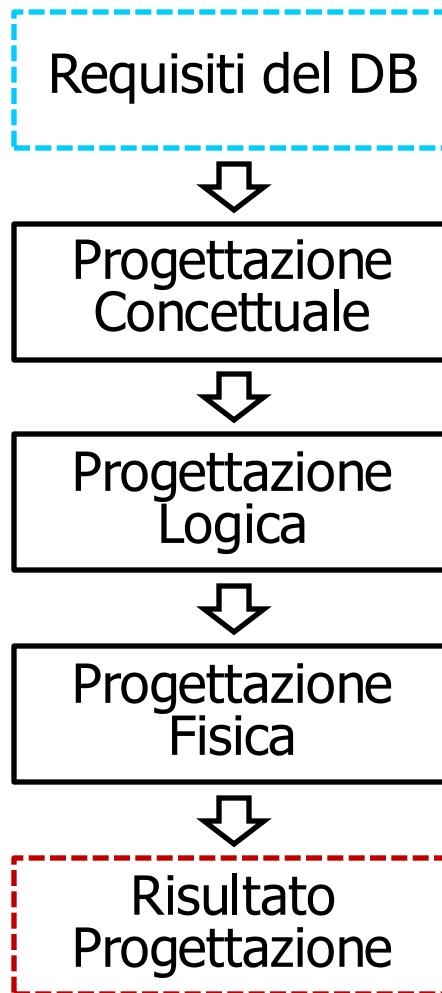




# Progettazione Concettuale: Il modello Entità-Relazioni



- **Sistema Informativo:** sistema in grado di gestire informazioni di interesse in modo strutturato
- **Basi di Dati (Database):** sistema in grado di gestire dati in modo integrato e flessibile, limitando i rischi di ridondanza e incoerenza
- **DBMS ( DataBase Management System):** sistema software in grado di gestire collezioni di dati (DB) in modo affidabile e persistente
- **Modello dei dati:** insieme di elementi utilizzati per organizzare e strutturare dati di interesse
  - modello relazionale, gerarchico, reticolare, NoSQL



- **Schema Concettuale**
  - Rappresenta le specifiche informali della realtà di interesse
  - Nessun riferimento sulle modalità di codifica/memorizzazione
- **Schema Logico**
  - Traduzione dello schema concettuale in termini di modello di rappresentazione dei dati
  - Si utilizzano tecniche di verifica dello schema logico
    - Indipendente dai dettagli fisici
- **Schema Fisico**
  - Specifica i parametri fisici di memorizzazione dei dati
  - Dipende dal DBMS utilizzato

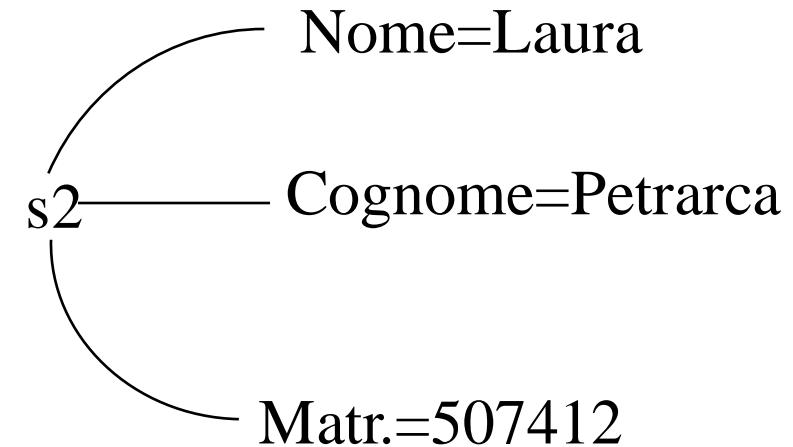
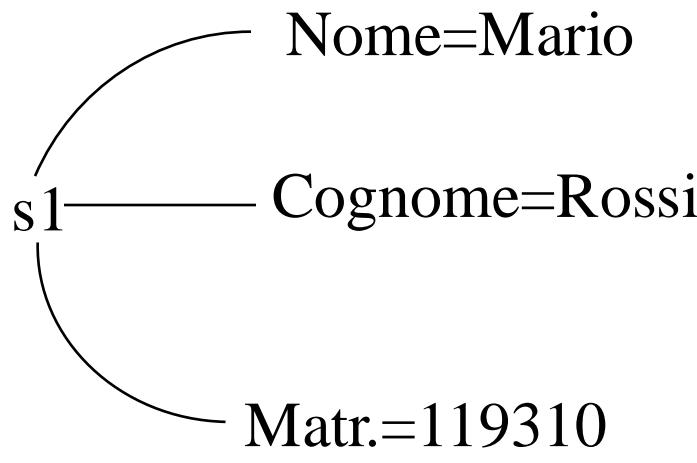
# Lo schema concettuale

