vendita contiene solo un esemplare di un tipo di pianta ad alto fusto, di cui si vuole conoscere il prezzo di vendita, che può cambiare ad ogni vendita. Solo per le vendite all'ingrosso, si interessa conoscere l'acquirente.

Si vuole realizzare una base di dati per l'organizzazione di una serra. Si noti che le instanze dell'entità "Pianta Alto Fusto" sono gli esemplari specifici, mentre le instanze di "Pianta Fiorita" si riferiscono ai tipi di piante (per es. *Palma Nana*) e non agli specifici esemplari di quel tipo

Ogni **pianta Alto Fusto** è caratterizzata da:

- Età
- Nome Botanico
- Codice Individuale

[1] Si noti quindi la differenza tra "piante fiorite" e "piante ad alto fusto", rappresentate come due entità separate nel diagramma ER. Un'instanza dell'entità "piante fiorite" rappresenta un tipo di pianta fiorita, che può essere disponibile in magazzino in un certo numero di entità. Viceversa, le instanze di "piante ad alto fusto" rappresentano i singolo esemplari: due instanze si possono riferire a due esemplari dello stesso tipo di pianta ad alto fusto.

- Clima

Ogni **Pianta Alto Fusto** appartiene ad una **Specie**, ma una Specie ha una o più Piante in vendita. La Specie è identificata da:

- Nome Specie

Ogni **Specie** di pianta è associata ad uno o più **Terricci**, ma un tipo di **Terriccio** può avere più **Specie** di piante associate. Il **Terriccio** è identificato da:

- Tipo Terreno

Ogni **Pianta Alto Fusto** può essere stata **Venduta** oppure no ad un determinato prezzo, ma una **Vendita** può contenere più di una **Pianta Alto Fuso**. Ogni **Vendita** è caratterizzata da:

- ID
- Data di vendita

Ogni **Pianta Fiorita** può essere stata **Venduta** oppure no in un determinato numero di esemplari, ma una **Vendita** può contenere più di una **Pianta Fiorita**. Ogni **Pianta Fiorita** è identificata da:

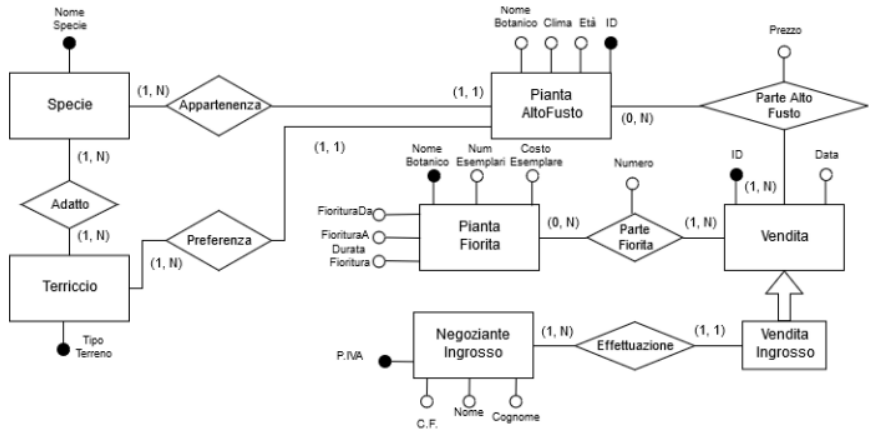
- Data inizio Fioritura
- Data fine Fioritura
- Durata della fioritura
- Numero di Esemplari
- Costo del singolo esemplare

Si noti l'assenza della Generalizzazione Pianta, in cui Pianta AltoFusto e Pianta Fiorita sarebbero state specializzazioni. Si ricordi che le istanze di Pianta AltoFusto sono i singoli esemplari in magazzino, mentre le istanze di Pianta Fiorita sono i tipi botanici. Non ha senso qui fare questa generalizzazione perché sono due concetti parecchio diversi.

Venditore Ingrosso è una specializzazione di **Venditore**. (**Venditore** può o non essere all'ingrosso, mentre ogni **Venditore Ingrosso** è anche un **Venditore**). Nel caso in cui una **Vendita** sia **all'Ingrosso**, essa è effettuata da un **Negoziante all'Ingrosso**, ma un **Negoziante all'Ingrosso** può prendere parte a più **Vendite all'ingrosso**. Ogni negoziante è caratterizzato da :

- Partita IVA (chiave)
- Codice Fiscale
- Nome
- Cognome

Diagramma ER:

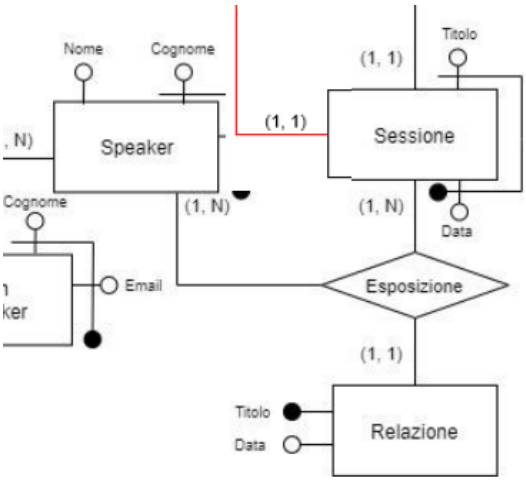


Vincoli di Integrità Non Rappresentabili in ER:

- Una **Vendita** riguarda almeno una **Piante Fiorita** o una **Piante Alto Fusto**.

Note utili sugli schemi concettuali/schemi logici:

- Se non diversamente specificato (con termini come *almeno uno/a* o altro), la cardinalità prende sempre opzionalità (0,1) o (0,N), che dipende naturalmente dal contesto rappresentato
- Ci possono essere anche attributi con cardinalità (1,N) → Esempio Primo Appello 21/22
  oppure con cardinalità (0,1) (a tal proposito, si ricordi nello schema logico di rappresentarli con un asterisco)
- Qualora si abbia una relazione per cui si preveda una cardinalità (1,1) e più cardinalità (1,N), va sia inserita la chiave esterna nella tabella di collegamento che inserita la chiave esterna nella relazione. Mi spiego meglio: normalmente le relazioni prevedono cardinalità (0/1 – 1/N). Il caso classico che normalmente viene modellato in merito a questo esempio è la (1,1) ed (1,N); basta il collegamento come chiave esterna da parte di una delle due tabelle. Se però si ha un caso come questo:



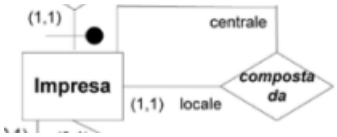
allora lo schema logico prevederà:

Relazione (Titolo, Data, Speaker, Sessione)

Esposizione (SpeakerCognome, SpeakerEmail, SessioneTitolo, SessioneData, Relazione)

- Esposizione.SpeakerCognome -> Speaker.Cognome
- Esposizione.SpeakerEmail -> Speaker.Email
- Esposizione.SessioneTitolo -> Sessione.Titolo
- Esposizione.SessioneData -> Sessione.Data
- Esposizione.Relazione -> Relazione.Titolo

- Esistono particolari casi di entità collegate ricorsivamente con sé stesse; per esempio, in una modellazione del prof, emerge Impresa formata da un'unità centrale e unità periferiche, facendo intendere non occorrano altri attributi specifici alla loro realizzazione.
  Quindi, da un punto di vista di schema logico, viene introdotta come chiave "ImpresaCentrale" riprendendo la chiave già esistente di Impresa.



Impresa (Codice,CodiceCentrale)

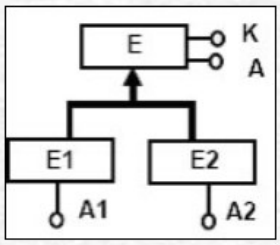
Impresa.Codice -->Soggetto.Codice

Impresa.CodiceCentrale --> Impresa.Codice

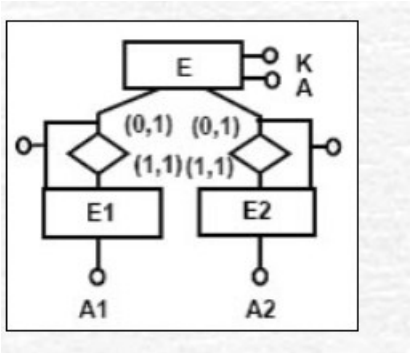
- Normalmente il prof adotta, come pattern, la minimizzazione dei valori nulli.
  Possiamo però approfondire in questo modo:

Per la generalizzazione esistono tre modi di eliminare le gerarchie:

Prendiamo come esempio il seguente schema E-R:

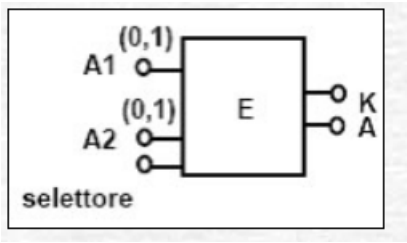


1) Mantenimento delle entità con associazioni



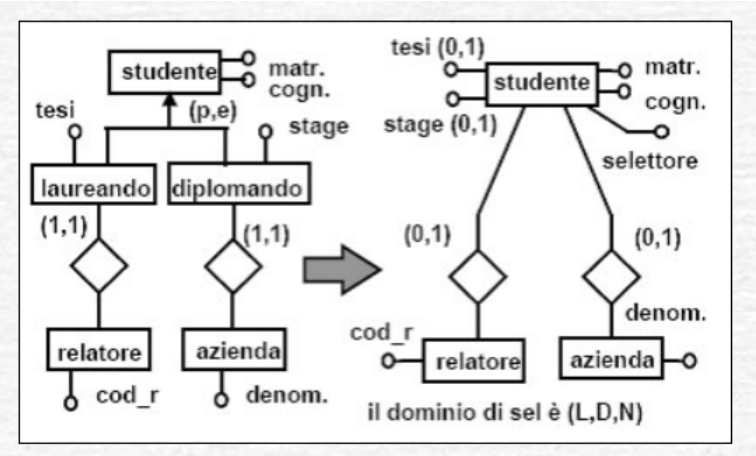
Tutte le entità vengono mantenute. Le entità figlie vengono messe in associazione con l'entità padre e sono identificate esternamente tramite l'associazione. La cardinalità (0,1) indica che per tale associazione l' entità padre può avere zero o una entità figlio. Mentre (1,1) che l' entità figlio può avere un solo padre.

2) Collasso verso l'alto



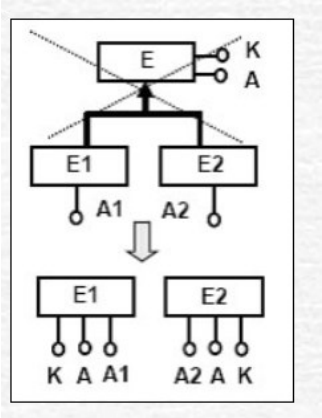
Esso riunisce tutte le entità figlie nell'entità padre.
E1 ed E2 vengono eliminate e le loro proprietà vengono aggiunte all' entità padre . Gli attributi obbligatori per le entità figlie divengono opzionali per il padre , indicato dalla cardinalità (0,1), con la conseguenza che si *avrà una certa percentuale di valori nulli*. All' entità ottenuta viene aggiunto un ulteriore attributo (selettore) che specifica se una istanza di E appartiene a una delle sottoentità.

Se la gerarchia era totale ed esclusiva il selettore ha N valori, quante sono le sottoentità. Se la gerarchia era parziale esclusiva il selettore ha N+1 valori; il valore in più serve per le istanze che non appartengono ad alcuna sottoentità. Se avessimo un overlapping occorrerebbero tanti selettori booleani quante sono le sottoentità. Vediamo di seguito un esempio di uno schema in cui è stata collassata la gerarchia verso l' alto:

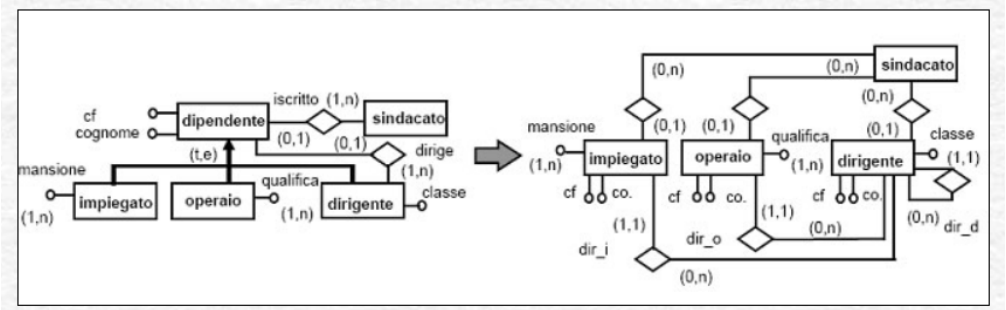


3) Collasso verso il basso

Si elimina l'entità padre trasferendone gli attributi su tutte le entità figlie. Una associazione del padre è replicata, tante volte quante sono le entità figlie *la soluzione è interessante in presenza di molti attributi di specializzazione* (con il collasso verso l'alto si avrebbe un eccesso di valori nulli).

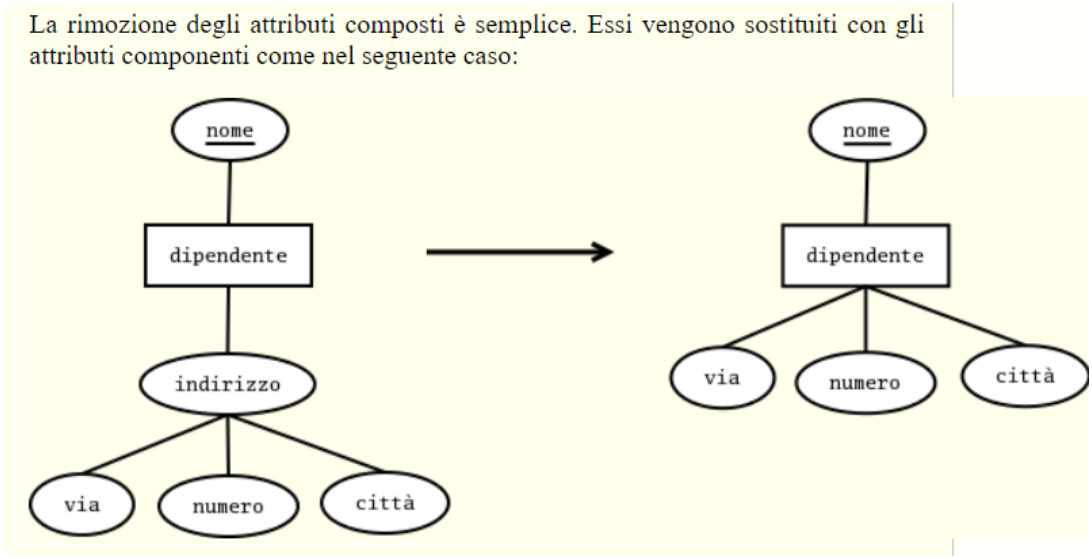


Si noti che se la copertura è parziale non si può fare il collasso verso il basso. Dove mettere gli E che non sono né E1, né E2 ? Mentre se la copertura è overlapping introduce ridondanza, infatti per una istanza presente sia in E1 che in E2 si rappresentano due volte gli attributi di E.
Vediamo di seguito un esempio di uno schema in cui è stata collassata la gerarchia verso il basso:



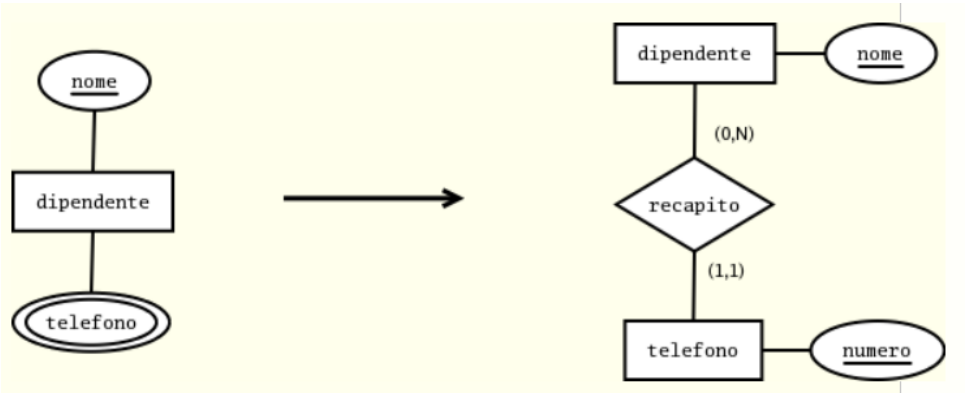
4) Nota di contorno: gli attributi multivalore e attributi composti.

Esistono campi come *Telefono* oppure *Indirizzo* per i quali sarebbe ridondante memorizzare tutti i campi in una sola tabella.
La rimozione degli attributi composti è semplice. Essi vengono sostituiti con gli attributi componenti come nel seguente caso:



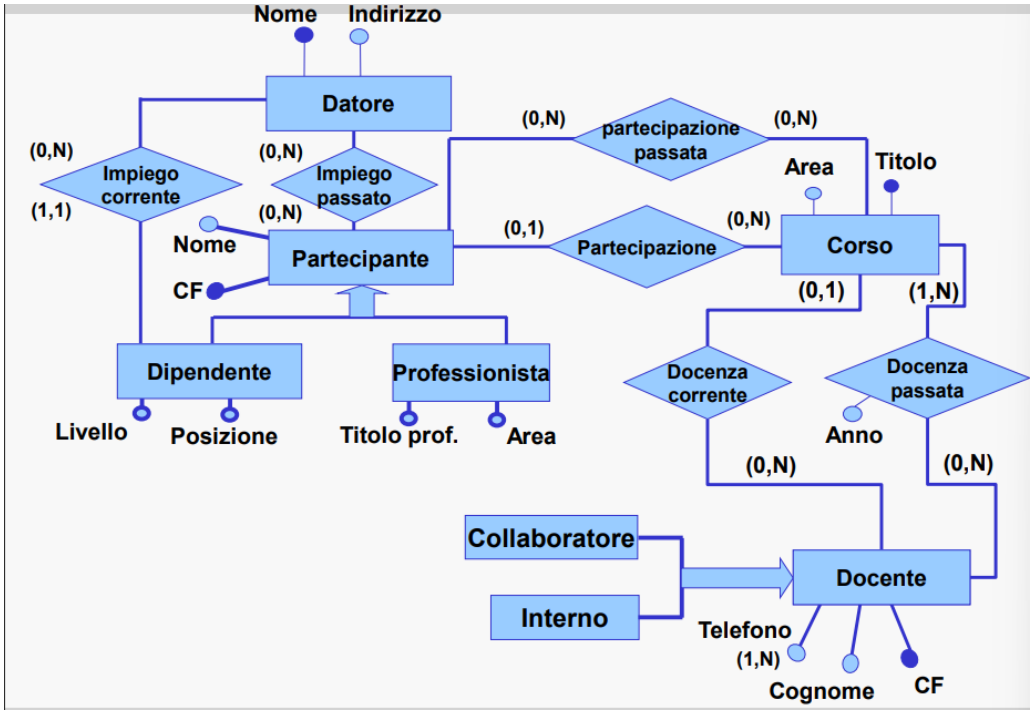
Eventuali gerarchie di attributi composti si traducono similmente procedendo dalle foglie verso la radice. A questo punto tutti gli attributi sono semplici, derivati, oppure multivalore.

Per quanto riguarda invece il campo Telefono che è multivalore, si crea una nuova entità che contiene i valori dell'attributo e la si collega all'entità che possedeva l'attributo mediante una nuova relazione uno a molti o molti a molti, a seconda dei casi.



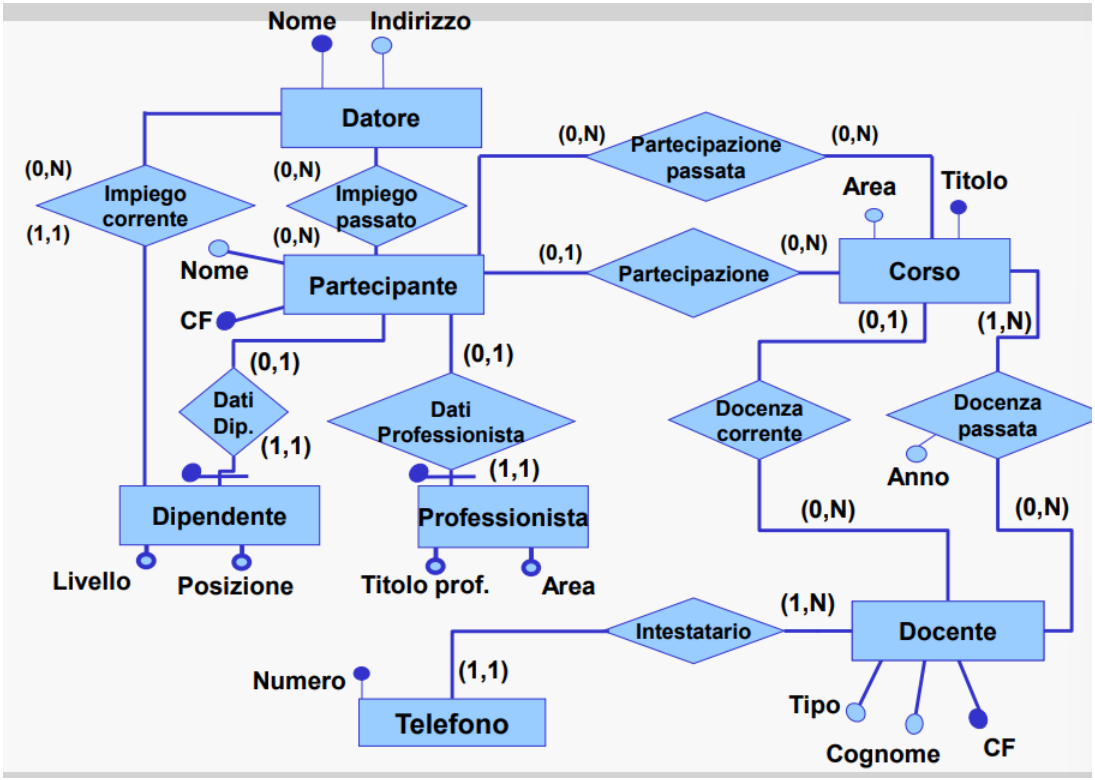
Se l'attributo telefono è opzionale, il vincolo di cardinalità (1,1) va sostituito con (0,1). Se inoltre un numero di telefono può essere condiviso da più persone (ad esempio, il telefono di casa), allora il vincolo di cardinalità (1,1) va sostituito con (0,N).

L'esempio visto in classe parte dallo schema:



E attua una minimizzazione dei valori nulli:

- mantenendo Docenza Corrente e Docenza Passata come relazioni
- usando il mantenimento delle entità con associazioni tra Partecipante e Dipendente/Professionista
- creando per l'attributo multivalore Telefono, un'entità apposita
- mantenendo Partecipazione e Partecipazione Passata come relazioni e similmente anche per Impiego (Corrente e Passato)

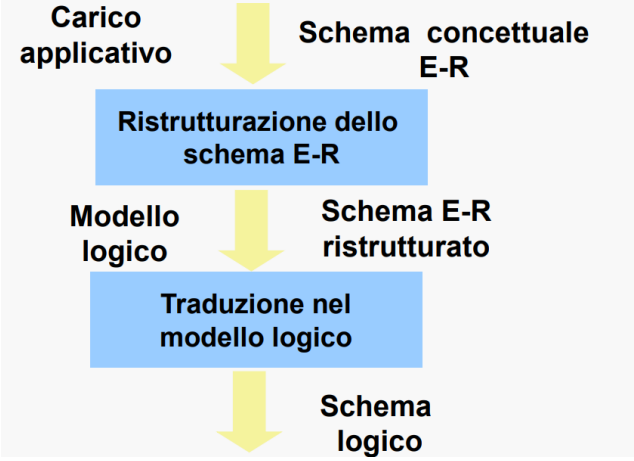


Creando come schema:

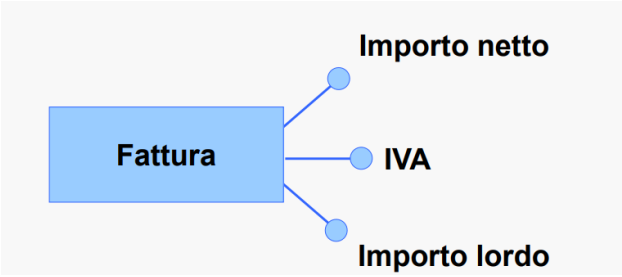
- Partecipante(CF, Nome, Corso-Attuale)
    - Partecipante.Corso-Attuale → Corso.Titolo
- Datore(Nome, Indirizzo)
- Corso(Titolo, Area, CF-Docente-Corrente)
    - Corso.CF-Docente-Corrente → Docente.CF
- Docente(CF, Cognome,Tipo)
    - Docente.Tipo ∈ { Collaboratore, Interno }
- Telefono(Numero, CF-Docente)
    - Telefono.CF-Docente → Docente.CF
- Dipendente(CF, Livello, Posizione, Datore)
    - Dipendente.CF → Partecipante.CF
    - Dipendente.Datore → Datore.Nome
- Professionista(CF,Titolo-Prof, Area)
    - Professionista.CF → Partecipante.CF
- ImpiegoPassato(CF-Partecipante,Nome-Datore)
    - ImpiegoPassato.CF-Partecipante → Partecipante.CF
    - ImpiegoPassato.Nome-Datore → Datore.Nome
- PartecipazionePassata(CF-Partecipante,Titolo-Corso)
    - PartecipazionePassata.CF-Partecipante → Partecipante.CF
    - PartecipazionePassata.Titolo-Corso → Corso.Titolo
- DocenzaPassata(CF-Docente,Titolo)
    - DocenzaPassata.CF-Docente → CF.Docente
    - DocenzaPassata.Titolo → Corso.Titolo

Progettazione logica

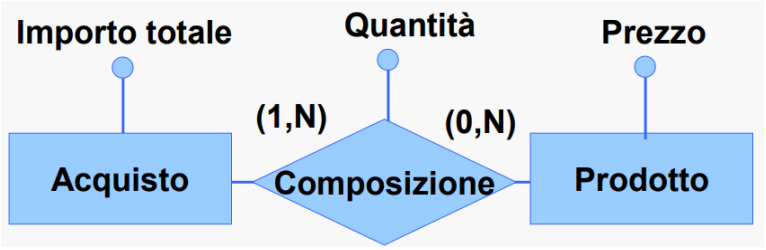
Successivamente all'analisi, ci dedichiamo direttamente alla base di dati, con alcuni aspetti delle tabelle rappresentabili direttamente e altri no, considerando le prestazioni. L'idea grafica è:



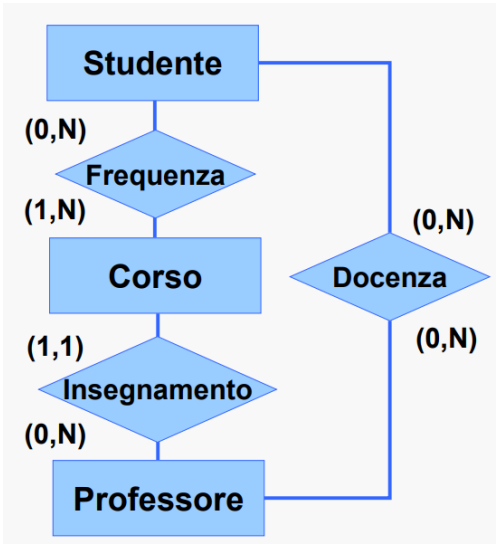
L'idea è quindi di semplificare la traduzione, ottimizzandolo chiamandolo schema ER "strutturato".
La prima fase è *l'analisi delle ridondanze*, semplificando quindi le interrogazioni, ma appesantendo gli aggiornamenti, con aumento dello spazio occupato e generazione di inconsistenze.
Essi possono essere attributi della stessa o altre entità/relationships. Ad esempio:



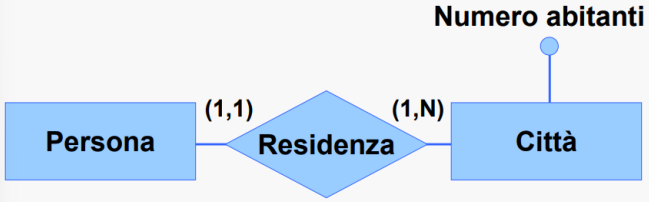
Similmente nel caso di *attributo derivabile da altre entità* (cioè la quantità che viene considerato sia nella fase di Acquisto che nel pagamento del Prezzo e, per evitare duplicazione, si mette come campo nella stessa relazione):



Similmente può esservi una *ridondanza dovuta ad un ciclo*, con informazioni duplicate in una serie di tabelle, riducendo se possibile il numero di join, computazionalmente impegnativi.
Non sempre è svantaggioso avere ridondanze, specie se si tratta di attributi molto richiesti.



Vediamo ad esempio *l'analisi di ridondanza* sulla relazione seguente, chiedendosi se serve aggiungere "Numero abitanti".



Concetto	Tipo	Volume
Città	E	200
Persona	E	1000000
Residenza	R	1000000

- Operazione 1 (500 volte al giorno): memorizza una nuova persona con la relativa città di residenza
- Operazione 2 (2 volte al giorno): stampa tutti i dati di una città (incluso il numero di abitanti, ca. 1M / 200 = 5000 per città)

Consideriamo la prima operazione; inseriamo la nuova Persona, la quale deve essere collegata a Città accedendovi in scrittura, aggiungendo poi una tupla alla relazione Residenza.

- Operazione 1 (500 volte al giorno): memorizza una nuova persona con la relativa città di residenza

Concetto	Costrutto	Accessi	Tipo	
Persona	Entità	1	S	x 500 volte/giorno
Residenza	Relazione	1	S	x 500 volte/giorno
Città	Entità	1	L	x 500 volte/giorno
Città	Entità	1	S	x 500 volte/giorno

Come seconda operazione si considera la stampa di tutti i dati di una Città, accedendo due volte in lettura al giorno, sommati ai calcoli visibili:

- Operazione 2 (2 volte al giorno): stampa tutti i dati di una città (incluso il numero di abitanti, ca. 1M / 200 = 5000 per città)

Operazione 2

Concetto	Costrutto	Accessi	Tipo	
Città	Entità	1	L	x 2 volte/giorno

con i seguenti costi:

- Costi:
  - Operazione 1: 1500 accessi in scrittura e 500 accessi in lettura al giorno
  - Operazione 2: 2 accessi in lettura.
- Assumendo costo doppio per gli accessi in scrittura

  Il costo giornaliero è 1500x2+500+2=3502

Ipotizziamo ora lo scenario di *assenza di ridondanza*. Qui dovremmo aggiungere la Persona, poi in Residenza e basta, avendo un costo di 1000 scritture:

- Operazione 1 (500 volte al giorno): memorizza una nuova persona con la relativa città di residenza

Concetto	Costrutto	Accessi	Tipo	
Persona	Entità	1	S	x 500 volte/giorno
Residenza	Relazione	1	S	x 500 volte/giorno

da cui il numero di abitanti, con un accesso in lettura e 5000 accessi per recuperare gli altri:

- Operazione 2 (2 volte al giorno): stampa tutti i dati di una città (incluso il numero di abitanti, ca. 1M / 200 = 5000 per città)

Concetto	Costrutto	Accessi	Tipo	
Città	Entità	1	L	x 2 volte/giorno
Residenza	Relazione	5000	L	x 2 volte/giorno

Nota: questa parte (calcolo costi e ridondanze), negli esami, non serve. Utile in ottica progetti, vi è infatti una sezione di analisi dedicata nella relazione per le operazioni tipicamente compiute e il costo possibile (per risolvere il quale, se accesso molto frequente in scrittura, si adotta un indice, cfr. lezione dedicata).

Assumendo costo doppio dunque:

- Costi:
  - Operazione 1: 1000 accessi in scrittura
  - Operazione 2: 10002 accessi in lettura al giorno
- Assumendo costo doppio per gli accessi in scrittura

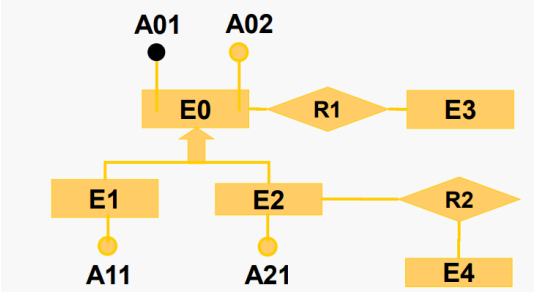
  Il costo giornaliero è 1000x2+10002=12002

Come si vede quindi non sempre le ridondanze sono uno svantaggio, come evidenziato dal costo decisamente più alto nel secondo caso, dato che non sapendo a priori con un campo quanti abitanti ci siano, volendo i dati di una città e dei suoi abitanti, è possibile recuperarli solamente con costo doppio, percorrendo ogni volta a doppio senso questa relazione.

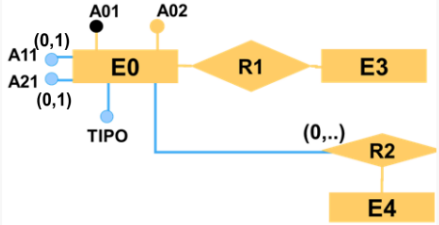
Successivamente vogliamo eseguire l'*eliminazione delle generalizzazioni/ristruttazione dello schema ER*, non rappresentando generalizzazioni, ridondanze, gerarchie nel corrispondente schema logico. Esse infatti verranno *sostituite con entity/relationships* oppure inserendo/eliminando attributìi. Ci sono 3 possibilità:

1. accorpamento delle figlie della generalizzazione nel genitore
2. accorpamento del genitore della generalizzazione nelle figlie
3. sostituzione della generalizzazione con relationship

Vediamo il seguente esempio (*eliminazione delle gerarchie*):



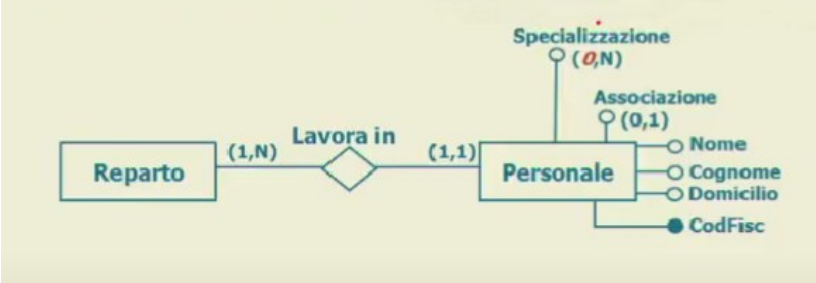
Il primo caso è di *cercare di accorpare le entità figlie nel padre* (quando ciò avviene gli attributi delle entità figlie diventano parte dell'entità padre, avendo per ciascuno di questi cardinalità (0,1) anche se erano attributi obbligatori; tale questione vale anche per le relazioni, quindi diventano tutte opzionali).
In generale questa operazione collassa verso l'alto, dunque.
Questa operazione genera molti valori nulli.



- Preferibile se gli accessi al padre e alle figlie sono contestuali
- Tabelle (es. E0) conterrà valori nulli.

L'esempio concreto è:

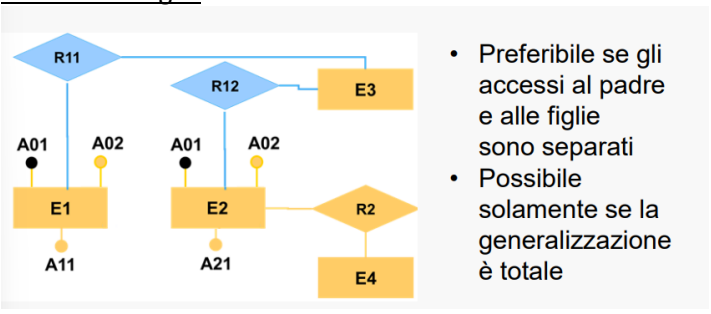
che diventa:



Per esempio, l'entità *Clienti* può essere rappresentata da sottoentità del tipo *Gold* e *Silver*. Se la generalizzazione non fosse totale, la cardinalità potrebbe essere (0,1). Il processo è l'estrazione di proprietà comuni e crearne un'entità generica (concettualmente), oppure inserirli come attributi all'interno dell'entità meno generale.
Si aggiunge finalmente un attributo discriminante, che permette di accorpare correttamente le figlie nel padre, e ciò funziona sia in caso di generalizzazione parziale che totale.

L'idea quindi è di *risparmiare l'accesso usando meno entità se possibile.*
Se nel caso concreto avessi come esempio una generalizzazione tra entità *Persona* e le entità *Uomo* e *Donna*, *Persona* va ristrutturata aggiungendo Sesso come campo. Nel caso di *Clienti* di prima, per esempio, aggiungerò un campo Tipologia/Categoria o una roba così.

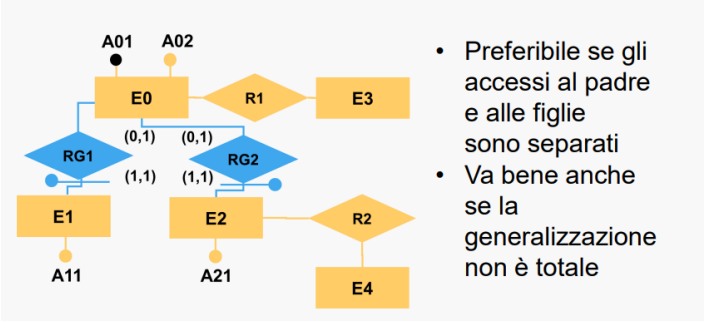
Nell'altro caso si cerca di avere solamente una generalizzazione totale al posto dei valori nulli (*accorpamento dell'entità genitore nelle entità figlie)*. Questa operazione crea ridondanza, duplicando i dati nelle entità figlie.



