APPUNTI

Basi Di Dati

Francesco De Lucchini

Università di Pisa

2022

1. Introduzione

Basi di dati e DBMS

Un **database** è un insieme organizzato di dati grandi, persistenti e condivisi, gestito da un **DataBase Managment System** che ne garantisce la privatezza, efficienza, efficacia, affidabilità e consistenza.

I DBMS si dividono in due categorie principali:

- Sistemi transazionali, cioè dotati di un meccanismo per la definizione e l'esecuzione di transazioni, ossia sequenze di operazioni che, se eseguite in modo corretto, producono una variazione nello stato di una base di dati, le quali assumono le proprietà **ACID**:

    - Atomicity *Viene eseguita per intero (commit) o per niente (abort)*
    - Consistency *Porta ad uno stato consistente*
    - Isolation *Non espone mai i suoi stati intermedi*
    - Durability *I cambiamenti che porta sono permanenti*

- Sistemi non transazionali, i quali privilegiano la disponibilità e la flessiblità dei dati, i quali assumono le proprietà **BASE**:

    - Basically Available *Base di dati sempre disponibile*
    - Soft State *Non esistono schemi o forme fisse dei dati*
    - Eventually Consistent *Prima o poi i dati saranno consistenti*

Big Data

Originariamente chiamati Very Large Data Bases sono insiemi di dati caratterizzati da:

- Volume *Immenso e sempre crescente*
- Velocità *Con cui vengono interrogati e aggiornati*
- Varietà *Delle informazioni che contengono*
- Veridicità *È possibile estrarne informazioni vere da dati contenenti gravi errori*

La Data Science è la disciplina che si occupa dello studio dei Big Data, tramite:

- Data cleaning *Selezionare dati di qualità*
- Data integration *Integrare differenti sorgenti di dati*
- Data mining *Estrarre informazioni utili*
- Machine learning

Modelli logici, interni e di esecuzione dei DBMS

Un **modello logico** è un insieme di costrutti che descrivono l'organizzazione e la dinamica dei dati, fornendone una vista astratta dei dati. I modelli logici più conosciuti sono il modello relazionale, gerarchico *(che utilizza puntatori fra record vari),* reticolare, a oggetti, XML e NoSQL.

Ogni tipo di dato nel modello logico ha uno **schema** *(struttura e caratteristiche dei dati)*, definito da un Data Definition Language, e un'istanza, ossia i valori attuali, interrogati e aggiornati da un Data Manipulation Language *(ad esempio il linguaggio SQL ricopre entrambi questi ruoli).*

Il **modello interno** rappresenta come vengono fisicamente memorizzati i dati *(lista, albero, hash ...)*

Tra i **modelli di esecuzione** spiccano le architetture distribuite, in particolare quella Client-Server, divisa in due principali rami:

- Two tier
  - Thin client *il server compie il lavoro sui dati*
  - Thick client *il client compie il lavoro sui dati*
- Three tier *presenta un server per il DB e uno per il lavoro sul DB*

Database NoSQL

Un database può essere scalato verticalmente *(aumentare la potenza dello stesso server)* oppure orizzontalmente *(distribuire i dati in più server).*

I database relazionali *(con proprietà ACID)* possono essere scalati solamente verticalmente *(mantenere informazioni coerenti su più server richiederebbe uno sforzo insostenibile)*, questo ha portato i progettisti a slivuppare database non relazionali, raggruppati sotto il movimento Not-only-SQL e dotati delle seguenti caratteristiche:

- Proprietà BASE
- Modelli logici semplici
- Mancanza di schemi fissi
- Mancanza di controlli sull'integrità dei dati
- Auto sharding *frammenti (shards) di dati vengono distribuiti automaticamente tra i server*
- Facile replicazione
- Query efficienti tramite API proprietarie

CAP *(Bewer)* THEOREM

Asserisce che un sistema distribuito *(dunque non si applica ai DBMS relazionali)* è in grado di supportare solamente due tra le seguenti caratteristiche:

- Consistency *tutti i nodi vedono gli stessi dati in ogni momento*
- Availability *ogni operazione deve sempre ricevere risposta*
- Partition Tolerance *sistema tollerante ad aggiunta/rimozione di un nodo*

I database NoSQL consentono strutture complesse, cosa impossibile in un database relazionale *(non si possono avere relazioni annidate, ossia relazioni come valori di un campo di un record)*, e hanno come modello logico l'aggregato, ossia il raggruppamento di dati secondo un particolare criterio, i **modelli logici NoSQL** più utilizzati sono:

DOCUMENT MODEL

Ogni record è memorizzato come un documento con un determinato schema e sono possibili interrogazioni su campi dello schema del documento

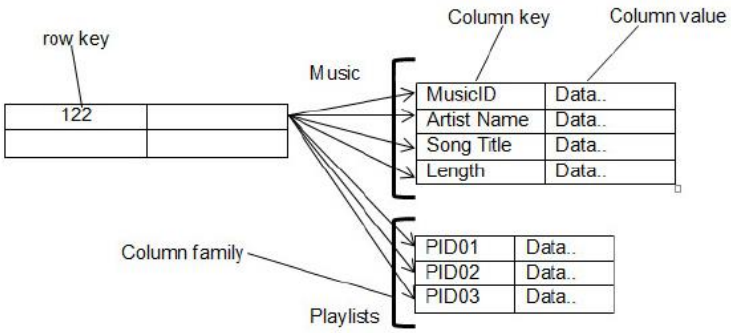
```
{
    Prodotto: "Macbook Air",
    Prezzo: 1199.99,
    Venditore: "Apple"
}
```

KEY-VALUE MODEL

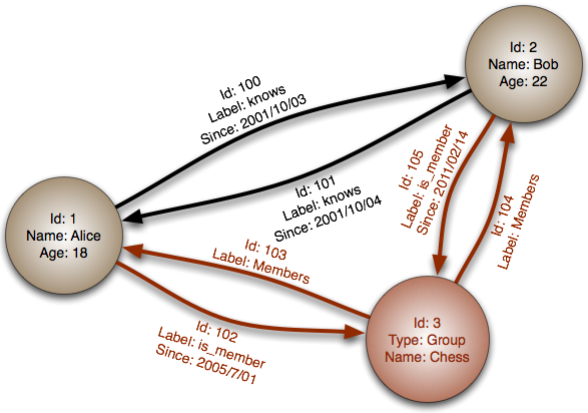
Ogni record consiste in uno o più campi *(per ogni record possono esssere diversi, a differenza dei database relazionali dove lo schema è fisso)* associati ad una chiave

```
KEY     VALUE
001     'GA876BK'
002     3, A12, BCC, 12/11/2019
```

COLUMN-FAMILY MODEL



GRAPH MODEL



2. Il modello relazionale

Nel modello logico relazionale lo schema è la relazione *(tabella)* con vari attributi o campi *(colonne)* e l'istanza *(anche detta occorrenza)* è l'iniseme dei valori che tali attributi assumo in ciascun **record** *(riga, anche detta tupla o matematicamente n-upla)*.

Nota: I DBMS relazionali salvano nella base di dati anche un dizionario dei dati *(metadati)*, ossia la descrizione di tutte le relazioni presenti.

Formalmente la relazione è un sottoinsieme del prodotto cartesiano tra i vari domini degli attributi:

$$\text{Inventario} \subseteq \text{codice}\ (intero)\ \times\ \text{prodotto}\ (stringa)\ \times\ \text{quantità}\ (intero)$$

Nota: Per definizione matematica di sottoinsieme del prodotto cartesiano una relazione non può avere n-uple uguali, inoltre essendo un insieme questa non ha ordinamento (due relazioni con le stesse n-uple in ordine diverso sono da considerarsi equivalenti)

Definizioni

- **Relazione** $R(X_i) = R(A_1, \dots, A_j)$ *Un nome R con un insieme di attributi X diversi*
- **Schema** $S = \{R_1(X_1), \dots, R_n(X_n)\}$ *L'insieme delle relazioni di una base di dati*
- **Tupla** $T: A \rightarrow Dom(A)\ \forall A \in X_i$ *Ad ogni attributo associa un valore del dominio*

A tutti i domini degli attributi *A* viene aggiunto il valore **NULL**, ossia l'elemento nullo, il quale indica un valore sconosciuto, inesistente o non interessante.

Viene definita **superchiave** di una relazione un qualsiasi insieme di attributi tali che identificano univocamente le tuple, dato che non possono esserci due righe uguali nel peggiore dei casi questo insieme sarà tutti gli attributi.

Una superchiave è detta semplicemente **chiave** *(o chiave candidata se ne è presente più d'una)* se è minimale, ossia se non contiene un'altra superchiave, in particolare una chiave viene detta **primaria** se non contiene valori nulli. Un attributo di una relazione è detto **primo** se è membro di una qualsiasi chiave canditata e **non primo** altrimenti.

Ma allora relazione ed una tabella sono la stessa cosa ?

> *Una relazione rispetta la teoria degli insiemi, dunque non può avere due ennuple uguali, mentre una tabella, che può avere più righe uguali, può essere usata per rappresentare una relazione; tuttavia, nel contesto delle basi di dati, i due termini sono utilizzati entrambi per indicare il concetto di relazione*

Vincoli di integrità

Per mantenere corenza nei dati ad ogni schema vengono associate delle proprietà dette **vincoli di integrità** che se soddisfatte esprimono la correttezza di quest'ultimo; si distringuono in:

- Vincoli **intrarelazionali**
  - Vincoli di dominio

    Esprimono vincoli sui valori assuminibli da un dominio, ad esempio $\text{ETA} \geq 18$

  - Vincoli di tupla

    Esprimono vincoli tra le singole tuple, ad esempio CODICE FISCALE IS NOT NULL

  - Dipendenze funzionali *(O vincoli di chiave interna)*

    Esprimono vincoli tra attributi, ad esempio $\text{STATO} \rightarrow \text{CAPITALE}$ perché ogni record della colonna stato determina ogni record della colonna capitale

Nota: Le dipendenze funzionali vengono riprese nel dettaglio nel capitolo 6 di questo documento

→ In caso di tentata violazione di un vincolo intrarelazionale il DBMS rifiuta sempre l'operazione

- Tra i vincoli **interrelazionali** consideriamo i Vincoli d'integrità referenziale *(chiave esterna)*

Impongono che gli attributi di una relazione *(slave)* possono assumere soltanto i valori assunti dalla chiave primaria di un'altra relazione *(master)* **o il valore nullo,** questi attributi nella relazione slave prendono il nome di **foreign key**

→ Reazioni del DBMS alla tentata violazione di un vincolo interrelazionale:

- Se modifico la tabella *slave* l'operazione non viene consentita
- Se modifico la tabella *master* posso adottare diverse politiche:
  - Cascade *Modifica a cascata sulla tabella slave delle righe corrispondenti*
  - Set null *Viene posto a NULL il valore dell'attributo referente*
  - Set default *Viene posto ad un valore di default l'attributo referente*
  - No action *L'operazione non viene consentita*

3. Algebra relazionale

L'algebra relazionale è un linguaggio **procedurale** che permette di esprimere interrogazioni a database organizzati secondo il modello relazionale, è importante perché il DBMS traduce le interrogazioni dall'SQL all'algebra relazionale per valutarne il costo ed eventuali alternative più efficienti. L'algebra relazionale sfrutta due categorie di operatori:

- Operatori su insiemi *(ossia su elementi di relazioni con lo stesso schema)*

    - **Unione** *Elementi di una relazione o dell'altra*
    - **Intersezione** *Elementi di una relazione e dell'altra*
    - **Differenza** *Elementi di una relazione e non dell'altra*

- Operatori su relazioni

    - **Ridenominazione** Modifica lo schema di relazione lasciandone inalterata l'istanza

      $\rho_{B \leftarrow A}(R)$ dove certamente $B \neq A$

    - **Selezione** *(anche chiamata decomposizione orizzontale)*

      Produce un sottoinsieme delle tuple che soddisfano una data condizione

      $\sigma_{Condition}(R)$

    - **Proiezione** *(anche chiamata decomposizione verticale)*

      Restringe gli attributi un sottoinsieme degli attributi originali, se questo non contiene una chiave avrò meno tuple della relazione originale