- Cosa è: una rappresentazione di alto livello dei requisiti sui dati raccolti nello URD (User Requirements Document)
- Cosa contiene: una descrizione dettagliata dei dati, delle relazioni e dei vincoli
- Cosa non contiene: dettagli implementativi
- Come si definisce: tipicamente mediante il modello Entità-Relazioni (Entity-Relationship)

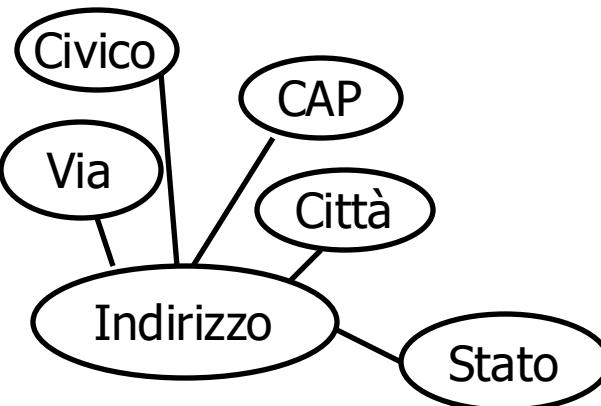
- **Entità:** Una classe di oggetti (astratti o tangibili) della realtà di interesse (mini mondo) distinguibili dagli altri
- **Attributi:** le particolari proprietà che caratterizzano ciascuna entità nel contesto di interesse



- L'occorrenza di una entità (1 singolo esemplare della classe) sarà caratterizzata da valori assunti dagli attributi



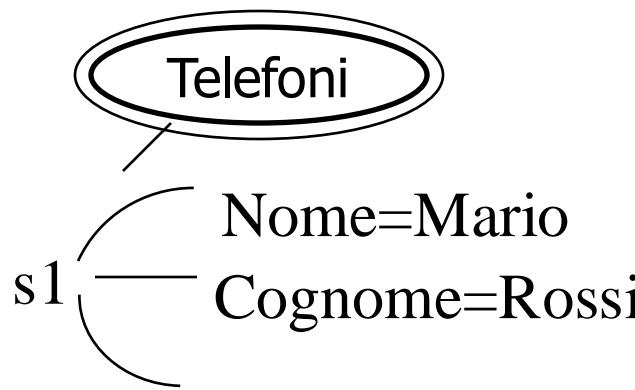
- Tipi particolari di attributo:



- Attributo Composto: è suddivisibile in parti più piccole che hanno ancora una propria specificità:

Indirizzo → Via, Civico, CAP, Città, Stato.

Utile quando preveda di riferirmi a singole parti dell'attributo (via e civico)



- Attributo Multivalore: può assumere più di un singolo valore per ciascuna occorrenza di una entità.

Es: Telefoni, titoli\_di\_studio, sedi...

Telefoni=080555333, 03481122345, 0335663452

- **Attributo derivato:** attributo che è possibile o conveniente determinare a partire da altri attributi immagazzinati.

Es.: Età è derivabile da data di nascita (è anche conveniente?)

Età

- **Dominio di un attributo:** L'insieme dei valori che possono essere assegnati ad un attributo di una entità.