In questi caso se avessimo una generalizzazione tra entità Persona, con campi Cognome, Età, Codice Fiscale (id) e le entità Uomo e Donna, a queste ultime vanno aggiunti Cognome, Età, Codice Fiscale.
Quindi appunto mettiamo tutti gli attributi del padre nelle figlie, ma dobbiamo tenere conto delle relazioni. Ad esempio, se avessi un Medico ed un Volontario che lavorano in un Reparto, dovremmo considerare di creare non una relazione ternaria (in quanto non valgono le due cose insieme), bensì dobbiamo sdoppiare le relazioni per ciascuna sottoentità.

Scritto da Gabriel

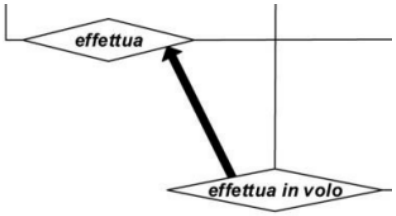
Altra idea dunque risulta la *sostituzione delle generalizzazioni con relationships*, perdendo alcune informazioni nel processo di eliminazione, introducendo associazioni (solitamente significa che uso delle relazioni per rappresentare il concetto, secondo l'idea dell'*is-a* (tipo Medico e Volontario, introducendo due relazioni apposite collegandolo a Personale), identificati esternamente dall'entità padre.
Questa soluzione si attua per <u>minimizzare il numero di valori nulli</u>.



- Preferibile se gli accessi al padre e alle figlie sono separati
- Va bene anche se la generalizzazione non è totale

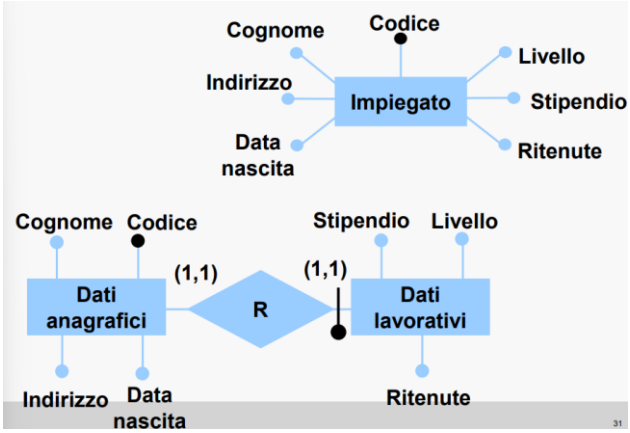
Attenzione: normalmente, *is-a* intende una cosa del tipo (Uomo e Donna *is-a* Persona, cioè fanno entrambe parte del concetto padre, si cerca di capire in base al contesto come).

Appunto (non visto in questo corso):
possono esistere le "is-a" tra le relazioni.

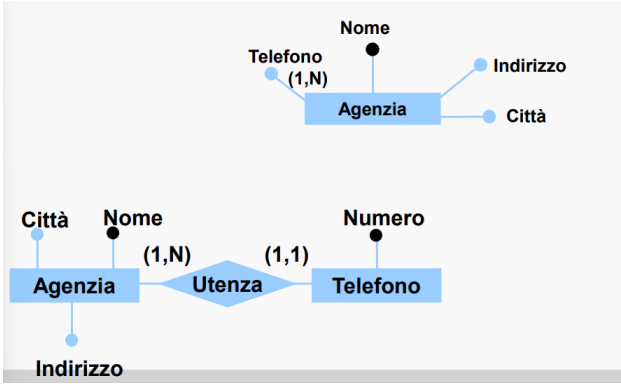


Nel concreto quindi significa sostituire tipo Uomo e Donna con delle relazioni apposite; vanno comunque aggiunti vincoli di partecipazione, in base alle occorrenze presenti (in quanto dipendenti logicamente dall'entità padre), avendo questa stessa entità padre come identificazione. In generale meglio garantire accessi separati, ma in molti altri casi ha senso fare così.
Questa soluzione può essere dispendiosa per trovare magari un dato facente parte di una certa relazione di partenza; infatti, generalmente vengono preferite le altre soluzioni.

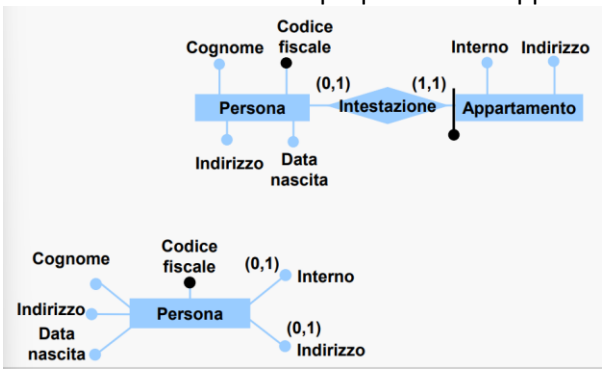
Vogliamo poi <u>*partizionare/accorpare entità/relationship*</u>, riducendo gli accessi separati e raggruppando attributi appartenenti a concetti diversi, prendendoli insieme.
Abbiamo l'esempio di *decomposizione/partizionamento verticale di entità*, suddividendo un concetto operando sugli attributi. Per esempio l'entità Impiegato, suddivisa in Anagrafica e Professione:



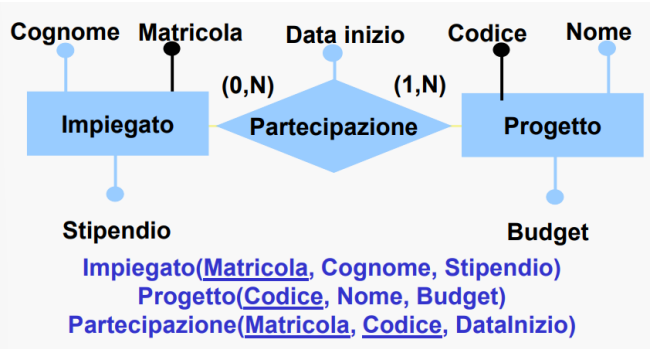
Un particolare partizionamento è l'*eliminazione di attributi multivalore* (come si vede, l'entità Agenzia che contiene l'attributo Telefono multivalore, mantenuta come entità a sé stante):



Eseguiamo inoltre *l'accorpamento entity/relationship*, operazione inversa al partizionamento, sapendo per esempio che vengono eseguite operazioni frequenti su una certa entità e quindi vogliamo ridurre gli accessi per risalire a quei dati, avendo dei valori nulli per entità non accorpate (ad esempio, valori Indirizzo/Interno nulli a fronte di assenza di proprietario di Appartamento).



Si opera poi la *scelta degli identificatori principali*, indispensabile per la traduzione relazionale, in maniera semplice e senza opzionalità per operazioni frequenti/importanti; se non esistono identificatori per certe entità vengono aggiunti *attributi codice.* In generale le entità usano identificatori attraverso le chiavi primarie, per poi scrivere gli attributi propri della relationship.



L'esempio *molti a molti/N-N*, che introduce delle chiavi esterne in una relazione intermedia per potersi collegare facilmente alle altre:
L'introduzione delle chiavi esterne è seguita dall'aggiunta dei relativi vincoli di entità referenziale:

Impiegato(Matricola, Cognome, Stipendio)
Progetto(Codice, Nome, Budget)
Partecipazione(Matricola, Codice, DataInizio)

Si aggiungono poi i vincoli di integrità referenziale
(Attributo_Rel_Esterna → Attibuto_Rel_Referenziata)
Partecipazione.Matricola → Impiegato.Matricola
Partecipazione.Codice → Progetto.Codice

Nelle relazioni derivate da relationships, inseriamo dei nomi espressivi per farne capire lo scopo:

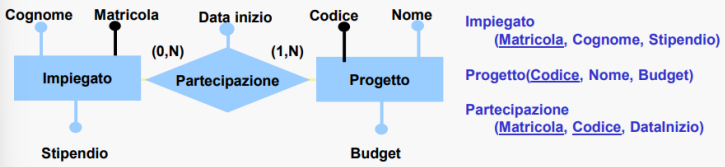
Impiegato(Matricola, Cognome, Stipendio)

Progetto(Codice, Nome, Budget)

Partecipazione(Matricola, Codice, DataInizio)

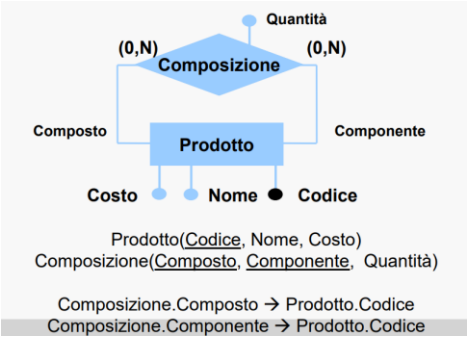
Partecipazione(Impiegato, Progetto, DataInizio)

La traduzione non garantisce vincoli di cardinalità minima in relationships N-N:

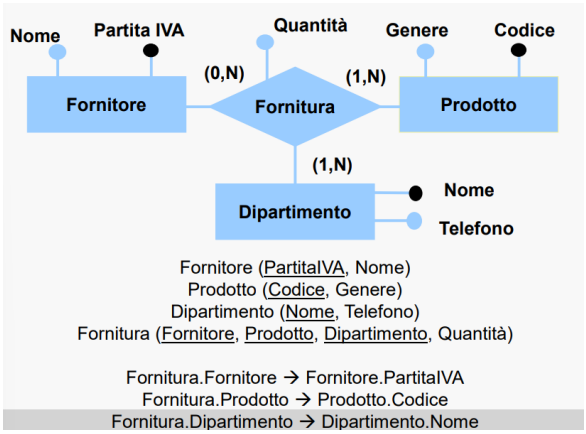


- Esempio: Possibile tupla (C, … , …) ∈ Progetto e nessuna tupla (…, C, …) ∈ Partecipazione
- Occorre vincoli di CHECK non supportati da quasi tutti DBMS.

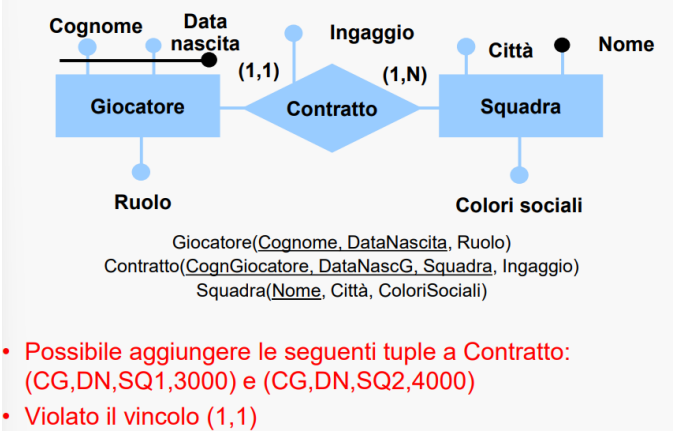
Similmente nel caso di *relationship ricorsive,* dove si inserisce una correlazione tra il Prodotto e la sua stessa Composizione, magari essendo interessati a recuperare in modo ridondante questa informazione:



Nel caso invece di *relationship n-arie*, si ha la stessa idea di aggiunta di vincoli di integrità referenziale:



Similmente posso avere soluzioni scorrette nel caso del Giocatore che fa parte di "n" Squadre firmando "n" Contratti, con il campo Squadra che viola il vincolo (1,1) (qui sarebbe (1:N) così, quindi sbagliato):

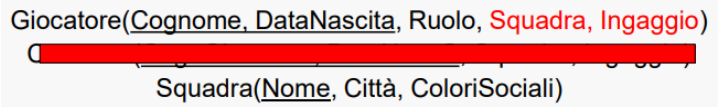


- Possibile aggiungere le seguenti tuple a Contratto: (CG,DN,SQ1,3000) e (CG,DN,SQ2,4000)
- Violato il vincolo (1,1)

Scritto da Gabriel

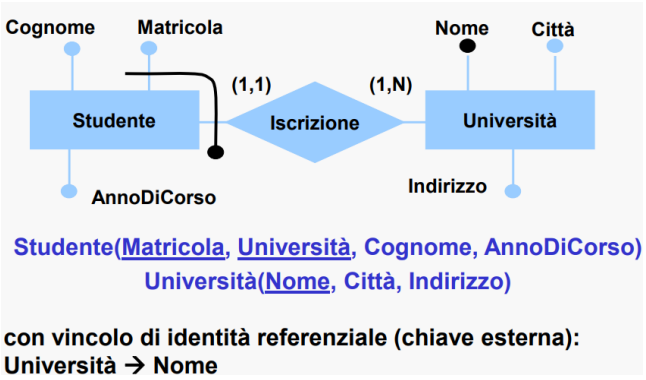
In questo caso per risolvere il conflitto "Squadra" non deve essere parte della chiave primaria, in quanto il giocatore può firmare *un solo* contratto per una squadra.

Notando che Contratto ha la stessa chiave di Giocatore, si toglie Contratto, e si accorpa in un'unica tabella:

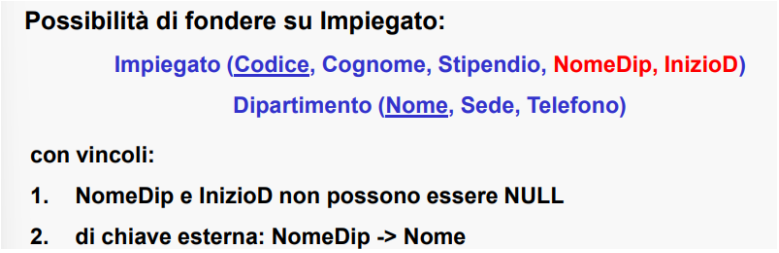


Le traduzioni permettono quindi capire che alcune relazioni ammettono normalmente valori nulli per specificare la loro correlazione logica/di cardinalità:

Similmente, l'attributo Università diventa parte della chiave primaria di Studente, facendone parte in quanto vincolo di integrità referenziale (*entità con identificazione esterna*):



Posso implementare l'uno ad uno/1-1 con l'idea di fusione/implementazione degli attributi con cardinalità minima e dunque diversa da NULL in vari modi, partendo dalla fusione su Impiegato:



similmente posso farlo su Dipartimento:

Possibilità di fondere su Dipartimento:

Impiegato (Codice, Cognome, Stipendio)

Dipartimento (Nome, Sede, Telefono, InizioD, CodDirettore)

con vincoli:

1. **CodDirettore e InizioD non possono essere NULL**
2. **di chiave esterna: CodDirettore -> Codice**

per poi volerlo farlo su entrambe (grazie alle cardinalità minime 1-1):

Possibilità di fondere su Dipartimento e Impiegato:

Impiegato (Codice, Cognome, Stipendio, NomeDip)

Dipartimento (Nome, Sede, Telefono, InizioD, CodDirettore)

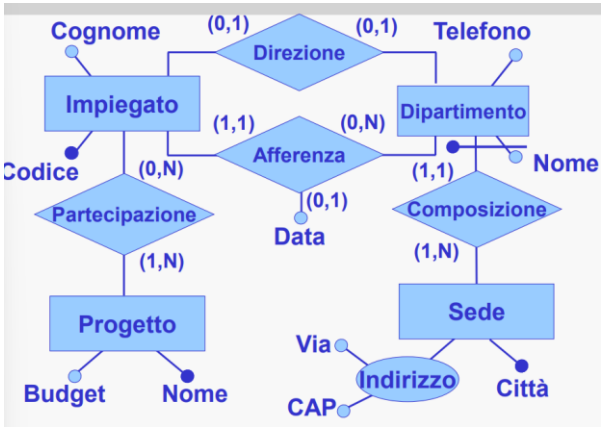
con vincoli:

1. **CodDirettore e NomeDip non possono essere NULL**
2. **di chiave esterna: CodDirettore -> Codice; NomeDip -> CodDirettore**

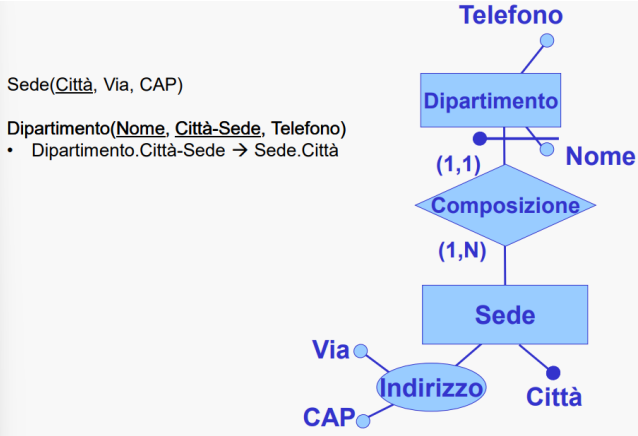
Progettazione logica: esercizi e conclusione

ESERCIZIO 1
DI PROGETTAZIONE LOGICA

basandosi su:



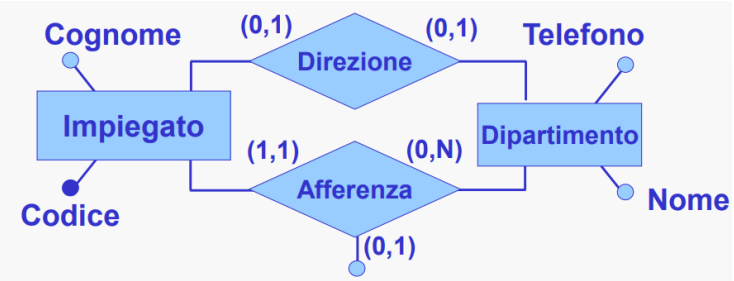
iniziando dalla Sede, collegando Città come campo di chiave esterna con Dipartimento, relazione di intermezzo.



Sede(Città, Via, CAP)

Dipartimento(Nome, Città-Sede, Telefono)
- Dipartimento.Città-Sede → Sede.Città

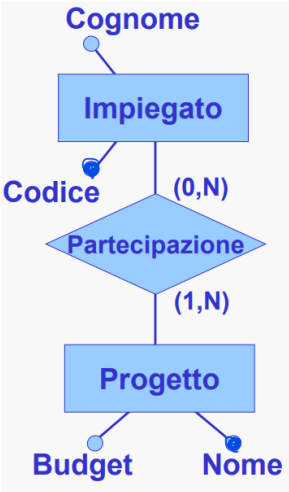
Proseguiamo con la parte superiore, con Direzione che, essendo relazione 1-1, definisce un collegamento con Impiegato, con Codice, Cognome, Dipartimento di Afferenza (chiave esterna). Il Direttore è chiave esterna al Codice dell'Impiegato. Attenzione: *le chiavi esterne possono, volendo, essere NULL,* come potrebbe essere Direttore. Successivamente diciamo che l'Impiegato fa parte di un Dipartimento.



Dipartimento(Nome, Città_Sede, Telefono, Direttore)
- Dipartimento.Direttore → Impiegato.Codice
- Direttore può essere NULL.

Impiegato(Codice, Cognome, DipartAffer, Sede, Data)
- Impiegato.DipartAffer, Impiegato.Sede → Dipartimento.Nome,Dipartimento.Citta_Sede
- Data può essere NULL

L'ultima parte definita, caso N a N, definisce il collegamento diretto di un Impiegato ad un Progetto, per mezzo di un'entità intermedia (definita Partecipazione).



Impiegato(Codice, Cognome, DipartAffer, Sede, Data)

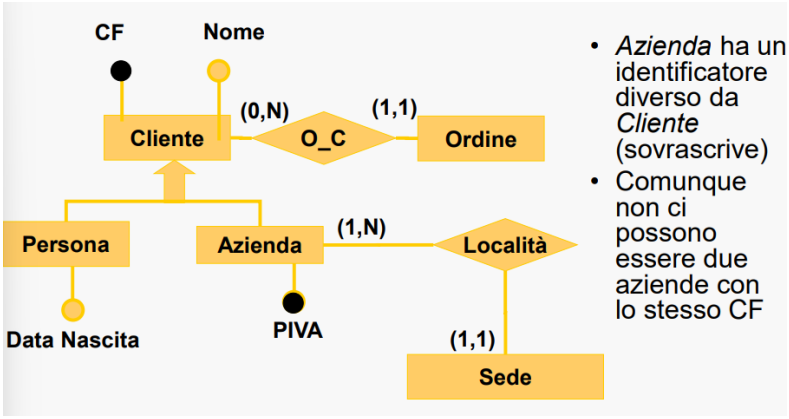
Progetto(Nome, Budget)

Partecipazione(Impiegato, Progetto)
- Partecipazione.Impiegato → Impiegato.Codice
- Partecipazione.Progetto → Progetto.Nome

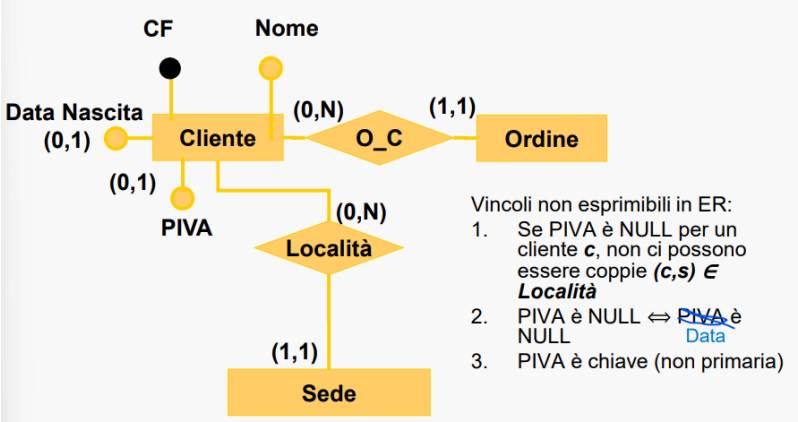
Lo schema finale dunque risulta essere:

- Impiegato(<u>Codice</u>, Cognome, DipartAffer, Sede, Data)
  - Impiegato.DipartAffer, Impiegato.Sede → Dipartimento.Nome,Dipartimento.Citta_Sede
  - Data può essere NULL
- Dipartimento(<u>Nome</u>, <u>Città_Sede</u>, Telefono, Direttore)
  - Dipartimento.Direttore → Impiegato.Codice
  - Direttore può essere NULL.
- Sede(<u>Città</u>, Via, CAP)
- Progetto(<u>Nome</u>, Budget)
- Partecipazione(<u>Impiegato</u>, <u>Progetto</u>)
  - Partecipazione.Impiegato → Impiegato.Codice
  - Partecipazione.Progetto → Progetto.Nome

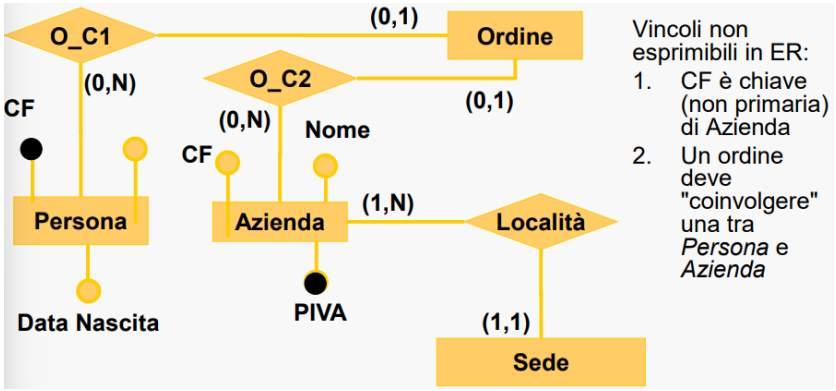
Vediamo l'esempio di *generalizzazione con identificatori diversi*, dicendo per esempio che un'entità figlia ha identificatore diverso da quello del padre, tipo due aziende che non hanno lo stesso CF:



Ad esempio, tolgo *Persona* come tabella e inserisco l'attributo *Data*, con cardinalità (0,1) e *Partita Iva*, anch'esso con cardinalità (0,1), definendo infine cardinalità (0, N) tra *Cliente* e *Località*.
Otteniamo come possibile *prima ristrutturazione (accorpamento dei figli nel padre)*:



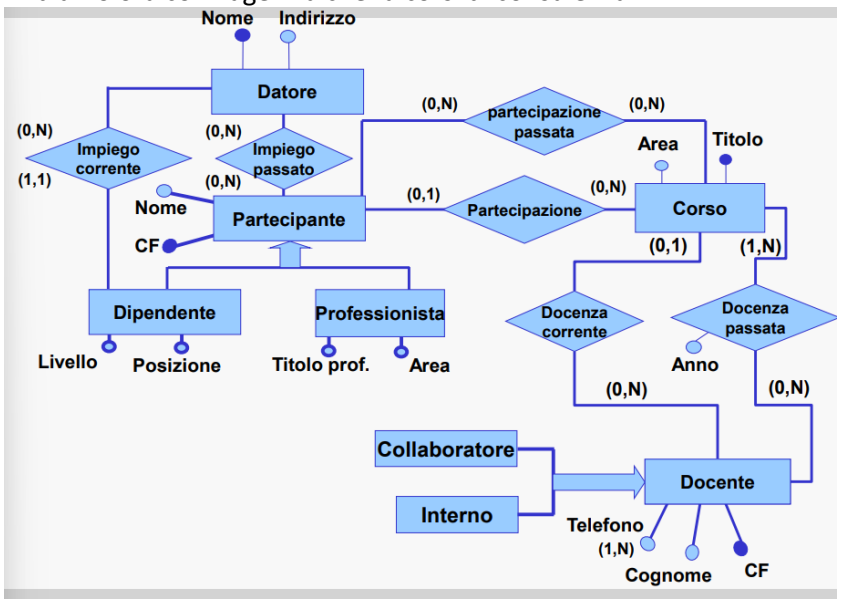
Segue la *seconda possibile ristrutturazione (accorpamento del padre nei figli).*
In pratica quindi *Cliente* non c'è più, *Persona* incorpora Nome, Codice Fiscale come campi (quest'ultimo chiave primaria), aggiungendo CF in *Azienda* come chiave non primaria (in essa ci saranno CF, Partita Iva, Nome).



Dato che un *Cliente* può essere direttamente un'*Azienda*, allora possiamo pensare di implementare un collegamento in questo modo.
CL(<u>CF</u>, NOME)
PERS(<u>CF</u>, DATA_N)
P.CF → CL.CF

ESERCIZIO 2
DI PROGETTAZIONE LOGICA

Andiamo ora con l'agenzia che fa corsi di consulenza:



L'idea è che il corso possieda "n" *Partecipanti*, con la distinzione della *partecipazione passata* e, tra i partecipanti, abbiamo *dipendenti* e *professionisti*. Il dipendente è *attualmente impiegato* oppure *era* impiegato presso un *Datore* di lavoro, altrimenti può essere *libero professionista*. I *Corsi* hanno un *Docente*, che *ha insegnato in passato* oppure *attualmente insegna*, distinguendo tra *collaboratore* ed *interno*.

Il primo passo è la *ristrutturazione*, togliendo le generalizzazioni.

SOLUZIONE CHE MINIMIZZA I VALORI NULLI

Scritto da Gabriel

seguendo l'idea di accorpamento del padre nei figli. La possibilità di ristrutturazione è di inserire delle relazioni (togliendo il collegamento diretto tra Partecipante e Dipendente/Professionista). Ovviamente si inglobano anche Collaboratore/Interno all'interno di Docente, mettendo Tipo come attributo. Il docente ha fino ad "n" numeri di telefono (caso attributo multivalore), inserendo una nuova relazione Telefono con un Numero, collegato relazionalmente al Docente.

In Telefono si avrà:
TELEFONO (NUMERO, INTESTAT)
T.INTEST → DOC.CF
DOC(CF, COGNOME, TIPO)
CORSO(TITOLO, AREA, DOC)
CORSO.DOC → DOC.CF

Essendoci la relazione N-N, tra Docente e Passato/Corrente mettiamo due tabelle extra.
DP(DOC, CORSO, ANNO)
DP.DOC → DOC.CF
DP.CORSO → CORSO.TITOLO

PARTECIP(CF, NOME, TITOLO_CORSO*)
P.TITOLO → CORSO.TITOLO
Risolvendo anche:
PART_PASS(CF, TITOLO)

Andiamo poi con il datore di lavoro
DATORE(NOME, INDIRIZZO) FK=Foreign Key
IMP-PASS(DATORE, PART) con ovviamente Datore FK con Nome e Part FK con CF di Partecipante

Successivamente consideriamo la professione:
PROF(CF, TITOLO, AREA)
PROF.CF → PART.CF
DIP(CF, LIVELLO, PROF, POSIZIONE, DATORE)
DIP.DATORE → DATORE.NOME
DIP.CF → PART.CF
Quindi lo schema finale:

- Partecipante(CF, Nome, Corso-Attuale)
  - Partecipante.Corso-Attuale → Corso.Titolo
- Datore(Nome, Indirizzo)
- Corso(Titolo, Area, CF-Docente-Corrente)
  - Corso.CF-Docente-Corrente → Docente.CF
- Docente(CF, Cognome,Tipo)
  - Docente.Tipo ∈ { Collaboratore, Interno }
- Telefono(Numero, CF-Docente)
  - Telefono.CF-Docente → Docente.CF
- Dipendente(CF, Livello, Posizione, Datore)
  - Dipendente.CF → Partecipante.CF
  - Dipendente.Datore → Datore.Nome
- Professionista(CF,Titolo-Prof, Area)
  - Professionista.CF → Partecipante.CF

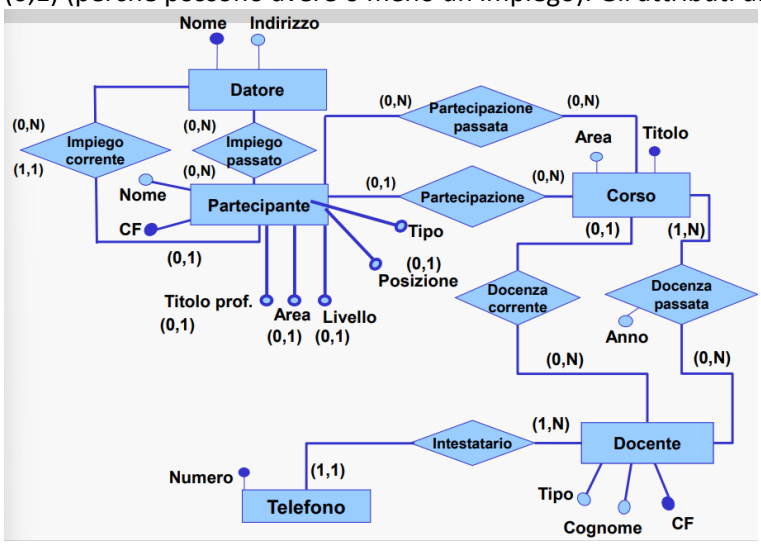
Gli attributi contrassegnati in rosso possono avere valori nulli

- ImpiegoPassato(CF-Partecipante,Nome-Datore)
  - ImpiegoPassato.CF-Partecipante → Partecipante.CF
  - ImpiegoPassato.Nome-Datore → Datore.Nome
- PartecipazionePassata(CF-Partecipante,Titolo-Corso)
  - PartecipazionePassata.CF-Partecipante → Partecipante.CF
  - PartecipazionePassata.Titolo-Corso → Corso.Titolo
- DocenzaPassata(CF-Docente,Titolo)
  - DocenzaPassata.CF-Docente → CF.Docente
  - DocenzaPassata.Titolo → Corso.Titolo

SOLUZIONE CHE RIDUCE IL NUMERO DI TABELLE

Scritto da Gabriel

Per minimizzare le tabelle tolgo *Dipendente/Professionista* e li metto dentro *Partecipante* con cardinalità (0,1) (perché possono avere o meno un impiego). Gli attributi diventano tutti opzionali.



- Partecipante(CF, Nome, Tipo, Tipo {"Dip", "Ref"}
  Corso-Attuale, Titolo-Prof, Area, Posizione, Datore)
  - Tipo="Dipendente" ⇔
    Livello IS NOT NULL ∧ Posizione IS NOT NULL
  - Tipo="Professionista" ⇔
    Area IS NOT NULL ∧ Titolo-Prof IS NOT NULL
  - Partecipante.Corso-Attuale → Corso.Titolo
  - Dipendente.Datore → Datore.Nome
- Datore(Nome, Indirizzo)
- Corso(Titolo, Area, CF-Docente-Corrente)
  - Corso.CF-Docente-Corrente → Docente.CF
- Docente(CF, Cognome,Tipo)
  - Docente.Tipo ∈ { Collaboratore, Interno }
- Telefono(Numero, CF-Docente)
  - Telefono.CF-Docente → Docente.CF

in cui aggiungiamo ad esempio con dei *CHECK* i vincoli seguenti:

- Tipo="Dipendente" ⇔
  Livello IS NOT NULL ∧ Posizione IS NOT NULL ∧ Area IS NULL ∧ Titolo−Prof IS NULL
- Tipo="Professionista" ⇔
  Area IS NOT NULL ∧ Titolo−Prof IS NOT NULL ∧ Livello IS NULL ∧ Posizione IS NULL

facendo così:

```
CHECK(
(Tipo="Dipendente" AND Livello IS NOT NULL AND Posizione IS
NOT NULL AND Area IS NULL AND Titolo-Prof IS NULL) OR
(Tipo="Professionista" AND Area IS NOT NULL AND Titolo-Prof IS
NOT NULL AND Livello IS NULL AND Posizione IS NULL))
```

tenendo tutto uguale a parte lo schema finale:

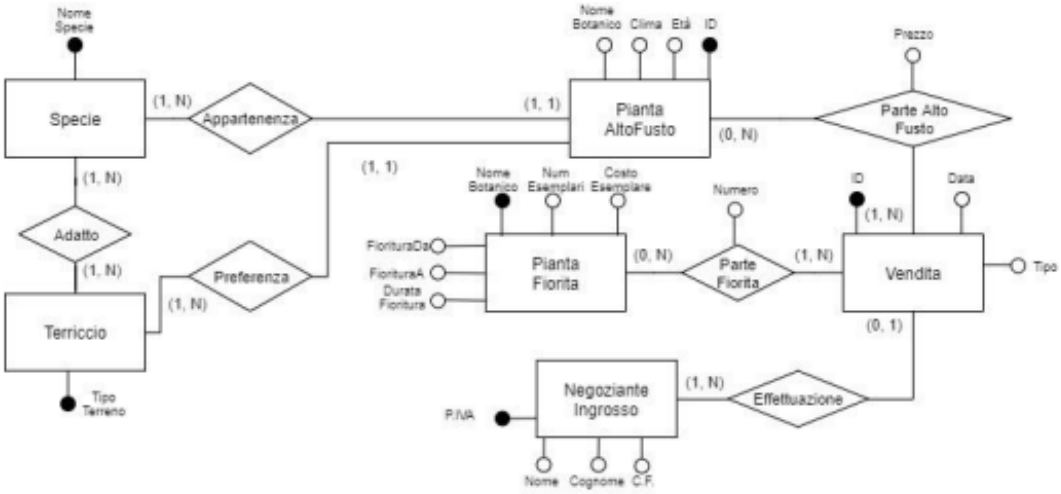
- ImpiegoPassato(CF-Partecipante,Nome-Datore)
  - ImpiegoPassato.CF-Partecipante → Partecipante.CF
  - ImpiegoPassato.Nome-Datore → Datore.Nome
- PartecipazionePassata(CF-Partecipante,Titolo-Corso)
  - PartecipazionePassata.CF-Partecipante → Partecipante.CF
  - PartecipazionePassata.Titolo-Corso → Corso.Titolo
- DocenzaPassata(CF-Docente,Titolo)
  - DocenzaPassata.CF-Docente → CF.Docente
  - DocenzaPassata.Titolo → Corso.Titolo

Note schema logico:

- Lo schema logico elimina *tutto ciò che non si può rappresentare nel concettuale.*
- In una tabella si vanno a sottolineare *solo* le chiavi di quella stessa tabella, non anche le chiavi esterne; ovviamente, se si parla di relazioni (che non hanno chiavi proprie ma solamente le chiavi esterne delle tabelle che collegano), allora si dovranno sottolineare tutte le chiavi (per il motivo indicato).
- Si ricordi, nella rappresentazione a schema logico, di rappresentare tutte le chiavi (se una tabella ha una chiave composta da due o più attributi, si deve considerare nei vincoli di integrità tra chiavi), eventualmente specificando se si hanno chiavi candidate/alternative.
  Il caso di esempio è visibile, ad esempio, nella foto di pag. 80, tra Dipartimento (due campi chiave) e Impiegato.

Esercitazione 4 – Progettazione logica

Relativamente all'esercizio della precedente esercitazione, si riporta in merito a "PianteBellissime" il seguente schema logico, partendo dal successivo schema ristrutturato:



Schema tradotto in modello relazionale:

Terriccio (TipoTerreno)

Specie (NomeSpecie)

Adatto (Specie, Terriccio)

- Adatto.Terriccio -> Terriccio.TipoTerreno
- Adatto.Specie -> Specie.NomeSpecie

PiantaAltoFusto (ID, NomeBotanico, Età, Clima, TipoTerreno, NomeSpecie)

ParteAltoFusto (Vendita, PiantaAltoFusto, Prezzo)

- ParteAltoFusto.Vendita -> Vendita.ID
- ParteAltoFusto.PiantaAltoFusto -> PiantaAltoFusto.ID

Vendita (ID, Data, Tipo, PIVANegoziante) (PIVANegoziante è chiave esterna a Negoziante all'ingrosso partita IVA)

- PIVA può essere NULL

ParteFiorita (Vendita, PiantaFiorita, Numero)

- ParteFiorita.Vendita -> Vendita.ID
- ParteFiorita. PiantaFiorita -> PiantaFiorita.NomeBotanico

PiantaFiorita (NomeBotanico, NumEsemplari, CostoEsemplare, FornituraDa, FornituraA, DurataFioritura)

Negoziante Ingrosso (PIVA, CF, Nome, Cognome)

Esercizio 2

Un'azienda che gestisce gli eventi di uno spazio di fiera vuole progettare una base di dati per la memorizzazione delle informazioni di suo interesse.