      $\pi_{Attributes}(R)$

    - **Prodotto cartesiano (CROSS JOIN)**

      Insieme di tutte le possibili combinazioni delle tuple di due relazioni

      $R_1 \times R_2$

    - **Join naturale (NATURAL JOIN)**

      Produce una relazione i cui attributi sono l'unione degli attributi e le cui tuple sono le tuple che presentano valori comuni in corrispondenza degli attributi omonimi (rispetto a tutte le possibili combinazioni)

      $R_1 \bowtie R_2$

Nota: In caso di mancanza di attributi omonimi equivale a fare il prodotto cartesiano, mentre in caso di attributi tutti omonimi equivale a fare l'intersezione. Se ogni tupla contribuisce ad almeno una tupla del risultato il join si dice **completo**, altrimenti si dice **incompleto** e le tuple che non contribuiscono si dicono **dangling tuples** (tuple dondolanti)

- **Theta join (INNER JOIN)**

  Esegue il prodotto cartesiano tra due relazioni, successivamente, per ogni tupla della relazione ottenuta, controlla che sia verificata la condizione imposta, scartando i record che non la soddisfano, *se la condizione è formata da uguaglianze eventualmente in congiunzione (AND) allora si parla di* ***equi-join***

  $R_1 \bowtie_{\text{Condition}} R_2$

Estensioni dell'algebra

- **Join esterno** **(RIGHT OUTER JOIN)** $R_1 ⟖_{Condition|\boldsymbol{opt}} R_2$
  (LEFT OUTER JOIN) $R_1 ⟕_{Condition|\boldsymbol{opt}} R_2$
  (FULL OUTER JOIN) $R_1 ⟗_{Condition|\boldsymbol{opt}} R_2$

Corrisponde ad un theta join dove i record che non fanno join vengono però mantenuti, a costo di essere impostati a NULL, il result set sarà composto dalla prima relazione con tutte le sue tuple da una parte e dalla seconda relazione con le tuple che fanno join oppure valori NULL dall'altra

- **Proiezione generalizzata** Crea una relazione con nuovi attributi che sono funzioni degli attributi esistenti, ad esempio da (ID, LORDO, NETTO) posso proiettare (ID, PROFITTO = LORDO-NETTO)

- **Operatori aggregati** *Più dettagli nel documento sull'SQL*

- **Raggruppamento** *Più dettagli nel documento sull'SQL* $_{\text{Attributi}}\text{G}_{\text{Funzioni}}(\text{R})$

- **Divisione** *Più dettagli nel documento sull'SQL* $\text{r}(\text{R}) \div \text{s}(\text{S})$

4. Calcolo relazionale

Il calcolo relazionale è una famiglia di linguaggi **dichiarativi**, basati sul calcolo dei predicati del prim'ordine: *un linguaggio formale che serve per gestire meccanicamente enunciati e ragionamenti che coinvolgono i connettivi logici, le relazioni e i quantificatori*. Algebra e calcolo relazionali sono equivalenti *(puoi tradurre un'interrogazione dall'algebra al calcolo e viceversa)* ma tuttavia hanno dei limiti: le interrogazioni ricorsive come la **chiusura transitiva** non sono rappresentabili *(ad esempio trovare la scala gerarchica di una azienda)*

Calcolo relazionale sui domini

Le espressioni hanno forma $\{ A_1 : x_1, \dots, A_n : x_n \mid f \}$ dove A_i sono i nomi degli attributi di una relazione ed esprimino la struttura del risultato, x_i sono gli effettivi valori che questi assumono nelle varie tuple ed f è una formula dove compaiono in congiunzione varie condizioni ed eventualmente quantificatori esistenziali (∃) ed universali (∀).

Difetti: è possibile scrivere espressioni insensate e i nomi delle variabili vengono ripetuti spesso

Calcolo relazionale su tuple con dichiarazioni di range *(il più vicino all'SQL)*

A differenza del calcolo sui domini ogni variabile qui rappresenta una tupla, le espressioni hanno forma $\{ x_1.Z_1, \dots, x_n.Z_n \mid x_1(R_1), \dots, x_n(R_n) \mid f \}$ dove x_i sono tuple prese sugli insiemi di attributi Z_i *(per indicare che voglio tutti gli attributi di un certo Z_i uso *)* delle una relazioni R_i *(range list)* ed f è una formula dove compaiono in congiunzione varie condizioni ed eventualmente quantificatori esistenziali (∃) ed universali (∀).

Difetti: non si possono esprimere interrogazioni i quali risultati possono provenire da due o più relazioni *(quelle che in algebra sfruttano le unioni)* perché ogni variabile *(tupla)* ha un solo range *(elenco di possibili attributi)*

Supponiamo di voler formulare un'interrogazione che restituisca l'unione di $R_1(A)$ *e* $R_2(A)$

- $\{ x_1.* \mid x_1(R_1) \mid true \}$ → Il risultato è composto dalle tuple di una sola relazione
- $\{ x_1.*, x_2.* \mid x_1(R_1), x_2(R_2) \mid true \}$ →Il risultato è composto solo dalle tuple di entrambe le relazioni, mentre l'unione vuol dire prendere le tuple di una o dell'altra relazione

Nota: è possibile esprimere l'unione di due relazioni identiche (il range è lo stesso).

Esempi di interrogazioni nei vari linguaggi

Data la base di dati : ***Impiegati***(*Matricola, Nome, Eta, Stipendio*) , ***Supervisione***(*Capo, Impiegato*)

1) *Trova nome e stipendio dei capi degli impiegati che guadagnano più di 40'000*
2) *Trova matricola e nome dei capi degli impiegati che guadagnano tutti più di 40'000*

In ***algebra relazionale***

$$\pi_{NomeC,StipendioC}\left(\begin{array}{c}\rho_{MatricolaC,NomeC,StipendioC,EtaC\leftarrow Matricola,Nome,Stipendio,Eta}(\boldsymbol{Impiegati})\\ \bowtie_{MatricolaC=Capo}\ \boldsymbol{Supervisione}\ \bowtie_{Impiegato=Matricola}\ \sigma_{Stipendio>40}(\boldsymbol{Impiegati})\end{array}\right)$$

$$\pi_{Matricola,Nome}\left(\begin{array}{c}\boldsymbol{Impiegati}\ \bowtie_{Matricola=Capo}\ [\pi_{Capo}(\boldsymbol{Supervisione})\ -\\ \pi_{Capo}(\boldsymbol{Supervisione}\ \bowtie_{Impiegato=Matricola}\ \sigma_{Stipendio\leq 40}(\boldsymbol{Impiegati})]\end{array}\right)$$

Nel ***calcolo sui domini*** *(andrebbero indicati tutti gli attributi anche con variabili inutilizzate)*

$$\{\ NomeC: nc, StipendioC: sc\ |$$
$$\boldsymbol{Impiegati}(Matricola: \boldsymbol{m}, Nome: n, Età: e, Stipendio: s)\ \wedge s > 40$$
$$\wedge\ \boldsymbol{Supervisione}(Impiegato: \boldsymbol{m}, Capo: \boldsymbol{c}\ )$$
$$\wedge\ \boldsymbol{Impiegato}(Matricola: \boldsymbol{c}, Nome: nc, Età: ec, Stipendio: sc)\}$$

$$\{\ Matricola: m, Nome: n\ |$$
$$\boldsymbol{Impiegato}(Matricola: \boldsymbol{c}, Nome: n, Età: e, Stipendio: s)$$
$$\wedge\ \boldsymbol{Supervisione}(Impiegato: m, Capo: \boldsymbol{c}\ )$$
$$\wedge\ \neg\ (\exists m'(\exists n'(\exists e'(\exists s'\ \boldsymbol{Impiegato}(Matricola: \boldsymbol{m'}, Nome: n', Età: e', Stipendio: s')$$
$$\wedge\ \boldsymbol{Supervisione}(Impiegato: \boldsymbol{m'}, Capo: \boldsymbol{c}\ )\ \wedge\ s' \leq 40))))\}$$

Nel ***calcolo sulle tuple***

$$\{\ NomeC, StipC: i'.(Matricola, Nome)\ |$$
$$i'(Impiegato), s(Supervisione), i(Impiegato)\ |$$
$$i'.Matr = s.Capo \wedge\ s.Impiegato = i.Matricola\ \wedge i.Stipendio > 40\ \}$$

$$\{\ i.(Matricola, Nome)\ |\ i(Impiegato),\ s(Supervisione)\ |$$
$$i.Matr = s.Capo \wedge\ \neg\ [\exists i'(Impiegato)(\exists s'(Supervisione)$$
$$(s.Capo = s'.Capo \wedge s'.Impiegato = i'.Matricola\ \wedge i'.stipendio \leq 40))]\ \}$$

5. Progettazione

La progettazione di una base di dati si articola in tre fasi principali: creazione di un **modello concettuale** con relativa documentazione, **ristrutturazione** del modello concettuale in base al carico applicativo e **traduzione** in un modello logico.

Modello concettuale ER

Sono modelli che definiscono una base di dati in maniera grafica, il più famoso è il modello **ER** *(Entity-Relationship)* il quale opera sui seguenti costrutti:

ENTITÀ

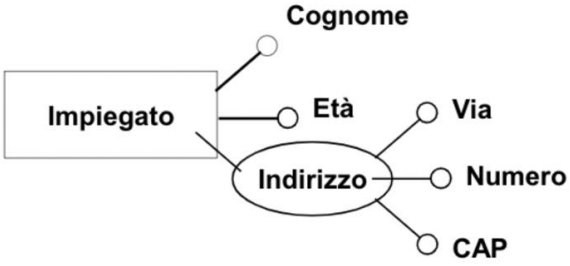
E' una classe di oggetti con proprietà comuni ed esistenza autonoma, graficamente è rappresentata da un rettangolo con al suo interno un nome espressivo e al singolare, che defisice gli oggetti espressi.

LEGAME

E' un legame logico tra due o più entità, graficamente è rappresentato da un rombo con al suo interno un sostantivo *(evitare verbi poiché definiscono un verso)* al singolare il quale definisce il legame. Sono consentiti anche legami diversi su entità uguali, legami ricorsivi con o senza ruoli.

ATTRIBUTO

E' una proprietà elementare di un'entità o di una relationship, può anche essere composto *(raggruppare attributi legati da un certo significato).*



CARDINALITÀ

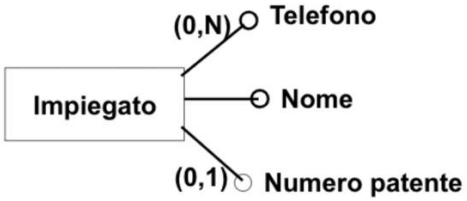
Si parla di **cardinalità di relationship** se considero le coppie di valori i quali esprimono il numero minimo e massimo di occorrenze della relationship in cui ciscuna occorrenza di entità può comparire *(si usa 0 per indicare partecipazione opzionale, 1 partecipazione obbligatorie ed N per non limitare)*.

Con riferimento alle cardinalità massime possiamo caratterizzare una relationship come:

- Uno a uno $(x, 1) \rightarrow (y, 1)$
- Uno a molti $(x, 1) \rightarrow (y, N)$ *o viceversa*
- Molti a molti $(x, N) \rightarrow (y, N)$

In questo esempio ogni impiegato ha almeno un incarico e al massimo 5 incarichi, mentre un incarico può non avere impiegati che ci lavorano o avere al massimo 50 impiegati che ci lavorano.