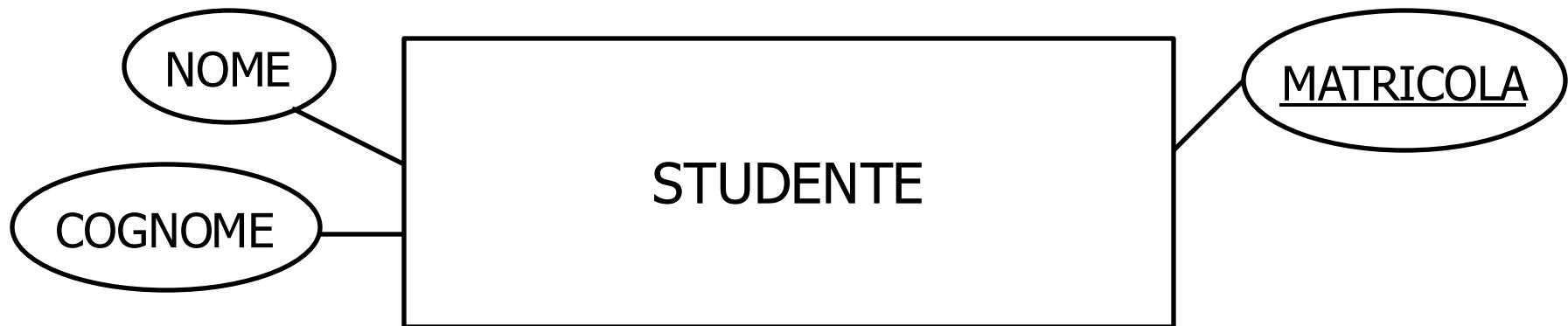
Es. età → 16-65, nome → insieme delle stringhe di caratteri

*Def.: un attributo A di una entità E il cui insieme di valori è V è definibile come  $A : E \rightarrow P(V)$ . Dove  $P(V)$  è l'insieme dei sottoinsiemi di V.*

- **Sui valori nulli:** quando per un attributo in un esemplare di entità non sia possibile determinare un valore viene creato il valore *NULL*. Il suo significato può essere duplice: (1) Ignoto; (2) Non applicabile.

Es.: (1) Voto\_di\_maturità=null (non è noto il valore, ma la maturità è stata conseguita)  
(2) Civico=null (non esiste una numerazione poiché l'urbanizzazione non è completa)

- **Attributi chiave:** Come distinguere tra occorrenze di una stessa entità?  
Assumiamo l'esistenza di un vincolo di unicità sugli attributi.
- **Vincolo di chiave:** esiste un sottoinsieme di attributi (che può ridursi ad uno) di una entità la cui combinazione di valori è distinta per ciascuna occorrenza di una entità.
- Chiamiamo Chiave questo sottoinsieme





# Note sul vincolo di chiave

- La proprietà di unicità è un vincolo sullo schema dell'entità, non su un particolare insieme di occorrenze di entità. Essa pertanto vale sempre
- La chiave va determinata quindi sulla base delle proprietà del mini mondo che la base di dati rappresenta
- Una entità può avere più di un attributo che verifica il vincolo di chiave
  - Es.: Numero di telaio e numero di targa per l'entità “autovettura”

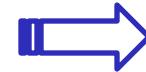
# Esempio di chiave (1/2)

- Utilizzando una rappresentazione tabellare:

Studente

<u>Nome</u>	<u>Cognome</u>
Mario	Rossi
Piero	Bianchi
Fabio	Bianchi
Giovanni	Bianchi

La chiave non deve dipendere dal particolare "stato"



<u>Nome</u>	<u>Cognome</u>	<u>Matricola</u>
Mario	Rossi	119310
Piero	Bianchi	514222
Fabio	Bianchi	125234
Giovanni	Bianchi	432541

## Esempio di chiave (2/2)

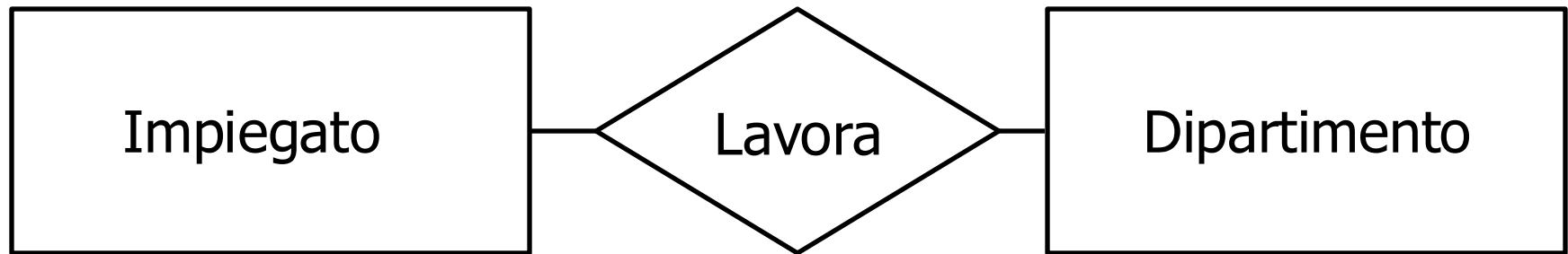
- La proprietà di un attributo di essere chiave dipende dal contesto:

### Esami superati

La Matricola da sola non è in grado di individuare una occorrenza di "esami superati"

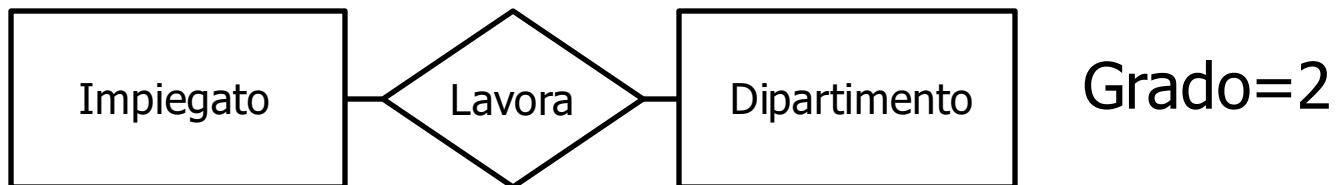
<u>Corso</u>	<u>Data</u>	<u>Votazione</u>	<u>Matricola</u>
AnalisiI	19/09/99	27	119310
AnalisiI	19/09/00	27	514222
AnalisiII	19/06/00	30	119310
FisicaII	14/07/00	28	119310

- Relazione: Associazione o legame logico esistente tra due o più entità

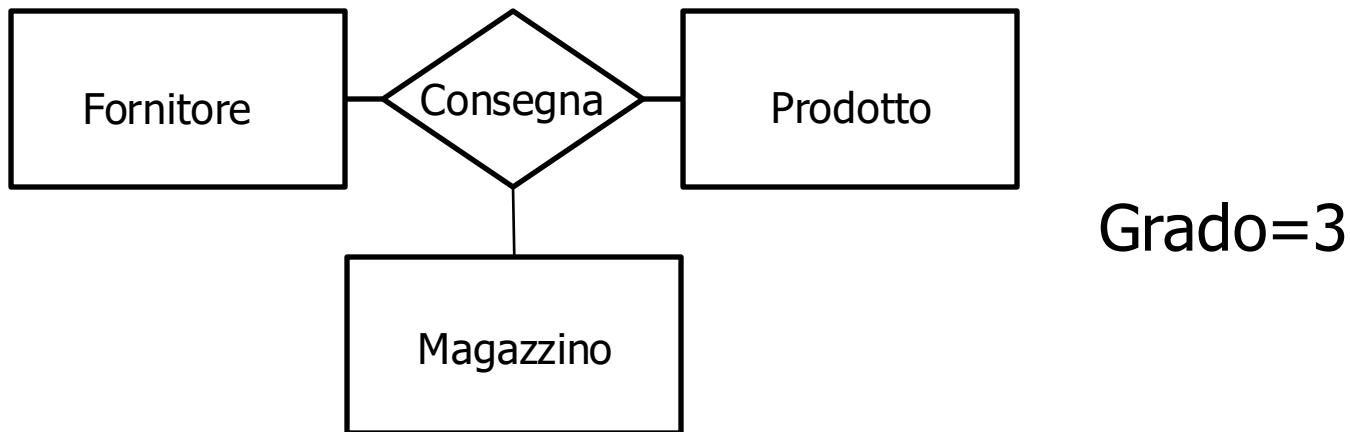


# Grado di una relazione

- Grado di una relazione: numero di entità partecipanti

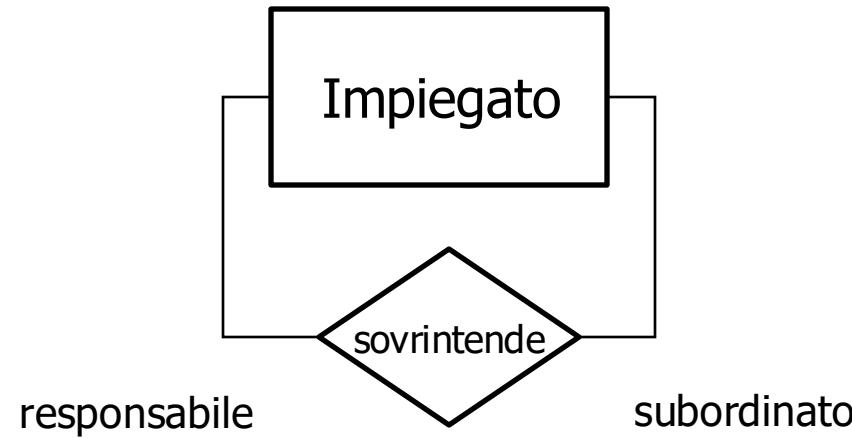


Grado=2



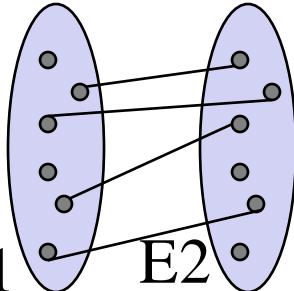
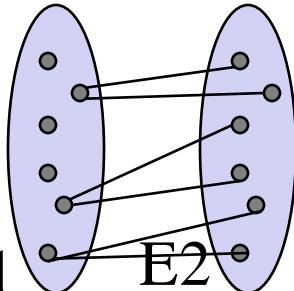
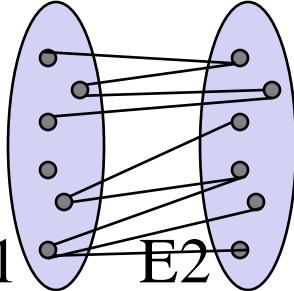
Grado=3

- Una entità può essere in relazione con se stessa. In tal caso si utilizzano esplicitamente nomi di ruolo per chiarire la partecipazione
- L'entità impiegato partecipa alla relazione sovrintende sia nel ruolo di responsabile che in quello di subordinato



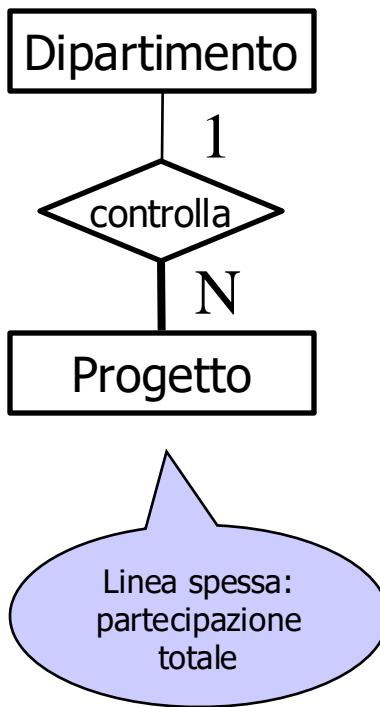
- Le relazioni possono avere vincoli che limitano le combinazioni delle entità partecipanti
  - I vincoli dipendono dal contesto, cioè dal minimondo che la relazione rappresenta
- 