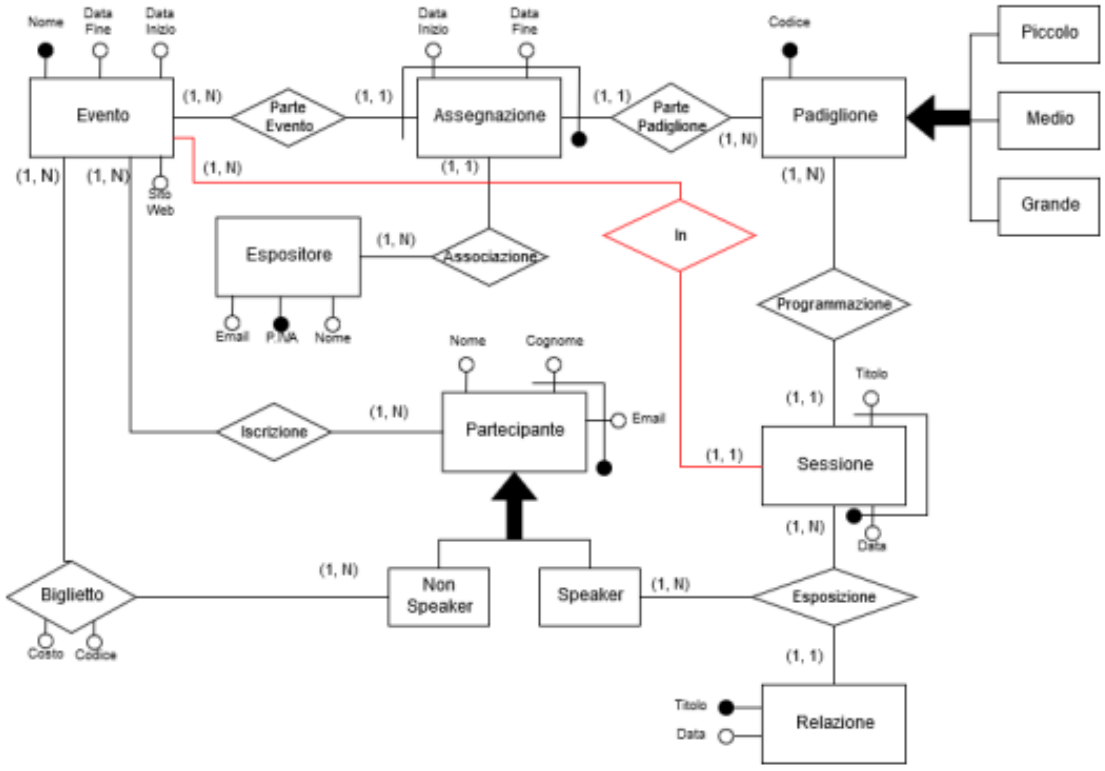
L'azienda ha il compito di gestire tutti gli eventi che sono organizzati nello spazio di fiera. Questi eventi sono caratterizzati da un nome, dalla durata dell'evento e, se disponibile, dal sito web dell'evento.

L'evento si svolge nei padiglioni dello spazio di fiera. È di interesse memorizzare quali padiglioni sono occupati in quali giorni. Inoltre i padiglioni, oltre ad essere identificati mediante un codice alfanumerico, sono classificabili in padiglioni grandi, medi e piccoli. Ogni padiglione ha associato un espositore, di cui interessa il nome, la partita iva, ed il riferimento ad una persona di contatto (email).

Il padiglione ha un programma delle giornate in cui si svolge l'evento che risulta organizzato in sessioni. A ciascuna sessione è associato un titolo, la giornata in cui si svolge e un insieme di speaker, di cui interessa nome, cognome, email, oltre che titolo e durata della relazione che dovranno esporre.

Si noti che uno speaker può effettuare più di una relazione all'interno dello stesso evento. I partecipanti all'evento devono iscriversi fornendo i loro dati anagrafici (nome, cognome, email). Anche gli speaker devono iscriversi all'evento. Tutti i partecipanti tranne gli speaker devono poi pagare un biglietto di iscrizione caratterizzato da un codice identificativo ed un costo.

Diagramma ER:

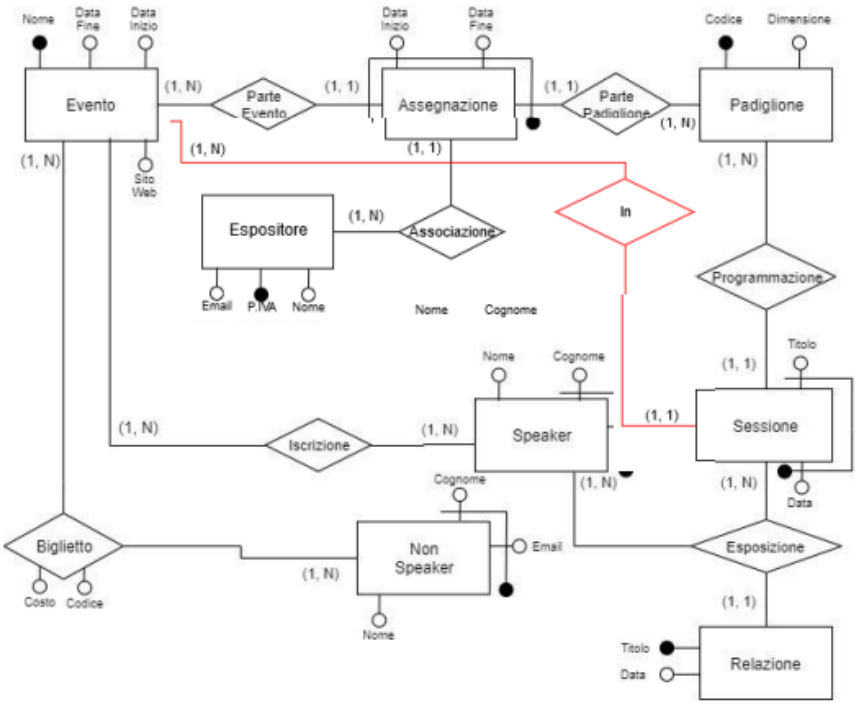


L'uso del colore rosso è per rendere la relazione "In" più visibile a causa delle sovrapposizioni.

Vincoli di Integrità Non Rappresentabili in ER:

- Se uno **Non-Speaker** partecipa ad un **Evento**, allora deve esistere anche un **Biglietto** ad egli associato.
- Il periodo in cui un **Padiglione** viene assegnato ad un evento deve essere compatibile con il periodo in cui l'**Evento** stesso si svolge (in termini di Data Inizio e Data Fine dell'evento).

Diagramma E-R ristrutturato:



Schema tradotto in modello relazionale:

Evento (Nome, DataFine, DataInizio, SitoWeb)

Assegnazione (DataInizio, DataFine, Padiglione, Nome-Evento, PIVA_Espositore,)

- Assegnazione.Padiglione -> Padiglione.Codice
- Assegnazione.PIVA_Espositore -> Espositore.PIVA
- Assegnazione.Nome-Evento -> Evento.Nome

Espositore (PIVA, Nome, E-mail)

Padiglione (Codice, Dimensione)

Sessione (Titolo, Data, Padiglione, Evento)

- Sessione.Padiglione -> Padiglione.Codice
- Sessione.Evento -> Evento.Nome

Note:
- SitoWeb avrebbe cardinalità (0,1) (andrebbe l'asterisco)
- Da parte di Relazione si ha che la chiave verso Sessione dovrebbe comunque essere composta da Titolo e Data; forse viene lasciato l'attributo singolo in virtù del caso relazionale Esposizione che copre già il collegamento.
Io, in ogni caso, lo farei nel modo classico descritto

Speaker (Cognome, Email, Nome)

Iscrizione (Evento, SpeakerCognome, SpeakerEmail)

- Iscrizione.Evento -> Evento.Nome
- Iscrizione.SpeakerCognome -> Speaker.Cognome
- Iscrizione.SpeakerEmail -> Speaker.Email

Biglietto (Evento, Non-SpeakerCognome, Non-SpeakerEmail, Costo, Codice)

- Biglietto.Evento -> Evento.Nome
- Biglietto.(Non-SpeakerCognome, Non-SpeakerEmail) -> Non-Speaker.(Cognome,Email)[2]

Non-Speaker (Cognome, Email, Nome)

Relazione (Titolo, Data, Speaker, Sessione)

Esposizione (SpeakerCognome, SpeakerEmail, SessioneTitolo, SessioneData, Relazione)

- Esposizione.SpeakerCognome -> Speaker.Cognome
- Esposizione.SpeakerEmail -> Speaker.Email
- Esposizione.SessioneTitolo -> Sessione.Titolo
- Esposizione.SessioneData -> Sessione.Data
- Esposizione.Relazione -> Relazione.Titolo

Esercizio 3

Si vuole realizzare una base di dati che gestisca procedimenti sanzionatori nel contesto di rilevazioni statistiche ufficiali di carattere nazionale. Per alcune rilevazioni statistiche ufficiali esiste, infatti, l'obbligo di risposta da parte dei soggetti contattati per la conduzione delle rilevazioni. Qualora il soggetto contattato non risponda al questionario inviatogli, dopo un prefissato intervallo di tempo, ha inizio un procedimento sanzionatorio che consta di due fasi principali: invio della diffida al soggetto non rispondente e, qualora tale soggetto continui ad essere inadempiente (cioè non risponda al questionario), invio della sanzione che il soggetto stesso dovrà pagare. I soggetti possono essere persone fisiche o imprese. Delle persone fisiche interessa memorizzare il codice fiscale, delle imprese il codice fiscale o la partita iva in maniera alternativa. Inoltre è di interesse l'indirizzo cui il soggetto è contattabile. Si noti che le imprese possono prevedere delle unità locali, ovvero l'impresa si articola secondo una struttura che consiste di un'impresa centrale ed eventualmente di un insieme di imprese "periferiche". Un procedimento viene avviato in relazione alla non-risposta ad una specifica edizione di un'indagine. Ogni indagine è caratterizzata da un nome (es. Forze di lavoro), da una frequenza con cui le sue edizioni occorrono (es. trimestrale) e dalle specifiche edizioni che sono occorse (es. primo trimestre 2011).

Le edizioni, che hanno un codice univoco nell'ambito dell'indagine in cui sono svolte, hanno una data di inizio ed una data di fine che caratterizzano l'inizio e la fine della rilevazione sul campo dei dati oggetto dell'indagine. Nell'ambito di un procedimento è prodotto un insieme di documenti che costituisce il fascicolo del procedimento. Un fascicolo ha un codice che lo identifica nell'ambito del procedimento a cui è legato. I documenti, che dispongono di un ID univoco nell'ambito del fascicolo in cui sono redatti, sono rappresentati da un nome, un tipo, una data di produzione e dal path relativo al file cui sono associati. Del procedimento, oltre alle informazioni necessarie a desumere il suo avanzamento, interessa memorizzare la data di inizio e, qualora sia stato archiviato, l'esito della archiviazione (ad esempio archiviato perché il soggetto ha risposto) e la data di archiviazione.

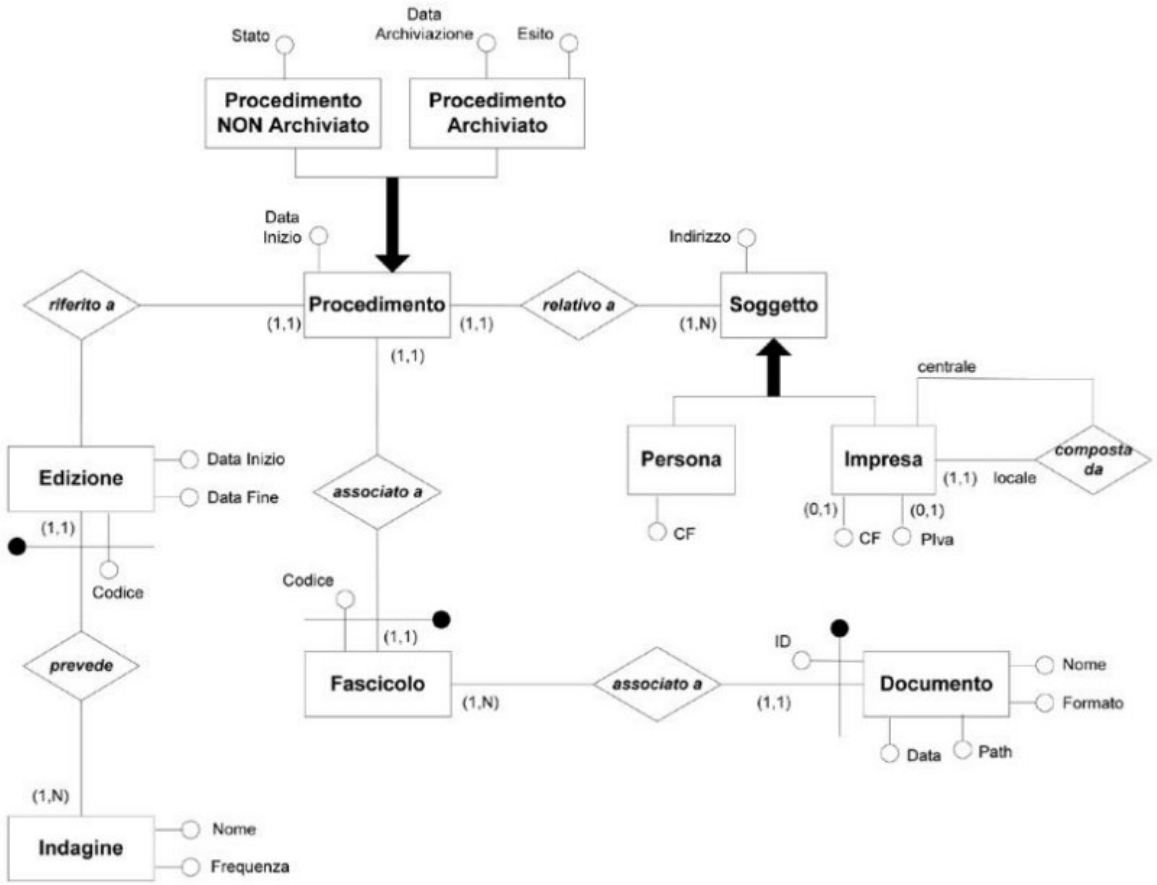
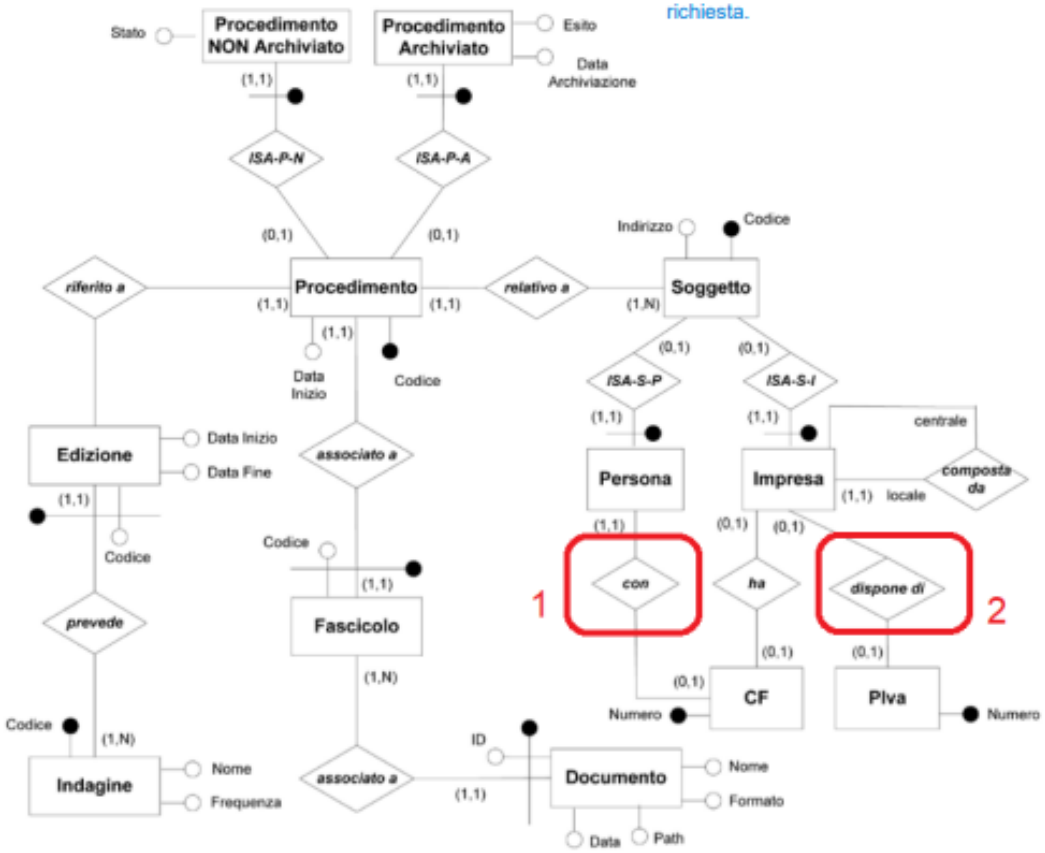


Diagramma E-R ristrutturato

"Informazioni necessarie a desumere l'avanzamento del procedimento" sembra sia l'unico motivo razionale per il quale è stato inserito l'attributo "Stato"; non sembra, infatti, comparire nel testo come esplicita richiesta.



Le scelte effettuate nella fase di ristrutturatone hanno avuto come obiettivo quello di evitare la presenza di valori NULL nella base di dati. Questo ha comportato una traduzione "ridondante" delle generalizzazioni, in cui sia l'entità padre che quelle figlie sono state mantenute nello schema, e una traduzione degli attributi con cardinalità (0,1) in nuove entità specifiche. Note che quest'ultimo aspetto ha comportato la modifica di uno dei vincoli esterni (quello identificato nel rettangolo num. 2) e l'aggiunta di un nuovo vincolo eterno (quello identificato nel rettangolo num. 1), rispetto a quelli definiti in fase di Progettazione Concettuale. Inoltre, dato che, per ogni entità, è necessario individuare principale, è stato introdotto un codice, i cui valori sono speciali ed hanno l'unico scopo di identificare le istanze dell'entità, per tutte quelle entità che naturalmente non ne dispongono.

Schema Logico:

Procedimento(Codice, Soggetto, Edizione, Indagine, DataInizio)

- Procedimento.Soggetto --> Soggetto.Codice)
- Procedimento.(Edizione,Indagine) --> Edizione.(Codice, Indagine)

ProcedimentoNonArchiviato(Codice,Stato)

- ProcedimentoNonArchiviato.Codice -->Procedimento.Codice)

ProcedimentoArchiviato(Codice,Esito,DataArchiviazione)

- ProcedimentoArchiviato.Codice -->Procedimento.Codice

Indagine(Codice,Nome,Frequenza)

Edizione(Codice,Indagine,DataInizio,DataFine)

- Edizione.Indagine --> Indagine.Codice

Fascicolo(Codice,Procedimento)

- Fascicolo.Procedimento --> Procedimento.Codice

Documento(ID,Fascicolo,Data,Path,Nome,Formato)

- Documento.Fascicolo -->Fascicolo.Codice

Soggetto(Codice,Indirizzo)

Persona(Codice,CF)

- Persona.Codice -->Soggetto(Codice)
- Persona.CF -->CF.Numero

Impresa (Codice,CodiceCentrale)

Impresa.Codice -->Soggetto.Codice

Impresa.CodiceCentrale --> Impresa.Codice

CF(Numero)

PIva(Numero)

ha(CF,Impresa)

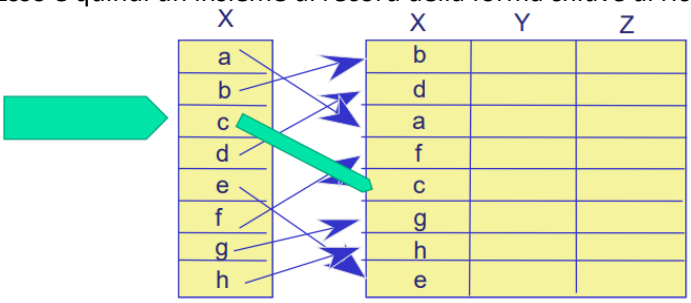
- ha.CF --> CF.Numero
- ha.Impresa --> Impresa.Codice

ImpresadisponeDi(PIva, Impresa)

- disponeDi.PIva --> PIva(Numero)
- disponeDi.Impresa --> Impresa.Codice

Gestione degli indici

Gli *indici* favoriscono l'accesso alla base di dati rispetto al valore di uno o più attributi, velocizzando l'accesso alle tuple. Viene quindi definita una "chiave di ricerca", rispetto ad 1+ attributi della relazione.
Fondamentalmente rappresentano dei puntatori alle tuple accedute in tempo costante O(1).
Esso è quindi un insieme di record della forma *chiave di ricerca – pointer(puntatore)*:



Parliamo quindi della struttura seguente:

Chiave Ricerca	pointer

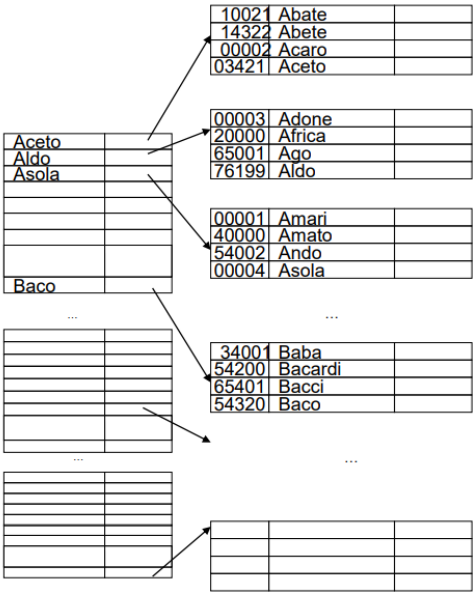
avendo l'indice che ha una dimensione molto più piccola della tabella-relazione associata.
Pro: Quando si cercano dei record, offrono dei sostanziali benefici
Contro: Creare indici comporta un overhead a causa dell'aggiornamento a catena se ce ne stanno troppi

Le *metriche di valutazione* sono:

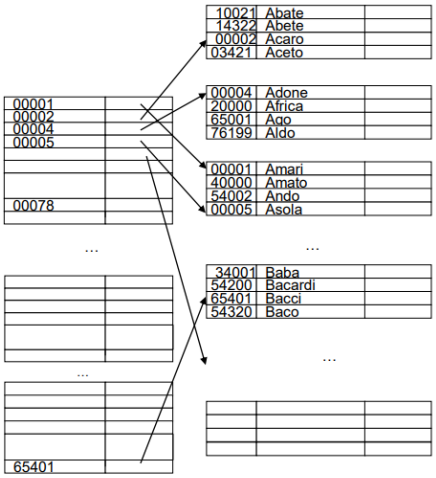
1) tempo di accesso ai valori/attributi entro un certo intervallo
2) tempo di inserimento
3) tempo di cancellazione
4) occupazione di spazio

Esistono due tipi di indici:

1) *primario*, le tuple sono ordinate sulla base delle chiavi primarie (tipiche chiavi di ricerca, avendo al più un singolo indice primario; esempio qui a destra)



2) *secondario*, le tuple sono mantenute in ordine diverso rispetto alla chiave di ricerca; essi possono essere molteplici (esempio qui sotto)

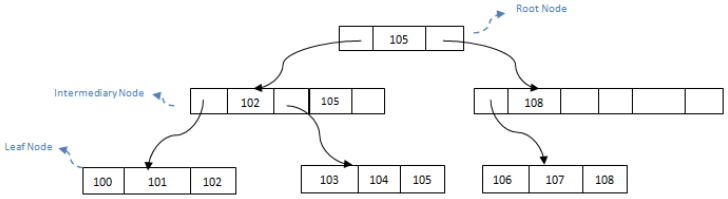


Iterare è efficiente su un indice primario; infatti i blocchi sono letti sequenzialmente. Tale osservazione vale anche per quelli secondari, leggendoli più volte. Per la creazione di un indice si usano i *BST/Binary Search Trees/alberi binari di ricerca*, tale da avere nel sottoalbero sinistro solo etichette minori rispetto al valore

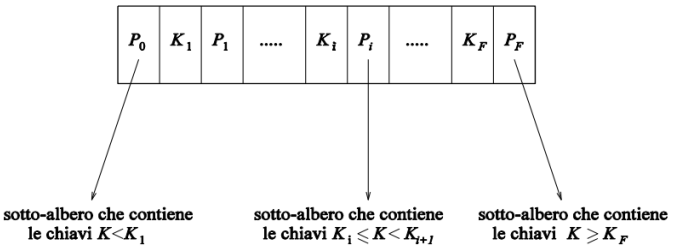
attuale, mentre in quello destro solo quelle maggiori. “Trovare” un nodo costa quanto la sua profondità; per minimizzare il costo medio si cerca di bilanciarlo.

Le strutture usate nelle basi di dati sono i *B+ Trees*, che sono dei BST bilanciati e i valori (puntatori ai dati) sono solo nelle foglie; i nodi hanno più di un valore e le foglie sono collegate.

STUDENT		
STUDENT_ID	STUDENT_NAME	ADDRESS
100	Joseph	Alaiedon Township
101	Allen	Fraser Township
102	Chris	Clinton Township
103	Patty	Troy
104	Jack	Fraser Township
105	Jessica	Clinton Township
106	James	Troy
107	Antony	Alaiedon Township
108	Jacob	Troy

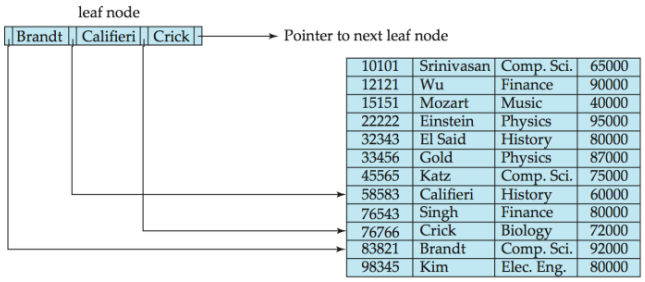


Nei nodi non foglia abbiamo questa struttura:



con tutti i valori K_i ordinati come $K_1 < K_2 <.... K_{n-1}$ facendo attenzione non vi siano duplicati.
Fondamentalmente i puntatori ragionano sequenzialmente, partendo da sinistra e ragionando per foglie.
Formalmente:

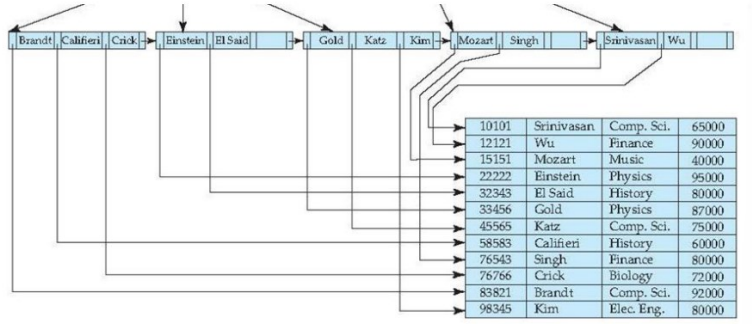
- $\forall$ i < F–1, il puntatore P_i si riferisce alla tuple con valore K_i per la chiave di ricerca
- P_F punta alla successiva foglia nell’ordine della chiave di ricerca



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

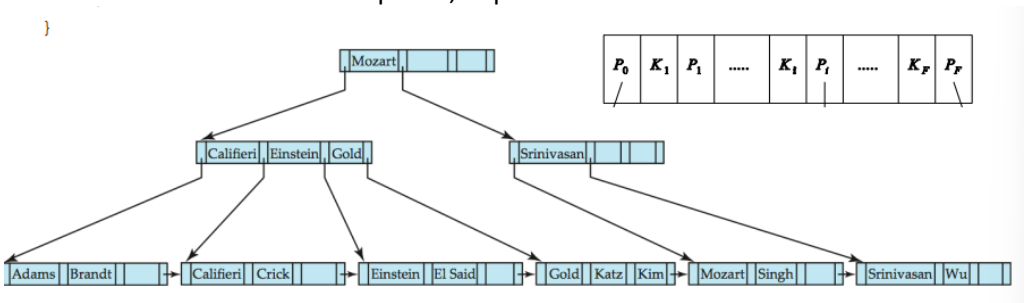
Similmente possiamo considerare solamente i valori minori di una certa foglia. Formalmente:

Se la foglia F_i e seguita dalla foglia F_j tutti i valori nella foglia F_i sono inferiori a tutti quelli della foglia F_j



Quindi per i B+ Trees:
Pro: Riorganizzazione con cambiamenti "locali" a seguito di inserimenti/cancellazioni e non si deve riorganizzare per mantenere le performance a seguito di queste operazioni.
Contro: Occupazione extra di spazio per le operazioni di inserimento/cancellazione e ottimizzazione solo parziale delle query con "=".

Assumendo un record senza duplicati, l'operazione di *ricerca di un record* avviene nel seguente modo:



Per trovare un valore devo considerare che, per N valori, la profondità è <= $\log_{[F/2]}(N)$, in cui F rappresenta il numero massimo di valori in un nodo. Un nodo tipicamente corrisponde a 4 KB; quindi se ci sono circa 40 bytes per index entry, F sarà circa 100 index entry per nodo.
Similmente per N=1,000,000 ed F=100 → $\log_{50}(1{,}000{,}000) = 4$ nodi

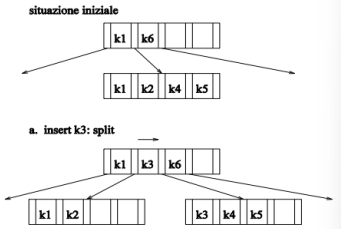
Vediamo l'operazione di *inserimento nei B+ Trees* seguendo i passi qui presenti:

1. Trova la foglia dove la chiave di ricerca dovrebbe apparire
2. Se il valore è presente, aggiorna il puntatore.
3. Se il valore non è presente, allora
   A. Se c'è spazio, aggiungi la coppia (chiave, puntatore al record)
   B. Se non c'è spazio, occorre creare un nodo

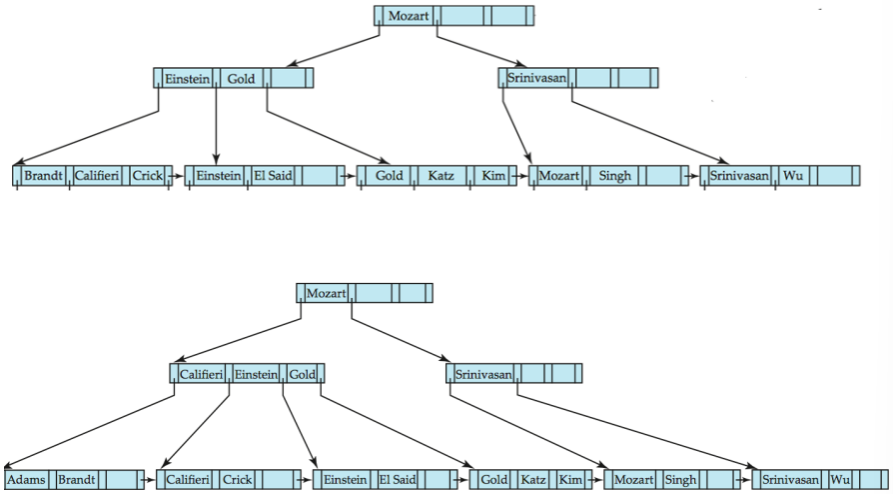
La descrizione completa è la seguente: di fatto si inserisce un nodo in mezzo alle foglie, se non ci fosse spazio. A seguito di questo, si cambia anche il nodo padre e, a seconda della posizione, (maggiore o minore rispetto al nodo di riferimento), si decide la collocazione finale.
Formalmente (qui a destra):

Si vuole aggiungere una coppia (k,p) dove p è puntatore alla tupla di dato.
Dividere la foglia con F coppie (valore chiave ricerca, puntatore):

A. Tenere $\lceil F/2 \rceil$ nel nodo originale, e il resto in un nuovo nodo incluso la coppia (k,p).
B. Modificare il nodo padre come da figura.
C. Se il nodo padre è pieno, ripetere la procedura per il nodo padre

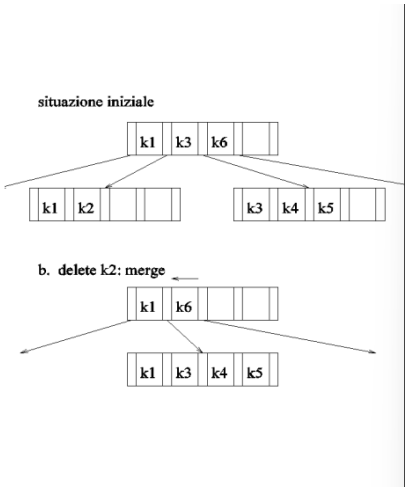


Vediamo il successivo esempio dell'inserimento del nodo "Adams", minore di tutti i nodi e va tutta a sinistra. Si prende la foglia, la divide in due sottofoglie e si mette "Adams" a sinistra. Per i valori più piccoli di Califieri si va a sx, per quelli più piccoli di Einstein si torna a prima e per valori più piccoli di Crick si va subito nella posizione successiva. Quindi completamente:



L'operazione di *cancellazione* è la stessa cosa al contrario, facendo attenzione a mantenere l'albero bilanciato. Se più di metà del nodo è libero, ne tolgo uno, viene messo per merge in un'altra foglia e cambio il puntatore mettendo k3 da sopra, al fine di mantenere la consistenza.

- Rimuovere la coppia (valore chiave ricerca, puntatore) dalla foglia.
- Se la foglia F ha meno di metà coppie utilizzate:
  - Se una foglia adiacente G ha sufficiente spazio per ospitare le entry di F:
    - Inserisci tutte le entry di F in G
    - Rimuovi la foglia G
    - Rimuovi la coppia (k,p) dal nodo padre dove p è il puntatore a G
    - Aggiorna i puntatori nel nodo padre
  - Se nessuna foglia adiacente ha sufficiente spazio
    - Ridistruisci le coppie con le foglie adiacenti
    - Aggiorna i puntatori nel nodo padre

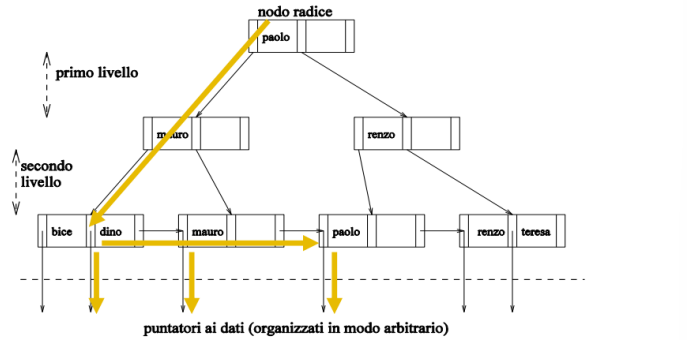


Nelle query su intervalli, si naviga l'indice come si vede da qui:

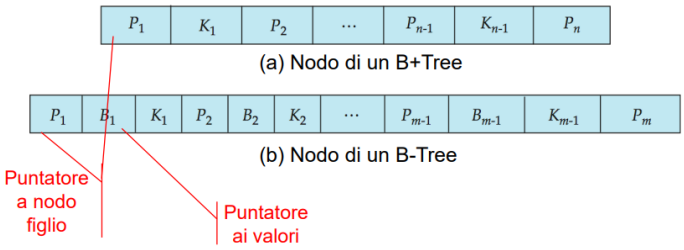
Supponiamo di voler fare

```
select * from Persone where Nome >= 'Dino' and Nome <='Paolo'
```

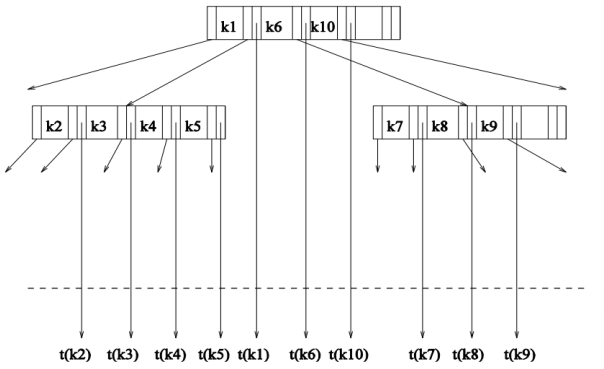
L'indice verrà navigato:



Gli indici dei B-Trees, simili a B+ Trees quindi dei nodi interni contengono valori e le foglie non sono connesse.