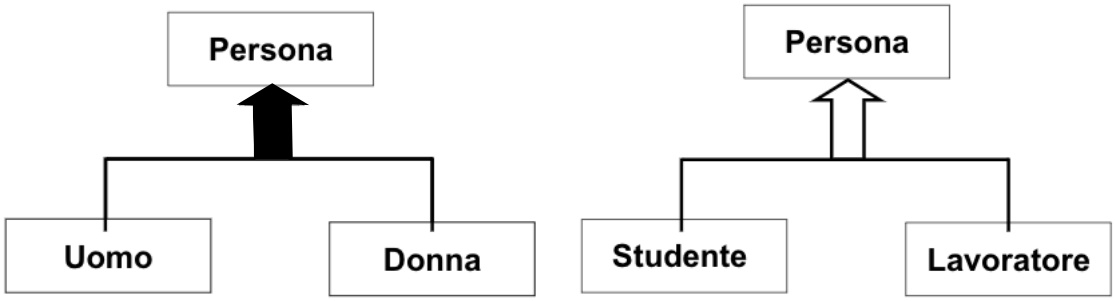
Si parla di **cardinalità di attributo** se associamo una cadinalità a un attributo per specificarne l'opzionalità oppure il *multivalore*



GENERALIZZAZIONE

E' uno strumento che mette in relazione una o più sottoentità figlie con un'entità genitore che le comprende come casi particolari, le entità figlie ereditano tutti gli attributi e i legami dell'entità genitore. Per non complicare le cose considereremo solo le generalizzazioni **esclusive**, ossia quelle in cui ogni occorrenza dell'entità genitore è al più una delle entità figlie *(ed esempio: una dipendente non può essere capo e tirocinante allo stesso tempo)*, in particolare distingueremo tra:

Generalizzazione **totale** *(figura a sinistra)* se ogni occorrenza dell'entità genitore è occorrenza di almeno una delle entità figlie e generalizzazione **parziale** *(figura a destra)* nel caso contrario.



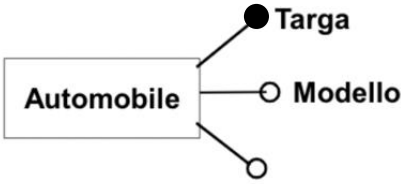
IDENTIFICATORE

E' uno strumento per identificare univocamente le occorrenze di un'entità, corrisponde alla chiave primaria e si indica con un pallino pieno. Si parla di **identificatore interno** se è costituito da attributi dell'entità mentre si parla di **identificatore esterno** se è costituito da attributi dell'entità uniti ad identificatori interni di altre entità.

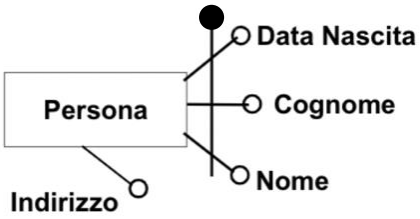
Un legame che collega entità con identificatore interno è detto **not identifying**
Un legame che collega entità con identificatore esterno è detto **identifying**
(Questo è possibile solo se l'entità da identificare partecipa in modalità uno ad uno)

***Attenzione*: Le relazioni non possono avere chiavi**, non ha alcun senso! Se in fase di progettazione non tornano le chiavi allora è necessario trasformare la relazione in un'entità a parte.

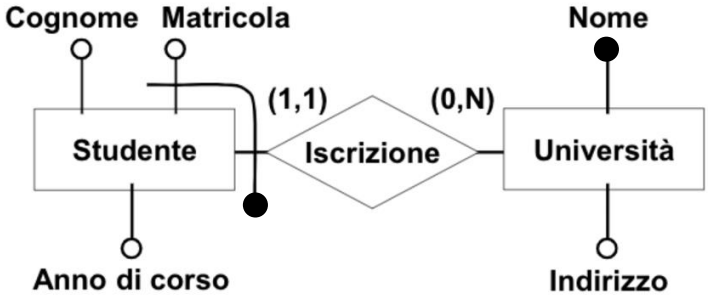
Identificatore interno (unico)



Identificatore interno (multiplo)



Identificatore esterno



In questo esempio la chiave di studente è formata dalla matricola e dal nome dell'università a cui è iscritto, durante la traduzione al modello relazionale l'attributo nome di Università verrà copiato in Studente e verrà fissato un vincolo di integrità referenziale tra i due.

Ristrutturazione del modello concettuale

Prima di tradurre uno schema concettuale in modello logico è necessario ristrutturarlo, questo perché in primo luogo l'algoritmo di traduzione non si applica a tutti i costrutti dello schema concettuale *(alcuni aspetti non sono direttamente rappresentabili)*, inoltre è dovesoro ottimizzare lo schema concettuale della base di dati in base alle operazioni che essa dovrà svolgere. La ristrutturazione è a cura del progettista e si articola in quattro fasi:

ELIMINAZIONE DELLE GENERALIZZAZIONI

Il modello relazionale non può rappresentare direttamente le generalizzazioni, abbiamo dunque tre diverse possibilità di ristrutturazione:

- **Accorpamento delle figlie nel genitore**

  Al genitore vengono aggiunti tutti gli attributi delle figlie più un nuovo attributo *tipo* per distinguere le figlie, questa soluzione presenterà diversi valori *null* e conviene se gli accessi al padre e alle figlie sono contestuali, ossia se ad ogni lettura mi interessa sapere sempre gli attributi del padre e delle figlie

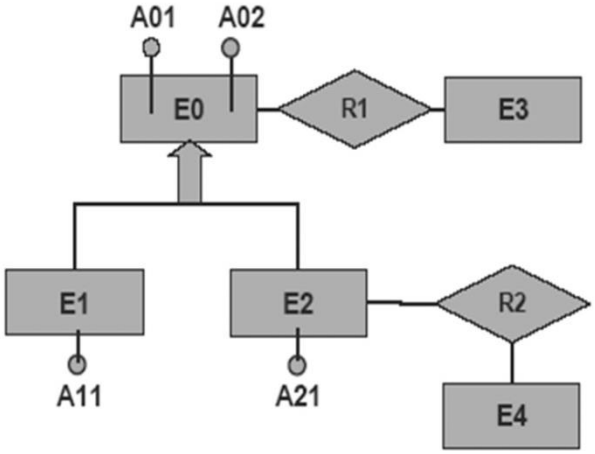
- **Accorpamento del genitore nelle figlie** *(Possibile solo se la generalizzazione è totale)*

  Gli attributi del genitore vengono distribuiti su tutte le figlie ed eventuali relazioni vengono moltiplicate. Questa soluzione conviene se gli accessi avvengono solamente alle figlie e solamente una figlia alla volta *(non mi servono sapere gli attributi di due o più figlie insieme ad ogni accesso)*

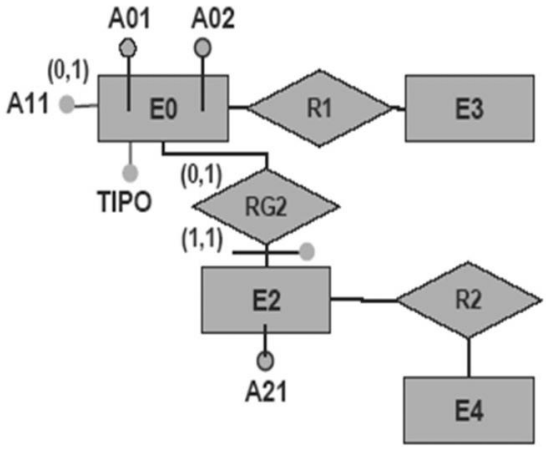
- **Sostituzione della generalizzazione con relazioni**

  E' la soluzione che più mantiene la struttura invariata, le figlie infatti diventano entità diverse con chiave nel padre. Conviene si effettuano accessi separati alle entità figlie e al padre.

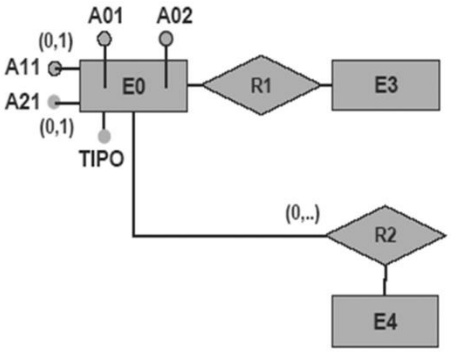
Esempio di modello da ristrutturare



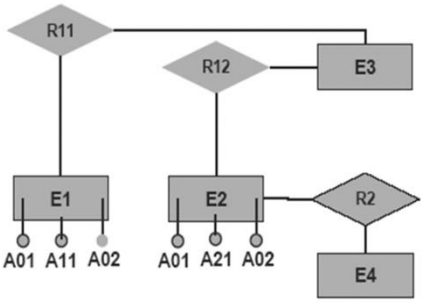
Ristrutturazione ibrida



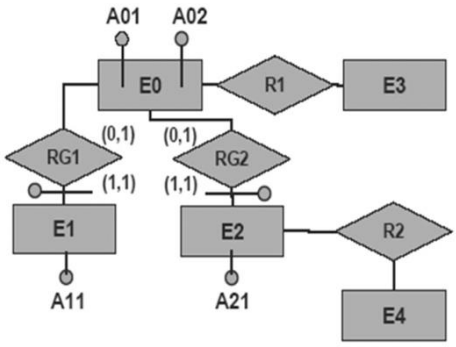
Accorpamento figlie nel genitore



Accorpamento genitore nelle figlie



Sostituzione della general.

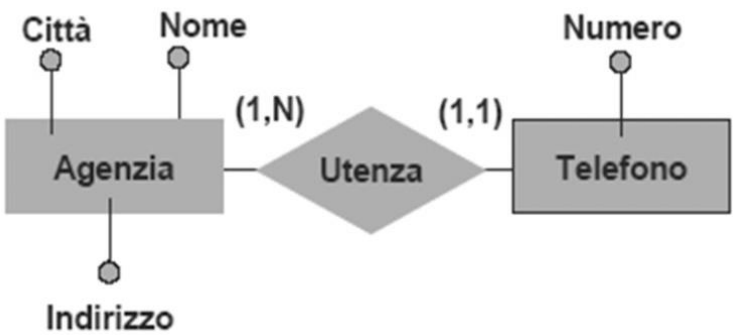


ELIMINAZIONE ATTRIBUTI MULTIVALORE

Il modello relazionale non può rappresentare direttamente gli attributi multivalore, potrebbe venire in mente di creare allora *n* record uguali con attributo multivalore diverso oppure *n* attributi in una relazione per replicare gli *n* attributi multivalore, tuttavia queste soluzioni sono entrambe scadenti, la soluzione ottimale in questo caso è quella di creare una nuova entità per rappresentare l'attributo multivalore

Originale

Dopo la ristrutturazione



PARTIZIONAMENTO O ACCORPAMENTO DI ENTITÀ E RELATIONSHIP

Questa sezione fa parte delle ristrutturazioni effettuate per rendere più efficienti le operazioni *(considerando che ad ogni accesso si legge sempre l'intero record)*, i casi principali sono:

- **Partizionamento di entità**

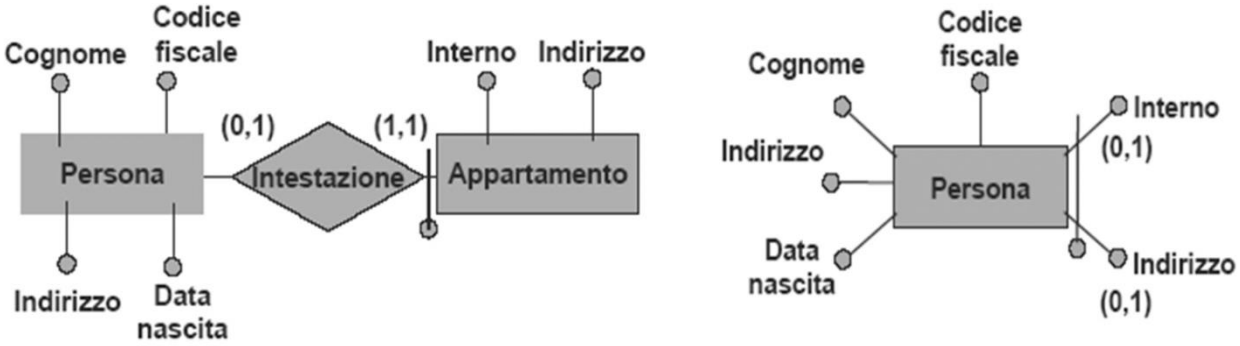
  Un'entità viene divisa in due o più parti *(con una certa logica)* per evitare che nel modello logico si venga a formare una relazione con un numero alto di attributi

  Prima *Dopo*

- **Accorpamento** di entità o relationship

  Diverse entità o relationship legate concettualmente vengono unite in un'unica entità