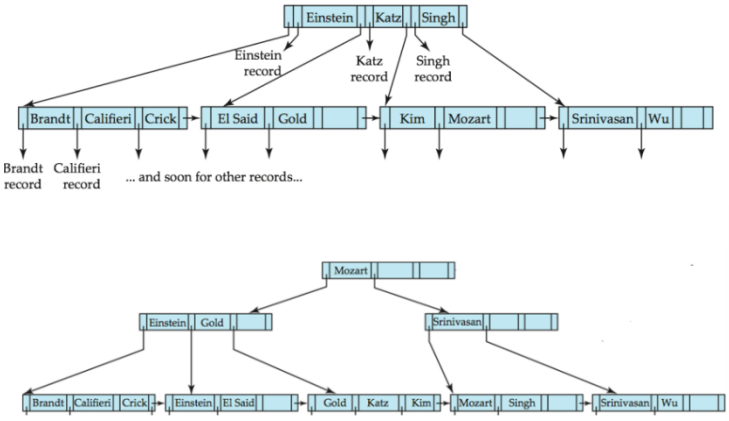


La profondità tende quindi ad essere più lunga:



Di solito nella pratica quindi i B- Trees:
Pro: Tende ad usare meno nodi dei B+ Trees ed è possibile trovare coppie chiave/valore prima di raggiungere una foglia, accumulando i valori utili il più in alto possibile compatibilmente con il loro uso
Contro: Non si hanno vantaggi nelle query per intervallo ed inserimenti/cancellazioni sono più complessi, richiedendo una manutenzione più accurata

Sugli stessi dati quindi si fa come si vede di fianco:



Su attributi singoli quindi le query possono essere ottimizzate, scorrendo la base di dati solo al verificarsi di una certa condizione:

Esempio:

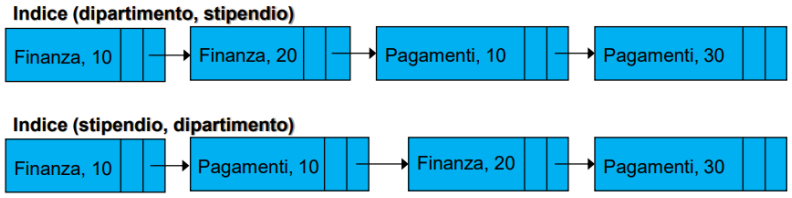
```
select ID
from Impiegato
where dipartimento = "Finanza" and stipendio = 80000
```

Possible strategie se ci sono indici su attributi **singoli**:

1. Usa indice su dipartimento per trovare gli impiegati del Dipartimento "Finanza"; scorri tutti, estraendo quelli con stipendio di € 80000
2. Usa indice su stipendio per trovare gli impiegati con stipendio di € 80000; scorri tutti, estraendo quelli del Dipartimento Finanza
3. Usa sia l'indice su Dipartimento che quello su stipendio. Prendi l'intersezione di entrambi

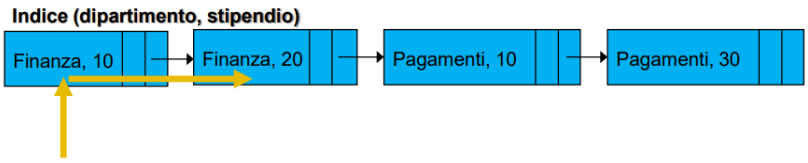
Posso anche usare *chiavi di ricerca su due attributi*; questo risulta efficiente, ma l'ordine delle chiavi conta:

- **(dipartimento,stipendo)** è diverso da **(stipendio,dipartimento)**
- Ordine lessicografico: (a1, a2) < (b1, b2) se:
  - a1 < b1, o
  - a1=b1 e a2 < b2



Un indice su una chiave di ricerca multipla *(dipartimento, stipendio)* può essere efficiente nei casi:

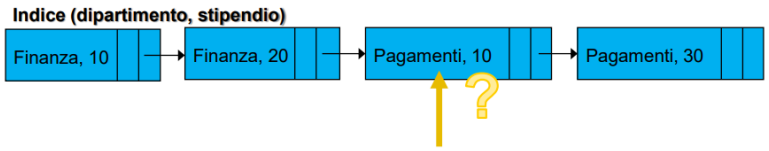
- *where dipartimento = 'Finanza' and stipendio=10*
- *where dipartimento = 'Finanza' and stipendio<=20*



Un indice su una chiave di ricerca multipla *(dipartimento, stipendio)* NON è efficiente nel caso:

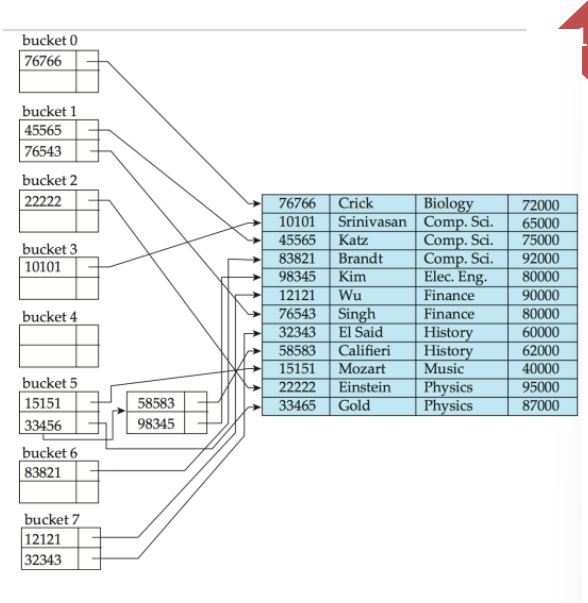
- *where dipartimento > 'Finanza' and stipendio=80*

- È possibile estrarre tutti gli impiegati di dipartimenti "maggiori di" finanza efficientemente
- Non è possibile estrarre efficientemente quelli con stipendio di 80



Non essendo ordinati secondo il criterio della query, non è particolarmente efficiente.

Si basa sul principio della *hash*, restituendo sulla base di un certo valore *n* restiuisce una stringa *s* della stessa lunghezza. Non è necessariamente invertibile, tornando alla stringa di partenza. Vediamo l'esempio della costruzione di un indice hash sui numeri di matricola, mappando ogni numero di matricola ad un certo numero come si vede qui a fianco.
Si basa quindi sul concetto dei "bucket", dunque gli insiemi di record sono associate sulla base di una certa chiave K.



La hash function viene usata per trovare il record di accesso, inserimento e cancellazione. Dunque, dati due valori K, K' è possibile che h(K)=h(K'), avendo possibili collisioni e cercando il valore K iterando nel bucket.
Essi sono sempre secondari e possono avere indice primario separato oppure avere sia hash che B- Tree.
Si possono avere *collisioni*, le quali devono essere gestite. Dipende anche dalla qualità stessa della funzione, sulla base di quanti valori gestisce. Come confronto quindi tra indici hash e B+-Trees:

Vantaggi

- Ricerca di un valore ha costo ca. 1
  - B+Tree è basso ma non 1
- Insertion ha costo ca. 1
- Cancellazione ha costo di ca. 1
  - Occorre inserire/rimuovere la entry dal bucket, creando/eliminando bucket di Collisione

Svantaggi

- Può essere solo usato per WHERE con atomi `variable=valore`
- Non può essere usato per velocizzare operazioni di ORDER BY.
  - Non è possible cercare la prossima entry in ordine

Possono essere definiti nel seguente modo nei vari DBMS:

- `create [unique] index` *IndexName* `on` *TableName*(*AttributeList*) [`using` *method*]
  - `unique` è opzionale: forza attributi degli indici ad avere valori unici
  - `using` *method* può essere `btree` o `hash`
  - *method* è opzionale: il default è `btree`.
  - altri *method* esistono ma non visti
- `drop index` *IndexName*

L'esempio in SQL:

Esempio: `create index DipStipendio on Impiegati(Dipartimento,Stipendio)`

PostgreSQL crea automaticamente indici hash su un singolo attributo, mentre in automatico viene creato un indice "unique" B+ Tree sulla chiave primaria di ogni tabella. La clausola *unique* è solo nei B+ Tree.

Un esempio pratico con un esercizio. La domanda che ci poniamo è: quale tra i due scegliere e come viene scelto l'ordine tra (A,B) e (B,A)?

Domanda 3 (2 Punti)

Si consideri le relazioni R(A, B, C, D) e la seguente query

SELECT MIN(A) FROM R WHERE B=10

Quale dei seguenti indici garantisce l'efficienza massima?

1. Indice Hash sulla coppia (B,A)
2. Indice Hash sulla coppia (A,B)
3. Indice B-TREE sulla coppia (A,B)
4. Indice B-TREE sulla coppia (B,A)

Visto che l'operazione non coinvolge una semplice uguaglianza si può escludere l'hash.
In un indice B-Tree se la coppia è (B,A) significa che l'ordinamento prioritizza B e poi A; quindi, i pointer a tutti i record con uguale valore di B saranno adiacenti. A quel punto per trovare il minimo basterà mettersi a cercare la foglia con quel valore di B e recuperare il primo/i record. In caso contrario (se A avesse avuto priorità) lo scorrimento da sinistra alla ricerca del valore più piccolo con B uguale proprio a 10 sarebbe potenzialmente più lungo.

Un buon confronto proviene da un file di riassunto di Mega del 2021:

B+ Tree	mediamente bene per = consigliati per intervalli < o > bene per cancellazione o inserimento
B- Tree	mediamente bene per = male per cancellazione o inserimento mediocri in generale
Hash	migliori per uguaglianze male ORDER BY

uguaglianza	Hash
intervallo	B+ Tree
uguaglianza e intervallo	B+ Tree
cancellazione/inserimento	B+ Tree
ORDER BY	B+ Tree

Accesso a PostgreSQL da software

Si prende l'esempio di una stampa del n. di tuple di una tabella T:

```
#include <iostream.h>
#include <libpq-fe.h>

void do_exit(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int main()
{
    PGconn *conn = PQconnectdb("dbname=testdb");                    //Connessione al database

    if (PQstatus(conn) == CONNECTION_BAD)                           //Se non è possibile connettersi
    {
        cerr << "Connection to database failed:" << PQerrorMessage(conn);
        do_exit(conn);
    }

    PGresult *res = PQexec(conn, "SELECT COUNT(*) FROM T");         //Esegui una query sulla connessione

    if (PQresultStatus(res) != PGRES_TUPLES_OK)                     //Se ci sono stati problemi
    {
        cerr << "Non è stato restituito un risultato" << PQerrorMessage(conn);
        PQclear(res);
        do_exit(conn);
    }

    cout << PQgetvalue(res, 0, 0);                                  //Estra il risultato e stampa su stdout

    PQclear(res);                                                   //Chiudi la connessione
    PQfinish(conn);

    return 0;
}
```

Da qui poi diamo l'apertura della connessione al database con *PQconnectdb*, partendo da una stringa di parametri. Esso ha la seguente struttura, partendo da *PGconn *PQconnectdb(const char *conninfo);:*

- Stringa in due formati:
  - `"host=… port=… dbname=… user=… password=…"`
    - Questi (ed altri) pararametri sono opzionali: se non vengono specificati, su usano i valori di default.
    - La lista completa è nella Sezione 33.1.2 della documentazione
  - `"postgresql://user:password@host:port/dbname`
    - Questi (ed altri) pararametri sono opzionali: se non specificati, si usano valori di default.
    - Esempi:
      - `postgresql://localhost:5432`
      - `postgresql://localhost/testdb`
      - `postgresql://user@localhost`
      - `postgresql://user:secret@localhost/testdb`

```
if (PQstatus(conn) == CONNECTION_BAD)
```

Esaminando rispetto al codice di prima, *PQstatus(conn)* restituisce lo stato della connessione tramite due costanti, *CONNECTION_OK* oppure *CONNECTION_BAD*.

Similmente abbiamo la restituzione di un certo messaggio di errore, generato attraverso *PQerrormessage(conn)*:

```
cerr << "Connection to database failed:" << PQerrorMessage(conn);
do_exit(conn);
```

La funzione *PQexec* invia un comando *command* al database tramite la connessione *conn*. Ad esempio:

```
PGresult *res = PQexec(conn, "SELECT COUNT(*) FROM T");
```

In generale il puntatore di ritorno:

- è nullo, dunque ci sono problemi importanti (mancanza di memoria, connessione al server, ecc.)
- non è nullo e può essere con diversi valori di ritorno, ad esempio la resttuzione di tuple con *PGRES_TUPLES_OK*, il fatto che NON siano state restituite tuple (a seguito di inserimenti, cancellazioni, ecc.), con *PGRES_COMMAND_OK*, oppure qualche altro bug a livello client.

Similmente vediamo l'esempio completo (prima di questo blocco di codice si ha *PGconn* che apre la connessione e *PQclear/PQfinish* che terminano la connessione chiudendola:

```
PGresult *res = PQexec(conn, "SELECT COUNT(*) FROM T");                    //Esegui una query sulla connessione

if (PQresultStatus(res) != PGRES_TUPLES_OK)                                //Se ci sono stati problemi
{
    cerr << "Non è stato restituito un risultato" << PQerrorMessage(conn);
    PQclear(res);
    do_exit(conn);
}
```

L'analisi del risultato viene poi esaminato tramite array di caratteri tramite:

```
char *PQgetvalue(const PGresult *res,
                 int row_number,
                 int column_number);
```

Essa restituisce, per esempio con idea implementativa di *PQgetvalue(res, i, j)* che restituisce il valore del j-esimo attributo per la i-esima tupla (la prima tupla è con indici i=j=0). Esempio pratico :

```
PGresult *res = PQexec(conn, "SELECT COUNT(*) FROM T");
char *value=PQgetvalue(res, 0, 0);
```

In generale il puntatore di ritorno:

- anche se l'attributo è numerico, rimane sempre rappresentato come stringa
- è una stringa vuota, se il risultato è stringa vuota o *NULL*. Per esempio:
  - La funzione `int PQgetisnull(PGresult *res, int row_number,int column_number)` permette di distinguere: 1 se null, 0 se non null
  - Per esempio se **`PQgetisnull`**`(res,0,0)` =1 allora il primo attributo della prima tupla ha un valore nullo

Abbiamo poi *PQclear(res)* che elimina il risultato dalla memoria e *PQfinish(conn)* che chiude la connessione:

```
cout << PQgetvalue(res, 0, 0);

PQclear(res);
PQfinish(conn);

return 0;
```

Similmente abbiamo i metodi *PQntuples(res)* che ottiene il numero di tuple del risultato ed un'estrazione ciclica dei valori per estrarli con l'accesso ad un i/j-esima posizione.
Esempio completo:

```
int numTuple=PQntuples(res);

    for(int i=0;i<numTuple;i++)
    {
         //Stampo i valori dei primi due attributi di ogni riga
        cout << PQgetvalue(res, i, 0) << "\t" << PQgetvalue(res, i, 1)
        cout << "\n";
    }
```

Siccome non saprò il numero di righe al momento della query, ma tipicamente conosco il numero di colonne, si ritiene utile questa applicazione.
Immaginiamo ad esempio di dover stampare la matricola e la data di nascita degli studenti ad esempio su questi dati:

Matricola	Cognome	Nome	Data di nascita
6554	Rossi	Mario	05/12/1978
…	…	…	…

Se volessimo l'attributo relativo all'i-esima colonna si usa *PQfname(res,i)*, come si vede prendendo in base al risultato di una query il nome di un certo attributo (nota: non si ha garanzia dell'ordine di tuple e/o colonne):

```
int numTuple=PQntuples(res);

cout << PQfname(res, 0) << "\t" << PQfname(res, 1);
cout << endl;

for(int i=0;i<numTuple;i++)
{
    cout << PQgetvalue(res, i, 0) << "\t" << PQgetvalue(res, i, 1)
    cout << endl;
}
```

Ora diamo un esempio di un codice C che stampa il risultato di una query generica.
Partiamo con l'inserimento della query da parte dell'utente:

```
char query[40];

cout >> "Inserire la query:"
cin << query;
```

prendendo il numero degli attributi con *PQnfields(res)* che restituisce il numero delle colonne/attributi della relazione, stampando i nomi di ciascuno di questi separati tra loro da tabulazione e, per ogni riga, si stampano i valori assegnati ad ogni attributo (anche qui tabulati):

```
int numTuple=PQntuples(res);
int numAttributi=PQnfields(res);

for(int i=0;i<numAttributi)
    cout << PQfname(res, i) << "\t";
cout << endl;

for(int i=0;i<numTuple;i++)
{
    for (int j=0;i<numAttributi;j++)
        cout << PQgetvalue(res, i, j) << "\t";
    cout << endl;
}
```

Prendiamo poi il caso della stampa di tutte le informazioni sugli studenti:

```
int numTuple=PQntuples(res);

cout << "Matricola" << "\t" << "Cognome" << "Nome" << "\t" << "Data di nascita";
cout << endl;

for(int i=0;i<numTuple;i++)
{
    cout << PQgetvalue(res, i, 0) << "\t" << PQgetvalue(res, i, 1) << PQgetvalue(res, i, 2) << "\t" << PQgetvalue(res, i, 3)
    cout << endl;
}
```

In generale la soluzione è dipendente dall'ordine di definizione degli attributi e si potrebbe non garantire il rispetto dell'ordine vero (come discusso sopra), dunque anche in caso di ristrutturazione si deve mantenere un design robusto. Meglio riferirsi alle colonne di un attributo, partendo dalla funzione *int PQfnumber(res, column_name),* restituendo la posizione della colonna *column_name* nel risultato *res*.

```
int numTuple=PQntuples(res);

cout << "Matricola" << "\t" << "Cognome" << "Nome" << "\t" << "Data di nascita" << endl;

for(int i=0;i<numTuple;i++)
{
    cout << PQgetvalue(res, i, Pqfnumber(res,"Matricola")) << "\t";
    cout << PQgetvalue(res, i, Pqfnumber(res,"Cognome")) << "\t";
    cout << PQgetvalue(res, i, Pqfnumber(res,"Nome")) << "\t";
    cout << PQgetvalue(res, i, Pqfnumber(res,"Data_di_Nascita"));
    cout << endl;
}
```

Descriviamo ora alcune funzioni definite dal prof come utili, come ad esempio la *sprintf(str, format, … (additional arguments)* memorizzando nell'array di caratteri *str*.

Poi definiamo gli altri parametri ed un esempio di riferimento:

- Format contiene la stringa che rappresenta il template e contiene i tag di form formattazione, tra cui:
  - `%d` per gli interi,
  - `%f` per i float,
  - `%s` per le stringhe.
- Esempio: `sprintf(query, "Il cognome è %s e la matricola è %d",cognome,numMatr);`
  1. Il valore di cognome (var. stringa) viene messo al posto del primo tag %s
  2. Il valore di matricola (var. intera) viene messo al posto del secondo tag %d
  3. Il risultato viene assegnato a query (puntatore a char).

Essa risulta comoda per esempio per costruire una stringa partendo da un template + una serie di variabili, per esempio a noi *stampa delle informazioni di studenti con un dato cognome e matricola maggiore di un valore*. Si costruisce quindi la query dinamicamente in base all'input fornito.

```
sprintf(query, "SELECT * FROM STUDENTI WHERE COGNOME=\'%s\' AND MATRICOLA>%d",cognome,numMatr);
```

Immaginiamo ad esempio di passare dei parametri da linea di comando, ad esempio con:

```
C:> nomeprog pippo 12 pluto 25
```

Nome Programma — Argomenti

Possiamo quindi pensare di scrivere il main nella maniera classica completa:

```
int main (int argc,char* argv[])
```

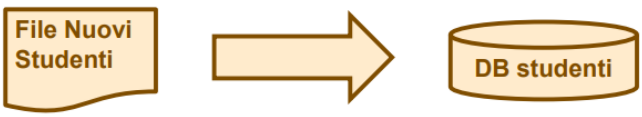
I parametri del main sono:

- *argc*, contenente il numero di stringhe inserite dall'utente
- *argv*, array che contiene le stringhe inserite a linea di comando, puntandole per carattere

L'esempio seguente prende i dati in formato carattere oppure prende un intero da una stringa come con la funzione *atoi(stringa)*.

```
char query[80];
char* cognome=argv[1];
int numMatr=atoi(argv[2]);
```

Immaginiamo di voler inserire nel codice la popolazione del DB partendo da un file di testo:




```
311 Cogn1 Nome1 13/3/2000
556 Cogn2 Nome2 17/6/2000
1056 Cogn3 Nome3 22/12/1999
.....
```

Matricola	Cognome	Nome	Data di nascita
...	...	...	...

Vediamo con *fstream*, le operazioni di apertura file in lettura/scrittura.

- Per aprire un file `n` in lettura: `ifstream infile(n);`
- Per aprire un file `n` in scrittura: `ofstream outfile(n);`

Possiamo quindi usare *cin/cout* per *ifstream* ed *ostream*:

- Si puo' usare ifstream come cin: `infile >> variabile;`
- Si puo' usare ofstream come cout: `outfile << variabile;`

La lettura arriva fino alla *end-of-token* con l'uso di *infile.close()* per chiudere il file e continuare a leggere finché *infile >> variabile* non restituisce *false* (nel qual caso esco e chiudo il file).
Questo ultimo comando legge una porzione di linea di testo fino al prossimo spazio.

Per leggere una intera riga: *getStream (infile, variabile)*

```
{
    […]
    char buffer[100]
    ifstream file("mioFile");

    while(file >> buffer)
          cout << buffer;

    file.close();

    […]
}
```

Esempio di lettura del contenuto di un file e scrittura dello stesso su console (senza nessuno spazio tra le righe):
Nota: l'ordine che seguiamo di lettura rispetto all'esempio di prima: *matricola – cognome – nome – data di nascita*

Vediamo l'esempio di aggiunta di studenti da file (assumendo che i cognomi non abbiano spazio). Leggo una ad una ogni riga del file secondo l'ordine descritto:

```
char matricola[10], char cognome[50], char nome[50], char dataNascita[10];
PGconn *conn = PQconnectdb("dbname=testdb");
[…]

ifstream infile(argv[1]);

while (infile >> matricola >> cognome >> nome >> dataNascita)
{
    if (aggiungiStudente(conn, matricola, cognome, nome, dataNascita)!=PGRES_COMMAND_OK)
        cerr << "Impossibile aggiungere la matricola " << matricola << ": " << PQerrorMessage(conn) << endl;
}
PQfinish(conn);
infile.close();
```

Il risultato della query segnala se l'inserimento è avvenuto, deallocando l'handler precedente e dando conferma tramite risultato dell'esito dell'inserimento:

```
#include <iostream.h>
#include <libpq-fe.h>
#include <fstream.h>

ExecStatusType aggiungiStudente(Pgconn *conn, char *matricola, char *cognomenome, char *nome, char *dataNascita
{
    char query[60];
    sprintf(query,
        "INSERT INTO STUDENTI(MATRICOLA,COGNOME,NOME,DATANASCITA) VALUES (%d, \'%s\', \'%s\', \'%s\')",
        matricola, cognome, nome, dataNascita);
    PGresult *res = PQexec(conn, query);
    ExecStatusType risultato=PQexec(conn, query);
    PQclear(res);
    return(risultato);
}
```

Quando ho fatto tutto, finisco tutto ed esco:

```
PQfinish(conn);
infile.close();
```

Normalmente gli statements/istruzioni SQL svolgono il parsing, l'ottimizzazione secondo "query plan" (regole interne delle query che consentono di ottimizzare l'accesso ai dati, in senso operazionale, tipo viene prima la selezione rispetto al join, ecc.) ed eseguono lo statement.
Ecco quindi che utilizziamo i "*prepared statements*", che permettono di evitare i passi 1/2 e risultano utili, in particolare, quando uno statement viene "preparato" (tale da garantire alta efficienza nelle serie successive di esecuzioni) per poi essere eseguito.

La sintassi di esecuzione è la seguente:

- `PGresult *PQprepare(PGconn *conn, const char *stmtName, const char *query, int nParams, const Oid *paramTypes)`

preparando uno statement *stmtName*, con *query* lo statement, poi *nparam* ben inteso come numero dei parametri. La query contiene anche $1, ... $i che sono dei "placeholders" per i valori e *lasciare sempre paramtype a NULL* (esso specifica i tipi di dati assegnati ai simboli dei parametri; non vedremo l'utilizzo di questo parametro nel corso e appunto si lasci sempre a NULL).

Scritto da Gabriel

Esempio:

```
PQprepare(conn , "aggStudenti",
        "INSERT INTO STUDENTI(MATRICOLA,COGNOME,NOME,DATANASCITA)
VALUES ($1,$2,$3,$4)", 4, NULL);
```

Similmente per l'esecuzione:

- `PGresult *PQexecPrepared(PGconn *conn, const char *stmtName, int nParams, const char * const *paramValues, const int *paramLengths, const int *paramFormats, int resultFormat)`

  Si usa per eseguire un prepared statement di nome `stmtName`, dove `paramValues` contiene la lista dei parametri e `nParams` è il numero di parametri. OK avere sempre:

  - `paramFormat` come un array di zeri, di lunghezza quanto il numero di parametri;
  - `paramLenght = NULL;`
  - `resultForm = 0.`

  Esempio:

```
PQexecPrepared(conn , "aggStudenti", 4, parametri , NULL ,
{0,0,0,0}, 0);
```

Vediamo l'esecuzione di un *prepared statement* composto di vari parametri e poi prendendo un altro insieme di parametri, preparando l'operazione precedente ed una specifica query.

```
ExecStatusType aggiungiStudente(Pgconn *conn, char *matricola, char *cognomenome, char *nome, char *dataNascita
{
    char* parametri={matricola, cognome, nome, dataNascita};
    ExecStatusType risultato=PQexecPrepared(conn , "aggStudenti", 4, parametri , NULL , {0,0}, 0);
    PQclear(res);
    return(risultato);
}

int main(int argc, char* argv[])
{
    char matricola[10], char cognome[50], char nome[50], char dataNascita[10];
    PGconn *conn = PQconnectdb("dbname=testdb");

    PGresult *stmt = PQprepare(conn , "aggStudenti",
        "INSERT INTO STUDENTI(MATRICOLA,COGNOME,NOME,DATANASCITA) VALUES ($1,$2,$3,$4)", 4, NULL);
    […]
```

Normalizzazione

Descriviamo le *forme normali*, quindi delle proprietà di basi di dati che ne garantiscono la "qualità", quindi l'assenza di determinati difetti. Quando una relazione non è normalizzata presenta ridondanze e si presta a comportamenti poco desiderabili durante gli aggiornamenti.
Utilizziamo quindi la *normalizzazione*, che ci permette di trasformare schemi non normalizzati in schemi che soddisfano le forme normali. Questa procedura si pone come una *tecnica di verifica* della buona progettazione del DB.
Vediamo l'esempio di una relazione con anomalie, perchè:

- lo stipendio di un impiegato è ripetuto in tutte le tuple (ridondanza)

Impiegato	Stipendio	Progetto	Bilancio	Funzione
Rossi	20	Marte	2	tecnico
Verdi	35	Giove	15	progettista
Verdi	35	Venere	15	progettista
Neri	55	Venere	15	direttore
Neri	55	Giove	15	consulente
Neri	55	Marte	2	consulente
Mori	48	Marte	2	direttore
Mori	48	Venere	15	progettista
Bianchi	48	Venere	15	progettista
Bianchi	48	Giove	15	direttore

- se lo stipendio di un impiegato varia, è necessario modificarne il valore in diverse n-uple (anomalia di aggiornamento)
- se un impiegato interrompe la partecipazione a tutti i progetti dobbiamo cancellarlo (anomalia di cancellazione)
- un nuovo impiegato senza progetto assegnato non può essere inserito (anomalia di inserimento, perché Progetto è parte della chiave)

Si deve introdurre quindi un vincolo di integrità, cioè la *dipendenza funzionale*, descritta con una serie di proprietà. Ad esempio:

- ogni impiegato ha un solo stipendio (anche se partecipa a più progetti)
- ogni progetto ha un bilancio
- ogni impiegato in ciascun progetto ha una sola funzione (può comunque avere funzioni diverse in progetti diversi)

Formalmente parlando:

- Data una $R(X)$ sugli attributi X
- Dati due insiemi di attributi $Y, Z \subset X$
- esiste in R una dipendenza funzionale (FD) da Y a Z se:
- Per ogni coppia di tuple $t_1, t_2 \in R$:
  se $\pi_Y(t_1) = \pi_Y(t_2)$ allora $\pi_Z(t_1) = \pi_Z(t_1)$

La notazione utile è: $Y \rightarrow Z$

Esempi:
Impiegato → Stipendio
Progetto → Bilancio
Impiegato_Progetto → Funzione

Vi sono alcune dipendenze banali, come:

- Impiegato Progetto $\rightarrow$ Progetto
- Si tratta però di una DF "banale":
- $Y \rightarrow A$ è banale se $A \subset Y$

Le anomalie sono legate ad alcune dipendenze funzionali:

- Gli impiegati hanno un unico stipendio (Impiegato → Stipendio)
- I progetti hanno un unico bilancio (Progetto → Bilancio)

Un esempio di dipendenza funzionale "buona" può essere: $\text{Impiegato Progetto} \rightarrow \text{Funzione}$ sapendo che non ci saranno mai impiegati con la stessa funzione in un progetto; viene definita buona in quanto non causa anomalie.

L'esempio migliore di forma normale è la *forma normale di Boyce e Codd (BCNF)*, formalmente:

- Una relazione R con chiavi $K_1, \ldots, K_n$, è in forma normale di Boyce e Codd:
  se ogni dipendenza funzionale non banale $X \rightarrow Y$ è «buona» cioè $\exists i.\ K_i \subseteq X$ (X è superchiave)

Significa inoltre che una tabella in BCNF deve avere ciascun campo chiave che preso da solo è effettivamente una chiave e ciascuna di queste, normalmente, va a formare una apposita tabella.
La forma normale richiede che i concetti in una relazione siano omogenei (quindi solo proprietà associate direttamente alla chiave; ad esempio nel caso d'uso, ogni informazione è legata alla coppia Impiegato-Progetto, ad esempio allo Stipendio).
Se una relazione non soddisfa la BCNF, normalmente la rimpiazziamo con altre relazioni che la soddisfano, decomponendo sulla base delle dipendenze funzionali per poter separare i concetti.

Scritto da Gabriel

Rispetto alle successive, la terza rispetta la BNCF, le altre no:

Impiegato → Stipendio
Progetto → Bilancio
Impiegato Progetto → Funzione

Similmente Impiegato-Stipendio-Progetto → Bilancio, per questa relazione, non violerebbe la BNCF.

Prendiamo ora "Impiegato → Stipendio", sapendo che viola, creandomi una tabella apposita, come si vede qui a fianco.

Impiegato	Stipendio
Rossi	20
Verdi	35
Verdi	35
Neri	55
Neri	55
Neri	55
Mori	48
Mori	48
Bianchi	48
Bianchi	48

Prendiamo poi "Progetto → Bilancio", creandomi la seconda tabella, togliendo tutti gli attributi della parte di sinistra.

Progetto	Bilancio
Marte	2
Giove	15
Venere	15
Venere	15
Giove	15
Marte	2
Marte	2
Venere	15
Venere	15
Giove	15

Si struttura poi la terza tabella con "Impiegato_Progetto → Funzione":

Impiegato	Progetto	Funzione
Rossi	Marte	tecnico
Verdi	Giove	progettista
Verdi	Venere	progettista
Neri	Venere	direttore
Neri	Giove	consulente
Neri	Marte	consulente
Mori	Marte	direttore
Mori	Venere	progettista
Bianchi	Venere	progettista
Bianchi	Giove	direttore

Nelle due tabelle formate prima, i campi "Impiegato" e "Progetto" diventano chiave, togliendo successivamente i duplicati.

L'idea intuitiva di normalizzazione è la seguente:

Per ogni dipendenza $X \rightarrow Y$ che viola la BCNF, definire una relazione su XY ed eliminare Y dalla relazione originaria

Vediamo quindi un esempio concreto con il seguente esercizio:

Data la seguente relazione:

Docente	Dipartimento	Facoltà	Preside	Corso
Verdi	Matematica	Ingegneria	Neri	Analisi
Verdi	Matematica	Ingegneria	Neri	Geometria
Rossi	Fisica	Ingegneria	Neri	Analisi
Rossi	Fisica	Scienze	Bruni	Analisi
Bruni	Fisica	Scienze	Bruni	Fisica

1. Individuare le proprietà della relazione:
   - Chiave/i della relazione
   - Dipendenze funzionali
2. Identificare ridondanze o anomalie
3. Decomporre in BCNF

1) Dipendenze funzionali:
   Facoltà – Preside Preside – Facoltà Docente – Dipartimento

- Chiave/i della relazione: **Dipart, Facoltà, Corso**
- Dipendenze funzionali: **Facoltà → Preside**

2) Per una stessa facoltà avremmo lo stesso preside, ripetuto ogni volta (ridondanza).
   Se il preside cambia, si devono cambiare tutte le tuple (anomalia aggiornamento).
   Se tutti i corsi vengono rimossi, si perde l'informazioni di chi fosse il preside (anomalia cancellazione).
3) La soluzione ignora la dipendenza Facoltà – Preside mantenendo come tabelle:
   Le tabelle risultanti sarebbero:
   Facoltà – Preside
   Docente – Dipartimento
   Docente – Facoltà – Corso

Graficamente, considerando che il libro (quindi l'immagine sotto) non considera la dipendenza Facoltà – Preside, allora il risultato sarà il seguente.

Facoltà	Preside
Ingegneria	Neri
Scienze	Bruni

Docente	Dipartimento	Facoltà	Corso
Verdi	Matematica	Ingegneria	Analisi
Verdi	Matematica	Ingegneria	Geometria
Rossi	Fisica	Ingegneria	Analisi
Rossi	Fisica	Scienze	Analisi
Bruni	Fisica	Scienze	Fisica

A seguito di questa decomposizione, le anomalie vengono risolte correttamente.
Non sempre è facile normalizzare:
Si pensa di attuare una decomposizione basata sulle dipendenze, partendo da Impiegato – Progetto – Sede.
L'idea è quindi di crearsi tabelle separate per:

- Impiegato → Sede
- Progetto → Sede

La dipendenza funzionale Impiegato → Sede non è più verificata dopo la decomposizione; dunque, non sarebbe più possibile mantenere i dati.

Impiegato	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Venere	Milano
Neri	Saturno	Milano
Neri	Venere	Milano

Impiegato → Sede
Progetto → Sede

Se la decomposizione è fatta bene, a seguito di join devo riottenere le stesse tuple, altrimenti ho perso dati oppure aggiunto informazioni che non servono.
Nel caso qui a sinistra, come si vede, dopo un join dovrei ottenere 5 tuple:

e partendo da qui:

Impiegato	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Venere	Milano
Neri	Saturno	Milano
Neri	Venere	Milano

Impiegato	Sede
Rossi	Roma
Verdi	Milano
Neri	Milano

Impiegato → Sede

Progetto	Sede
Marte	Roma
Giove	Milano
Saturno	Milano
Venere	Milano

Progetto → Sede

in realtà ne ottengo 7:

Impiegato	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Venere	Milano
Neri	Saturno	Milano
Neri	Venere	Milano
Verdi	Saturno	Milano
Neri	Giove	Milano

Diversa dalla relazione di partenza!

ottenendo quindi una *decomposizione con perdita* (perché necessaria ma non sufficiente, quindi gli attributi di join sono chiavi di una delle due relazioni).

Se come detto prima con il join si riottengono gli stessi dati, si ha una *decomposizione senza perdita*:

- Una relazione R(X) si decompone senza perdita su $X_1, X_2 \subset X$ con $X1 \cup X2 = X$ se $\pi_{X1}(X) \bowtie \pi_{X2}(X) = X$
- La decomposizione senza perdita è garantita se gli attributi in $X1 \cap X2$ sono una chiave della relazione $\pi_{X1}(X)$ e/o di $\pi_{X2}(X)$

In questo caso però si ha perdita, in quanto l'attributo comune Sede non è né chiave in R1 né in R2:

R1

Impiegato	Sede
Rossi	Roma
Verdi	Milano
Neri	Milano

Impiegato → Sede

R2

Progetto	Sede
Marte	Roma
Giove	Milano
Saturno	Milano
Venere	Milano

Progetto → Sede

sapendo che poi non ho perdita tramite i join, per esempio Impiegato (relazione di sinistra tra Impiegato e Stipendio) e Progetto (relazione di sinistra tra Impiegato_Progetto → Funzione).

Impiegato	Stipendio
Rossi	20
Verdi	35
Verdi	35
Neri	55
Neri	55
Neri	55
Mori	48
Mori	48
Bianchi	48
Bianchi	48

Impiegato → Stipendio

⋈

Impiegato	Progetto	Funzione
Rossi	Marte	tecnico
Verdi	Giove	progettista
Verdi	Venere	progettista
Neri	Venere	direttore
Neri	Giove	consulente
Neri	Marte	consulente
Mori	Marte	direttore
Mori	Venere	progettista
Bianchi	Venere	progettista
Bianchi	Giove	direttore

Impiegato Progetto → Funzione

⋈

Progetto	Bilancio
Marte	2
Giove	15
Venere	15
Venere	15
Giove	15
Marte	2
Marte	2
Venere	15
Venere	15
Giove	15

Progetto → Bilancio

Attributi Comuni:

Impiegato
(Chiave di Relazione di Sinistra)

Attributi Comuni:

Progetto
(Chiave di Relazione di Sinistra)

Tramite il join si ha la relazione di partenza:

Docente	Dipartimento	Facoltà	Preside	Corso
Verdi	Matematica	Ingegneria	Neri	Analisi
Verdi	Matematica	Ingegneria	Neri	Geometria
Rossi	Fisica	Ingegneria	Neri	Analisi
Rossi	Fisica	Scienze	Bruni	Analisi
Bruni	Fisica	Scienze	Bruni	Fisica

Si prova quindi a decomporre senza perdita sfruttando solo Impiegato → Sede, sapendo che ciò non funziona;
noi vogliamo dimostrarne il perché.
Creiamo quindi la tabella Impiegato – Sede, ottenendo così la tabella Impiegato – Progetto:

Impiegato	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Venere	Milano
Neri	Saturno	Milano
Neri	Venere	Milano

Impiegato	Sede
Rossi	Roma
Verdi	Milano
Neri	Milano

Impiegato	Progetto
Rossi	Marte
Verdi	Giove
Verdi	Venere
Neri	Saturno
Neri	Venere

La decomposizione è senza perdita, perché Impiegato è chiave; tuttavia ci sta la dipendenza funzionale tra Progetto e Sede, avendo applicato l'algoritmo a metà.
Questo porta ad un problema di *conservazione delle dipendenze*: immaginiamo infatti di voler aggiungere una tupla in verde in merito all'impiegato Neri, che opera a Milano, al progetto Marte:

Impiegato	Sede
Rossi	Roma
Verdi	Milano
Neri	Milano

Impiegato	Progetto
Rossi	Marte
Verdi	Giove
Verdi	Venere
Neri	Saturno
Neri	Venere
Neri	Marte

Impiegato → Sede ~~Progetto → Sede~~

Attraverso il join riottengo tutte le tuple della relazione di partenza; tuttavia, non vengono conservate tutte le dipendenze funzionali:

Impiegato	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Venere	Milano
Neri	Saturno	Milano
Neri	Venere	Milano
Neri	Marte	Milano

Impiegato → Sede ~~Progetto → Sede~~

Una decomposizione conserva la dipendenze se ciascuna delle dipendenze funzionali dello schema originario coinvolge attributi che compaiono tutti insieme in uno degli schemi decomposti.
In questo specifico caso, Progetto → Sede non è conservata.