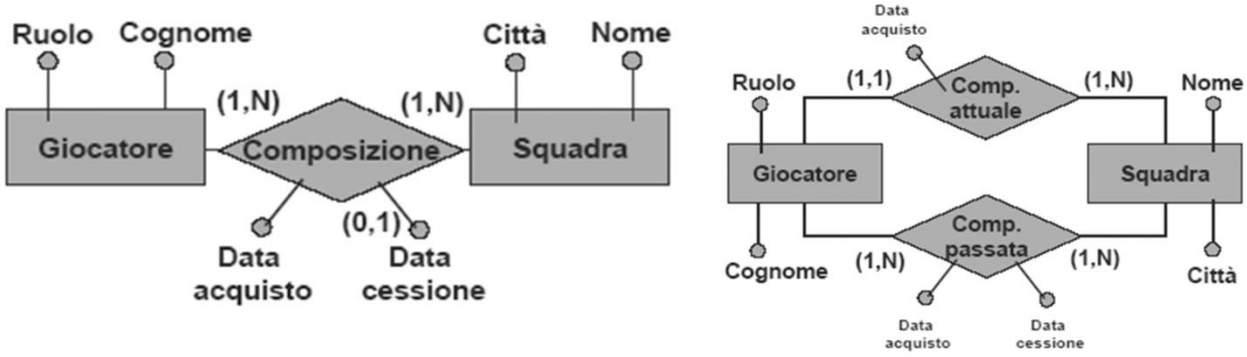
  Prima *Dopo*



- **Partizionamento di relationship**

  Una relationship viene divisa in due o più parti, questo serve a dividere una parte poco usata ma importante da mantenere da una più recente e dunque più usata

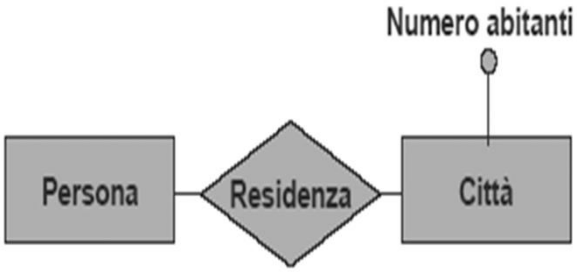
  Prima *Dopo*



ANALISI DELLE RIDONDANZE

Una ridondanza è un'informazione significativa ma derivabile da altre, ridondanze possono essere: attributi derivabili dalla stessa o altre entità, associazioni derivabili dalla composizione di altre associazioni *(dunque la presenza di cicli tra associazioni)*. Le ridondanze offrono uno strumento utile per evitare complessi calcoli ad ogni operazione, bisogna valutare con cura quali hanno senso e quali vanno rimosse in base alle operazioni che dovrà compiere la base di dati. Per effettuare questa distinzione è necessario avere degli indicatori delle prestazioni di ogni operazione, questi indicatori sono: *spazio*, ossia il numero di occorrenze previste nelle tavole dei volumi e *tempo*, ossia il numero di occorrenze visitate per portare a termine un'operazione *(in particolare le letture L valgono l'unità mentre le scritture S valgono il doppio delle letture)*.

Esempio di valutazione di una ridondanza



Operazione 1 : Inserimento di una nuova persona con relativa residenza 500 volte al giorno

Operazione 2 : Stampa di tutti i dati di una città 2 volte al giorno

1. Creare una **tavola degli accessi** per ogni operazione e per ogni ridondanza

In presenza di ridondanza

Operazione 1

Concetto	Costrutto	Accessi	Tipo
Persona	Entità	1	S
Residenza	Relazione	1	S
Città	Entità	1	L
Città	Entità	1	S

Operazione 2

Concetto	Costrutto	Accessi	Tipo
Città	Entità	1	L

In asssenza di ridondanza

Operazione 1

Concetto	Costrutto	Accessi	Tipo
Persona	Entità	1	S
Residenza	Relazione	1	S

Operazione 2

Concetto	Costrutto	Accessi	Tipo
Città	Entità	1	L
Residenza	Relazione	5000	L

2. Combinare la **tavole dei volumi** e degli accessi 3. **Confrontare** i costi ottenuti

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

Ridondanza	Operazione 1	Operazione 2
Senza	2000	10'002
Con	3500	2

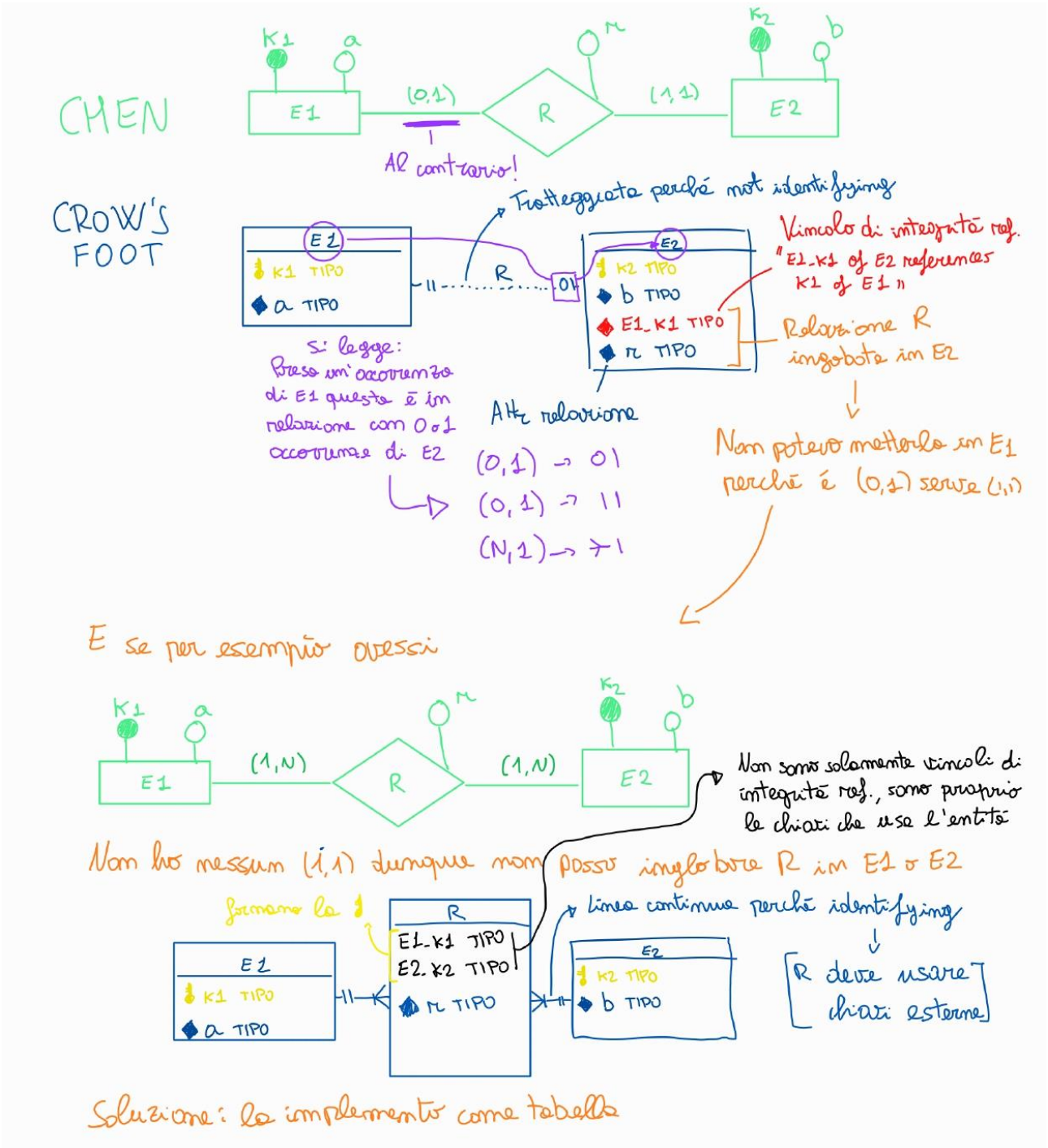
In questo esempio chiaramente conviene mantenere la ridondanza

Traduzione verso il modello logico *(in particolare quello relazionale)*

Una volta ristrutturato il modello concettuale è pronto per essere tradotto in modello logico, questo avviene secondo la semplice idea di base che ogni entità diventa una relazione sugli stessi attributi ed ogni associazione diventa una relazione sugli attributi propri più le chiavi delle entità coinvolte, tutto questo avviene senza mai modificare la struttura espressa dal modello concettuale.

Notazione Crow's Foot

Fa da intermediario tra la notazione ER e il modello relazionale



6. Normalizzazione

L'obbiettivo della normalizzazione è quello di valutare la qualità della progettazione degli schemi relazionali ed eventualmente modificarli per soddisfare certi standard di qualità

Una dipendenza funzionale esprime un legame semantico tra due gruppi di attributi di una relazione, grazie alle dipendenze è possibile definire forme normali della relazione, ognuna delle quali garantisce l'assenza di determinati difetti, il processo che porta una relazione in una forma normale è detto **normalizzazione**

Prima di parlare di normalizzazione è necessario introdurre alcuni concetti che serviranno più avanti

Elenco di definizioni

- Data una relazione $R(U)$ e dati X, Y non vuoti $\subset U$ esiste una **dipendenza funzionale** o *Functional Dependency (FD)* da X a Y (*Notazione:* $X \to Y$ *"X determina Y")* se, per ogni coppia di tuple t_1 e t_2 di R con gli stessi valori su Z, risulta che t_1 e t_2 hanno gli stessi valori anche su Y

- Una generica FD $X \to Y$ si dice **completa** se per ogni $W \subset X$ non vale $W \to Y$, in altre parole se togliendo qualcosa all'insieme di attributi che genera la dipendenza questa non è più valida

- Se la condizione di prima non è verificata si dice che la FD è **parziale**

- Una generica FD $X \to Y$ si dice **non banale** se nessun attributo in Y appartiene ad X

- Sia F un insieme di dipendenze funzionali di una relazione e sia $X \to Y$ una generica dipendenza funzionale, si dice che F **implica** $X \to Y$ se in ogni istanza della relazione che verifica tutte le dipendenze di F è verificata anche $X \to Y$

- Sia F un insieme di dipendenze funzionali, la **chiusura di F** è l'insieme di tutte le dipendenze funzionali che esso implica: $F^+ = \{X \to Y \mid F \Longrightarrow X \to Y\}$

- Sia X un insieme di attributi di una relazione, e sia F un insieme di dipendenze funzionali per tale relazione, si dice **chiusura di X** e si indica con $X^{+(F)}$ l'insieme di attributi della relazione che dipendono da X secondo F

- Due insiemi di FD F **e** G **sono equivalenti** se $F^+ = G^+$, ossia se ogni dipendenza in F è implicata da G *(cioè appartiene a G^+)* e vicenersa, oppure in altre parole se per ogni $X \to Y$ in F ottengo che $Y \in X^{+(G)}$ e viceversa
- Si verifica una **ridondanza di attributi** quando questi possono essere semplificati:

- A destra: $\{ X \to YZ \} = \{ X \to Y, X \to Z \} \to$ *ho semplificato YZ*
  Le FD con a destra un solo attributo si dicono **semplici** o standard

- A sinistra: $\{ X \to Y,\ Y \to Z,\ XZ \to W \} = \{ X \to Y,\ Y \to Z,\ X \to W \}$
  Gli attributi semplificabili a sinistra si dicono **estreanei** perché inutili

- Una generica **dipendenza** X → Y in *F* è **ridondante** se, dopo averla rimossa, $Y \subseteq X^{+(F)}$, in altre parole se le dipendenze rimanenti erano già sufficienti per determinare *Y* da *X*, ad esempio $\{ X \to Y,\ Y \to Z,\ X \to Z \} \to X \to Z$ *è ridondante e va rimossa*

- Un insieme *F* di FD è detto **minimale** se non presenta ridondanze di attributi o di dipendenze

- Dato un generico insieme *F* di *FD*, con **copertura minimale** di *F* si intende un insieme minimale ed equivalente ad *F*, la copertura minimale generalmente non è unica.

  Posso **ottenere una copertura minimale** di *F* seguendo questi semplici passi:
  1. Portare ogni dipendenza in dipendenza semplice
  2. Rimuovere gli attributi estranei
  3. Rimuovere le dipendenze ridondanti

Dai ragionamenti sopra segue che, dato un insieme di FD F su una generica relazione:
- Gli attributi che stanno sempre a sinistra in ogni FD di *F* compaiono in tutte le chiavi, poiché essi determinano tutti gli attributi
- Gli attributi che stanno sempre a destra in ogni FD di *F* non compaiono in alcuna chiave poiché essi sono sempre determinati da altri attributi

Alla luce di questi nuovi concetti possiamo formulare le seguenti equivalenze:
Data una relazione R definita su un insieme di attributi U e dato un insieme F di FD di R
Le seguenti affermazioni sono esattamente equivalenti!

- La dipendenza "*X* determina *U*" è presente tra tutte le possibili dipendenze della relazione *R*
- $X \to U \in F^+$ *(X → U appartiene alla chiusura di F)*
- $X \to U$ è implicata da *F*
- $U \subseteq X^{+(F)}$ *(Gli attributi U dipendono da X secondo F)*
- *X* è superchiave di *R*

Teorema

Generalizzando quanto detto sopra, la dipendenza $X \to Y$ è implicata da *F* se e solo se gli attributi *Y* dipendono da *X* secondo *F,* in linguaggio matematico $X \to Y \in F^+ \iff Y \subseteq X^{+(F)}$

Vediamo ora delle regole utili per manipolare insieme di FD che ci serviranno nella prossima pagina

Regole di inferenza di Armstrong

1. **Riflessività** Se $Y \subseteq X$ allora $X \to Y$ *(FD banali)*

2. **Additività** Se $X \to Y$ allora $XZ \to YZ$ per qualunque Z

   Dimostrazione

   Supponiamo per assurdo che:

   - $X \to Y \Longrightarrow (1)\ t_1[X] = t_2[X]\ ,\ (2)\ t_1[Y] = t_2[Y]$
   - $XZ \nrightarrow YZ \Longrightarrow\ (3)\ t_1[XZ] = t_2[XZ]\ ,\ (4)\ t_1[YZ] \neq t_2[YZ]$

   Ma allora:

   - $Da\ (1)\ e\ (3) \Longrightarrow (5)\ t_1[Z] = t_2[Z]$
   - $Da\ (2)\ e\ (5) \Longrightarrow\ t_1[YZ] = t_2[YZ]$

   E questo è assurdo

3. **Transitività** Se $X \to Y$ e $Y \to Z$ allora $X \to Z$

   Dimostrazione

   Supponiamo per assurdo che:

   - $X \to Y \Longrightarrow (1)\ t_1[X] = t_2[X]\ ,\ (2)\ t_1[Y] = t_2[Y]$
   - $Y \to Z \Longrightarrow (2)\ t_1[Y] = t_2[Y]\ ,\ (4)\ t_1[Z] = t_2[Z]$
   - $X \nrightarrow Z \Longrightarrow (5)\ t_1[X] = t_2[X]\ ,\ (6)\ t_1[Z] \neq t_2[Z]$

   Ma questo è assurdo

4. **Unione** Se $X \to Y$ e $X \to Z$ allora $X \to YZ$

   Dimostrazione

   Dalle ipotesi ottengo $(1)\ XZ \to YZ\ \ (2)\ XX = X \to XZ$

   Allora $X \to XZ \to YZ$ e per transitività $X \to YZ$

5. **Pseudotransività** Se $X \to Y$ e $WY \to Z$ allora $XW \to Z$

6. **Decomposizione** Se $Z \subseteq Y$ e $X \to Y$ allora $X \to Z$

 # Algoritmo per il calcolo della chiusura di un insieme di FD # [Utilizzando le regole di Armstrong] (Complessità esponenziale)

/*

Questo algoritmo è puramente teorico e ha zero utilità pratica, serve solamente a definire un procedimento formale ed implementabile in un calcolatore *per ricavare la chiusura di un insieme di FD, che potrà essere successivamente usata per trovare le chiavi della relazione oppure per verificare l'equivalenza di due insiemi di dipendenze.*

*/

Per iniziare $F^+ = F$

Finchè F^+ cambia **ripeti :**

Per ogni dipendenza f nella chiusura F^+ :
Cerca le dipendenze banali di f ***(riflessività)***
Cerca le dipendenze arricchite di f ***(additività)***
Aggiungi alla chiusura F^+ le eventuali dipendenze trovate

Per ogni coppia di dipendenze f_1 e f_2 nella chiusura F^+ :
Controlla se possono essere combinate ***(transitività)*** e in tal caso aggiungi le combinazioni trovate alla chiusura F^+

 # Algoritmo per il calcolo della chiusura di un insieme di attributi # (Complessità lineare)

/*

Questo algoritmo ha **notevole utilità pratica**, spesso infatti quello che ci interessa è verificare solamente se una certa dipendenza appartiene alla chiusura, e non l'intera chiusura (come abbiamo visto nelle pagine precedenti)

*/

Per iniziare $X^+ = X$

Finchè X^+ cambia **ripeti :**

Per ogni dipendenza $Sx \rightarrow Dx$ in F:
Se a sinistra ci sono attributi di X^+ e a destra no
Sicuramente dipendono dall'intero X dunque aggiungili a X^+

Forme normali

BOYCE-CODD *(BCNF)*

Una relazione è in forma normale di Boyce-Codd **se in ogni sua possibile dipendenza funzionale non banale il membro di sinistra è una superchiave**, grazie ad un teorema sappiamo che se un insieme F di FD è costruito in questo modo allora anche la sua chiusura lo è *(e dunque la relazione è in BCNF)*, allora possiamo definire un algoritmo per la verifica:

```
# Algoritmo per verificare se una relazione R(U) è in BCNF #
                (Complessità quadratica)

        # Questo algoritmo ha notevole utilità pratica #

Scorri tutte X → Y in F :
        Calcola la chiusura di X per F
        Se X → Y è non banale e U ⊄ X^+(F)  (X non è superchiave) ritorna FALSO

Ritorna VERO (è in BCNF)
```

Se una relazione è in BCNF ?

Ottimo, vuol dire che la relazione non presenta anomalie di alcun tipo *(ripetizioni e ridondanze che portano ad errori e problemi di aggiornamento, inserimento ed eliminazione di record)*

Se invece non è in BCNF ?

Se voglio evitare anomalie devo normalizzarla, ossia decomporla in altre relazioni che siano BCNF

Come la decompongo ?

Il **primo tentativo** che posso fare per normalizzare in BCNF una relazione è semplicemente dividerla in tante relazioni quante sono le dipendenze *(e sperare che vada tutto bene)*

Come faccio a capire se è andato tutto bene ?

Semplicemente ricompongo le relazioni ottenute tramite join naturale:

- Se ottengo di nuovo la relazione originale è andato tutto bene
- Se ottengo qualcosa di diverso dalla relazione originale vuol dire che ho perso un legame importante *(si dice che ho avuto una **decomposizione con perdita**)*

Esiste un modo più veloce di controllare che è andato tutto bene senza fare il join ?

Condizione sufficiente *(non necessaria!)* **per decomporre** R in R_1 ed R_2 **senza perdita** è che gli attributi comuni ad R_1 ed R_2 siano chiave in almeno una relazione, in gerco tecnico:

Per dividere senza perdita $R(X)$ *in* $R_1(X_1)$ *ed* $R_2(X_2)$ *dove* $X = X_1 \cup X_2$
ed $X_0 = X_1 \cap X_2$ *devo avere che* $X_0 \to X_1 \in F^+$ *oppure* $X_0 \to X_2 \in F^+$

Esiste un procedimento che mi assicuri di ottenere una decomposizione in BCNF senza perdita ?

 # Algoritmo per decomporre *R(U)* senza perdita in BCNF #
[Assumendo che le FD in *F* siano semplici](Complessità esponenziale)

```
/*

Questo algoritmo è puramente teorico e ha zero utilità pratica, serve solamente
a definire un procedimento formale ed implementabile in un calcolatore

*/

Decomponi (R,F):
        Se una dipendenza di F non va bene (X → Y con Y non superchiave)
        Allora :
                Divido la relazione in cui vale F in due parti:
                    - R1 su U − Y → Tolgo a R gli attributi che dipendono
                    - R2 su X ∪ Y → Lascio solo gli attributi della dipendenza

                Decomponi ( R1, F_{U−Y}) → Dipendenze che non coinvolgono Y
                Decomponi ( R2, F_{X∪Y}) → Dipendenze che coinvolgono solo X e Y

Decomposizione conclusa
```

Il problema di questo algoritmo sta nel trovare la proiezione di un gruppo di dipendenze funzionali, in altre parole dato un gruppo di FD su un'insieme di attributi U bisogna tenere solo le dipendenze che coinvolgono *(sia a sinistra che a destra)* un sottoinsieme di attributi $X \subseteq U$

 # Algoritmo per calcolare la proiezione di *F* su *X* #
(Complessità esponenziale)

```
/*

Questo algoritmo è puramente teorico e ha zero utilità pratica, serve solamente
a definire un procedimento formale ed implementabile in un calcolatore

*/

Per iniziare F_x = ∅

Per ogni S ⊂ X: → Sottoinsieme proprio S degli attributi X
    Per ogni A ∈ X: → Attributo singolo A appartenente ad X
        Se A ∉ S → A non è in S
        E ∄S' ⊆ S | S' → A ∈ F_x → Non esiste alcun sottoinsieme di S tale che
                                   S' → A è nel risultato
        Ed inoltre A ⊆ S^{+(F)} → A dipende da S secondo F
        Allora F_x = F_x ∪ S' → A → Aggiungo S' → A al risultato
```

Normalizzando una relazione in BCNF tramite l'algoritmo di prima potrebbe non essere sufficiente a garantirne la correttezza: può verificarsi una **perdita di dipendenze** *(ad esempio una dipendenza che dopo la decomposizione interessa attributi su relazioni diverse, dunque non più in relazione tra loro)*, per essere sicuri che eventuali modifiche alle relazioni decomposte rispettino ancora tutte le FD della relazione originale bisognerebbe ad ogni modifica verificare tramite join che questa equivalga a modificare allo stesso modo la tabella originale, ma ciò è chiaramente insostenibile.

Come posso sapere se la decoposizione che ho scelto ha conservato le dipendenze ?
Una decomposizione conserva le dipendenze se ciascuna delle dipendenze della relazione originaria coinvolge attributi che compaiono tutti in una singola relazione decomposta, in gerco tecnico:

La decomposizione di $R(U)$ in $R_1(X)$ ed $R_2(Y)$ dove $X, Y \subseteq U$ è senza perdita di dipendenze se $(F_X \cup F_Y)$ è equivalente ad F (ossia hanno le chiusure uguali)

<u>Nota</u>: ***Prima di decomporre è sempre bene minimizzare F*** *perché altrimenti potrebbero risultare mancanti delle dipendenze che in realtà erano ridondanti!*

<u>Ricapitolando</u>: noi vorremmo una decomposizione che garantisse sempre una forma BCNF senza perdite e con conservazione delle dipendenze, ma al momento abbiamo un algoritmo che garantisce solo l'assenza di perdite *(e tra l'altro con complessità esponenziale)*.