Nota:
Il prof spesso chiede negli esercizi di normalizzazione se, a seguito di decomposizioni in 3FN/BCNF, ci sia conservazione delle dipendenze. Quella che segue ne è la definizione:
"Una decomposizione conserva le dipendenze se ciascuna delle dipendenze funzionali dello schema originario coinvolge attributi che compaiono tutti insieme in uno degli schemi decomposti."

Quindi: se nelle relazioni finali a seguito delle decomposizioni (o in 3FN di cui esiste un algoritmo oppure in BCNF, che significa semplicemente che ogni attributo non chiave dipende solo dalla sua chiave/superchiave) ci sono tutti gli attributi delle FD originarie, allora siamo a posto.
La decomposizione è sempre senza perdita se "gli attributi in comune sono superchiave di almeno una delle due relazioni".

Normalizzazione: proseguimento lezione

Partendo dalla tabella sotto, allora:

1. Trovare le Dipendenze Funzionali che violano la forma normale di Boyce Codd e non sono banali
2. Decomporre in Forma Normale di Boyce-Codd

Prodotto	Componente	Tipo	Q	PC	Fornitore	PT
Libreria	Legno	Noce	50	10.000	Forrest	400.000
Libreria	Bulloni	B212	200	100	Bolt	400.000
Libreria	Vetro	Cristal	3	5.000	Clean	400.000
Scaffale	Legno	Mogano	5	15.000	Forrest	300.000
Scaffale	Bulloni	B212	250	100	Bolt	300.000
Scaffale	Bulloni	B412	150	300	Bolt	300.000
Scrivania	Legno	Noce	10	8.000	Wood	250.000
Scrivania	Maniglie	H621	10	20.000	Bolt	250.000
Tavolo	Legno	Noce	4	10.000	Forrest	200.000

Q = Quantità
PC = Prezzo Unitario (per tutte le quantità insieme e non per unità di quantità)
PT = Prezzo Totale

Le dipendenze funzionali che violano la BNCF sono:

- Prodotto – PT (non è superchiave Prodotto)
- Prodotto, Componente – Fornitore (similmente anche Prodotto → Componente)

Andiamo ad operare una decomposizione per Prodotto, togliendola dalla relazione:

Prodotto	Componente	Tipo	Q	PC	Fornitore
Libreria	Legno	Noce	50	10.000	Forrest
Libreria	Bulloni	B212	200	100	Bolt
Libreria	Vetro	Cristal	3	5.000	Clean
Scaffale	Legno	Mogano	5	15.000	Forrest
Scaffale	Bulloni	B212	250	100	Bolt
Scaffale	Bulloni	B412	150	300	Bolt
Scrivania	Legno	Noce	10	8.000	Wood
Scrivania	Maniglie	H621	10	20.000	Bolt
Tavolo	Legno	Noce	4	10.000	Forrest

Prodotto	PT
Libreria	400.000
Scaffale	300.000
Scrivania	250.000
Tavolo	200.000

Rimane comunque un'ulteriore decomposizione da fare, togliendo Fornitore ed ottenendo il resto:

Prodotto	Componente	Tipo	Q	PC
Libreria	Legno	Noce	50	10.000
Libreria	Bulloni	B212	200	100
Libreria	Vetro	Cristal	3	5.000
Scaffale	Legno	Mogano	5	15.000
Scaffale	Bulloni	B212	250	100
Scaffale	Bulloni	B412	150	300
Scrivania	Legno	Noce	10	8.000
Scrivania	Maniglie	H621	10	20.000
Tavolo	Legno	Noce	4	10.000

Prodotto	Componente	Fornitore
Libreria	Legno	Forrest
Libreria	Bulloni	Bolt
Libreria	Vetro	Clean
Scaffale	Legno	Forrest
Scaffale	Bulloni	Bolt
Scrivania	Legno	Wood
Scrivania	Maniglie	Bolt
Tavolo	Legno	Forrest

La trasformazione finale consegue:

Prodotto	Componente	Tipo	Q	PC
Libreria	Legno	Noce	50	10.000
Libreria	Bulloni	B212	200	100
Libreria	Vetro	Cristal	3	5.000
Scaffale	Legno	Mogano	5	15.000
Scaffale	Bulloni	B212	250	100
Scaffale	Bulloni	B412	150	300
Scrivania	Legno	Noce	10	8.000
Scrivania	Maniglie	H621	10	20.000
Tavolo	Legno	Noce	4	10.000

Prodotto	Componente	Fornitore
Libreria	Legno	Forrest
Libreria	Bulloni	Bolt
Libreria	Vetro	Clean
Scaffale	Legno	Forrest
Scaffale	Bulloni	Bolt
Scrivania	Legno	Wood
Scrivania	Maniglie	Bolt
Tavolo	Legno	Forrest

Prodotto	PT
Libreria	400.000
Scaffale	300.000
Scrivania	250.000
Tavolo	200.000

L'idea è che la *decomposizione sia sempre senza perdita*, in maniera tale da poter sempre ricreare le informazioni originarie, nonché la *conservazione delle dipendenze*, mantenendo gli originari vincoli di integrità. Prendiamo una relazione di questo tipo:

Dirigente	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Marte	Milano
Neri	Saturno	Milano
Neri	Venere	Milano

Progetto Sede → Dirigente
Dirigente → Sede

Andando a decomporre con Dirigente → Sede, rimarrei con:

Progetto Sede → Dirigente

che coinvolge tutti gli attributi e si perderebbe la dipendenza funzionale; dunque la BNCF non è sempre raggiungibile.

Una possibile riorganizzazione può essere:

Dirigente	Progetto	Sede	Reparto
Rossi	Marte	Roma	1
Verdi	Giove	Milano	1
Verdi	Marte	Milano	1
Neri	Saturno	Milano	2
Neri	Venere	Milano	2

Dirigente → Sede Reparto
Sede Reparto → Dirigente
Progetto Sede → Reparto

tale che la decomposizione sia di porre in una tabella il Dirigente, dipendente dalla coppia Sede – Reparto e Reparto una relazione con la coppia Progetto – Sede. Inserendo ad esempio Rossi – Milano – 2, sarebbe una violazione per il Dirigente, ma non sulla superchiave vista come esterna, dunque la coppia Sede – Reparto.

In generale, comunque, si intende che la decomposizione senza perdite permetta di avere una serie di dipendenze funzionali per cui, per ogni coppia di relazioni, ci sia almeno una chiave/superchiave di collegamento (possibilmente minimale, quindi chiave); nel caso in cui non ci fosse una chiave per tutte le relazioni e, rispetto alle dipendenze funzionali iniziali, vi sono relazioni per cui non sono tutte rappresentate (quindi, qualche dipendenza funzionale non sta in qualche relazione), allora si ha una decomposizione con perdita.

Chiavi:
- Dirigente
- Sede, Reparto

Dirigente	Sede	Reparto
Rossi	Roma	1
Verdi	Milano	1
Neri	Milano	2

Chiavi:
- Sede, Progetto

Progetto	Sede	Reparto
Marte	Roma	1
Giove	Milano	1
Marte	Milano	1
Saturno	Milano	2
Venere	Milano	2

- **Decomposizione senza perdita: gli attributi comuni {Sede, Reparto} sono una chiave della relazione sx**
- **Tutte le dipendenze sono rispettate**

Introduciamo la *Terza Forma Normale*, definita su una relazione R con chiavi K_1, …… K_N, definita se:

Per ogni dipendenza funzionale non banale $X \rightarrow Y$, almeno una delle seguenti condizioni sono valide:
- X è superchiave (BCNF)
- ogni attributo in Y è contenuto in almeno una tra le chiavi $K_1,\ldots, K_n$.

In particolare, si definisce che per ogni DF:
- X è superchiave della relazione
- Y è membro di una chiave della relazione

Quando non viene rispettata si attuano delle decomposizione secondo algoritmi seguenti.

Essa risulta essere sempre raggiungibile, tuttavia risulta essere *meno restrittiva della BNCF*.
Alcune dipendenze vengono escluse e di fatto ammette alcune anomalie.
Ad esempio, Progetto Sede → Dirigente non è in BNCF, tuttavia essendo Progetto Sede superchiave ed essendo contenuto in Dirigente Sede, allora è ammesso dalla stessa.
Si ha quindi:
- ridondanza tra Dirigente e Sede
- possibilità di un Dirigente di "apparire" in più sedi

Per poter fare una *decomposizione in Terza Forma Normale*:
- si crea una relazione per ogni gruppo di attributi coinvolti in una dipendenza funzionale
- si verifica che alla fine una relazione contenga una chiave della relazione originaria

Se la relazione non è normalizzata, si decompone nella Terza Forma Normale, in maniera tale che sia anche in BNCF. *Se è in BNCF, allora è in 3NF; il viceversa non è verificato.*

Data dunque una relazione R ed un insieme di dipendenze funzionali definite su R, si vuole generare una decomposizione che sia:
- senza perdita
- conservi le dipendenze
- contenga solo relazioni normalizzate (possibilmente in BCNF, ma sicuramente in 3NF)

Un insieme F di FD (dip. funz) implica un'altra FD f se ogni relazione che soddisfa tutte le FD in F soddisfa anche f (transitività):

Esempio: R(Impiegato, Categoria, Stipendio)

Impiegato→Categoria
Categoria→Stipendio

implicano

Impiegato→Stipendio.

Dati uno schema di Relazione R(U), un insieme F di Dip. Funz. definite su U ed un insieme di attributi $X \subseteq U$, si definisce il concetto di *chiusura*:

- La **chiusura** X^+_F di X rispetto ad F, è l'insieme degli attributi che dipendono funzionalmente da X:
  $X^+_F = \{ A \mid A \in U \text{ e } F \text{ implica } X \rightarrow A \}$
- Si noti che se $\{ X1, X2 \} \rightarrow A$ allora $\{ X1, X2, \dots \} \rightarrow A$
- Se A appartiene a X^+_F allora $X \rightarrow A$ è implicata da F

Esempio:
X_1 X_2 X
$X_1 \rightarrow Y$
(se a X_1 aggiungo altri attributi, farà parte lo stesso della relazione verso Y)
$X_1X_2 \rightarrow y$

Generalmente, comunque la chiusura comprende:

- tutti gli attributi di sinistra e quelli raggiunti a destra transitivamente
  ES: $A^+ = \{A, B, C, D\}$ su $A \rightarrow B, B \rightarrow C, B \rightarrow D$
- se l'attributo a sinistra è composto da vari attributi, la chiusura comprende anche le chiusure dei sottoattributi.
  ES: $AB^+ = \{A, B, C, D, E\}$ su $AB \rightarrow C, B \rightarrow C, B \rightarrow D, D \rightarrow E$

Dunque, se $X_F^+ = U$, allora $X \rightarrow U$. Non avendo due tuple con lo stesso valore, allora X necessariamente potrà essere chiave. In questo modo, si dice che X è candidata ad essere chiave.
Vogliamo quindi trovare tutti gli attributi che hanno lo stesso valore se quest'ultimo ammette la chiusura transitiva, calcolando la chiusura X_F^+. Definita la relazione con queste DF:

Impiegato	Stipendio	Progetto	Bilancio	Funzione
Rossi	20	Marte	2	tecnico
Verdi	35	Giove	15	progettista
Verdi	35	Venere	15	progettista
Neri	55	Venere	15	direttore
Neri	55	Giove	15	consulente
Neri	55	Marte	2	consulente
Mori	48	Marte	2	direttore
Mori	48	Venere	15	progettista
Bianchi	48	Venere	15	progettista
Bianchi	48	Giove	15	direttore

Impiegato → Stipendio
Progetto → Bilancio
Impiegato Progetto → Funzione

Semplicemente prendendo ad esempio:

$X = \{ \text{Impiegato, Progetto} \}$ $X_P = \{ \text{Impiegato, Progetto} \}$

andando ad aggiungere altri attributi, la chiusura comunque rimane, come definito dai successivi esempi, coinvolgendo comunque tutti gli attributi (Impiegato, Progetto è considerata dunque una buona chiave):

Da Impiegato → Stipendio:
$X_P = \{ \text{Impiegato, Progetto, Stipendio} \}$
Da Progetto → Bilancio aggiungiamo Bilancio:
$X_P = \{ \text{Impiegato, Progetto, Stipendio, Bilancio} \}$
Da Impiegato Progetto → Funzione aggiungiamo Funzione:
$X_P = \{ \text{Impiegato, Progetto, Stipendio, Bilancio, Funzione} \}$

Prendiamo l'esempio di R (A,B,C,D) con le dipendenze F = {A → B, B → D}
Nella chiusura transitiva avremo $A_F^+ = \{A,B,D\}$ sarà quindi anche vero A → D
La chiusura transitiva si considera a partire da A; se mettessi AC, non sarebbe valida nel caso di A_F^+.
Sarebbe invece contenuta in $AC_F^+ = \{A,C,B,D\}$

L'algoritmo si struttura in questo modo:

Input: un insieme X di attributi e un insieme F di dipendenze funzionali

Output: un insieme X_P di attributi.

1. Inizializziamo X_P con l'insieme di input X.
2. Se esiste una FD $Y \rightarrow A$ in F con $Y \subseteq X_P$ e $A \notin X_P$, allora aggiungiamo A a X_P .
3. Ripetiamo il passo (2) fino a quando non ci sono ulteriori attributi che possono essere aggiunti a X_P.

Similmente due insiemi F_1 ed F_2 sono equivalenti se F_1 implica ciascuna dipendenza in F_2 e viceversa.
Ad esempio: {A → B; AB → C; A → C} e {A → B; AB → C} sono equivalenti
Se due insiemi sono equivalenti, ognuno è *copertura* dell'altro; quando si ha l'equivalenza, conviene sceglierne uno più semplice (per esempio nel caso visto, preferirò il secondo insieme, avendo meno FD e dato che agiscono sullo stesso insieme di dati; altro esempio ABC → C ed AB → C, preferirò la seconda).
Un insieme di dipendenze F viene definito:

- **non ridondante** se non esiste dipendenza $f \in F$ tale che $F - \{f\}$ implica f;
- **ridotto** se (i) è non ridondante e (ii) non esiste un insieme F′ equivalente a F ottenuto eliminando attributi dai primi membri di una o più dipendenze di F.

Se ridondante, si toglie perché i vincoli rimangono gli stessi rispetto alle tuple ammissibili nella relazione.

- Esempio (parte in rosso rimovibile):
  - {A → B; AB → C; A → C} è ridondante;
  - {A → B; AB → C} non è ridondante né ridotto;
  - {A → B; A → C} è ridotto

Partendo da un insieme di dipendenze funzionali, tolgo tutte quelle che sono ridondanti, riducendo all'osso.

F: M→RSDG, MS→CD, G→R, D→S, S→D, MPD→AM.

All'inizio si vuole avere una serie di FD dove a destra si abbiano solo attributi singoli, a sinistra possono anche non esserlo. Ad esempio, MS → CD è equivalente ad MS → C ed MS → D.

1. Sostituiamo l'insieme dato con quello equivalente dove i secondi membri sono singoli attributi;

M→R, M→S, M→D, M→G, MS→C, MS→D, G→R, D→S, S→D, MPD→A, MPD→M

Dobbiamo togliere le dipendenze ridondanti. Vediamo quindi che:

- si può togliere M → R, perché si ha dipendenza transitiva con M → G e G → R
- si può togliere M → S, perché si ha M → D e D → S
- si può togliere MPD → M, avendo M da entrambi i lati

2. Eliminiamo le dipendenze ridondanti;

M→D, M→G, MS→C, MS→D, G→R, D→S, S→D, MPD→A

3. Per ogni dipendenza verifichiamo se esistono attributi eliminabili dal primo membro

M→D, M→G, MS→C, MS→D, G→R, D→S, S→D, MPD→A

Da quello che capisco qui, va a togliere attributi che non aggiungono informazioni alla relazione.
Per esempio, viene tolto S da MS → C. Dato che S appare da altre parti, va tenuta come informazione e teniamo S → D, togliendo M. Esistendo già una forma S → D ne togliamo una delle due.
Similmente, ho MPD → A. M → A, essendo M in varie relazioni è sensato da mantenere. D → A, sempre data la presenza di D in tutte le relazioni si mantiene. Chi rimane fuori sarà P → A, dato che P non serve a nessuno.

Esempi utili:

- Un insieme F di dipendenze funzionale puo' contenere dipendenze ridondanti, ovvero che possono essere ottenute dalle altre dipendenze di F
  - Esempio: $A \to C$ e' ridondante in $\{A \to B, B \to C, A \to C\}$
- Anche degliattributi di una dipendenza funzionale potrebbero essere ridondanti:
  - A destra: $\{A \to B, B \to C, A \to CD\}$ puo' essere semplificata in $\{A \to B, B \to C, A \to D\}$
  - A sinistra: $\{A \to B, B \to C, AC \to D\}$ puo' essere semplificata in $\{A \to B, B \to C, A \to D\}$

Quindi così

1) viene calcolata una *copertura ridotta (o minima)* G di F:

M→D, M→G, M→C, G→R, D→S, S→D, MP→A

Calcoliamo nell'ordine le chiusure (tutte le relazioni che riguardano anche il valore X):

- la chiusura di MP: $PM_F^+ = \{M, D, G, C, R, S, P, A\}$
- la chiusura di M $M_F^+ = \{M, D, G, C, R, S\}$
- la chiusura di G: $G_F^+ = \{G, R\}$
- la chiusura di D: $D_F^+ = \{D , S\}$

2) Si riuniscono tutte le dipendenze sulla base delle chiusura, quindi M con tutte le relazioni che lo riguardano direttamente, similmente G, D ed MP

2. G viene partizionato in sottoinsiemi tali che due dip. funzion. $X \to A$ e $Y \to B$ sono insieme se $X^+_G = Y^+_G$:

   {M→D, M→G, M→C}, {G→R}, {D→S, S→D}, {MP→A}

3) Per ogni sottoinsieme viene costruita una relazione:

$R_1(M, D, C, G), R_2(G, R), R_3(D, S), R_4(M, P, A)$

4) Se esistono due relazioni S(X) e T(Y) con $X \subseteq Y$, S viene eliminata:
Nel caso nostro, se non ci fosse una relazione che contiene M,P non devo aggiungere nulla (come nel nostro caso); ciò viene fatto per garantire che M rimanga chiave e come tale ne venga preservato il vincolo di chiave. Si sarebbe quindi dovuta creare una relazione R_5(M,P).
Al di là di discorsi del prof, banalmente, se ho (A,B) ed (A,B,C) andrei ad eliminare (A,B) sulla base di questa operazione. Nota di contorno, sarebbe "Non accade" e non "Non accede"

Non accede. Quindi stesse relazioni:
$R_1(M, D, C, G), R_2(G, R), R_3(D, S), R_4(M, P, A)$

5) Se, per qualche *i*. esiste una chiave K per la quale non esiste una relazione che contiene tutti gli attributi di K, viene aggiunta una relazione $T(K_i)$. Per esempio, P, che non fa parte di nulla. Se non ci fosse stato in R_4 avrei dovuto aggiungerlo in una nuova.

Non accede. Perché R_4 contiene M e P:
$R_1(M, D, C, G), R_2(G, R), R_3(D, S), R_4(M, P, A)$

Dunque, le chiavi sarebbero come:
R_1 (<u>M, D</u>, C, G)
R_2 (<u>G</u>, R)

Scritto da Gabriel

R_3 (D, S)
R_4 (M, P, A)

Due esempi davvero utili passo per passo (calcolo copertura ridotta e forme normali):

Esercizio 13. Siano dati lo schema di relazione $R(A, B, C, D, E, F, G, H, I, J)$ ed il relativo insieme di dipendenze funzionali $F = \{ABD \to E, AB \to G, B \to F, C \to J, CJ \to I, G \to H\}$.

(8.1) Stabilire se F e' o meno una copertura minimale. In caso di risposta negativa, determinare una copertura minimale di F.
(8.2) Determinare l'insieme delle chiavi candidate di R.

1) Tutti gli attributi sono già unitari a sinistra, secondo l'algoritmo della copertura ridotta.
Ora cerchiamo di eliminare le dipendenze ridondanti.

Andando con ordine:

- In ABD → E, l'attributo A è ridondante sse E appartiene alla chiusura BD.
  B → F e dunque, come visto sopra BD → F.
  Dunque, essendo BD^+ = {BDF}, allora A non è presente e non è ridondante
- In ABD → E, l'attributo B è ridondante sse E appartiene alla chiusura AD.
  AD → E quindi non è ridondante
- In ABD → E, l'attributo D è ridondante sse E appartiene a AB^+.
  AB^+ = {ABFGH} avendo AB → G, B → F e G → H e dunque D non è ridondante

Su questa linea:

L'attributo A e' ridondante in $AB \to G$ sse $G \in B^+$. $B^+ = \{BF\} \Rightarrow A$ non e' ridondante in $AB \to G$.
L'attributo B e' ridondante in $AB \to G$ sse $G \in A^+$. $A^+ = \{A\} \Rightarrow B$ non e' ridondante in $AB \to G$.
L'attributo C e' ridondante in $CJ \to I$ sse $I \in J^+$. $J^+ = \{J\} \Rightarrow C$ non e' ridondante in $CJ \to I$.
L'attributo J e' ridondante in $CJ \to I$ sse $I \in C^+$. $C^+ = \{CIJ\} \Rightarrow C$ e' ridondante in $CJ \to I$. Sostituiamo dunque $CJ \to I$ con $C \to I$, ottenendo $F = \{ABD \to E, AB \to G, B \to F, C \to J, C \to I, G \to H\}$

Otteniamo quindi un insieme di dipendenze con relative chiusure. Eliminiamo quelle ridondanti ottenuto dal passo precedente. A questo punto:
Per ogni dipendenza X → Y è sufficiente verificare se *y* appartiene alla chiusura di X rispetto ad F \ {X → Y}

$X \to Y$	X^+ rispetto a $F \setminus \{X \to Y\}$	$X \to Y$ e' ridondante?
$ABD \to E$	$ABD^+ = \{ABDGFH\}$	No
$AB \to G$	$AB^+ = \{ABF\}$	No
$B \to EF$	$B^+ = \{B\}$	No
$C \to J$	$C^+ = \{CI\}$	No
$C \to I$	$C^+ = \{CJ\}$	No
$G \to H$	$G^+ = \{G\}$	No

Dunque, la copertura minimale richiesta è proprio:

$$F = \{ABD \to E, AB \to G, B \to F, C \to J, C \to I, G \to H\}$$

2) Giustamente, le chiavi candidate sono tutti gli attributi a sinistra che non compaiono a destra (ABCD, G invece appare).
Si ha inoltre che $ABCD^+$ = ABCDEFGHIJ
Includendo tutto per vincolo di minimalità, ABCD è superchiave.

Esercizio 14. Siano dati lo schema relazionale $R(A, B, C, D, E, F)$ e gli insiemi di dipendenza funzionali $G = \{AB \to C, B \to A, AD \to E, BD \to F\}$ ed $H = \{AB \to C, B \to A, AD \to EF\}$

(9.1) Determinare una copertura minimale per G ed una copertura minimale per H.

(9.2) Stabilire se G ed H sono equivalenti.

(9.1) Determinare una copertura minimale per G ed una copertura minimale per H.

- Soluzione. Calcoliamo una copertura minimale per G con l'algoritmo visto a lezione.
  - Passo 1. I membri destri sono gia' unitari e dunque il primo passo non apporta modifiche a G.
  - Passo 2. Rimuoviamo gli attributi ridondanti da ogni dipendenza.
    L'attributo A e' ridondante in $AB \to C$ sse $C \in B^+$. $B^+ = \{BAC\} \supseteq \{B\}$. A e' dunque ridondante in $AB \to C$ che viene sostituita con $B \to C$.
    L'attributo A e' ridondante in $AD \to E$ sse $E \in D^+$. $D^+ = \{D\}$ $\Rightarrow$ A non e' ridondante in $AD \to E$.
    L'attributo D e' ridondante in $AD \to E$ sse $E \in A^+$. $A^+ = \{A\}$ $\Rightarrow$ D non e' ridondante in $AD \to E$.
    L'attributo B e' ridondante in $BD \to F$ sse $F \in D^+$. $D^+ = \{D\}$ $\Rightarrow$ B non e' ridondante in $BD \to F$.
    L'attributo D e' ridondante in $BD \to F$ sse $F \in B^+$. $B^+ = \{B\}$ $\Rightarrow$ D non e' ridondante in $BD \to F$.
  - Passo 3. Eliminiamo infine le dipendenze ridondanti dall'insieme ottenuto al passo precedente. Per ogni dipendenza $X \to Y$ e' sufficiente verificare se y appartiene alla chiusura di X rispetto ad $F \setminus \{X \to Y\}$

$X \to Y$	X^+ rispetto a $F \setminus \{X \to Y\}$	$X \to Y$ e' ridondante?
$B \to C$	$B^+ = \{BA\}$	No
$B \to A$	$B^+ = \{BC\}$	No
$AD \to E$	$AD^+ = \{AD\}$	No
$BD \to F$	$BD^+ = \{BDCAE\}$	No

L'insieme di DF:

$$\{B \to C, B \to A, AD \to E, BD \to F\}$$

e' dunque una copertura per G. Operando analogamente su H otteniamo la seguente copertura minimale:

$$\{B \to C, B \to A, AD \to E, AD \to F\}$$

(9.2) Stabilire se G ed H sono equivalenti.

- Soluzione. Dobbiamo verificare che g e' coperto da H ed H e' coperto da G. Verifichiamo se G e' coperto da H ovvero $G \subseteq H^+$. Le dipendenze $AB \to C, B \to A, AD \to E$ in G appartengono anche ad H e dunque ad H^+. Vediamo se $BD \to F \in H^+$. $BD \to F \in H^+$ sse $F \in BD^+$ (rispetto ad H). BD^+ rispetto ad H equivale a $\{BDACEF \supseteq \{F\}$. Possiamo dunque concludere che $G \subseteq H^+$. Al fine di provare $H \subseteq G^+$ dobbiamo verificare se $AD \to F \in G^+$. Si ha $F \notin AD^+$ (rispetto a G). Infatti $AD^+ = \{ADE\}$. Dunque $H \not\subseteq G^+$ e G ed H non sono equivalenti.

Esercizio 15. Si considerino lo schema di relazione $R(A, B, C, D, E, F)$ e l'insieme di dipendenze associato: $G = \{A \to B, C \to AD, AF \to EC\}$.

(10.1) Si determinino le chiavi candidate di R.

(10.2) Si stabilisca se R e' in 3NF. Qualora non lo sia, si definisca una decomposizione di R in 3NF che conservi le dipendenze date.

1) F non compare nella parte destra di alcuna DF, dunque appartiene ad ogni chiave candidata. Dunque, A, C, AF e
   AF^+ = AFBDEC = R
   CF^+ = CFADEC = R
   F^+ = F
   e includendo tutto sia AF che CF sono le uniche chiavi candidate.

2) L'unica relazione che non rispetta la 3NF è A → B con A che fa parte di superchiave ma B non è membro di chiave. Calcolando una copertura minimale (facciamo in modo di avere a sinistra un solo attributo):

$$G' = \{A \to B, C \to A, C \to D, AF \to E, AF \to C\}$$

da cui otteniamo la decomposizione: $R1 = (AB), R2 = (CAD), R3 = (AFEC)$.

Esercizio 16. Si considerino lo schema di relazione $R(A, B, C, D, E, G)$ e l'insieme di dipendenze associato: $F = \{E \to D, C \to B, CE \to G, B \to A\}$.

(11.1) Si stabilisca se R e' in 3NF. Qualora non lo sia, si definisca una decomposizione di R in 3NF che conservi le dipendenze date.

(11.2) Se R non e' in BCNF, determinare una decomposizione losslessjoin di R in BCNF.

(11.1) Si stabilisca se R e' in 3NF. Qualora non lo sia, si definisca una decomposizione di R in 3NF che conservi le dipendenze date.

- Soluzione. Determiniamo innanzitutto le chiavi della relazione. C ed E non compaiono a destra in alcuna DF, quindi devono appartenere ad ogni chiave. Si ha:
  - $CE^+ = CEDBGA = R$
  - $C^+ = CBA$
  - $E^+ = ED$

  Dunque CE e' l'unica chiave candidata di R.
  R non e' in $3NF$: La DF $E \to D$ e' tale che E non e' una superchiave e D non e' un attributo primo. Applichiamo dunque ad R l'algoritmo visto a lezione per definire una decomposizione 3NF che preserva le dipendenze.
  - Passo 1. Mediante l'algoritmo apposito, ci assicuriamo che F sia una copertura minimale (lo e').
  - Passo 2. Da $F = \{E \to D, C \to B, CE \to G, B \to A\}$ otteniamo la decomposizione: $R1 = (ED), R2 = (CB), R3 = (CEG), R4 = (BA)$
  - Passo 3. $R3$ contiene una chiave di R e dunque la decomposizione rimane invariata.

(11.2) Se R non e' in BCNF, determinare una decomposizione losslessjoin di R in BCNF.

- Soluzione. La scomposizione e' in BCNF. Infatti:
  - DE,BC,AB sono relazioni binarie e dunque in BCNF
  - R3=CEG rispetta la BCNF poiche' per ogni $X \subseteq \{C, E, G\}$, si ha: X^+ contiene tutti gli attributi di $R3$ oppure X^+ non include attributi di $R3 \setminus X$.

Esercitazione 5: Normalizzazione

Si inseriscono i cenni fondamentali:

Terza Forma Normale

Una relazione R con chiavi $K_1,\ldots, K_n$ è in Terza Forma Normale se:

Per ogni dipendenza funzionale non banale $X \rightarrow Y$, almeno una delle seguenti condizioni sono valide:

- X è superchiave (BCNF)
- ogni attributo in Y è contenuto in almeno una tra le chiavi $K_1,\ldots, K_n$.

Copertura ridotta

- Un insieme di dipendenze F è una copertura ridotta:
  - **non ridondante** se non esiste dipendenza $f \in F$ tale che $F - \{f\}$ implica f;
  - **ridotto** se
    - **non ridondante** se non esiste dipendenza $f \in F$ tale che $F - \{f\}$ implica f;
    - non esiste un insieme F' equivalente a F ottenuto eliminando attributi dai primi membri di una o più dipendenze di F.
- Esempio (parte in rosso rimovibile):
  - {A → B; AB → C; A → C} è ridondante;
  - {A → B; AB → C} non è ridondante né ridotto;
  - {A → B; A → C} è ridotto

I passi per calcolare la copertura ridotta di una relazione sono i seguenti:

1. Sostituzione dell'insieme dato con quello equivalente che ha tutti i secondi membri costituiti da singoli attributi;
2. Per ogni dipendenza verifica dell'esistenza di attributi eliminabili dal primo membro;
3. Eliminazione delle dipendenze ridondanti.

Attenzione. Di seguito alcune osservazioni utili sull'algoritmo e sulla sua esecuzione.
Questo nasce dall'aver fatto tanti esercizi e aver capito cosa, effettivamente, vada fatto.

1) Il primo punto significa scomporre a destra (quindi, dipendenze funzionali che a sinistra possono avere più di un attributo e a destra basta averne uno: easy).

2) Il secondo punto indica che possiamo eliminare attributi dal primo membro e significa che:

   a. Raggiungiamo già un attributo del primo membro in qualche modo sempre attraverso uno degli attributi del primo membro (ad esempio: AD → C avrà D ridondante se ho già A → D)
   b. Alternativamente, si vede se l'attributo a secondo membro (a destra della freccia) della dipendenza funzionale sia contenuto nella chiusura di un attributo a sinistra (primo membro) meno l'attributo che si pensa essere ridondante.
      Primo esempio:
      AD → C A → C
      Allora (questa volta ragioniamo con A)
      D è ridondante se C fosse già contenuto nella chiusura di A $A^+ = \{D, C\}$ $AD^+ = \{A, D, C\}$
      Allora, eliminiamo D; in questo caso avremmo una doppia dipendenza A → C e ne togliamo una.

      Secondo esempio:
      AD → C D → C
      Allora (questa volta ragioniamo con D)
      A è ridondante se C fosse contenuto già nella chiusura di D $D^+ = \{D, C\}$ $AD^+ = \{A, D, C\}$
      Allora, eliminiamo A; in questo caso avremmo una doppia dipendenza D → C e ne togliamo una.

3) Infine, consideriamo che eliminare le dipendenze ridondanti significhi vedere quelle che si raggiungono o sono raggiunte transitivamente e si elimina quella doppia (teniamo "il giro più lungo e non "il più corto").
   Se avessi ad esempio:
   A → D D → C A → C
   Toglierò A → C, in quanto posso già raggiungerlo tramite A → D, D → C (giro più lungo, per come la vedo e spiego io) e togliamo appunto A → C (giro più corto).

Esercizio 1

Data la relazione R(A,B,C,D) con dipendenze funzionali {C→D, C→A, B→C}.

1) Mostrare tutte le chiavi di R e motivare perché ognuna è chiave
2) Dire quali dipendenze violano la BNCF spiegandone la ragione
3) Decomporre in BNCF

1) Partiamo dalle chiusure:
   C^+ = {C, D, A}
   B^+ = {B, C}
   Per proprietà transitiva sappiamo che vale anche B^+ = {B, C, D, A} e similmente B → C → A.
   Siamo quindi certi che la ch. transitiva di B contenga tutti gli attributi.

2) X → Y è BNCF se X è superchiave
   C → D e C → A violano ma B → C no perché B è superchiave

3) Dunque, si formeranno:
   R(A,B,C,D)
   / \
   ~~R_1(C,D)~~ R_2(A,B,C) (tolgo la parte sinistra)
   R_4(C,A,D) / \
   R_2(C,A) R_3(B,C)

   Dato che C è chiave, la decomposizione riporta tutti gli attributi a seguito di join.

Si otterranno quindi come relazioni:
R_1(C, D) R_2(C, A) R_3(B, C)

Esercizio 2

Considerare uno schema di relazione R(E,N,L,C,S,D,M,P,A) con le seguenti dipendenze funzionali:
E → NS
NL → EMD
EN → LCD
C → S
D → M
M → D
EPD → A
NLCP → A
Calcolare una **copertura ridotta** della relazione data e decomporre la relazione in **terza forma normale**.
Passi della copertura:

1) Sostituzione dell'insieme dato con quello equivalente che ha tutti i secondi membri costituiti da singoli attributi

E → NS
NL → EMD
EN → LCD
C → S

Scritto da Gabriel

D → M
M → D
EPD → A
NLCP → A

Risultato:
E → S
E → N
NL → E
NL → M
NL → D
EN → L
EN → C
EN → D
C → S
D → M
M → D
EPD → A
NLCP → A

2) Per ogni dipendenza verifica dell'esistenza di attributi eliminabili dal primo membro (si consiglia di guardare direttamente a destra delle frecce, verificando di controllare subito gli attributi raggiunti per dipendenza transitiva e/o due volte da parte di relazioni)

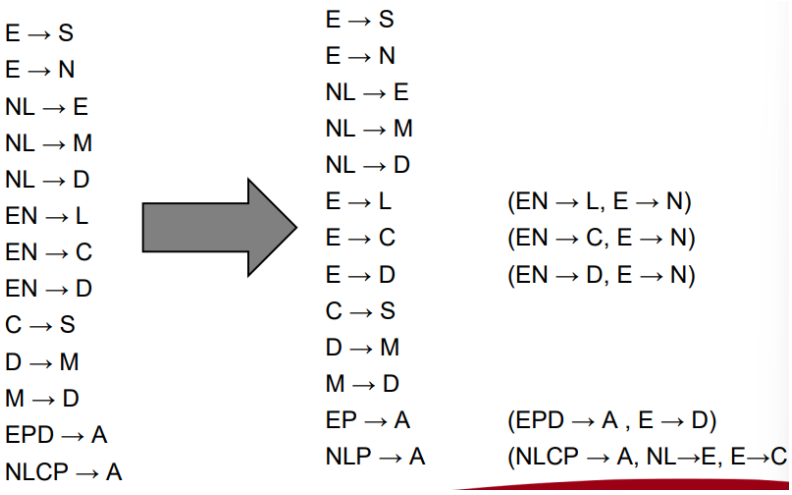
Le prime cinque dipendenze sono a posto, in quanto sono tutte univoche.

- EN → L presenta una ridondanza; si ha infatti E → N ma EN → L e dunque E può andare direttamente ad L
  Quindi EN → L diventa E → L
- EN → C similmente presenta la stessa ridondanza, avendo EN → C ed E → N, dunque E può andare direttamente a C
  Quindi EN → C diventa E → C
- EN → D similmente presenta la stessa ridondanza, avendo EN → D ed E → D, dunque E può andare direttamente a D. Quindi EN → D diventa E → D

Similmente sono a posto anche C → S, D → M, M → D (queste ultime due, anche se si richiamano tra loro, non costituiscono ridondanza).
Però abbiamo:

- EPD → A che presenta E → D (ex EN → D); P non compare da nessuna parte e va solo in A, dunque ci interessa mantenerlo. In particolare, EN → D ma anche EPD → A, dunque raggiungeremmo D due volte. Ecco quindi che l'eliminazione si ha su D, portando ad avere EP → A
- NLCP → A, avendo NL → E ed E → C, dunque andiamo a togliere C in quanto già raggiunto da NL. Dunque, NLCP → A diventa NLP → A.