Ma allora che alternative abbiamo ?
Possiamo ricorrere ad una forma normale meno stringente *(dunque che permette alcune anomanie e ridondanze)* chiamata **TERZA FORMA NORMALE** *(3NF)*, secondo la quale è sufficiente che, per ogni FD non banale $X \rightarrow Y$ sia verificata almeno una tra le seguenti:

- X è superchiave della relazione *(Ossia la dipendenza va bene per la BCNF)*
- Ogni attributo in Y è primo *(ossia è parte di almeno una chiave, NON superchiave)*

<u>Nota</u>: *verificare se una relazione è in 3NF è un problema di complessità esponenziale in quanto, per ogni possibile dipendenza, devo verificare che ogni possibile sottoinsieme di attributi a destra sia parte di almeno una chiave*

Teorema
Ogni relazione può essere portata in 3NF senza perdita e con conservazione delle dipendenze

Come posso normalizzare una relazione in 3NF ?
Prima di tutto minimizzo l'insieme delle dipendenze, poi scelgo uno dei due algoritmi illustrati

Se la chiave è unica *(ne esiste solo una)* **questi algoritmi producono una relazione in BCNF!**
Potrei ottenere una relazione BCNF ***anche se la chiave non è unica****, ma non in ogni caso*

 # Primo algoritmo per la decomposizione in 3NF #
(Assumiamo che le FD in F siano semplici e sia R relazioni sugli attr. U)

/

Questo algoritmo, sebbene abbia utilità pratica, **è sconsigliato** poiché l'algoritmo successivo è equivalente a questo ma di più facile comprensione

**/*

Passo 1: conserva il join (Simile a BCNF)
Se una dipendenza di F non va bene *($X \to Y$ con Y non superchiave)*
Allora :
Divido la relazione in cui vale F in due parti:
- R_1 su $U - Y$ → *Tolgo a R gli attributi che dipendono*
- R_2 su $X \cup Y$ → *Lascio solo gli attributi della dipendenza*

Passo 2: conserva le dipendenze
Per ogni dipendenza persa creo una relazione che ha come attributi gli stessi attributi della dipendenza

 # Secondo algoritmo per la decomposizione in 3NF #
(Assumiamo che le FD in F siano semplici e sia R relazioni sugli attr. U)

Questo algoritmo ha **notevole utilità pratica**

Passo 1: conserva le dipendenze
Per ogni dipendenza creo una relazione che ha come attributi gli stessi attributi della dipendenza

Aggiungo una relazione in cui metto gli eventuali attributi esclusi dalle FD
Se due FD si determinanpìo a vicenda le fondo in una relazione a più chiavi

Passo 2: conserva il join
Se non c'è già una relazione la quale contiene una chiave della relazione originaria allora la creo

Nota finale: esistono anche altre forme normali *(anche se quasi totalmente inutili)*

- **Prima forma normale** *(1NF)*, che è parte della definizione del modello relazionale e dunque sempre verificata
- **Seconda forma normale** *(2NF)* è verificata quando in una relazione le dipendenze di ogni attributo non primo da ogni chiave della relazione sono complete *(implicato da 3NF e BCNF)*

7. Guida per la progettazione di una base di dati

I seguenti step sono delle linee guida sulle fasi di progettazione di una base di dati e della sua documentazione:

1. **Introduzione** *(Descrivere lo scopo della base di dati)*

2. **Glossario** dei termini

Termine	Descrizione	Sinonimi	Collegamenti *(Entità)*

3. Descrizione del **diagramma ER** secondo le *"business rules" (vedi pagina dopo)*

4. Spiegazione delle **ristrutturazioni** eseguite sul diagramma ER

5. **Operazioni** da compiere sui dati
   a. Tavola dei volumi *(anche ipotizzati o derivati)* della base di dati

Concetto	Tipo	Volume	Origine *(Derivato, ipotizzato…)*

   b. Elenco delle operazioni strutturato come segue:
      i. *Descrizione operazione, input ed output*
      ii. *Porzione di diagramma ER interessata*
      iii. *Porzione della tavola dei volumi interessata*
      iv. *Tavola degli accessi e studio delle ridondanze*

ID	Concetto	Costrutto	Tipo	Numero	Motivazione *(Attributi)*

6. Traduzione al **modello logico**
   a. Schema relazionale
   b. Elenco dei vincoli di integrità referenziale
   c. Elenco dei vincoli di dominio e di tupla
   d. Utenti e privilegi

7. **Normalizzazione**

8. **Analitycs** (*Analisi sull'efficienza, variano a seconda delle specifiche del progetto*)

9. **Implementazione in Oracle MySQL** (*Popolamento, operazioni …*)

10. **Conclusione** (*Eventuali API, Extra …*)

Note aggiuntive

Un buon diagramma ER dovrebbe:

- Essere sin da subito in BCNF se possibile
- Essere tutto collegato attraverso cammini di join
- Evidenziare con colori diversi le diverse aree tematiche
- Evidenziare bene le ridondanze
- Avere al centro le entità con più relazioni
- Avere solo linee perpendicolari e poche intersezioni
- Avere i genitori sopra ai figli nelle generalizzazioni

Le *"**business rules**"* citate prima sono la descrizione di ciò che un diagramma ER non può esprimere:

- Spiegazione discorsiva dei concetti principali *con eventuale tavola delle entità e relazioni*
  - [Entità, descrizione, attributi, identificatori (se FK mettere nome tabella)]
  - [Relazioni, descrizione, entità coinvolte e cardinalità, attributi]

- Frasi di carattere generale *in modo discorsivo o tramite il seguente elenco*
  - [RVi] Concetto DEVE/NON DEVE espressione

- Derivazioni *in modo discorsivo o tramite il seguente elenco*
  - [RDi] Concetto SI OTTIENE operazione

8. Gestione fisica del DBMS

Gestore del BUFFER *(Generalmente i DBMS adottano la politica Steal / No-Force)*

Ogni memoria è generalmente organizzata in blocchi, ossia l'unità più più piccola sulla quale un sistema può leggere o scrivere, un sistema è sempre dotato di due tipi di memoria:

- **Memoria principale** *(RAM)* è volatile e viene eliminata ad ogni riavvio
- **Memoria secondaria** *(HDD o SSD, anche detta stabile o di massa)* è persistente nel tempo

In una base di dati il componente del DBMS che gestisce la memoria principale prende il nome di **gestore del buffer**, questo componente ritaglia una zona di memoria più o meno ampia chiamata **buffer** da riservare alla base di dati. Il buffer è organizzato in **pagine**, che corrispondono ad un numero intero di blocchi di memoria, ogni pagina ha un contatore, che rappresenta il numero di transazioni attive che la stanno utilizzando, ed un bit che indica se è stata modificata o meno.

Il gestore del buffer risponde alle seguenti richieste *(da parte delle transazioni)*

- **Fix**

  Rappresenta una richiesta di accesso ad una pagina, è eseguita come segue:

  Il gestore controlla se la pagina richiesta è nel buffer
  - Se è nel buffer la restituisce alla transazione che l'ha chiesta
  - Se non è nel buffer allora la deve inserire, dunque cerca nel buffer pagine per farle spazio, preferibilemente pagine vuote o non modificate, ossia con contatore a 0
    - Se ne trova allora deve scegliere quali sovrascrivere
      - Politica **LRU** *Last Recently Used*
      - Politica **FIFO** *First In First Out*
    - Se non ne trova ha due strade
      - Politica **Steal** *Viene rubata una pagina ad una transazione che la sta utilizzando e viene salvata in memoria secondaria*
      - Politica **No Steal** *La transazione attuale viene messa in coda*

    Adesso che le ha scelte deve controllare se sono state modificate
    - Se si le salva il memoria secondaria *(operazione force o flush)*
    - Se no sono giò pronte per essere sovrascritte
  - In ogni caso una volta aperte le pagine richieste ne incrementa il contatore

- **Unfix** La transazione ha finito di utilizzare la pagina e ne decrementa il contatore
- **Set Dirty** Indica che la pagina è stata modificata
- **Force** Trasferisce in modo sincrono *(immediatamente)* una pagina in memoria stabile
- **Flush** La stessa operazione sopra ma in modo asincrono *(quando ritiene opportuno)*

Nota: La politica di Flush è anche detta No-Force

Gestore della memoria secondaria

La memoria principale è notevolemente più veloce della secondaria e per questo ogni file a cui voglio accedere deve prima essere copiato su di essa, è dunque necessario un componente del DBMS il quale tramite il file system, un componente del sistema operativo che conosce quali blocchi sono liberi o meno, interagisca con la memoria secondaria.

Il **fattore di blocco**, calcolato dividendo la dimensione di un blocco per la dimensione di un record, indica il numero di record che possiamo scrivere su un unico blocco di memoria, generalmente ogni blocco è dedicato a record di un'unica relazione, oppure di più relazioni ma correlate (*i join sono favoriti*), l'eventuale spazio residuo in ogni blocco può essere lasciato vuoto (*unspanned*) oppure può essere occupato da 'mezzi' record (*spanned*).

Solitamente, per favorire operazioni sull'intera relazione, le tuple di una relazione risiedono in blocchi contigui, il cui numero si può calcolare moltiplicando il numero di record per la lunghezza dei record e dividendo il tutto per la dimensione di un blocco.

Per memorizzare le relazioni il DBMS costruisce:

- **Strutture primarie** → Contengono effettivamente i dati
    - **Organizzazione sequenziale** → Record in blocchi contigui
        - **Seriale**
          I record non sono ordinati, gli inserimenti nuovi avvengono in coda, con riorganizzazioni periodiche, oppure al posto di record eliminati.
        - **Ordinata**
          I record sono ordinati, ad esempio rispetto al valore della chiave, ma mantenere l'ordinamento è relativamente costoso.
        - **Accesso calcolato**
          Una funzione hash calcola ogni volta la posizione del blocco da accedere in base al valore della chiave, l'accesso su un intervallo di valori è relativamente costoso.
    - **Strutture ad albero** → Utilizzano puntatori a blocchi non contigui
      Sono utilizzate sia come strutture primarie sia, soprattutto, come strutture secondarie, in particolare per gestire gli indici vengono utilizzati alberi di ricerca mantenuti bilanciati dove le foglie sono collegate in una lista (*B-tree+*).

- **Strutture secondarie** → Favoriscono l'accesso ai dati

  - **Indici**

Un indice di un file è un file secondario dove vengono memorizzate coppie formate da una chiave e l'indirizzo dei blocchi corrispondenti, ordinando per il valore della chiave, come ogni file ordinato la sua modifica è relativamente costosa. Generalmente gli indici primari vengono sempre definiti dal DBMS mentre i secondari posso essere implementati a seconda di particolari esigenze.

- **Primario**

L'ordine della chiave nell'indice è uguale all'ordine della chiave nella relazione, ogni file può averne solamente uno.

- **Secondario**

L'ordine della chiave nell'indice è diverso dall'ordine della chiave nella relazione, ogni file può averne anche diversi.

- **Denso**

Contiene un record per ogni record della relazione, proprietà sicuramente assunta dagli indici secondari, altrimenti non saprei come raggiungere tutti i valori.

$$N_{blocchi\ indice} = {N_{record} * (Dim_{chiave} + Dim_{puntatore})}\Big/{Dim_{blocchi}}$$

- **Sparso**

Contiene un record solo per alcuni dei record della relazione, questa è una proprietà dei solo indici primari poiché anche se l'elemento manca riesco a capire se guardare sopra o sotto grazie all'ordinamento.

$$N_{blocchi\ indice} = {N_{blocchi\ relazione} * (Dim_{chiave} + Dim_{puntatore})}\Big/{Dim_{blocchi}}$$

- **Multilivello**

Essendo gli indici dei file, per una migliore organizzazione dei dati è possibile costruire indici di indici

$$N_{blocchi\ livello\ i} = {N_{blocchi\ livello\ i-1} * (Dim_{chiave} + Dim_{puntatore})}\Big/{Dim_{blocchi}}$$

Gestore dell'ottimizzazione delle interrogazioni

Nel modello relazionale le interrogazioni vengono tradotte dall'SQL in operazioni dell'algebra relazionale ed ottimizzate dal *query processor*, un componente interno al DBMS che cerca espressioni equivalenti a quelle date ma meno costose, alcune equivalenze importanti sono:

- **Pushing selections down** *Esegue le selezioni il prima possibile (e le più selettive prima)*
- **Pushing projections down** *Esegue le proiezioni il prima possibile*

E' importante sapere che le ottimizzazioni sulle query avvengono non solo dal punto di vista algebrico ma anche dal punto di vista dei costi per la particolare istanza su cui si sta operando, ogni relazione infatti ha un **profilo** che contiene informazioni quantitative su dimensioni e valori delle tuple, in modo da poter stimare il numero di trasferimenti in memoria da effettuare e la dimensione dei risultati intermedi.

Le operazioni base che un DBMS compie sono:

- Accesso diretto
- Scansione
- Ordinamento (*tramite quicksort se la relazione è piccola altrimemti merge sort*)
- Join, implementato tramite una delle seguenti alternative:

    - **Nested loop**

      Per ogni record della relazione esterna si esaminano tutti i record di quella interna e così via, la relazione interna viene interamente ricopiata nel buffer per garantire un accesso rapido ai record, mentre per quella esterna verrà trasferito nel buffer un record alla volta, a seconda di quale sta esaminando. Il costo è proporzionale al numero di trasferimenti in memoria e la scelta di quale relazione è esterna e quale è interna dipende da quale delle due riesce ad entrare interamente nel buffer.

    - **Merge scan**

      Le relazioni vengono ordinate in base agli attributi di join, il costo risiede tutto nell'ordinamento, questo metodo funziona bene per join naturale.

    - **Hash based**

      Utilizza una funzione hash per confrontare il valore degli attributi di join, è utile solamente per il join naturale ed è relativamente veloce, ma occupa memoria per salvare i risultati della funzion hash. Il costo di questo algoritmo è la lettura e scrittura completa di entrambe le relazioni.

Database

C → Customer (Idcust, Name, Age, City cust)

I → Insurance (Idcom, Iddirector, NumberEM, city)

P → Policy (Idpol, Idcust, Idcom, Date)

Tavola dei volumi

$nC = 2000$	$nI = 20$	$nP = 100000$
	$nCity = 5$	$nIdcust = 2000$
		$nIdcom = 20$
		$nDate = 20$

Query da ottimizzare

$$\pi_{Name}(\sigma_{City='m' \wedge Date='m'}(C \bowtie P \bowtie I))$$

Ottimizzazione algebrica

$$\pi_{Name}\left(\pi_{Name,Idcust}(C) \overset{①}{\bowtie} \pi_{Idcust,Idcom}\left(\sigma_{Date='m'}(P)\right) \overset{②}{\bowtie} \pi_{Idcom}(\sigma_{city='m'}(I)) \overset{③}{}\right)$$

Ottimizzazione basata sui costi

① La proiezione contiene la chiave dunque gli elementi restano gli stessi (2000)

② La selezione lascia in media $\frac{Record}{Date} = \frac{100000}{20} = 5000$ elementi
La proiezione non contiene la chiave, dunque è uguale al minimo tra i record che ho e i record diversi che posso proiettare con tali attributi → nota che non possono essere più di 5000 singolarmente!

$$\min(5000, 20 \cdot 2000) = \min(5000, 40000) = 5000$$

③ La selezione lascia in media $\frac{Record}{citta} = \frac{20}{5} = 4$ elementi
La proiezione contiene la chiave dunque restano 4 elem.

Con questi risultati posso ora calcolare l'ordine di join

① ⋈ ② Il join viene fatto sulla chiave di ① e ② ha un vincolo di integrità referenziale dunque si producono esattamente ② record ossia 5000

① ⋈ ③ Il join non contiene attributi comuni dunque equivale a fare il prodotto cartesiano ossia $2000 \cdot 4 = 8000$ elementi

② ⋈ ③ Il join viene fatto sulla chiave di ③ ma non vale più il vincolo di integrità referenziale tra ② e ③ perché in entrambi ho effettuato selezioni.

Il join produce $\min$ (elem ② · elem uguali ③, elem ③ · elem uguali ②)

elem uguali sull' attributo di join

Dunque $\min\left(5000 \cdot \frac{4}{4}, 4 \cdot \frac{5000}{20}\right) = (5000, 1000) = 1000$

L'ordine di esecuzione dei join è dunque $((② \bowtie ③) \bowtie ①)$

Query finale

$$\pi_{Nome}\left(\pi_{IdCust, IdCom}\left(\sigma_{Data = '\_'}(P)\right) \bowtie \pi_{IdCom}\left(\sigma_{City = '\_'}(I)\right) \bowtie \pi_{Nome, IdCust}(C)\right)$$

Gestore dell'affidabilità *(Garantisce atomicità e durabilità)*

E' un componente del DBMS il quale assicura che le transazioni non vengano lasciate a metà e che i loro effetti siano permanenti, per ottenere questo utilizza il LOG, un file scritto in memoria secondaria che raccoglie in ordine cronologico tutte le azioni compiute dal DBMS e permette di ricostruire il contenuto della base di dati a seguito di guasti.

Dati i possibili stati di una base di dati:

- Corretto *Ultima versione dei dati, completa, aggiornata e consistente*
- Valido *Parte dell'informazione di uno stato corretto è andata persa*
- Consistente *La base di dati è valida e i vincoli sono mantenuti*

Il gestore dell'affidabilità garantisce di poter tornare sempre ad uno stato corretto

All'interno del log compaiono principalmente

- Record di transazione
  - **B**egin (T) *T è la transazione che ha eseguito l'operazione*
  - **I**nsert (T, O, AS) *O è l'oggetto (in memoria) che ha subito l'operazione*
  - **U**pdate (T, O, BS, AS) *BS è una copia dell'oggetto prima dell'operazione*
  - **D**elete (T, O, BS) *AS è una copia dell'oggetto dopo l'operazione*
  - **C**ommit (T)
  - **A**bort (T)

- Record di sistema
  - **Dump** Sono una copia completa della base di dati
  - **Checkpoint** Sono dei punti di ripristino intermedi, per costruirne uno devo:
    - Congelare le transazioni *(Smetto di accettare commit o abort)*
    - Trasferire forzatamente in memoria di massa le pagine modificate da transazioni che sono andate in commit
    - Scrivere sul log che ho effettuato il checkpoint indicando quali sono le transazioni attive *(non andate in commit o abort)*
    - Scongelare le transazioni

Ma una transazione viene prima scritta nel log e poi eseguita oppure il contrario ?

Il gestore dell'affidabilità mantiene la consistenza del log seguendo due regole:

- **Write Ahead Log** *Consente di disfare le transazioni che possono aver già scritto dei dati*
  Le parti riguardanti i BeforeState vengono prima scritte nel log, e poi effettivamente eseguite
- **Commit Precedence** *Consente di rifare le transazioni andate in commit ma perse nel buffer*
  Le parti riguardanti gli AfterState vengono scritte nel log prima di effettuare il commit

Nota: negli esercizi si utilizza una notazione semplificata la quale prevede che nel log venga scritto per intero il record di transazione prima della sua esecuzione, senza distinguere tra BS e AS

A seguito di un guasto farà fede ciò che si trova nel file di log, dunque ogni transazione per cui al momento del guasto non compare il record di commit *(rimane a metà oppure abortisce)* andrà disfatta (*ad esempio se faccio un insert l'undo corrispondente è un delete)*, mentre ogni transazione per cui compare il record di commit andrà ripetuta *(redo)* in quanto non possiamo sapere se questa sia stata già salvata sulla memoria secondaria o se le modifiche siano andate perse insieme al buffer e alla memoria primaria, dato che, come detto prima, il DBMS utilizza la politica No-Force e duqnue le modifiche apportate sul buffer vengono riportate sulla memoria secondaria quando il gestore lo ritiene più opportuno.

Abbiamo visto come funziona il buffer ed il file di log, ma quando vengono effettivamente riportate le operazioni scritte sul log sulla memoria secondaria ossia sulla base di dati ?

Ci sono tre modalità diverse di gestire la scrittura delle nuove operazioni sulla base di dati:

IMMEDIATA

Tutte le modifiche apportate sul buffer vengono applicate immediatamente alla base di dati; in caso di guasto non sono richieste operazioni di redo poiché le operazioni sono immediatamente salvate in memoria stabile e dunque non vengono perse se viene perso il buffer, bisogna tuttavia disfare le transazioni che al momento del guasto erano attive, perché queste essendo rimaste a metà potrebbero aver sporcato la base di dati

DIFFERITA

Tutte le modifiche apportate sul buffer vengono applicate alla base di dati non appena le transazioni vanno in commit; in caso di guasto non occorre disfare nulla, poiché tutte le modifiche delle transazioni attive vengono perse con il buffer, bisogna tuttavia rifare le transazioni che al momento del guasto erano andate in commit, perché queste potrebbero non essere state scritte in tempo

MISTA *(Comportamento assunto generalmente dai DBMS)*

Vengono mescolate le due procedure viste sopra, il risultato è che il gestore dell'affidabilità riporta le modifiche sulla base di dati quando ritiene più opportuno

Ma come si recupera effettivamente una base di dati in seguito ad un guasto ?

Il DBMS adotta strategie di recovery diverse a seconda del tipo di guasto:

Guasti **software → Ripresa a caldo**

Guasti **hardware → Ripresa a freddo**

Indipendentemente dal tipo di guasto, viene adottato il cosidetto **modello fail-stop**, secondo il quale nell'istante del guasto le transazioni vengono interrotte, il sistema viene riavviato, viene letto l'ultimo checkpoint, viene avviata la procedura di recovery adeguata ed infine vengono fatte ripartire le transazioni con il buffer pulito

RIPRESA A CALDO

Viene letto l'ultimo checkpoint e in base a quello vengono creati due file: l'**undo list** contentente le transazioni da disfare, ossia tutte quelle attive tra il checkpoint ed il guasto più quelle abortire, e la **redo list** contenente le tranaszioni da rifare, ossia tutte quelle andate in commit tra il checkpoint ed il guasto; una volta create le liste viene prima eseguita la undo list *(prima le transazioni più recenti poi le più vecchie)* e successivamente la redo list *(prima le transazioni più vecchie poi le più recenti)*

RIPRESA A FREDDO

Si accede al record di dump più recente e si ripristina selettivamente la parte danneggiata della base di dati, successivamente vengono eseguite tutte le transazioni che, dal record di dump fino al momento del guasto, hanno interagito con la parte danneggiata della base di dati, infine si esegue una ripresa a caldo.

Esempio di file LOG

DUMP B(T1) B(T2) I(T1,O1,A1) D(T2,O2,B2) B(T3) B(T4) U(T3,O3,B3,A3) C(T2) CK(T1,T3,T4) U(T1,O4,B4,A4) A(T3) B(T5) D(T4,O5,B5) C(T1) C(T4) I(T5,O6,A6) GUASTO

Ripresa a caldo

Ultimo checkpoint	Istante del guasto
UNDO → T1, T3, T4	UNDO → T3, T5
REDO → *Vuota*	REDO → T1, T4

Operazioni da disfare *(ordine cronologico inverso)*
I(T5,06,A6) → Delete O6
U(T3,03,B3,A3) → Update O3 = B3

Operazioni da rifare *(ordine cronologico)*
I(T1,O1,A1) → Insert O1 := A1
U(T1,O4,B4,A4) → Update O4 = A4
D(T4,O5,B5) → Delete O5

Ripresa a freddo

Supponiamo un guasto su O1, O2, O3

DUMP → *Ripristino O1, O2, O3*

I(T1,O1,A1) → Insert O1 := A1
D(T2,O2,B2) → Delete O2
U(T3,O3,B3,A3) → Update O3 := A3

Checkpoint → *Inizio una ripresa a caldo*

Gestore della concorrenza *(Garantisce l'isolamento)*

Generalmente una base di dati è accessibile a più utenti e di conseguenza possono arrivare nello stesso istante un numero notevole di richieste, è dunque necessario un componente *(detto **scheduler**)* che sappia prevedere lo stato finale della base di dati in seguito alle richieste, potendo così di stabilire quali accettare e quali no, e in che ordine eseguile per mantenere uno stato consistente dei dati.

Uno **schedule** è una sequenza di operazioni e si dice **seriale** se tutte le operazioni di ogni transazione della sequenza avvengono di seguito, senza essere interfogliate *(mischiate)*, per le proprietà ACID delle transazioni un qualisasi schedule seriale porta sempre ad uno stato consistente dei dati.

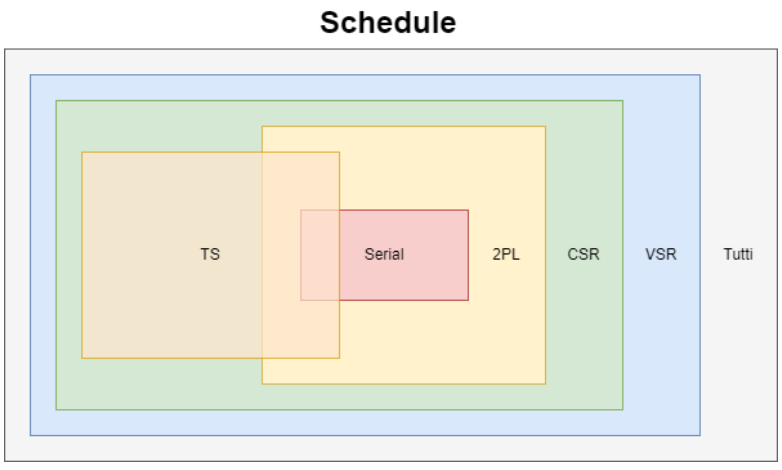
Ci sono cinque principali anomalie che devono essere evitate dal gestore della concorrenza:

- **Perdita di aggiornamento** *Due transazioni aggiornano contemporaneamente lo stesso oggetto (uno dei due aggiornamenti si perde)*
- **Lettura sporca** *Lettura di un valore modificato da una transazione abortita*
- **Lettura inconsistente** *Più letture sequenziali di un oggetto producono valori diversi*
- **Aggiornamento fantasma** *Lettura di un valore prima del suo aggiornamento in memoria*
- **Inserimento fantasma** *La stessa operazione su un insieme di valori dà risultati diversi poiché ne viene inserito uno nuovo (ad esempio media dei voti)*

L'obbiettivo dello scheduler è evitare tutte le anomalie costruendo uno schedule **serializzabile**, ossia equivalente ad un qualsiasi schedule seriale costrutio a partire dalle stesse transazioni.