Risultato finale:



3) Eliminazione delle <u>dipendenze ridondanti</u>

Partendo dallo schema sopra:

- E → S risulta essere ridondante in quanto E → C e C → S
- E → N non è ridondante
- NL → E non è ridondante
- NL → M è ridondante perché NL → D e D → M, oltre che M → D
- E → L non è ridondante
- E → C non è ridondante
- E → D non è ridondante
- C → S non è ridondante
- D → M non è ridondante
- M → D non è ridondante
- EP → A è ridondante perché NL → E ed NLP → A
- NLP → A non è ridondante quindi

Risultato:

E → N
NL → E
E → L
E → C
E → D
C → S
D → M
M → D
NLP → A

Abbiamo la copertura ridotta; tuttavia ora occorre individuare le chiavi partendo dalla copertura ridotta.
Si vanno quindi ad eseguire le chiusure di tutti i membri:
E^+ = {E,N,L,C,D,M,S}
NL^+ = {E,N,L,D,C,S,M}
C^+ = {C,S}
D^+ = {D,M}
NLP^+ = {A,E,P,N,L,E,C,D,M,S}

Per capire chi è chiave dobbiamo capire l'insieme con più attributi; questo chiaramente è NLP.
Esso viene scelto in quanto contiene anche {A,P}, evidentemente non presenti in E ed NL (peraltro sono la stessa chiusura).
Tuttavia, se decido di includere P nell'insieme di E, automaticamente avrò anche A; quindi anche EP sarebbe chiave.

Chiavi:
NLP^+ = {N,L,P,A,E,D,C,S,M}
EP^+ = {E,N,L,D,C,S,M,P,A}

Ora dobbiamo, partendo dalle chiavi NLP - EP, applicare lo *schema della terza forma normale*:

Dati uno schema R(U) e un insieme di dipendenze F su U, con chiavi $K_1, \dots, K_n$

1. Viene calcolata una copertura ridotta G di F
2. G viene partizionato in sottoinsiemi tali che due dipendenze funzionali X→ A e Y → B sono insieme se $X_G^+ = Y_G^+$
3. Viene costruita una relazione per ogni sotto-insieme
4. Se esistono due relazioni S(X) e T(Y) con X ⊆ Y, S viene eliminata
5. Se, per qualche *i*, non esiste una relazione S(X) con $K_i \subseteq$ X, viene aggiunta una relazione T(K_i)

Prendiamo il passo 2:
G è partizionato in sottoinsiemi tali che due DF X→ A e Y → B sono insieme se $X_G^+ = Y_G^+$

Come si diceva prima, in questo caso si parla di E ed NL con le chiusure coincidenti, con il resto che rimane uguale:

E^+ = { E, N, L, C, D, S, M }

C^+ = { C, S }

CHIUSURE COINCIDONO

D^+ = { D, M } ; M^+= { D, M }

NL^+ = { E, N, L, C, D, S, M }

NLP^+ = { E, N, L, C, D, C, S, M, A }

Prendiamo il passo 3:
Viene costruita una relazione per ogni sottoinsieme
R_1 (E,N,L,C,D)
R_2 (C,S)
R_3 (D,M)
R_4 (N,L,P,A)

Prendiamo il passo 4:
Se esistono due relazioni S(X) e T(Y) con X ⊆ Y, S viene eliminata

Non succede nulla, resta tutto invariato, come si vede a fianco.

E → N
E → L
E → C
E → D
NL → E
} R_1 (E, N, L, C, D)

C → S } R_2 (C, S)

D → M
M → D
} R_3 (D,M)

NLP → A } R_4 (N, L, P, A)

Non accade

Andiamo al passo 5:
Se, per qualche i, non esiste una relazione S(X) con $K_i \subseteq$ X, viene aggiunta una relazione T(K_i)

Siccome tra tutte le relazioni, non ne esiste una che comprenda EP (chiave), allora si aggiunge proprio R_5(E,P).

Dunque tra tutte le relazioni, si struttura:
R_1 (<u>E</u>, <u>N,L</u>, C,D) con E chiave ed NL chiave esterna
R_2 (<u>C</u>, S)
R_3 (<u>D</u>, <u>M</u>) con D chiave ed M chiave esterna

Scritto da Gabriel

R_4 (N,L,P, A) con NLP chiave
R_5 (E,P) con EP chiave

Esercizio 3
Dato lo schema R(A, B, C, D, E, F) con dipendenze:
CE → A, C → D, A → B, D → BE, B → F, AD → CF

1) Trovare la copertura ridotta G
2) Trovare tutte le chiavi
3) Dire se ci sono e quali dipendenze violano la 3NF
4) Normalizzare lo schema in 3NF
5) Lo schema normalizzato al punto 4 è anche in BCNF?

1) La copertura ridotta è data da:

$$C \rightarrow A,\ C \rightarrow D,\ A \rightarrow B,\ D \rightarrow B,\ D \rightarrow E,\ B \rightarrow F,\ AD \rightarrow C$$

2) In merito alle chiavi, si vedono le chiusure

$C^+ = \{A, B, C, D, E, F\}$
$AD^+ = \{A, B, C, D, E, F\}$
Dunque, queste sono AD e C

3) Per essere in 3NF si ha che per ogni FD X → Y si abbia che:
   - X contiene chiave K di r
   - ogni attributo di Y è contenuto in almeno una chiave di r

   Dunque, per ogni dipendenza:
   - C → A non viola la dipendenza perché C è chiave
   - C → D non viola la dipendenza perché C è chiave
   - A → B viola perché A non è super chiave e B non è presente nella chiave
   - D → B viola perché D non è super chiave e B non è presente nella chiave
   - D → E viola perché D non è super chiave ed E non è presente nella chiave
   - B → F viola perché B non è super chiave ed F non è presente nella chiave
   - AD → C non viola perché AD è chiave

   Si conclude che lo schema non sia in 3FN

4) Per normalizzare lo schema partiamo dalle chiusure:
   $C^+ = \{A, B, C, D, E, F\}$
   $A^+ = \{A, B, F\}$
   $D^+ = \{D, B, E, F\}$
   $B^+ = \{B, F\}$
   $AD^+ = \{A, B, C, D, E, F\}$

Come si discuteva prima, le chiusure di C e AD coincidono e fanno parte della stessa partizione.
Ciò comporta la creazione delle successive relazioni:

R_1 = {A, C, D} con chiavi C, AD
R_2 = {A, B} con chiave A
R_3 = {B, E, D} con chiave D
R_4 = {B, F} con chiave B

5) Dato che tutte le dipendenze sono parte di chiave, non violano la BNCF.
Nel dettaglio:

C→A, C→D, AD→C	R1 (C, A, D)	chiavi C, AD
A→B,	R2 (A, B)	chiave A
D→B, D→E,	R3 (B, E, D)	chiave D
B→F,	R4 (B, F)	chiave B

Tutte le dipendenze funzionali non violano BCNF:

- C→A e C→D si applicano su R1 dove C è chiave
- AD→C si applica su R1 dove AD è chiave.
- A→B si applica su R2 dove A è chiave
- …

Concludiamo dunque con:

R (A, B, C, D, E, F, **G**)
con:
AF→BE, EF→BCD , A→F, B→C

1. Trovare copertura ridotta

   A→B, A→E, EF→B, EF→D, A→F, B→C

2. Trovare tutte le chiavi

   A^+ contiene tutti gli attributi, inoltre G è già una chiave

   Quindi le chiavi sono: A, G

3. Dire se ci sono e quali dipendenze violano 3NF

4. Normalizzare in 3NF

2) Come si vede la chiave dalle chiusure:
A^+ = {A,B,C,D,E,F}
EF^+ = {B,D}
A è chiave di sicuro e l'esercizio ci dà G come se fosse tale

3)

A→B, non viola 3NF
A→E, non viola 3NF
EF→B, viola (EF non super chiave e B non presente in chiave)
EF→D, viola (EF non super chiave e D non presente in chiave)
A→F, non viola 3NF
B→C viola (B non super chiave e C non presente in chiave)

Con chiavi: A, G

Non è in 3NF

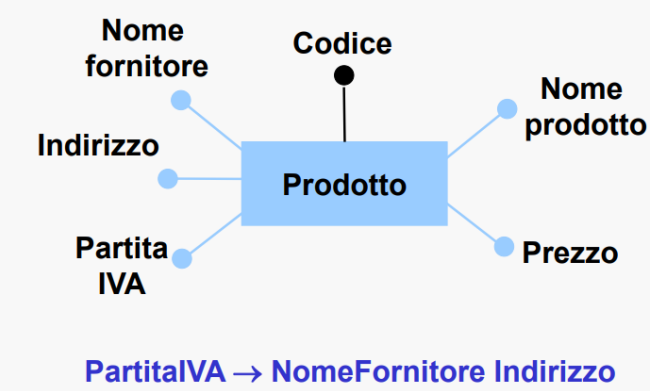
4) Abbiamo già la copertura ridotta e si ha un partizionamento di questo tipo:
{A→B, A→E, A→F}, {EF→B, EF→D}, {B→C}
Si costruiscono le relazioni per ogni sottoinsieme
R_1 (A,B,E,F)
R_2 (E,F,B,D)
R_3 (B, C)

Se esistono due relazioni S(X) e T(Y) con $X \subseteq Y$, S viene eliminata.
Ciò non accade e rimane tutto uguale

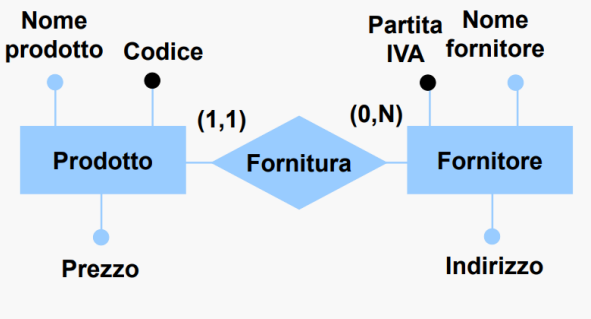
Se, per qualche i, non esiste una relazione S(X) con $K_i \subseteq X$, viene aggiunta una relazione $T(K_i)$
Nessuna relazione contiene G e viene aggiunta una relazione:

$R_1(A, B, E, F), R_2(E, F, B, D), R_3(B, C), \mathbf{R_4(G)}$

La teoria della normalizzazione può essere usata per verificare lo schema relazionale finale verificando la qualità dello schema concettuale. Ad esempio su:

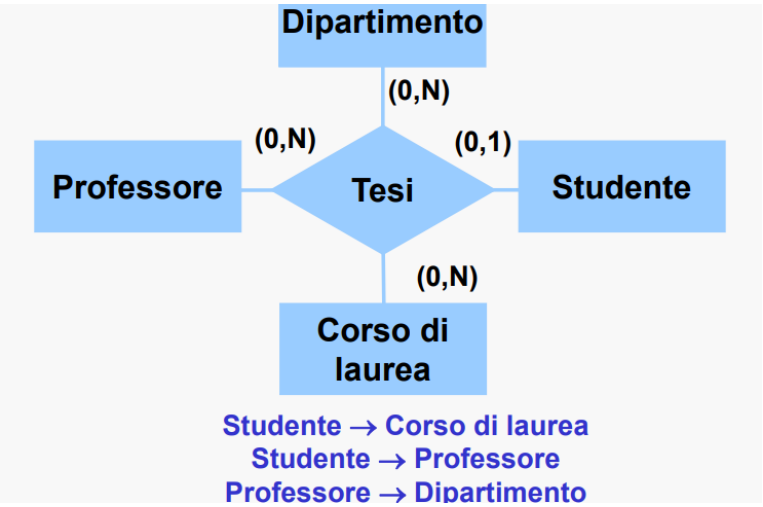


L'entità viola la forma normale a causa della dipendenza: *PartitaIVA → NomeFornitore Indirizzo*

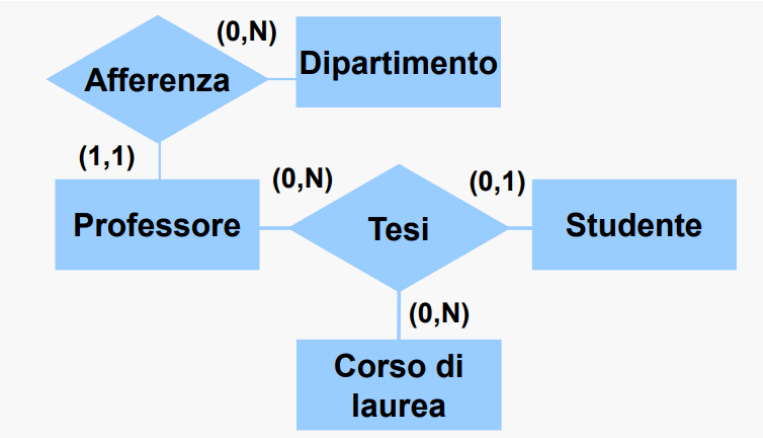


E decomponiamo sulla base di questa:

Lo Studente sarà chiave primaria nella trasformazione dello schema concettuale che passa a logico; cominciamo ad eseguire la ristrutturazione:



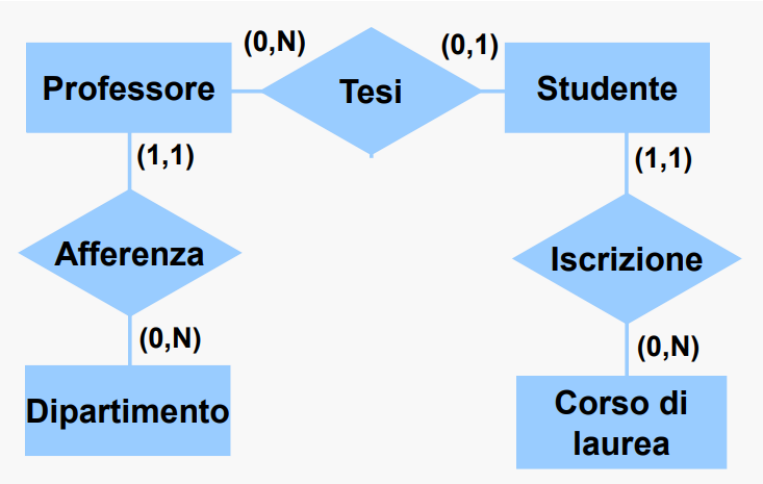
La dipendenza che viola la 3NF è *Professore → Dipartimento* e decomponiamo da qui:



Similmente Tesi è in BNCF sulla base delle dipendenze:

$$\text{Studente} \rightarrow \text{CorsoDiLaurea}$$
$$\text{Studente} \rightarrow \text{Professore}$$

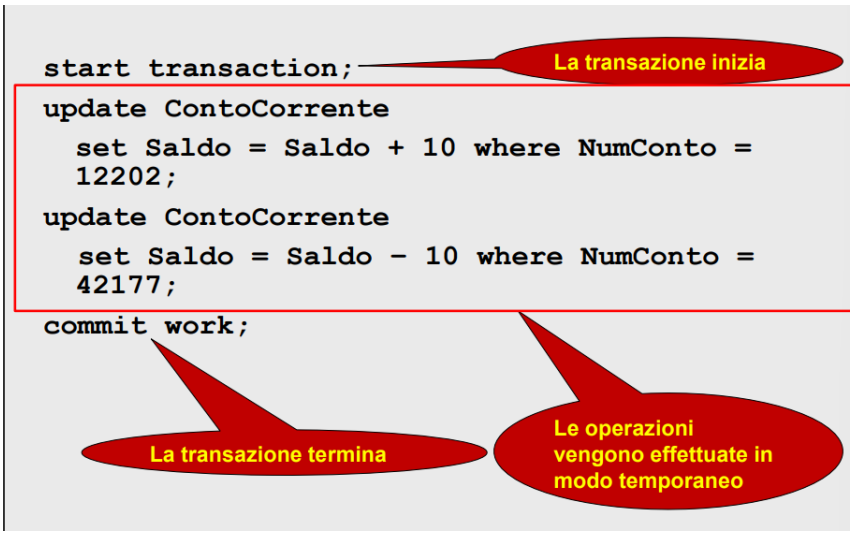
Le proprietà sono indipendenti e ciò suggerisce una decomposizione ulteriore:



L'idea è che la forma normale di Boyce-Codd permetta di fare *reverse engineering* allo schema logico e successivamente relazionale di una certa base di dati, nel concreto utile per ristrutturazione.
Si parla ora delle *transazioni*, parte di programma caratterizzata da:

- inizio di transazione (*begin-transaction*)
- corpo di transizione (serie di insert/delete/update in SQL)
- fine di transazione (*end-transaction*) che porta a:
  1. *commit work:* terminazione corretta e che rende i cambiamenti definitivi
  2. *rollback work/abort*: abortire la transazione

Un sistema transazione *OLTP* è dunque in grado di definire ed eseguire transazioni per un certo numero di applicazioni concorrenti. Un esempio di transazione:

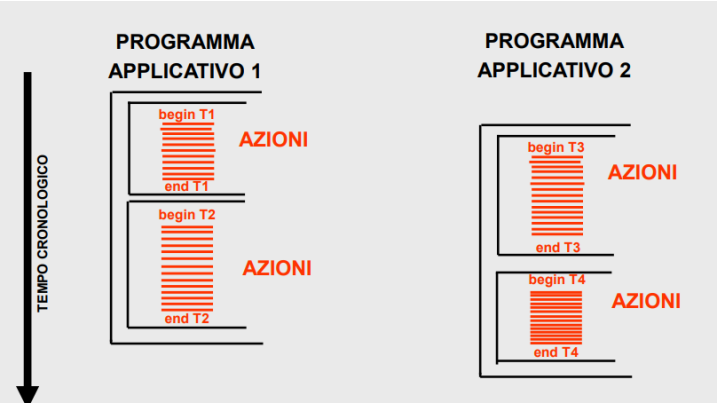


Normalmente il sistema di gestione delle transazioni, se ci fosse un'interruzione od un blackout, previene la scorretta esecuzione ed applicazione delle singole operazioni.

```
start transaction;
  update ContoCorrente
  set Saldo = Saldo + 10
  where NumConto = 12202;
  update ContoCorrente
  set Saldo = Saldo - 10
    where NumConto = 42177;
  select Saldo into A
     from ContoCorrente
     where NumConto = 42177;
  if (A>=0)
     then commit work
     else rollback work;
```

Potremmo decidere di eseguire un *update* del conto corrente, togliendoli da un certo conto corrente e verifichiamo che il saldo non sia negativo; nel qual caso viene eseguito il commit, altrimenti viene fatto il rollback.

La struttura grafica segue una serie di azioni eseguite in ordine cronologico (esempio su 2 programmi applicativi):



Naturalmente, se non viene eseguito il *commit*, le transazioni non vedono (e non devono vedere) i cambiamenti fatti. Ciò segue il principio *ACID:*

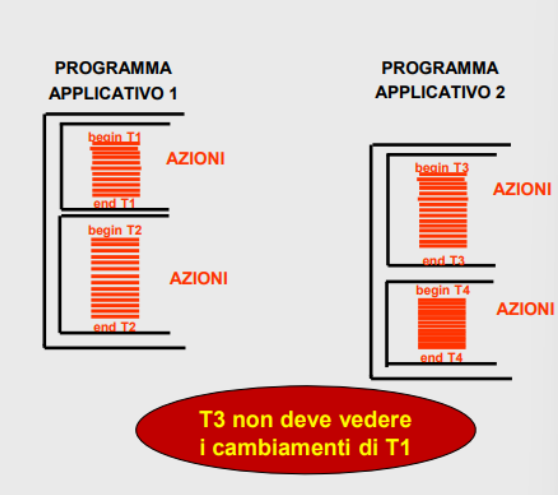
1) *Atomicity/Atomicità*: La transazione o è fatta interamente o per nulla. La base di dati non può essere in uno stato intermedio. Un guasto o un errore:
   - prima del commit, deve causare l'UNDO delle operazioni svolte
   - dopo il commit non deve avere conseguenze e, se necessario, le operazioni svolte vanno ripetute con REDO.

Esempio (Agenzia Viaggi):

1. Acquisto Biglietto Roma – New York
2. Acquisto Biglietto New York – Roma
3. Prenotazione hotel a New York

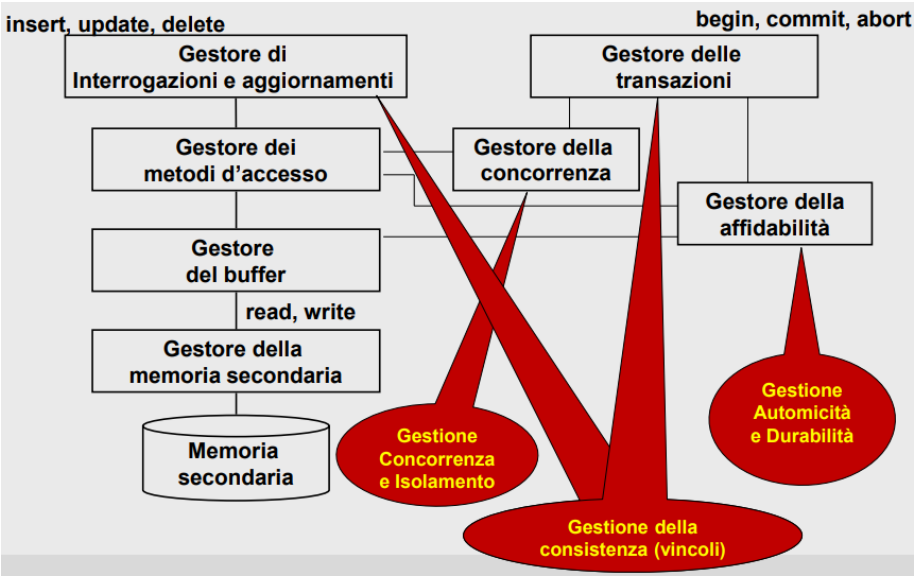
– Se non si riescono ad acquistare entrambi i biglietti e a prenotare hotel, allora occorre fare il "rollback" di tutto!

2) *Consistency/Consistenza:* La transazione rispetta i vincoli del DB (chiave, check, valori, ecc.). I vincoli vanno verificati alla fine della transazione, dato che la violazione si ha nel mentre. Se i vincoli sono violati nello stato finale, l'unica operazione fattibile è la rollback.
3) *Isolamento/Isolation*: La transazione non risente degli effetti delle altre transazioni concorrenti (l'esecuzione concorrente deve produrre un risultato paragonabile ad una esecuzione sequenziale). Di conseguenza, una transazione non espone i suoi stati intermedi:



4) *Durabilità/Durability*, in ogni caso gli effetti di un commit di una transazione non vanno perduti e anche in presenza di guasti di sistema (dispositivo, termini hardware, che si rompe) o di sistema (applicazione, termini software, che va in crash)

A livello architetturale in maniera sintetica:



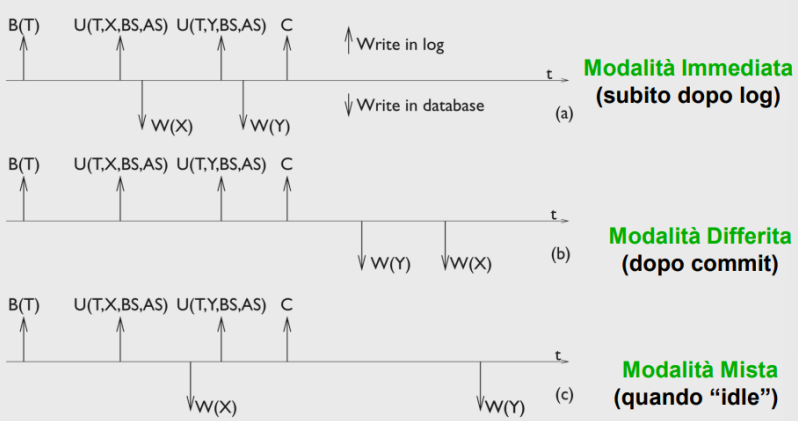
L'affidabilità assicura atomicità e durabilità e gestisce l'esecuzione dei comandi transazionali (start, commit, rollback), usando un log per archiviare permanentemente delle operazioni svolte. Esso è memorizzato in una memoria stabile che non può danneggiarsi (es. anche sistemi di consistenza, tipo RAID/nastri, ecc.) o anche sistemi distribuiti. Il *log* è descritto come file sequenziale gestito dal controllore dell'affidabilità che riporta tutte le operazioni in ordine. Il record nel log ha questa struttura:

- *operazioni delle transazioni*
  - **B(T):** begin transazione T,
  - **I(T,O,AS):** T inserisce l'oggetto O con valore AS,
  - **D(T,O,BS):** T cancella l'oggetto O con valore BS,
  - **U(T,O,BS,AS):** T aggiornata il valore dell'oggetto O da BS a AS
  - **C(T):** commit transazione T
  - **A(T):** abort transazione T
- *record di sistema*
  - **dump**
  - **checkpoint**

Nel log si hanno due regole:

- *write-ahead-log*, scrivendo il log prima del database
- *commit-precedenza*, scrivendo il log prima del commit e consente di rifare le azioni

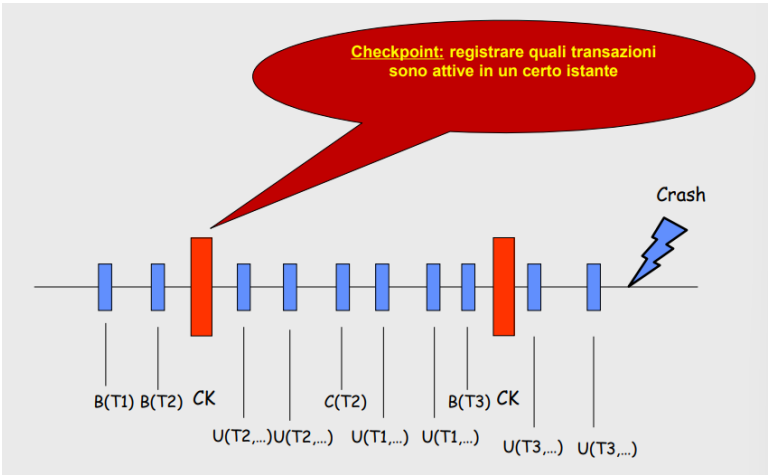
La scrittura segue questa logica (magari di tanto in tanto, appena per il DB è fattibile):



In generale quindi:

- Modalità Immediata, il DB contiene i valori AS (cioè i nuovi valori aggiornati) provenienti da transazioni uncommitted. Richiede Undo delle operazioni di transazioni uncommited al momento del guasto.
- Modalità Differita, il DB non contiene valori AS provenienti da transazioni uncommited. In caso di abort, avendo già eseguito il commit, non occorre fare nulla.
- Modalità Mista, in cui la scrittura può avvenire in modalità sia immediata che differita e consente l'ottimizzazione delle operazioni di Flush.

La struttura del log è composta dalla seguente struttura, utilizzando anche dei *checkpoint*, descrivendo le transazioni attive (*begin*), senza aver fatto il *commit*.



Il sistema, in fase di *checkpoint*, "fa il punto" delle transazioni attive e registra quelle presenti fino a quel momento. Normalmente si sospende l'accettazione di richieste di ogni tipo, si trasferiscono in memoria di massa (tramite *force*) tutte le pagine sporche relative a transazioni in commit, registrando sul log (sempre tramite *force*) in modo sincrono quelle ancora in corso, riprendendo in ultimo l'accettazione delle operazioni. L'obiettivo è quindi la classificazione in:

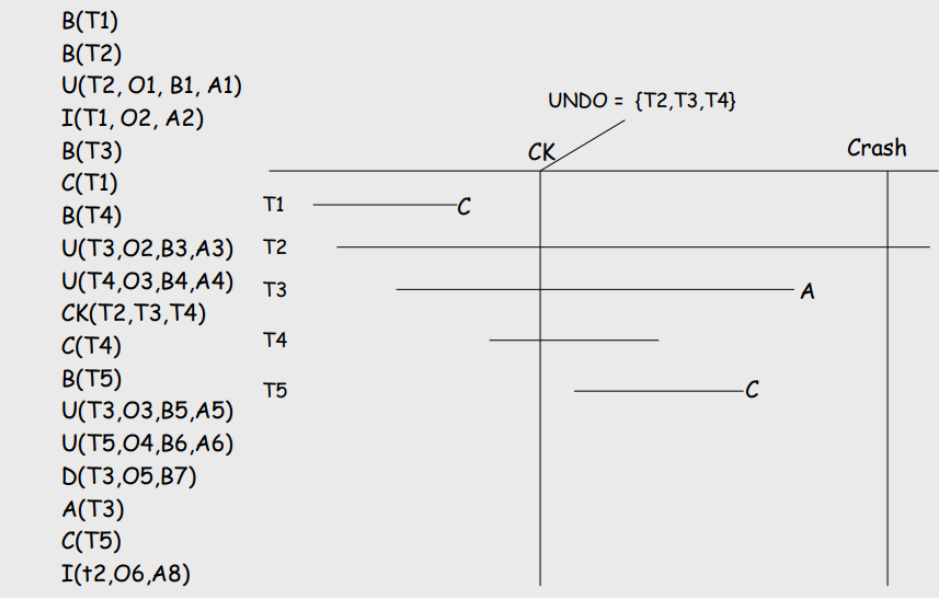
- completate (in memoria stabile tutti i dati)

- in commit ma non per forza completate (può servire *REDO*)
- senza commit (vanno annullate, *UNDO*)

La ripresa a caldo quindi:

- trova l'ultimo checkpoint ripercorrendo il log a ritroso
- costruisce gli insiemi *UNDO/REDO*
- ripercorre il log all'indietro fino alla più vecchia fra le transazioni *UNDO/REDO*, disfacendo tutte le azioni delle transazioni in *UNDO*
- ripercorre il log in avanti, rifacendo tutte le azioni delle transazioni in *REDO*

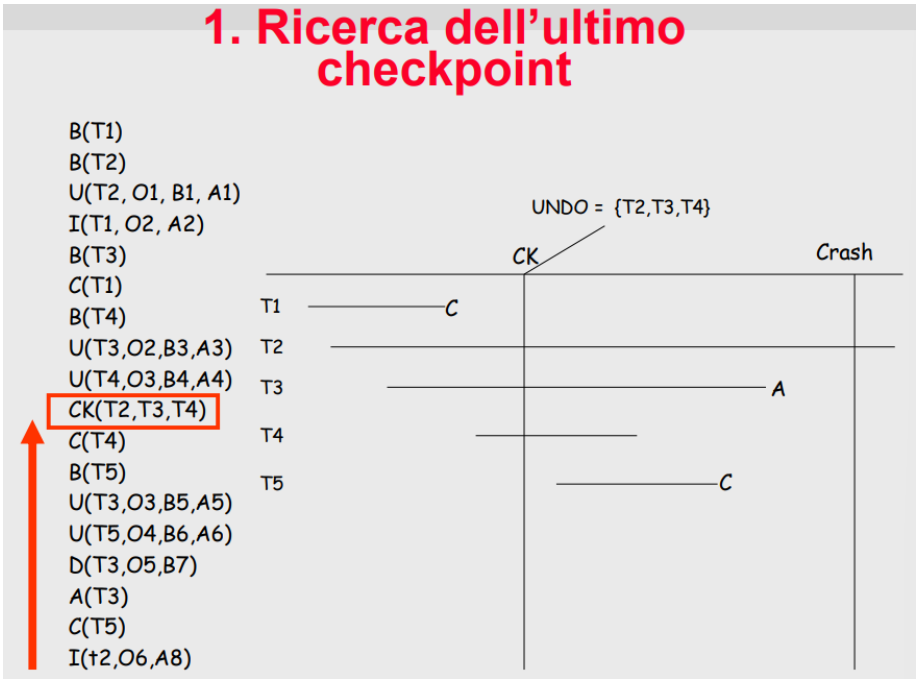
Un esempio pratico:



Glossario iniziali:

I = Insert
D = Delete
B = Begin
C = Commit
U = Update
A = Abort
CK = Check

1) Si cerca l'ultimo checkpoint, partendo da sotto e arrivando a:



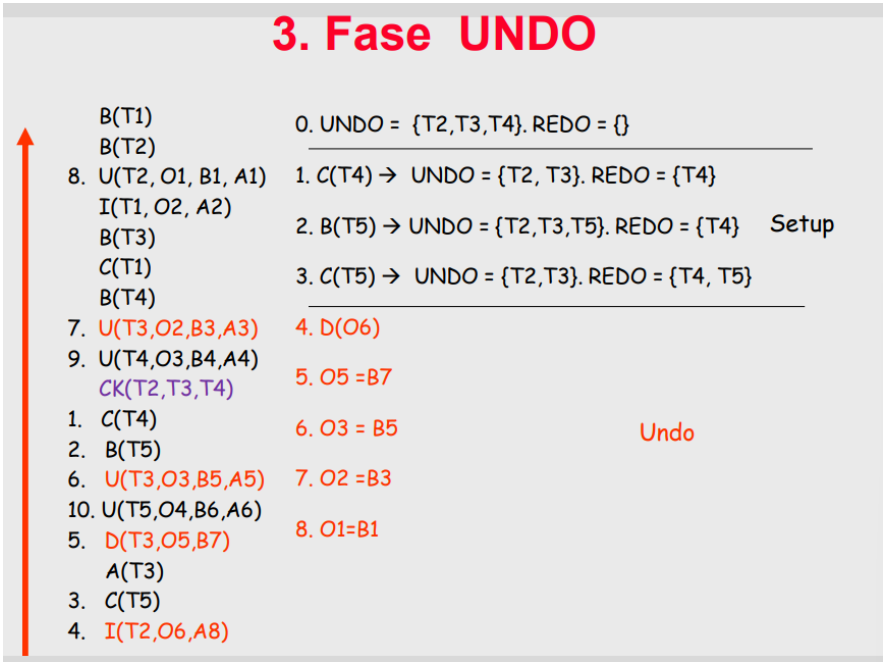
2) Si costruiscono gli insiemi UNDO e REDO (nota: questi due insiemi si costruiscono partendo dal basso e arrivando al primo CK/CHECK. Le operazioni vanno comunque rifatte per UNDO e REDO su tutto il file, ma si considerano *solo* le operazioni che coinvolgono questi due insiemi. Quindi, se devo costruire UNDO/REDO non mi serve scorrere tutto il file di log, ma solo arrivare all'ultimo checkpoint, partendo appunto dal basso)
All'interno di UNDO andremo ad inserire:

Scritto da Gabriel

- inizialmente T2 e T3, dato che su di esse pone controllo il check
- si ha il commit di T4 in C(T4) e la transazione viene posta in REDO
- qualsiasi altra operazione (D/A/I/U) prevede che la transazione rimanga in UNDO; è il caso di T5 che inizia (B/BEGIN), di T3 che aggiorna l'oggetto O3 in B5, di T5 che aggiorna l'oggetto O4 in B6, la cancellazione dell'oggetto O5 in B7 e l'abort di A3 (che fanno in modo T2,T3,T5 rimangano in UNDO
- si ha il commit di T5 e alla fine si rimane con:
  C(T5) → UNDO = {T2,T3}. REDO = {T4, T5}

3) Dato l'insieme degli UNDO, si ripercorre dalla fine all'inizio tutto il log, rifacendo le operazioni che riguardano T2 e T3. Quindi:

- si ha un INSERT di T2 su O6; questo dovrà essere cancellato (quindi quando si ha un INSERT si ha la cancellazione, per permettere di rifarla → D(O6)
- si ha il DELETE di O5 in B7 e dunque → O5 = B7 (che era lo stato precedente, ma è la convenzione che si adotta)
- si ha un UPDATE DI T3 su O3 e → O3=B5
- prima del CHECK si ha un UPDATE di T3 su O2 e → O2=B3
- finalmente, si ha un UPDATE di T2 su O1 in B1 → O1=B1

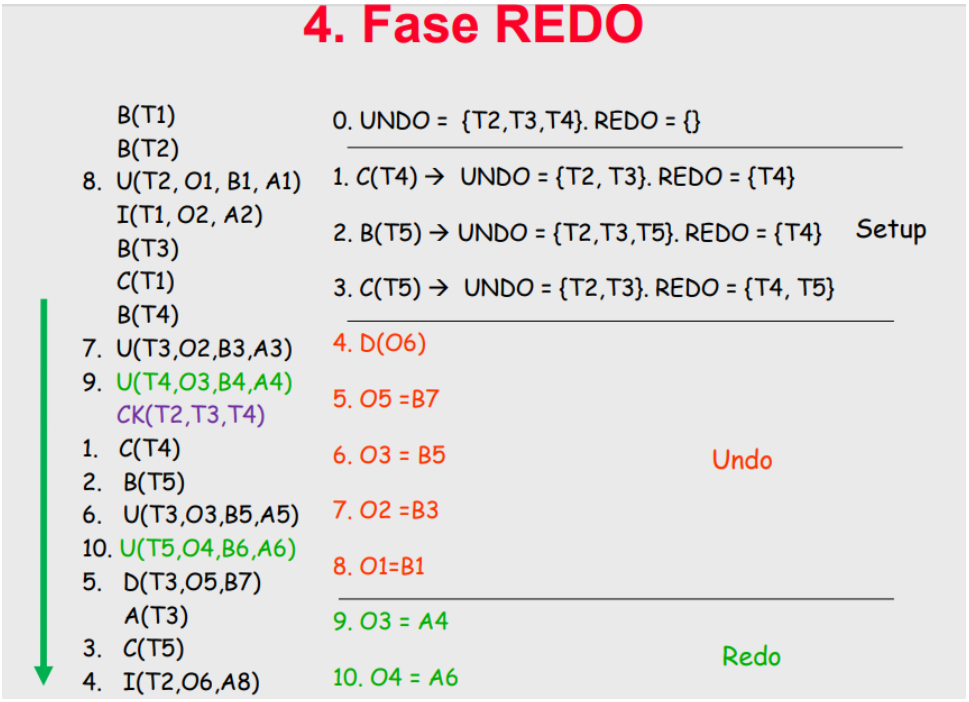


4) Per la fase di REDO, si fa la stessa cosa al contrario, dunque partendo dall'inizio e arrivando alla fine.

Scritto da Gabriel

Sull'insieme si ha da considerare T4 e T5
Si nota che le operazioni che li riguardano sono:

- U(T4,O3,B4,A4) e quindi porta a dire O3 = B4
- U(T5,O4,B6,A6) e quindi porta a dire O4 = A6



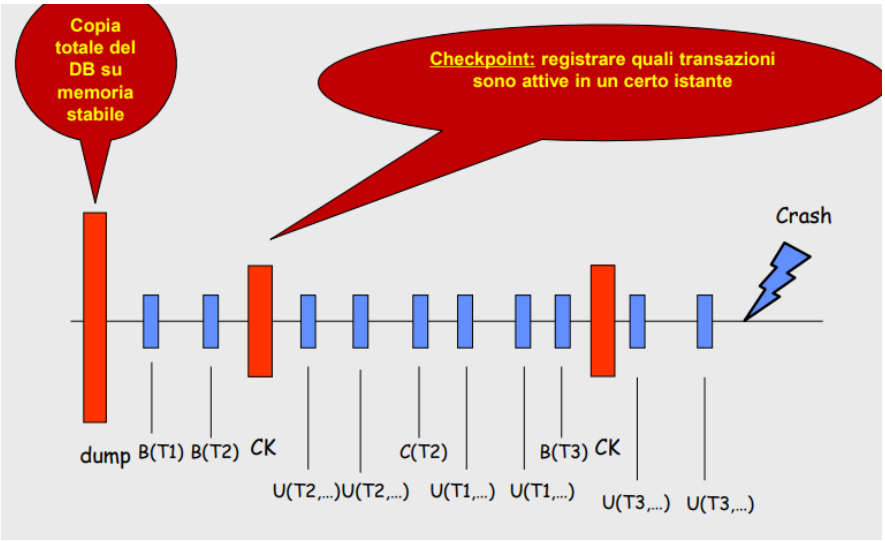
Sempre seguendo l'ordine, tiene traccia di quali operazioni rifare.

Se il DB è scritto in Modalità Immediata (immediatamente dopo il log), il REDO non è necessario.
Invece se il DB è scritto in Modalità Differita (solo dopo un commit), l'UNDO non è necessario.
È possibile venga fatto l'UNDO oppure il REDO di operazioni con effetto nullo nel database, come se fosse fatto una sola volta (ad esempio: undo(undo(A)) = undo(A) redo(redo(A)) = redo(A)

In merito ai *guasti* abbiamo:

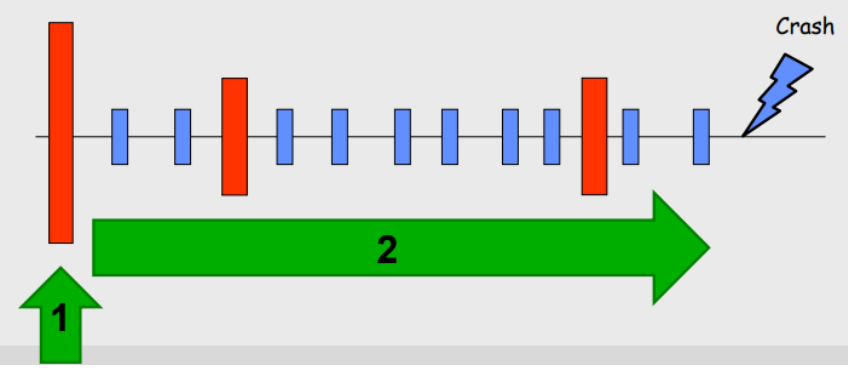
- *guasti di sistema (soft)*, intendendo errori di programma, crash di sistema, caduta di tensione; si opera la ripresa a caldo (*warm restart*), ripetendo le transazioni
- *guasti di dispositivo (hard)*, intendendo guasti su dispositivi di memoria secondaria; tuttavia si ha ridondanza su varie memorie stabili del log, non perdendo la memoria stabile

La struttura dei guasti, oltre al checkpoint, periodicamente esegue dei *dump* (copie del database, ad esempio su nastro) di tutto il contenuto del DB. Grazie a questo evito di dover ogni volta ricominciare.



La ripresa a freddo:

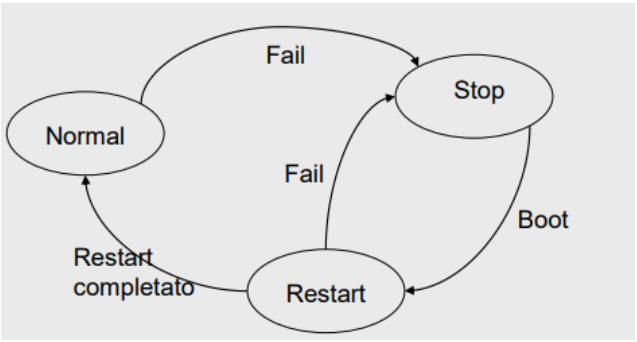
1) ripristina i dati partendo dal backup
2) esegue le operazioni registrare sul log fino all'istante del guasto
3) esegue una ripresa a caldo



Il dump esegue una copia completa della base di dati e viene fatta mentre il sistema non è operativo. Dunque, i dati sono salvati in memoria stabile ed un record di dump indica il momento in cui il log è stato effettuato.

Il modello seguito è il "fail-stop":

- si va in "stop" quando c'è un problema e il DBMS viene fatto ripartire (boot)
- se si ha una failure durante "boot", si va di nuovo in "stop"
- se il "boot" è completo, si fa una *warm/cold restart*



Il problema di Atomicità e Persistenza è garantito da Commit/Abort e Warm/Cold Restart. Normalmente si agisce in maniera concorrente (es. tramite semafori). Questa pratica è fondamentale in maniera tale da avere decine/centinaia di transizioni al secondo in parallelo.

Il modello delle transazioni è costituito da un insieme di input-output su oggetti astratti x, y, z:

- Transazione t1 : r(x), x = x + 1, w(x)
- Transazione t2 : r(x), y = x + 1, w(y)

Scritto da Gabriel

Problema: Anomalie dettate dall'esecuzione concorrente
Classico esempio di transazioni identiche partendo da $x = 2$.

- **t1 :** $r(x), x = x + 1, w(x)$
- **t2 :** $r(x), x = x + 1, w(x)$

L'esecuzione seriale produce $x = 4$.
L'esecuzione concorrente (a destra) produce $x = 3$.

t_1	t_2
bot	
$r_1(x)$	
$x = x + 1$	
	bot
	$r_2(x)$
	$x = x + 1$
$w_1(x)$	
commit	
	$w_2(x)$
	commit

Dunque, le operazioni generalmente portano ad una perdita di aggiornamento, portando alcune operazioni ad essere eseguite dopo.
T1 va a modificare x solo quando va a fare W1(X).
Questo perché quando T1 fa X = X + 1 è solo un dato calcolato da T1, ma non è stato effettivamente ancora scritto. Per questo motivo quando R2(X) va a leggere X, trova ancora il valore iniziale, cioè 2, poi anche T2 fa il suo calcolo con la X appena letta. L'esecuzione continua con W1(X) che va a sovrascrivere X con 3, facendo poi il commit e successivamente anche T2 conclude, ma va a scrivere sempre 3 perché era il valore di X che aveva calcolato.

Vediamo due casi di *letture inconsistenti*:

t_1	t_2
bot	
$r_1(x)$	
$x = x + 1$	
$w_1(x)$	
	bot
	$r_2(x)$
abort	
	commit

Si parte da $x = 2$; la transazione legge $x = 3$ e dall'altra parte leggo lo stesso dato. In quel momento viene eseguito un *abort* durante la transazione e si ha una *lettura sporca*.
Dunque: t_2 ha letto uno stato intermedio ("sporco") e si ha necessità dello stesso *scheduling*. Le transizioni *devono* essere correttamente *serializzate*.

Successivamente:

t_1	t_2
bot	
$r_1(x)$	
	bot
	$r_2(x)$
	$x = x + 1$
	$w_2(x)$
	commit
$r_1(x)$	
commit	

Si parte sempre da $x = 2$; la seconda lettura porta ad avere $x = 3$.
Se t_1 non cambia valore, non può leggere la stessa tupla con due valori diversi.
Dunque: la transazione deve avvenire nella sequenza giusta.
Infatti, t_1 legge due volte e legge ben due valori diversi

Vediamo invece un esempio di *aggiornamento fantasma*:

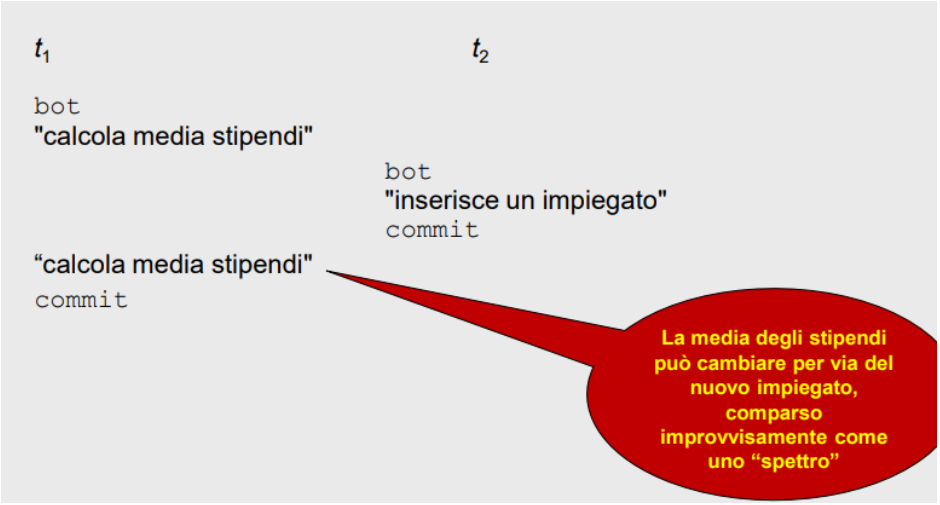
Assumiamo vincolo $y + z = 1000$:

t_1	t_2
bot	
$r_1(y)$	
	bot
	$r_2(y)$
	$y = y - 100$
	$r_2(z)$
	$z = z + 100$
	$w_2(y)$
	$w_2(z)$
	commit
$r_1(z)$	
$s = y + z$	
commit	

$s = 1100$: il vincolo sembra non soddisfatto, t_1 vede un aggiornamento non coerente

L'inserimento fantasma è una versione particolare delle altre anomalie; il valore restituito da una transazione è influenzato dalle altre e ciò non deve accadere.

Scritto da Gabriel



Riassumendo:

Anomalia	Quando potenzialmente accade
Perdita di aggiornamento	Due transazioni scrivono lo stesso dato
Lettura sporca	Transazione legge un dato scritto da un'altra transazione che poi ha abortito
Letture inconsistenti	Transazione legge lo stesso dato in due momenti ma la seconda volta legge un dato aggiornato da un'altra transazione
Aggiornamento fantasma	Un dato appare "improvvisamente" aggiornato
Inserimento fantasma	Un nuovo dato appare "improvvisamente"

Le transazioni possono essere definite come *read-only*; il livello di isolamento può essere scelto per ogni transazione: *read uncommitted, read committed, repeatable read, serializable*.
In sintassi SQL: *BEGIN TRANSACTION ISOLATION LEVEL [valore];*

Anomalia	RU	RC	RR	S
Perdita di aggiornamento	X	X	X	X
Lettura sporca	-	X	X	X
Letture inconsistenti	-	-	X	X
Aggiornamento fantasma	-	-	X	X
Inserimento fantasma	-	-	-	X

X = il livello di isolamento garantisce l'assenza dell'anomalia

Non sempre la gestione è serializable, dato che definire un livello che dà più garanzie richiede più risorse e blocca le transazioni; occorre dunque definire il livello che serve in funzione delle operazioni della transazione.

Transazioni: view/conflict serializzabili/grafo dei conflitti

Si individua lo *schedule* come sequenza di operazioni di I/O di transazioni concorrenti. Ad esempio:

$S_1 : r_1(x)\ r_2(z)\ w_1(x)\ w_2(z)$

che va intesa come (lettura della transizione 1 su X, lettura della transizione 2 su z, scrittura della tr. 1 su x, scrittura della tr. 2 su z)

Ignoriamo quindi le transazioni che vanno in abort, rimuovendo tutte le loro azioni dallo schedule (*commit-proiezione*). Il controllo della concorrenza e delle transazioni è dato dallo *scheduler*, eseguendo o riordinando le operazioni in un certo modo.
Esso può essere:

- seriale, transazioni separate una alla volta

  $S_2 : r_0(x)\ r_0(y)\ w_0(x)\ r_1(y)\ r_1(x)\ w_1(y)\ r_2(x)\ r_2(y)\ r_2(z)\ w_2(z)$

- serializzabile, produce lo stesso risultato di uno schedule seriale

Se sono equivalenti, lo schedule può essere definito buono.
Lo schedule viene costruito nel mentre si eseguono le transazioni e quando una transazione fa il commit/abort, le operazioni sono rimosse dallo schedule e, se una di queste producesse uno schedule non serializzabile, l'operazione viene messa in attesa finché la sua esecuzione non viola la serializzabilità (magari venendo posticipata).
Si danno due definizioni preliminari:

- $r_i(x)$ ***legge-da*** $w_j(x)$ in uno schedule S se $w_j(x)$ precede $r_i(x)$ in S e non c'è $w_k(x)$ fra $r_i(x)$ e $w_j(x)$ in S
- $w_i(x)$ in uno schedule S è ***scrittura finale*** *se* è l'ultima scrittura dell'oggetto x in S

- Esempio: $S_3 : w_0(x)\ r_2(x)\ r_1(x)\ w_2(x)\ w_2(z)$
  - $r_2(x)$ legge da $w_0(x)$
  - $r_1(x)$ legge da $w_0(x)$
  - *Scritture finali:* $w_0(x)$, $w_2(x)$, $w_2(z)$

Diremo che due schedule sono *view-equivalenti* $(S_i \approx_V S_i)$ se hanno la stessa relazione *legge-da* e le stesse scritture finali. Nell'esempio che segue, tutti leggono da w_0 e le scritture finali sono le stesse, quindi le schedule sono view-equivalenti:

$S_3 : w_0(x)\ r_2(x)\ r_1(x)\ w_2(x)\ w_2(z) \approx_V$
$S_4 : w_0(x)\ r_1(x)\ r_2(x)\ w_2(x)\ w_2(z)$

Uno schedule è *view-serializzabile* (VSR) se è view-equivalente ad un qualche schedule seriale.
Esempio: S_3 è view-serializzabile perché view-equivalente con S_4 che è seriale.
Le garanzie che fornisce sono l'assenza di:

- Perdita di Aggiornamento
  $S_7 : r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)$
- Letture Inconsistanti
  $S_8 : r_1(x)\ r_2(x)\ w_2(x)\ r_1(x)$
- Aggiornamento Fantasma
  $S_9 : r_1(x)\ r_1(y)\ r_2(z)\ r_2(y)\ w_2(y)\ w_2(z)\ r_1(z)$

S_7, S_8, S_9 sono tutte non view-serializzabili.

Per esempio, posso decidere di fare tutte le operazioni della transazione 1, oppure tutte le operazioni della transazione 2. A tale scopo, se fossero equivalenti, si avrebbe lo stesso ordine di esecuzione su:

- $r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)$
- $r_2(x)\ w_2(x)\ r_1(x)\ w_1(x)$ (Questo legge (con r_1) da w_2 mentre S_7 no

Scritto da Gabriel

Inoltre, la verifica della view-equivalenza di due dati schedule è lineare sulla lunghezza dello schedule e decidere sulla view serializzabilità di uno schedule S è un problema "difficile" perchè occorre provare tutte i possibili schedule seriali, ottenuti per permutazioni dell'ordine delle transazioni (coefficiente binomiale).

Introduciamo poi la *conflict-serializzabilità*:

- Un'azione a_i è in *conflitto* con a_j ($i \neq j$), se operano sullo stesso oggetto e almeno una di esse è una scrittura. Due casi:
  - conflitto *read-write* (*rw* o *wr*)
  - conflitto *write-write* (*ww*).

Schedule conflict-equivalenti ($S_i \approx_C S_j$): includono le stesse operazioni e ogni coppia di operazioni in conflitto compare nello stesso ordine in entrambi

Dunque, ad esempio qui le operazioni non devono essere invertite, in particolare:
S_1 : $r_1(x)$ $w_1(x)$ $w_2(x)$
S_2: $w_2(x)$ $r_1(x)$ $w_1(x)$
Quelli segnati in giallo rappresentano i conflitti, dato che operano tutti su *x*.

Ad esempio, anche $r_1(x)$ e $w_2(x)$ sono in conflitto.
Oppure similmente $w_2(x)$ e $r_1(x)$ sono in conflitto.

1_a 2_a
S_1 : $r_1(x)$ $w_1(x)$ $w_2(x)$
S_2: $w_2(x)$ $r_1(x)$ $w_1(x)$
1_a 2_a

L'oggetto è la tupla; di fatto che l'operazione nella seconda riga, legge un valore diverso rispetto al valore prodotto in origine. Devono leggere gli stessi attributi sulla stessa tupla; se hanno un diverso attributo sono due oggetti diversi. Quindi semplicemente i conflitti devono essere nello stesso ordine di apparizione.

Uno schedule è *conflict-serializable* se è conflict-equivalente ad un qualche schedule seriale.
Inoltre, l'insieme degli schedule conflict-serializzabili si indica con **CSR.**
Inoltre:

- Ogni schedule conflict-serializzabile (CSR) è view-serializzabile (VSR)
- Ci sono schedule view-serializzabili (VSR) che non sono conflict-serializzabili (CSR)
- CSR implica VSR
- VSR non implica CSR

Esempio per dimostrare che VSR ⇏ CSR

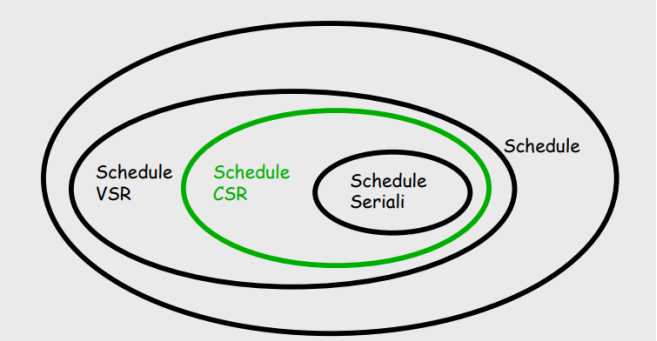
$S_1 : r_1(x)\ w_2(x)\ w_1(x)\ w_3(x)$

- VSR: View-equivalenza: $S_1 \approx_V r_1(x)\ w_1(x)\ w_2(x)\ w_3(x)$
- non CSR
  - $r_1(x)\ w_1(x)\ w_2(x)\ w_3(x)$ inverte $w_1(x)$ e $w_2(x)$
  - $w_2(x)\ r_1(x)\ w_1(x)\ w_3(x)$ inverte $r_1(x)$ e $w_2(x)$

Dato che le operazioni complessivamente compaiono nello stesso ordine, allora sono view-serializzabili. Si ha invece un ordine diverso rispetto ai conflitti, in quanto come spiegato qui si ha un diverso ordine di applicazione di letture e scritture che porta ad operare con valori diversi e anche inconsistenti

$w_2(x)\ r_1(x)\ w_1(x)\ w_3(x)$ inverte $r_1(x)$ e $w_2(x)$

Insiemisticamente parlando:

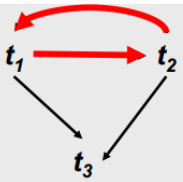


Citiamo i *grafi*, intesa come struttura tale da collegare i singoli punti nel piano con un insieme di archi e descrivere un ciclo se, partendo anche da un solo nodo *n*, è possibile seguire questi e tornare ad *n* percorrendo un arco in più.
Ci serve per creare un *grado dei conflitti* così generato:

- un nodo per ogni transazione t_i
- un arco (orientato) da t_i a t_j se :
  c'è almeno un conflitto fra un'azione a_i e un'azione a_j tale che a_i precede a_j

Uno schedule è in CSR sse il grafo è aciclico.

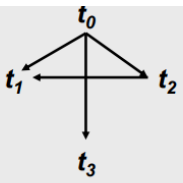
$r_1(x)\ w_2(x)\ w_1(x)\ w_3(x)$



Ciclo ⇒ non CSR

In questo secondo caso, si va da t_2 a t_1 in quanto la lettura/scrittura agisce su *x*

$w_0(x)\ r_1(x)\ w_0(z)\ r_2(x)\ r_3(z)\ w_3(z)\ w_1(x)$



Nessun Ciclo ⇒ CSR

Normalmente costa troppo la view-serializzabilità, ma non la conflict-serializzabilità, perché interessa trovare conflitti e capire se possono essere CSR e quindi risolvibili, piuttosto che esaminare tutte le singole operazioni per capire se l'ordine di letture/scritture è effettivamente quello indicato.

Supponendo di avere un sistema con 100 transazioni al secondo, ciascuna di durata di 5 secondi e che accede a 10 oggetti/pagine, leggendone 2 per secondo. In ogni secondo ci sono quindi 500 transazioni e quindi il sistema deve costruire un grafo con 500 nodi e almeno fino a 5000 archi.
In pratica se non con basi di dati "poco usate", questa situazione non è trattabile.

Si usano quindi i *lock*, per cui:

- Tutte le letture per una risorsa *x* sono precedute da *r_lock(x)* (lock condiviso) e seguite da unlock
- Tutte le scritture per una risorsa *x* sono precedute da *w_lock(x)* (lock esclusivo) e seguite da unlock

Scritto da Gabriel

Quando una transazione prima legge e poi scrive una risorsa *x* può:

- richiedere subito *w_lock(x)*
- chiedere prima *r_lock(x)* e poi *w_lock(x) (lock escalation)*

Si tiene quindi conto dei lock e dei possibili conflitti, in una apposita *tavola dei conflitti*. Per ogni risorsa:

- Un contatore tiene conto del numero di "lettori"; la risorsa è rilasciata quando il contatore scende a zero
- Un valore booleano tiene conto se c'è un *w_lock*

	libera	***r_locked***	***w_locked***
r_lock(x)	OK / r_locked conta(x)++	OK / r_locked conta(x)++	NO / w_locked
w_lock(x)	OK / w_locked	NO / r_locked	NO / w_locked
unlock(x)	error	OK / if (--conta(x)=0) libera else r_locked	OK / not w_locked

← libera

Va in *error* nel caso della terza casella della prima colonna perché la risorsa è già libera. Se la risorsa non è concessa, la transazione richiedente è posta in attesa (eventualmente in coda), fino a quando la risorsa non diventa disponibile.

Normalmente tutti i sistemi implementano un *locking a due fasi* (2PL), che garantisce "a priori" CSR sulla base di:

- fase crescente, acquisendo i lock necessari
- fase decrescente, perché i lock si rilasciano pian piano

Il deadlock si evita perché *una transazione, dopo aver rilasciato un lock, non può acquisirne altri*.
Si rimane in coda per tutte le richieste del lock; se non è fattibile eseguire quella richiesta, poi si rilascia se e quando serve.
Ritornando al problema dell'*aggiornamento fantasma,* cioè un dato che improvvisamente appare aggiornato e per t_1 non è coerente.

- Assumiamo vincolo $y + z = 1000$:

t_1	t_2
bot	
$r_1(y)$	
	bot
	$r_2(y)$
	$y = y - 100$
	$r_2(z)$
	$z = z + 100$
	$w_2(y)$
	$w_2(z)$
	commit
$r_1(z)$	
$s = y + z$	
commit	

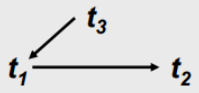
- $s = 1100$: il vincolo sembra non soddisfatto, t_1 vede un aggiornamento non coerente

t_1	t_2	x	y	z
bot		free	free	free
$r\_lock_1(x)$		1:read		
$r_1(x)$				
	bot			
	$w\_lock_2(y)$		2:write	
	$r_2(y)$			
$r\_lock_1(y)$			1:wait	
	$y = y - 100$			
	$w\_lock_2(z)$			2:write
	$r_2(z)$			
	$z = z + 100$			
	$w_2(y)$			
	$w_2(z)$			
	commit			
	$unlock_2(y)$		1:read	
$r_1(y)$				
$r\_lock_1(z)$				1:wait
	$unlock_2(z)$			1:read
$r_1(z)$				
	eot			
$s = x + y + z$				
commit				
$unlock_1(x)$		free		
$unlock_1(y)$			free	
$unlock_1(z)$				free
eot				

Attenzione alla relazione tra 2PL (Locking a due fasi) e CSR.

Consideriamo:

$S: r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)\ r_3(y)\ w_1(y)$



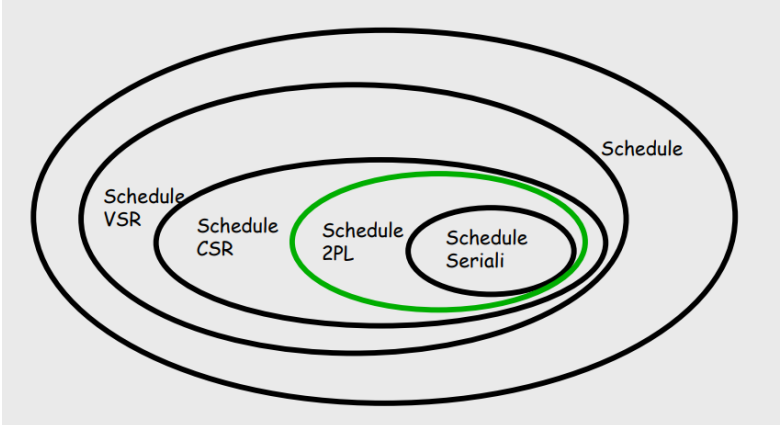
Si nota che S sia CSR.
Affinché sia 2PL dovrebbe essere strutturato in questo modo (*con il locking che va fatto solamente nel caso di una relazione leggi-da* e *se le due transizioni operano sulla stessa risorsa*):

- w_lock1(x) r1(x) w1(x) unlock1(x) r2(x) w2(x) r3(y) w1(y)

Dato che esiste anche w1(y) che va in conflitto con r3(y), allora per precedenza da parte del grafo dei conflitti (il cui ordine sarebbe T3, T1, T2)
Si dovrebbe mettere prima w_lock1(y) prima di unlock1(x)

- w_lock1(x) r1(x) w1(x) w_lock1(y) unlock1(x) r2(x) w2(x) r3(y) w1(y)

Sempre per il grafo dei conflitti se si ha la lettura di y da parte di r3, essa deve precedere T2 e non è compatibile con w1(y) che segue
Quindi *CSR ⇏ 2PL*. Tuttavia, 2PL ⇒ CSR



Scritto da Gabriel

Due transazioni si dicono *serializzabili* se il risultato delle transazioni è uguale a quello eseguito sequenzialmente, dunque non avendo sovrapposizioni temporali.
Per esempio avendo:
r2(x) r1(x) r2(y) w2(y) r1(y) w1(x)
Vediamo che le operazioni sono eseguite in maniera "ordinata", dunque a delle letture seguono delle scritture. In particolare, una lettura di T1 su x e due letture di T2 su x e y, una scrittura di T2 su y e una lettura/scrittura di t1 su x. *Non ci sono transazioni sovrapposte.*
Se si scambiassero:

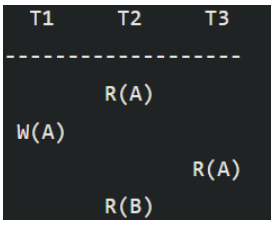
- r1(x) con r2(y)
- r1(x) con w2(y)

avremmo:
r2(y) r2(y) w2(y) r1(x) r1(y) w1(x)
Anche qui, avremmo una lettura e scrittura di T2, poi due letture di T1 su x e y e una operazione in scrittura in modo isolato, quindi senza che l'altra legga quel valore "pendente".
Dunque, tutto bene.

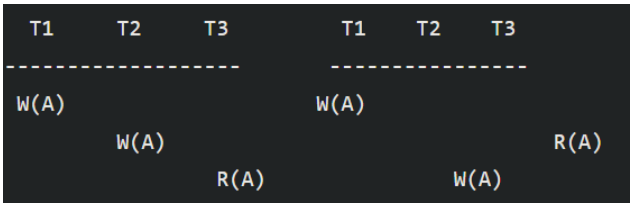
Per introdurre il concetto di *view-derivabilità* tra due schedule devono essere soddisfatte due condizioni:

- lettura iniziale, cioè se una transazione T1 legge il dato A dal database nello schedule S1, allora anche nello schedule S2 anche T1 deve leggere A dal database.

Ad esempio T2 che legge da:

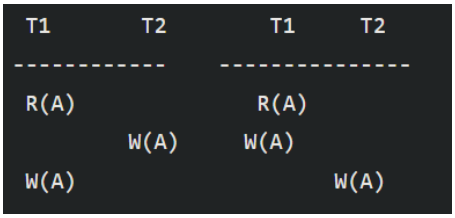


- lettura aggiornata, Se T_i stesse leggendo A che viene aggiornato da T_j in S1, allora in S2 anche T_i dovrebbe leggere A che è aggiornato da T_j.



Qui sopra abbiamo l'esempio in cui T3 legge un valore aggiornato da T2 (in S1) e anche T3 legge A aggiornato da T1 (dentro S2). Questo non è view equivalente.

- scrittura finale, se una transazione T1 ha aggiornato A per ultimo in S1 allora in S2 anche T1 dovrà eseguire le scritture finali. Una scrittura è detta finale quando sia l'ultima su quell'oggetto.
  Per esempio su:
  r1(x) w1(x) w1(y) r2(x) w2(y) → w1(x) è scrittura finale per l'oggetto x, mentre w2(y) lo è per y



In questo esempio vediamo che non sono view-equivalenti perché l'operazione di scrittura finale in S1 è fatta da T1, mentre in S2 è fatta da T2.

Buon riassunto di *view-equivalenza*:

Two schedules T1 and T2 are said to be view equivalent, if they satisfy all the following conditions:

1. **Initial Read:** Initial read of each data item in transactions must match in both schedules. For example, if transaction T1 reads a data item X before transaction T2 in schedule S1 then in schedule S2, T1 should read X before T2.

Read vs Initial Read: You may be confused by the term initial read. Here initial read means the first read operation on a data item, for example, a data item X can be read multiple times in a schedule but the first read operation on X is called the initial read. This will be more clear once we will get to the example in the next section of this same article.

2. **Final Write:** Final write operations on each data item must match in both the schedules. For example, a data item X is last written by Transaction T1 in schedule S1 then in S2, the last write operation on X should be performed by the transaction T1.

3. **Update Read:** If in schedule S1, the transaction T1 is reading a data item updated by T2 then in schedule S2, T1 should read the value after the write operation of T2 on same data item. For example, In schedule S1, T1 performs a read operation on X after the write operation on X by T2 then in S2, T1 should read the X after T2 performs write on X.

Per il discorso di *conflict-serializzabile*, diciamo semplicemente che due operazioni sono in conflitto se:

- operano sullo stesso dato
- appartengono a due transizioni differenti
- almeno una delle operazioni è una scrittura

Detto in tre parole:

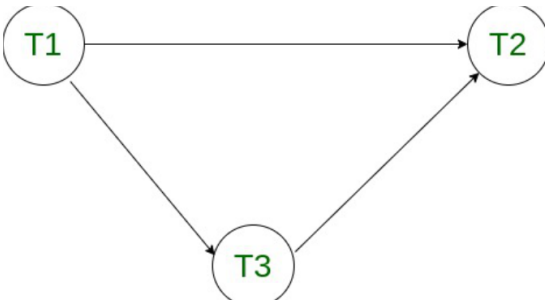
1) una transazione non potrà andare in conflitto con sé stessa a seguito di lettura/scrittura
2) non sapendo in un contesto reale in che ordine si eseguono le operazioni, banalmente, ogni volta che si ha una lettura/scrittura da parte di due transizioni diverse, si ha un conflitto.

Ad esempio:

Le operazioni che vanno in conflitto (si indicano dalla precedente alla successiva sono):

1) R3(X) e W2(X) [T3 -> T2]
2) W1(Y) e R3(Y) [T1 -> T3]
3) W1(Y) e W2(Y) [T1 -> T2]
4) R3(Y) e W2(Y) [T3 -> T2]

Va costruito il grafo dei conflitti, che segue le frecce indicate e ci si accorge che *dato che non ha cicli*, lo schedule è conflict-serializzabile.



Un esempio invece come, ragionando come prima:

S = r1(x), r2(y), r3(x), w3(x), w1(x), w1(y), r2(x), w2(x)

Si disegna il grafo dei conflitti avendo che:

- R1 dipende da W3 (w3(x) – r1(x))
- R3 dipende da W1 (w1(x) – r3(x))
- R2 dipende da W1 (w1(y) – r2(y))
- R2 dipende da W3 (w3(x) – r2(x))
- W2 dipende da R1 (w2(x) – r1(x))
- W2 dipende da R3 (w2(x) – r3(x))

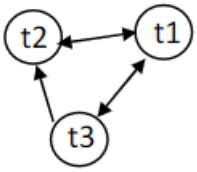
E quindi dato che si ha un ciclo e le operazioni non sono fatte in ordine, S non è né conflict-serializzabile né view-serializzabile.

Attenzione: per capire veramente se un conflict è view-serializzabile, basta semplicemente usare il grafo dei conflitti e ordinarsi le operazioni come per il grafo; se si vede che le scritture finali cioè tutte le scritture) sono nello stesso ordine e le dipendenze sono le stesse, allora è view-serializzabile.

Se è CSR, allora comunque è VSR.

Inoltre:

- per ordinare le transazioni, basta vedere dove vanno le frecce nel grafo dei conflitti oppure verificare l'ordine delle scritture. Spesso, se non sempre, l'ordine delle scritture finali corrisponde al giusto riordinamento delle transazioni.



$W_1(A)\ R_2(A)\ R_2(B)\ W_2(D)\ R_3(C)\ R_1(C)\ W_3(B)\ R_4(A)\ W_3(C).$

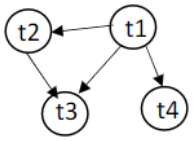
Nello schedule qui sopra:

1) scrive prima T1, T1 stessa e T2 leggono
2) scrive T2, T3 e T1 leggono
3) scrive T3, legge T4
4) scrive T3

Le operazioni sono fatte in maniera ordinata, le scritture non si sovrappongono.

È view serializzabile; i conflict ce ne stanno e di diversi.

Infatti, il grafo dei conflitti riporta;

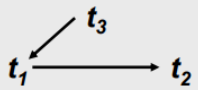


Attenzione alla relazione tra 2PL (Locking a due fasi) e CSR.

Scritto da Gabriel

Consideriamo:

$S: r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)\ r_3(y)\ w_1(y)$



Si nota che S sia CSR.
Affinché sia 2PL dovrebbe essere strutturato in questo modo (*con il locking che va fatto solamente nel caso di una relazione leggi-da* e *se le due transizioni operano sulla stessa risorsa*):

- w_lock1(x) r1(x) w1(x) unlock1(x) r2(x) w2(x) r3(y) w1(y)

Dato che esiste anche w1(y) che va in conflitto con r3(y), allora per precedenza da parte del grafo dei conflitti (il cui ordine sarebbe T3, T1, T2)
Si dovrebbe mettere prima w_lock1(y) prima di unlock1(x)

- w_lock1(x) r1(x) w1(x) w_lock1(y) unlock1(x) r2(x) w2(x) r3(y) w1(y)

Sempre per il grafo dei conflitti se si ha la lettura di y da parte di r3, essa deve precedere T2 e non è compatibile con w1(y) che segue.

Esercitazione 6: Transazioni

Le transazioni vengono registrate su disco fintanto che, eseguendo un update, prima di un CHECK sono effettivamente salvate ed attive; dobbiamo quindi registrare le transazioni per cui dobbiamo fare l'UNDO.
Infatti, normalmente, si cerca l'ultimo checkpoint partendo da sotto e si costruiscono gli insiemi di UNDO e REDO.
Ad esempio, le transazioni che non hanno eseguito un commit, potenzialmente con B/U/D/A, possono andare in fase di UNDO.
Dunque, una transazione che ha fatto il commit (quindi C) si mette in REDO (non ho nessuna garanzia dell'esecuzione corretta dell'operazione).

Sull'esempio delle slide:
UNDO = {T2, T3}
REDO = {T4, T5}
In un esempio: U(T2, O1, B1, A1)
Update per T2 di T1 indicando *B – Before ed A – After* per B1 (oggetto vecchio) e B2 (oggetto nuovo).
Vado poi a rieseguire, partendo dal basso, tutte le operazioni che sono in UNDO

1) D(06)
2) I(05, B7)
3) U(O3, B5)
4) U(O2, B3)
5) U(O1, B1)

T1 non ci va, dato che aveva anche fatto il commit prima del checkpoint.
T5 non si considera perché è in REDO; quindi si fa al secondo passaggio, rieseguendo tutte le operazioni di T5.
Andrò quindi a ripetere:

- B(T4)
- U(T4, O3, B4, A4)
- U(T5, O4, B6, A6)

È indifferente eseguire prima il REDO o l'UNDO.

Esercizio 1
Descrivere la ripresa a caldo, indicando la costituzione progressiva degli insiemi di UNDO e REDO e le azioni di recovery, a fronte del seguente log:

DUMP, B(T1), B(T2), B(T3), I(T1, O1, A1), D(T2, O2, B2), B(T4),
U(T4, O3, B3, A3), U(T1, O4, B4, A4), C(T2), CK(T1, T3, T4), B(T5), B(T6),
U(T5, O5, B5, A5), A(T3), CK(T1, T4, T5, T6), B(T7), A(T4),
U(T7, O6, B6, A6), U(T6, O3, B7, A7), B(T8), A(T7), guasto

Si percorre il log a ritroso fino al più recente checkpoint, cioè CK(T1, T4, T5, T6).
Si mette tutto in UNDO → UNDO = {T1, T4, T5, T6}
Si considera dentro l'UNDO anche T7 che fa l'ABORT e T8 che era già nell'UNDO.
Nessuna transazione ha fatto il commit e si ha REDO{}
UNDO = {T1, T4, T5, T6, T7, T8}

A questo punto si parte al contrario e si ripetono le seguenti operazioni: (mettendo precedente e successivo all'operazione):

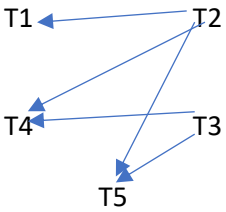
1) U (O3, B7)
2) U (O6, B6)
3) U (O5, B5)
4) U (O4, B4)
5) U (O3, B3)
6) Delete di O1 (essendo l'oggetto perso O1, vado a cancellarlo così lo reinserirò ancora)

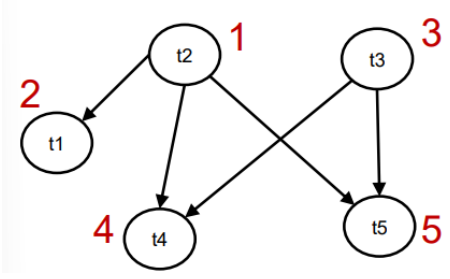
Viene poi ripercorso in avanti tutto il log per rieseguire le operazioni di REDO.
Prima quindi faccio la ripresa a freddo, prendendo lo stato del disco al momento del dump e ripeto tutte le operazioni fino a prima del guasto.
Con la ripresa a caldo, si crea un hardware utile che esegue correttamente le operazioni.