Ma tutti gli schedule sono serializzabili ?

No, come vedremo ci sono diverse categorie di schedule serializzabili, ognuna delle quali garantisce l'assenza di certe anomalie *(un po' come per le forme normali)*, riassunte in questo schema:



SCHEDULE VIEW-SERIALIZZABILI (VSR)

Bisogna innanzitutto definire due proprietà:

- Si dice che un'operazione di lettura **legge da** una scrittura quando la scrittura precede la lettura e non vi è alcun altra scrittura in mezzo *(la lettura legge l'ultimo valore effettivo)*
- Un'operazione di scrittura su un oggetto è detta **scrittura finale** se è l'ultima scrittura su tale oggetto che compare nello schedule

Uno schedule è **view-equivalente** ad un altro se per ogni oggetto valgono le stesse proprietà elencate sopra *(ogni operazione di lettura legge dalla stessa scrittura e gli stati finali degli oggetti combaciano)*

Uno schedule è **view-serializzabile** se è view-equivalente ad un qualsiasi schedule seriale

La **verifica della view-equivalenza** di due schedule è lineare perché basta scorrere gli schedule e controllare che le proprietà siano rispettate, invece controllare se uno schedule è VSR è un problema enorme poiché bisognerebbe vedere se è view-equivalente ad ogni possibile schedule seriale *(fattoriale)* e dunque questa tecnica **non è utilizzabile in pratica**.

Esempio

S_1: $W_0(X)\ R_2(X)\ W_2(X)\ W_2(Z)\ R_1(X)$ è seriale perché vengono eseguite in ordine $T_0\ T_2\ T_1$

S_2: $W_0(X)\ R_2(X)\ W_2(X)\ R_1(X)\ W_2(Z)$ è serializzabile perché view-equivalente ad S_1

SCHEDULE CONFLICT-SERIALIZZABILI (CSR)

Bisogna innanzitutto definire il termine conflitto: un'operazione è in **conflitto** con un'altra se operano sullo stesso oggetto e almeno una di esse è una scrittura.

Due schedule sono **conflict-equivalenti** se compiono le stesse operazioni e ogni coppia di operazioni in conflitto compare nello stesso ordine in entrambi *(ossia se i due grafi dei conflitti sono uguali)*

Uno schedule è **conflict-serializzabile** se è conflict-equivalente ad un qualsiasi schedule seriale

La **verifica della conflict-serializzabilità** avviene per mezzo del **grafo dei conflitti**, un grafo in cui ogni transazione ha un nodo e ogni conflitto è rappresentato per mezzo di un arco *(orientato)*, **uno schedule è CSR se e solo se il grafo è aciclico**, ossia se non riesco a trovare un percorso tramite il quale partendo da un qualsiasi nodo del grafo ritorno ad esso

Esempio

Provare che il seguente schedule è CSR e trovarne uno seriale equivalente

$$S: R_1(Y)\ W_3(Z)\ R_1(Z)\ R_2(Z)\ W_3(X)\ W_1(X)\ W_2(X)\ R_3(Y)$$

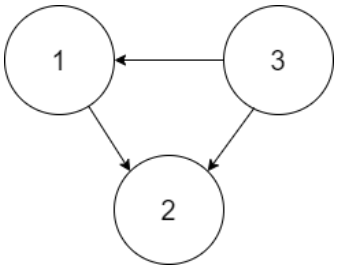
A. Separare le operazioni in base all'oggetto su cui operano per trovare i conflitti

$$X \rightarrow\ W_3\ W_1\ W_2$$
$$Y \rightarrow\ R_1\ R_3$$
$$Z \rightarrow\ W_3\ R_1\ R_2$$

B. Individuare i conflitti e rimuovere i duplicati

$$Per\ X\ i\ conflitti\ sono: 3 \rightarrow 1, 3 \rightarrow 2, 1 \rightarrow 2$$
$$Per\ Y\ non\ ci\ sono\ conflitti\ (solo\ letture)$$
$$Per\ Z\ i\ conflitti\ sono: 3 \rightarrow 1, 3 \rightarrow 2\ (duplicati\ in\ X)$$

C. Disegnare il grafo orientato dei conflitti



D. Verificare se il grafo è aciclico *(in questo caso lo è)*

E. Se richiesto trovare uno schedule seriale equivalente
   $T_3\ T_1\ T_2$ è uno schedule seriale equivalente perché rispetta l'ordine imposto dal grafo

La **verifica della confict-serializzabilità** è lineare, tuttavia necessita della costruzione, aggiornamento e verifica del grafo dei conflitti, che in un sistema reale richiederebbe troppe risorse, inoltre non è fisicamente possibile adoperare questa tecnica nel caso di basi di dati distributite, dunque anche questa tecnica è da scartare.

SCHEDULE BASATI SU LOCK

Nei sistemi reali si utilizzano tecniche di scheduling che garantiscono a priori la serializzabilità, introduciamo dunque il concetto di lock, sulla quale si basa la prossima tecnica che andremo a considerare: secondo questo modello tutte le letture e le scritture sono precedute da richieste di **lock** e **unlock**, che acquistano o rilasciano risorse, gestite dal **lock manager**.

Ogni operazione di lettura viene preceduta da un r_lock *(read lock)* ossia un **lock condiviso**, infatti più operazioni possono tranquillamente leggere lo stesso oggetto allo stesso momento.

Ogni operazione di scrittura viene preceduta da un w_lock *(write lock)* ossia un **lock esclusivo**, infatti una e una sola operazione può modificare lo stesso oggetto allo stesso momento.

Se una transazione fa sia operazioni di lettura che di scrittura sullo stesso oggetto può richiedere da subito un lock esclusivo oppure chiedere prima un lock condiviso e poi passare ad uno esclusivo nel momento in cui inizia a scrivere ***(lock upgrade)***.

TWO PAHSE LOCKING (2PL)

È una tecnica che implementa il modello basato sui lock tramite una semplice regola: una volta rilasciato un lock una transazione non può più acquisirne altri, le transazioni avranno dunque una fase iniziale dove richiederanno tutte le risorse e una fase finale dove rilasceranno tutte le risorse per poi terminare, eventuali *lock upgrade* potranno svolgersi solo nella fase di acquisizione ed eventuali *lock downgrade* solo nella fase di rilascio.

Questa tecnica presenta tuttavia due anomalie: **lettura sporca e deadlock**. La prima è già stata spiegata all'inizio di questo paragrafo, la seconda si verifica quando due transazioni vogliono ognuna risorse bloccate dall'altra e dunque si aspettano a vicenda. Per risolvere l'anomalia di lettura sporca è stata introdotta una tecnica chiamata STRICT TWO PHASE LOCKING in cui i lock possono essere rilasciati solo dopo il commit. Per gestire i casi di deadlock ci sono invece tre possibili alternative:

Timeout È il più semplice e usato di tutti, le transazioni rimangono in attesa di una risorsa per un tempo limite prefissato dopo il quale la transazione viene abortita

Rilevamento Viene scansionata a intervalli regolari la tabella dei lock alla ricerca di deadlock

Prevenzione Vengono uccise le transazioni sospette tramite:

- **Politche interrompenti**: viene uccisa una delle due transazioni in stallo, tipicamente quella che ha svolto meno lavoro, questo tuttavia può portare alla **starvation**, ossia quando una transazione come prima operazione deve accedere ad una risorsa molto richiesta, dunque andrà sempre in conflitto e verrà sempre uccisa perché avrà sempre svolto meno lavoro delle altre *(una possibile soluzione a questo problema è dare priorità alle transazioni più anziane)*

- **Politiche non interrompenti**: una transazione continua ad attendere finchè non effettua una nuova richiesta

TIMESTAMP LOCKING (TS)

È una tecnica di scheduling alternativa al 2PL ma meno usata, basata sul concetto di timestamp: ogni transazione ha un timestamp unico che rappresenta l'istante d'inizio, in base al quale viene ipotizzato uno schedule seriale, che verrà provato ad essere costruito.

Per ogni oggetto lo scheduler memorizza due contatori: **RTM** e **WTM**, che **rappresentano i timestamp dell'ultima lettura o scrittura su un oggetto**, servono per assicurare che nessuna transazione legga dove ha già scritto una transazione più recente e che nessuna transazione scriva dove hanno già letto o scritto transazioni più recenti, in caso contrario la transazione viene uccisa, ossia fatta ripartire successivamente, dunque con un timestamp più alto e con maggiori possibilità di passare i controlli.

Esempio

Applica 2PL Strict e TS allo schedule S: $R_1(Y)\ W_3(Z)\ R_1(Z)\ R_2(Z)\ W_3(X)\ W_1(X)\ W_2(X)\ R_3(Y)$

2PL Strict

$R_1(Y) \rightarrow R\_lock(Y)$
$W_3(Z) \rightarrow W\_lock(Z)$
$R_1(Z) \rightarrow T_1\ wait\ T_3 \rightarrow Queue\ (T_1)$
$R_2(Z) \rightarrow T_2\ wait\ T_3 \rightarrow Queue\ (T_2)$
$W_3(X) \rightarrow W\_lock(X)$
$W_1(X)\ ignorata\ perchè\ T_1\ in\ attesa$
$W_2(X)\ ignorata\ perchè\ T_2\ in\ attesa$
$R_3(Y) \rightarrow R\_lock(Y)$
$Commit\ (T_3) \rightarrow unlock(Z, X, Y)$
$Dequeue\ (T_1, T_2)$
$R_1(Z) \rightarrow R\_lock(Z)$
$R_2(Z) \rightarrow R\_lock(Z)$
$W_1(X) \rightarrow W\_lock(X)$
$Commit\ (T_1) \rightarrow unlock(Y, Z, X)$
$W_2(X) \rightarrow W\_lock(X)$
$Commit\ (T_2) \rightarrow unlock(Z, X)$

TS

$R_1(Y) \rightarrow RTM(Y) = 1$
$W_3(Z) \rightarrow WTM(Z) = 3$
$R_1(Z) \rightarrow Abort \rightarrow T_1 \Rightarrow T_4$
$R_4(Y) \rightarrow RTM(Y) = 4$
$R_4(Z) \rightarrow RTM(Z) = 4$
$R_2(Z) \rightarrow Abort \rightarrow T_2 \Rightarrow T_5$
$R_5(Z) \rightarrow RTM(Z) = 5$
$W_3(X) \rightarrow WTM(X) = 3$
$W_4(X) \rightarrow WTM(X) = 4$
$W_5(X) \rightarrow WTM(X) = 5$
$R_3(Y) \rightarrow RTM(Y) = 4$

Comunicazione tra il DBMS e altri linguaggi

In applicazioni reali il programmatore non vuole eseguire solo comandi SQL, ma interi programmi che si appoggiano ad SQL per comunicare con una base di dati, per ottenere ciò ci sono tre strade:

- **Embedded** SQL

  Le istruzioni SQL sono immerse nel codice di un altro programma e vengono tradotte tramite funzioni di libreria nel linguaggio che le ospita.

  - **Static** Se le istruzioni SQL sono note a tempo di compilazione
  - **Dynamic** Se le istruzioni SQL non sono note a tempo di compilazione

  Tuttavia questo approccio presenta un importante difficoltà (***impedance mismatch***): i linguaggi di programmazione standard non supportano il tipo di dato *relazione*, per questo viene dichiarato preventivamente un **cursore** *(un costrutto SQL)* che legge record per record la *relazione* e la trasferisce nella struttura dati scelta

- Call Level Interface (**CLI**)

  È un'interfaccia tra un programma e il DBMS che permette al programma, per mezzo di parametri trasmessi a funzioni di libreria, di comunicare con una base di dati *(non vengono dunque scritte istruzioni in SQL)*