Esercizio 2
Considera il seguente schedule:
S = r2(x) r1(x) w3(t) w1(x) r3(y) r4(t) r2(y) w2(z) w5(y) w4(z)
Esempi di conflitto/conflitto:

S è conflict-serializable? Se sì, mostrare uno schedule che è conflict-equivalente.

Si listano i conflitti:





Quindi S è conflict-serializzabile.
(*Attenzione: capire l'ordine è semplice, senza scervellarsi. Basta vedere l'ordine che c'è scritto dallo schedule*). Oppure, equivalentemente, si ragiona con:
"Se S è conflict-serializzabile, è possibile costruire un grafo dei conflitti di S che è aciclico. Il percorso nel grafo che tocca tutte le transazioni, fornisce un ordine delle transazioni. Scrivendo le operazioni nell'ordine delle transazioni e rispettando l'ordine all'interno delle transazioni, si ottiene uno schedule seriale che è conflict-equivalente e quindi anche view-equivalente."

Scritto da Gabriel

Lo schedule CSR-equivalente è quindi:

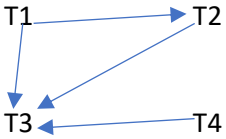
r2(x) r2(y) w2(z) r1(x) w1(x) w3(t) r3(y) r4(t) w4(z) w5(y)

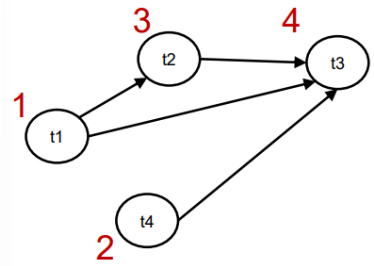
In questo modo i conflitti non sono stati invertiti (come se fossero eseguite da un unico client). Quindi, dato che CSR implica VSR, questa è corretta.

Esercizio 3

S = r1(x) w2(x) r3(x) w1(u) w3(v) r3(y) r2(y) w3(u) w4(t) w3(t)

Dire se è conflict-serializzabile e trovare uno schedule seriale conflict-equivalente

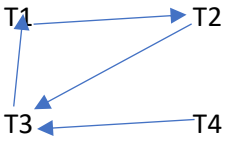




Esso è conflict-serializzabile:

Se per esempio si decidesse di spostare w1(u) nella posizione (prosegue la frase in pagina dopo):
S = r1(x) w2(x) r3(x) w3(v) r3(y) r2(y) w3(u) w1(u) w4(t) w3(t)

si genererebbe un ciclo:



Qualsiasi cosa succede, w1(u) genera conflitti. Se volessi eseguire operazioni di scrittura, dovrei quindi evitare un ciclo in qualche modo. L'operazione non può essere effettuata in quel punto; pima di effettuarla, occorre "rompere" il ciclo, cioè fare il commit o abort di t2 o t3 per togliere il nodo delle transazioni attive dal grafo.

Esercizio 4

Indicare se i seguenti schedule sono VSR:

1. r1(x), r2(y), w1(y), r2(x) w2,(x)
2. r1(x), r2(y), w1(x), w1(y), r2(x) w2,(x)
3. r1(x), r1(y), r2(y), w2(z), w1(z), w3(z), w3(x)
4. r1(y), r1(y), w2(z), w1(z), w3(z), w3(x), w1(x)

Sappiamo individuare se sono VSR se invertendo le transazioni, non si hanno relazioni del tipo *"legge da"*.

1) Non è VSR: infatti invertendo ad esempio w1(y) con r2(y) oppure w2(x) con r1(x) si hanno relazioni di dipendenza.
2) Non è VSR, infatti si hanno varie dipendenze conflittuali, come w1(y) per r2(y) oppure r2(x) per w1(x). Dato che si hanno più relazioni di lettura/scrittura, cambiando l'ordine dei fattori, il risultato cambia.
3) In questo caso è VSR: infatti, non ci sono letture/scritture che concorrono, avendo una sola scrittura su X mentre tutte le altre agiscono su Z oppure su Y. Anche per w3, unico che esegue su x e z, quindi due valori diversi, non si ha ordine diverso di scrittura.
4) Non è VSR: infatti ho due scritture con T1 per x e z e due scritture su T3 per x e z; potenzialmente queste transazioni possono essere invertite e non avere lo stesso ordine di esecuzione finale.

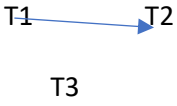
Esercizio 5

Classificare i seguenti schedule (come: NonSR, VSR, CSR). Nel caso uno schedule sia VSR oppure CSR, indicare tutti gli schedule seriali e esso equivalenti.

1. r1(x), w1(x), r2(z), r1(y), w1(y), r2(x), w2(x), w2(z)
2. r1(x), w1(x), w3(x), r2(y), r3(y), w3(y), w1(y), r2(x)

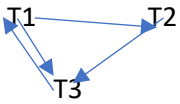
1) r1(x), w1(x), r2(z), r1(y), w1(y), r2(x), w2(x), w2(z)

Indicando al solito i conflitti con i colori, si vede che disegnando si avrebbe:



Vedendo anche che ci sono due scritture di w2 in x e z e una sola scrittura di w1 in x, possiamo dire che è certamente conflict-equivalente (dal grafo), mentre per la VSR, si avrebbero le stesse scritture finali, in quanto le dipendenze presenti hanno una lettura iniziale su z e y che non interferiscono l'una con l'altra e le stesse scritture finali su x e z, anche qui che non interferiscono.

2) r1(x), w1(x), w3(x), r2(y), r3(y), w3(y), w1(y), r2(x)



Come si vede, non è CSR essendoci un ciclo. Si vede che non è neanche VSR in quanto l'ordine delle scritture finali non viene rispettato dalle letture iniziali, portando quindi a differenti viste sul dato. Dunque, S non è né VSR che CSR.

Esercitazione 7: Esercizi vari per esame

Dato il seguente schema:

```
AEROPORTO(Città,Nazione,NumPiste)
VOLO(IdVolo,GiornoSett,CittàPart,OraPart,CittàArr,OraArr,TipoAereo)
AEREO(TipoAereo,NumPasseggeri,QtaMerci)
```

Esprimere le seguenti query in Algebra Relazionale:
1. Le nazioni da cui parte e arriva il volo con codice AZ274;
2. Le città da cui partono voli internazionali
3. Le città da cui partono solo voli nazionali
4. Le città con più piste;

1) $\pi_{Nazione}((\sigma_{IdVolo} = 'AZ274' (VOLO)) \bowtie_{CittaPart=Citta \text{ OR } CittaArr=Citta} AEROPORTO))$
oppure
$\pi_{Nazione}((\sigma_{IdVolo} = 'AZ274' (VOLO)) \bowtie_{CittaPart=Citta} AEROPORTO))$ U
$\pi_{Nazione}((\sigma_{IdVolo} = 'AZ274' (VOLO)) \bowtie_{CittaArr=Citta} AEROPORTO))$
2) VP= (VOLO $\bowtie_{CittaPart=Citta}$ AEROPORTO);
VA=(VOLO $\bowtie_{CittaArr=Citta}$ AEROPORTO);
$\pi_{VP.CittaPart}$(VP $\bowtie_{VP.IdVolo=VA.IdVolo \text{ AND } VP.Nazione<>VA.Nazione}$ VA);
3) VP= (VOLO $\bowtie_{CittaPart=Citta}$ AEROPORTO);

VA=(VOLO ⋈ $_{CittaArr=Citta}$ AEROPORTO);
π_{Citta} (AEROPORTO) - π_{Citta}($\rho_{Citta \leftarrow VP.CittaPart}$(VP ⋈ $_{VP.IdVolo=VA.IdVolo\ AND\ VP.Nazione<>VA.Nazione}$ VA))

4) A1 = Aeroporto
A2 = Aeroporto
$\pi_{Città}$(AEROPORTO) - $\pi_{A1.Città}$ (A1 ⋈ $_{A1.NumPiste > A2.NumPiste}$ A2)

Viene usato il maggiore al posto del maggiore uguale perché, in questo modo, vado a togliere eventuali duplicati nella conta del massimo. Con i colori sotto si indica il join (fatti sulla stessa riga, quindi nel caso di due celle ha due colori per i join di entrambe).

Città	AereoP
A	10
B	10
C	8

Città	AereoP
A	10
B	10
C	8

Allora effettivamente toglie 8, e prende i due 10.

Facendo però:

Città	AereoP
A	11
B	10
C	8

Città	AereoP
A	10
B	10
C	8

Restituirebbe 11 e 10 per effetto del >= e non sarebbe corretto. Si deve usare il maggiore.

Sia R(A,B,C,D) uno schema di relazione su cui siano definite le dipendenze funzionali:

$F = \{AB \rightarrow CD,\ C \rightarrow A,\ D \rightarrow B\}$.

Si richiede di:
1. determinare le possibili chiavi
Se R non è in 3NF rispetto a F, allora:
2. mostrare le dipendenze che violano la 3NF
3. portare lo schema in terza forma normale.

1) Calcoliamo le chiusure:
$AB^+ = \{A,B,C,D\}$ $C^+ = \{C,A\}$ $D^+ = \{D, B\}$
AB è chiave e superchiave e quindi anche chiave
Verificando se ci sono altri attributi che uniti possono portare a delle chiavi che esiste CD
e si vede che CD^+ contiene tutti gli attributi:
$CD^+ = \{A,B,C,D\}$ CD è superchiave CD è chiave
Dunque, le chiavi sono Ce AB (volendo, si aggiunge B a C oppure si aggiunge A verso D e si ottiene che anche AD e BC sono chiavi.

2) AB è chiave
C non è chiave ma A fa parte di una chiave (perché A fa parte di AB)
D non è chiave ma B fa parte di una chiave (perché B fa parte di AB)

Siamo in terza forma normale

3) Lo schema è già in 3NF

Sia R(A, B, C, D, E, G) uno schema di relazione su cui siano definite le dipendenze funzionali:

$F = \{A \rightarrow B, CD \rightarrow A, CB \rightarrow D, AE \rightarrow G, CE \rightarrow D\}$.

Si richiede di:
1. determinare le possibili chiavi

Se R non è in 3NF rispetto a F, allora:
2. mostrare le dipendenze che violano la 3NF
3. portare lo schema in terza forma normale.

1) Determiniamo le chiusure:

$A^+ = \{A,B\}$
$CD^+ = \{C,D,A,B\}$
$CB^+ = \{C,B,D,A\}$
$AE^+ = \{A,E,G,B\}$
$CE^+ = \{C,E,D,A,G,B]$

Se proviamo a mettere la E a CB^+, si ottiene $ECB^+ = \{ C, B, D, A, E \}$ ma B è superflua perchè $ECB^+=CE^+$ .Stesso dicasi per altre "estensioni". L'unica chiave è CE, in quanto D non aggiunge informazioni e CE è la più piccola chiave.

2) Escludendo CE → D in cui si ha che CE è chiave:
A→B vìola perché A non è chiave e B non fa parte di CE (non nel senso di chiusura, ma nel senso proprio di lettere)
Similmente per CD → A, CB → D, AE → G.

3) Le cinque relazioni che abbiamo fanno parte già di una copertura ridotta.
Si nota, seguendo il passo 2:
"G viene partizionato in sottoinsiemi tali che due dipendenze funzionali X→ A e Y → B sono insieme se $X_G^+ = Y_G^+$"
Si nota che CD/CB hanno la stessa chiusura, quindi si mettono assieme.

Seguendo il passo 3:
"*Costruiamo una relazione per ogni sotto-insieme*"
Quindi:
$R_1(A,B,C,D)$
$R_2(A,B)$
$R_3(A,E,G)$
$R_4(C,D,E)$

Seguendo il passo 4:
"Se esistono due relazioni S(X) e T(Y) con $X \subseteq Y$*, S viene eliminata"*
Esiste R_2 che contiene A,B che sta dentro già A,B,C di R_1.
Quindi R_2 viene eliminata.

Seguendo il passo 5:
"Se, per qualche i, non esiste una relazione S(X) con $K_i \subseteq X$*, viene aggiunta una relazione* $T(K_i)$"
Dopo l'eliminazione di R_2 si ha correttamente un insieme di relazioni che hanno già tutte le chiavi e non occorre aggiungere altro.
Infatti:
CD/CB stanno in R_1

AE sta in R_3
CE sta in R_4

È in BNCF?
No, in quanto tra le relazioni A non è superchiave, mentre rispettivamente CD, CB, AE, CE sono tutte superchiavi/chiavi.
Quindi avremo come relazioni, togliendo Y dalla relazioni originaria, quindi B, quindi B e facendo una sua relazione:
R_1 (A, C, D)
R_2 (A, B, C)
R_3 (A, E, G)
R_4 (C, D, E)

In questo modo mantengo tutte le chiavi e rispetto anche BNCF.

Data una relazione R(ABCD) con dipendenze funzionali
{C →D, C →A, B →C}

(a) Identificare le possibili chiavi
(b) Enumerare le dip.funz. che violano BCNF
(c) Proporre una decomposizione in BCNF.
(d) La decomposizione proposta è in BCNF e senza perdite nel join?

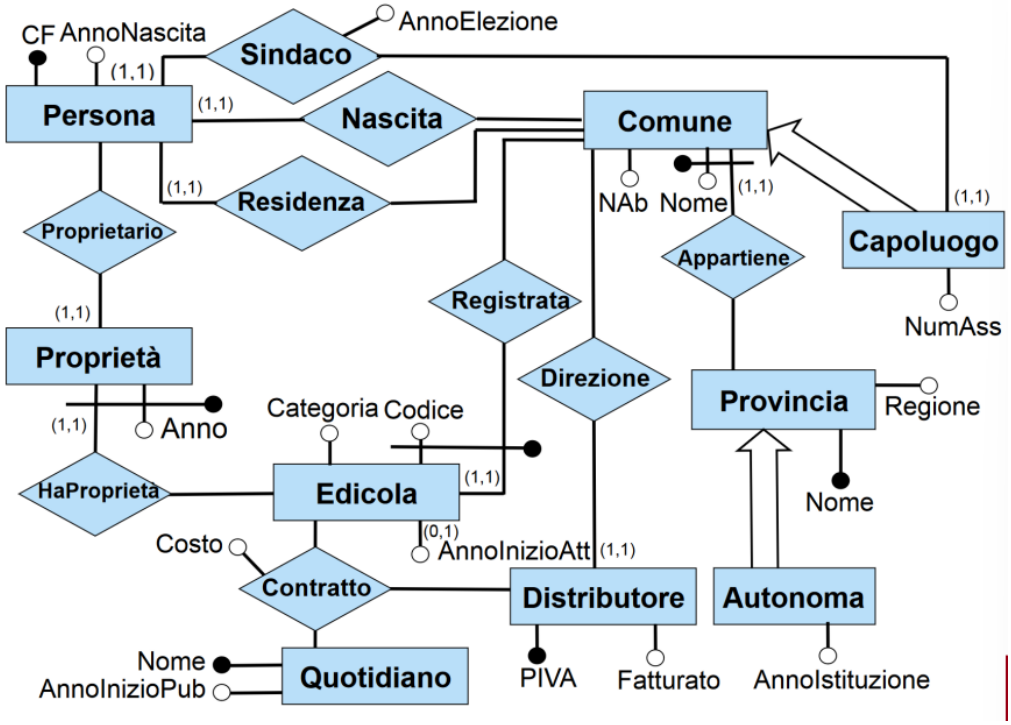
a) Calcoliamo le chiusure:
C^+ = {C, D, A}
B^+ = {B, C, A, D}
B comprende tutti gli attributi e comprende funzionalmente anche la chiusura di B e si rivela la sola possibile chiave.
b) Le dipendenze che violano sono C → D e C → A perché C non è superchiave e né D né A fanno parte della chiave.
c) La decomposizione in BCNF segue due possibili strade:
Partendo da R1 = {A,B,C,D}
- decomponendo per C → D si ottiene
R_1(A,B,C) R_2(C,D)
- decomponendo per C → A si ottiene
R_1(B,C,D) R_2(A,C) R_3(C,D)
d) A tale scopo, le decomposizioni proposte si pongono in BNCF e sono senza perdite, nel senso che tramite i join e tramite la scomposizione delle chiavi presenti, riotterremmo gli stessi dati di partenza.

Progettazione concettuale

Si richiede di progettare lo schema ER concettuale di un'applicazione relativa alle edicole per la vendita di giornali. Di ogni edicola interessa il comune in cui essa è registrata, il codice, che è unico nell'ambito del comune in cui l'edicola stessa è registrata, la categoria, l'anno di inizio attività (non sempre disponibile), e i contratti che l'edicola ha con i distributori per l'approvvigionamento dei quotidiani.

Ogni contratto riguarda un'edicola, un quotidiano ed un distributore, ed è caratterizzato dal costo mensile a carico dell'edicola. Infine, per ogni edicola, interessa conoscere le varie persone che sono state proprietarie dell'edicola nei diversi anni, tenendo conto del fatto che in ogni anno un'edicola ha al massimo un proprietario. Di ogni persona interessa il codice fiscale (id), l'anno di nascita, il comune di nascita, ed il comune di residenza. Di ogni distributore di quotidiani interessa la partita IVA (id), il fatturato ed il comune in cui è situata la direzione. Di ogni quotidiano interessa il nome (identificativo), e l'anno di inizio pubblicazione. Di ogni comune interessa la provincia di appartenenza, il nome (unico nella provincia), ed il numero di abitanti. Dei comuni che sono capoluogo di provincia interessa l'attuale sindaco (con l'anno di elezione), ed il numero di assessori comunali. Di ogni provincia interessa il nome (identificativo) e la regione di appartenenza. Alcune province sono "autonome", e di esse interessa anche l'anno di istituzione.

Soluzione:



Dato il seguente schema:

```
AEROPORTO(Città,Nazione,NumPiste)
VOLO(IdVolo,GiornoSett,CittàPart,OraPart,CittàArr,OraArr,TipoAereo)
AEREO(TipoAereo,NumPasseggeri,QtaMerci)
```

Le città francesi da cui partono più di 20 voli alla settimana diretti in Italia

```
SELECT CittàPart
FROM AEROPORTO as A1 join VOLO on A1.Città=CittàPart
JOIN AEROPORTO as A2 on CittàArr=A2.Città
WHERE A1.Nazione='Francia'
AND A2.Nazione= 'Italia'
GROUP BY CittàPart
HAVING COUNT(*) >20
```

Dato il seguente schema:

```
AEROPORTO(Città,Nazione,NumPiste)
VOLO(IdVolo,GiornoSett,CittàPart,OraPart,CittàArr,OraArr,TipoAereo)
AEREO(TipoAereo,NumPasseggeri,QtaMerci)
```

Per ogni giorno della settimana, il tipo di aereo più utilizzato

```
CREATE VIEW Tipo_Per_Giorno (Numero_voli, Tipo_aereo, Giorno_Settimana) AS
SELECT Count(*), TipoAereo, GiornoSett
FROM Volo V
GROUP BY TipoAereo, GiornoSett

SELECT *
FROM Tipo_Per Giorno T
WHERE Numero_voli >= (SELECT Max(Numero_voli)
                      FROM Tipo_per_giorno
                      WHERE T2.GiornoSett = T.Giorno_Settimana)
```

oppure

```
CREATE VIEW TIPOAEREO_X_GIORNO(GiornoSett, TipoAereo, Numero) AS
SELECT GiornoSett, TipoAereo, COUNT(*) AS Numero
FROM VOLO
GROUP BY GiornoSett, TipoAereo;

CREATE VIEW TIPOAEREO_MAX_X_GIORNO(GiornoSett,MaxNumero)
AS SELECT GiornoSett, MAX(Numero)
FROM TIPOAEREO_X_GIORNO
GROUP BY GiornoSett;

SELECT T.GiornoSett, T.TipoAereo
FROM TIPOAEREO_X_GIORNO T JOIN TIPOAEREO_MAX_X_GIORNO M
WHERE M.GiornoSett=T.GiornoSett AND T.Numero=M.MaxNumero
```

Scritto da Gabriel

Riepilogo e discussione Esempio d'esame

Di ogni blocco di marmo estratto interessa il codice (identificativo), l'anno di estrazione, il peso e la cava da cui è stato estratto.

Per il dominio di estrazione del marmo, è interesse memorizzare i dati di certi luoghi geografici. Ogni luogo geografico ha un codice identificativo e l'area che occupa in chilometri quadrati.

Le cave di interesse sono quelle dalle quali è stato estratto almeno un blocco. Ogni cava è un luogo geografico di cui interessa anche conoscere l'altitudine e la regione (esattamente una) in cui è situato il suo territorio.

Da ogni blocco di marmo si producono almeno una lastra di marmo e di ogni lastra interessa il blocco da cui è stato prodotto (uno ed uno solo), la superficie, il numero (unico nell'ambito del blocco di marmo da cui è stato prodotto), l'eventuale abitazione in cui viene usata.

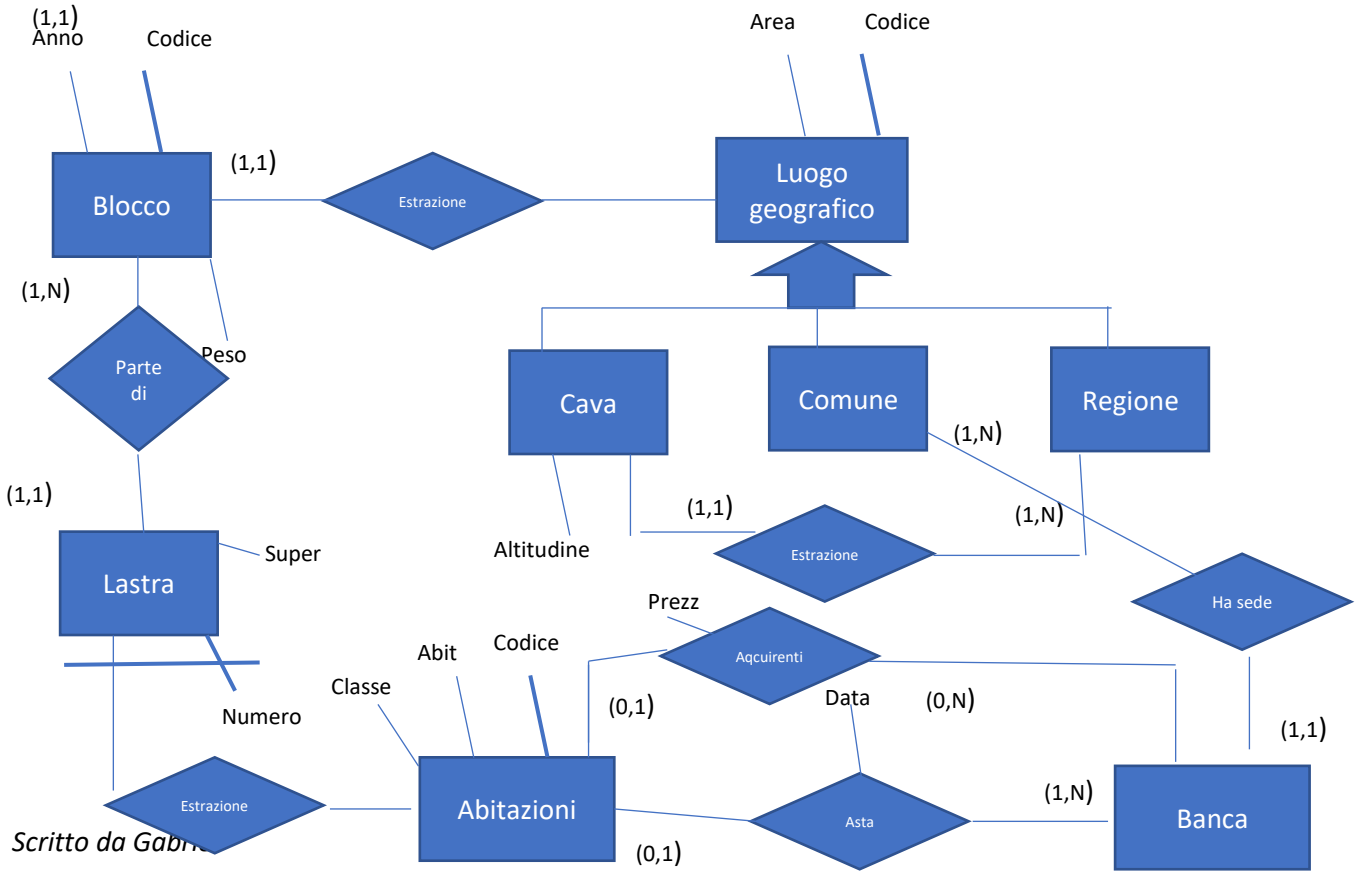
Ogni abitazione che interessa all'applicazione usa almeno una lastra di marmo e di ognuna di tali abitazioni interessa il codice (identificativo), la classe e le eventuali banche che hanno partecipato all'asta per quell'abitazione, con la data di partecipazione della banca all'asta.

Tra le banche che hanno partecipato all'asta per una certa abitazione, dopo tale asta, è di interesse sapere la banca, se esiste, che ha acquistato l'abitazione stessa, con il relativo prezzo di acquisto.

Di ogni banca interessa il "codice unico bancario" (identificativo), il capitale sociale ed il comune in cui si trova la sede centrale.

Di ogni comune, oltre alle proprietà di tutti i luoghi geografici, interessa il livello del PIL, la regione in cui si trova ed il nome (unico nell'ambito della regione in cui si trova).

Di ogni regione, oltre alle proprietà di tutti i luoghi geografici, interessa l'anno della sua fondazione.



Passaggio allo schema logico:
Eliminiamo le generalizzazioni e collasso l'entità Padre nelle figlie e metto tutti i campi di Luogo in Cava, Comune, Regione

Occorre far vedere la ristrutturazione
(Si può farlo ricopiando la parte che cambia ricopiando gli attributi)

Entità:
Cava (Codice, Area, Altitudine, Regione)
Cava.Regione → Regione.Codice
Regione (Codice, Anno, Area)
Comune (Nome, Regione, Area, PIL, Codice)
Comune.Regione → Regione.Codice
Banca (CUB, Capitale, Comune, Regione)

1) Banca.Comune → Comune.Nome
   Banca.Regione → Comune.Regione
2) Banca (Comune,Regione) → Comune (Nome, Regione)

Prendiamo la seconda strada, perché quando ho una chiave esterna composta da più campi, è bene fare in quel modo.
Abitazione (Codice, Classe, CUB, Prezzo)
Abitazione.CUB → Banca.CUB
Asta (CUB, Cod_abit, Data)
Asta.CUB → Banca.CUB

Discussione tema d'esame parte 2

Si consideri la seguente base di dati per la registrazione degli impiegati che lavorano in aziende:
• IMPIEGATO(CF, Nome, Cognome, Residenza)
• LAVORA(CF, PIVA, DataInizio, DataFine, StipendioMensile)
• AZIENDA(PIVA, Citta, Regione)

dove nessun attributo ammette valori nulli tranne DataFine che può essere NULL se l'impiegato lavora ancora in una azienda. Si noti che una persona può lavorare contemporaneamente in più aziende.

A. *Nel riquadro, scrivere una query in algebra relazionale che restituisca il codice fiscale delle persone che attualmente hanno esattamente un solo lavoro.*

$La - L2 \quad L' = La \quad L = L'$

$La = (\sigma_{DATAFINE\ IS\ NULL}\ LAVORA)$

$L2 = \Pi_{CF}(L \bowtie_{L.PIVA \neq L'.PIVA\ AND\ L.CF = L'.CF} L')$

$\Pi_{CF}(La) - \Pi_{CF}(L2)$

B. Nel riquadro, scrivere una query in Standard SQL che restituisce la partita IVA dell'azienda con più dipendenti impiegati alla data attuale.

```
CREATE VIEW Dip_Att_X_Azienda AS
SELECT Piva, Count(*) AS Num  (Count(CF) conta tutti i CF non nulli; essendo non chiave succede)
FROM Lavora
```

Scritto da Gabriel

```
WHERE Datafine IS NULL
GROUP BY Piva

SELECT Piva
FROM Dip_Att_X_Azienda
WHERE Num = (SELECT(MAX(Num)) FROM Dip_Att_X_Azienda)
```

C. Nel riquadro, scrivere una query in Standard SQL che, per ogni partita IVA in Veneto relativo ad aziende con almeno dieci dipendenti, restituisce lo stipendio medio

```
SELECT Avg(StipendioMensile), Piva
FROM Lavora JOIN Azienda
ON Lavora.Piva = Azienda.Piva
WHERE Regione = 'Veneto'
GROUP BY Azienda.Piva
HAVING Count(*) >= 10
```

Sia data la seguente relazione R(ABCDE), con copertura ridotta G={B→C, B→E, C→A, CD→E}.

a. Trovare la/e chiave/i di R, motivando la risposta.
b. Effettuare una decomposizione in 3NF ed indicare le chiavi delle relazioni finali ottenute.
c. Indicare se la decomposizione ottenuta al punto b è anche in BCNF rispetto all'insieme di dipendenze in G. Motivare la risposta.
d. Indicare se c'è conservazione delle dipendenze. Motivare la risposta

a) $B^+ = \{B, C, E, A\}$
$C^+ = \{C, A\}$
$CD^+ = \{C, A, D, E\}$
$BD^+ = \{B, C, E, A, D\}$

La seguente chiusura è superchiave perché C è in più:
$BCD^+ = \{B, C, E, A, D\}$
Pertanto, BD è chiave

b) Inizializziamo i gruppi delle relazioni:
B → C, B → E CD → E C → A
Dunque, creiamo le relazioni:
R_1 (<u>C, D</u>, E)
R_2 (<u>C</u>, A)
R_3 (<u>B</u>, C, E)

Non c'è una relazione che contiene anche D chiave e si aggiunge:
R_4 (<u>B, D</u>)

c) Siamo già in BNCF

d) Per ogni dipendenza funzionale ci deve essere una relazione che contiene tutto per la conservazione delle dipendenze. Infatti, le dipendenze funzionali

Data la relazione R(A, B, C) con dipendenze funzionali A→BC e B→A. Una delle seguenti affermazioni è vera. Quale?

Basta fare una chiusura:
$A^+ = \{A, B, C\}$

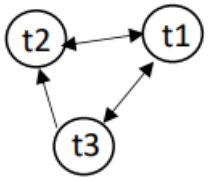
Scritto da Gabriel

$B^+ = \{A, B\}$
$AB^+ = \{A, B, C\}$

Sia data la seguente porzione di log fino al guasto: CK(T5,T6), B(T7), U(T7,O6,B6,A6), U(T6, O3, B7, A7), B(T8), C(T7), I(T8,O5,A5). Quali transazioni richiedono l'UNDO?

UNDO = {T5, T6, T7, T8}
Viene tolta T7 che ha fatto il commit.
Quindi
UNDO = {T5, T6, T8}

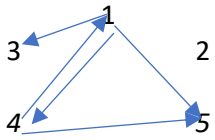
Dato il seguente schedule S, con grafo dei conflitti in figura:
S = r1(x), r2(y), r3(x), w3(x), w1(x), w1(y), r2(x), w2(x)



Per forza la seconda transazione va per ultima
L'ordine sarà dato da:

1) t3
2) *t1*
3) *t2*

r4(y) w1(z) r2(y) w3(x) w1(y) r1(x) r3(z) w5(y) w4(z) r4(x)



Avendo trovato il ciclo, non è conflict-serializzabile.

Esercizio 1
Verificare se i seguenti due schedule sono view equivalenti.

- r1(x) r2(x) w1(x) r2(y) w1(y) w2(z) r3(z) r1(z) w3(x) w1(z) w3(z) r3 (y) w3(y)
- r1(x) r2(x) w1(x) r2(y) w1(y) w2(z) r1(z) w3(x) w1(z) r3(z) w3(z) r3 (y) w3(y)

I due schedule non sono view equivalenti perché la lettura r3(z) legge valori diversi in ognuno.

Esercizio 2
Verificare se esiste e qual è uno schedule seriale conflict equivalente al seguente.

- r1(x) r2(x) w1(x) r2(y) w1(y) w2(z) r3(z) r1(z) w3(x) w1(z) w3(z) r3(y) w3(y)

Il grafo dei conflitti di questo schedule causa un ciclo tra la Transazione 1 e la 3, quindi non esiste nessun schedule seriale equivalente.

Esercizio 3
Verificare se esiste e qual è uno schedule seriale conflict equivalente al seguente.

- r2(x) r1(x) r2(y) w2(y) w1(z) r1(z) r3(z) w1(x) w3(z) r2(y) w3(y)

Basta disegnare il grafo dei conflitti e risulterà che la transazione da cui escono più frecce è la 2 (quindi T2 andrà per prima, similmente 3 è ultima perché raggiunta da più frecce, mentre 1 è la seconda visti i due ragionamenti precedenti. Ne consegue che l'ordine sarà T2 T1 T3.

Esercizio 4
Verificare se il seguente schedule sia conflict o view serializzabile e, nel caso definire uno schedule seriale equivalente: r3(y),w1(x),r1(y),r2(x),r3(z), r1(z),w2(x),w2(z),w3(y),w1(x)
Il grafo dei conflitti mostra un ciclo, quindi non è CSR (né VSR).

Scritto da Gabriel

Esercizio 5
Verificare se il seguente schedule sia conflict o view serializzabile e, nel caso definire uno schedule seriale equivalente: *r3*(y),w2(x),r1(y),r3(x),r2(z),r2(y),r3(z),r1(z),w3(x),w1(z),w1(y),w1(x)
Lo schedule è conflict serializzabile ed equivalente a T2 T3 T1

Esercizio 6
Verificare se i seguenti due schedule sono conflict-equivalenti e, in ogni casi, dire se uno o entrambi sono conflict-serializzabili

- w2(x), r1(x), w1(x), r2(y), w3(x), r1(z), r3(y), r3(z), w2(y), w3(z)
- w2(x), r1(x), w1(x), w3(x), r3(y), r1(z), r3(z), r2(y), w2(y), w3(z)

I due schedule sono conflict-equivalenti (stesso ordine dei conflitti) ed entrambi non conflict-serializzabili.

- r1(y) r1(z) r3(x) w3(x) w1(y) w1(x) r2(x) r4(z) w4(z) r2(y) w2(z) w2(x)

Questo schedule è CSR e dunque VSR.
Un possibile schedule conflict-equivalente è dato da
T3 - T1 - T4 - T2

- w3(x) r2(y) w2(x) w3(y) r3(z) r2(z) w1(x) r1(y) w3(z) r1(x)

Questo schedule non è CSR e dunque non è VSR.
Sussiste infatti un ciclo tra T2 e T3 (disegnando il grafo dei conflitti)
Seguendo l'ordine delle scritture, T3 dovrebbe precedere T2 (scrittura su x), tuttavia, esiste una leggi-da da T2 su T3 su y, facendo capire che T2 dovrebbe precedere invece T3. Chiaramente, non esiste nessuno schedule in grado di soddisfare tale ordine.

DUMP, B(T1), D(T1,O1,B1), I(T1,O2,A2), B(T2), I(T2,O3,A3), B(T3), U(T1,O4,B4,A4), CK(. . .), C(T2), B(T4), D(T3,O5,B5),
U(T4,O6,B6,A6), A(T4), I(T3,O7,A7), CK(. . .), B(T5), U(T5,O8,B8,A8), C(T3), B(T6), C(T5), D(T6,O10,B10), B(T7),
A(T1), GUASTO
Si richiede di:
1. scrivere, in corrispondenza di ogni record di checkpoint, le transazioni attive;
- al primo checkpoint, a seguito delle operazioni, risultano essere attive
T1, T2, T3, a seguito di rispettivi begin e insert/delete/update --> CK(T1,T2,T3)
- eccezion fatta per T2 che presenta un commit, le transazioni di prima rimangono
attive, ora viene inserita ----> CK(T1,T3,T4)

Dunque, ora siamo in questa situazione:
DUMP, B(T1), D(T1,O1,B1), I(T1,O2,A2), B(T2), I(T2,O3,A3), B(T3), U(T1,O4,B4,A4), CK(T1,T2,T3), C(T2), B(T4), D(T3,O5,B5),
U(T4,O6,B6,A6), A(T4), I(T3,O7,A7), CK(T1,T3,T4), B(T5), U(T5,O8,B8,A8), C(T3), B(T6), C(T5), D(T6,O10,B10), B(T7),
A(T1), GUASTO

2. illustrare dettagliatamente i passi da compiere per effettuare la ripresa a caldo

- Si percorre a ritroso fino al primo checkpoint, creando gli insiemi di UNDO - REDO:
UNDO = {T1, T3, T4} REDO = {T3, T5}
- Si percorre in avanti il log creando gli insiemi di UNDO - REDO:
UNDO = {T1, T3, T4, T6, T7} REDO = {T3, T5}
- Il log viene percorso fino all'inizio per disfare le azioni in UNDO con le seguenti operazioni:
I(O10)
O8 = B8
D(O7)

Scritto da Gabriel

O6=B6
O4=B4
D(O2)
I(O1)
- Il log viene percorso fino alla fine per rifare le azioni in REDO:
I(O5)
O7=A7
O8=B8