1. **Cardinalità:** specifica il numero di occorrenze di relazione cui le occorrenze di entità possono partecipare
  2. **Partecipazione:** specifica se l'esistenza di una occorrenza di entità dipende dal suo essere in relazione con un'altra occorrenza di entità

Le cardinalità vengono espresse normalmente come:

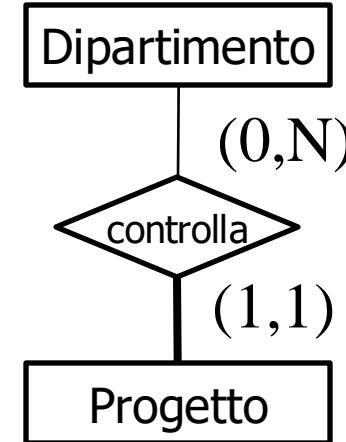
- 1:1 (uno a uno), alla relazione partecipa una singola occorrenza di entità per ciascuna delle 2 entità partecipanti
- 1:N (uno a molti), alla relazione possono partecipare, per una singola occorrenza di entità di una delle entità partecipanti, svariate occorrenze dell'altra entità
- M:N (molti a molti), vale anche il viceversa della precedente definizione

- Si considerano due tipi di partecipazione: totale e parziale.

- Partecipazione **totale** → dipendenza esistenziale  
Ogni occorrenza di entità partecipa alla relazione  
Es.: I **requisiti** dichiarano che un progetto (una occorrenza della entità Progetto) dove essere gestito da un dipartimento, altrimenti non ha senso che esista
- Partecipazione **parziale** → Una occorrenza di entità può partecipare alla relazione.  
Es.: Un impiegato può essere direttore di un dipartimento, ma non necessariamente.



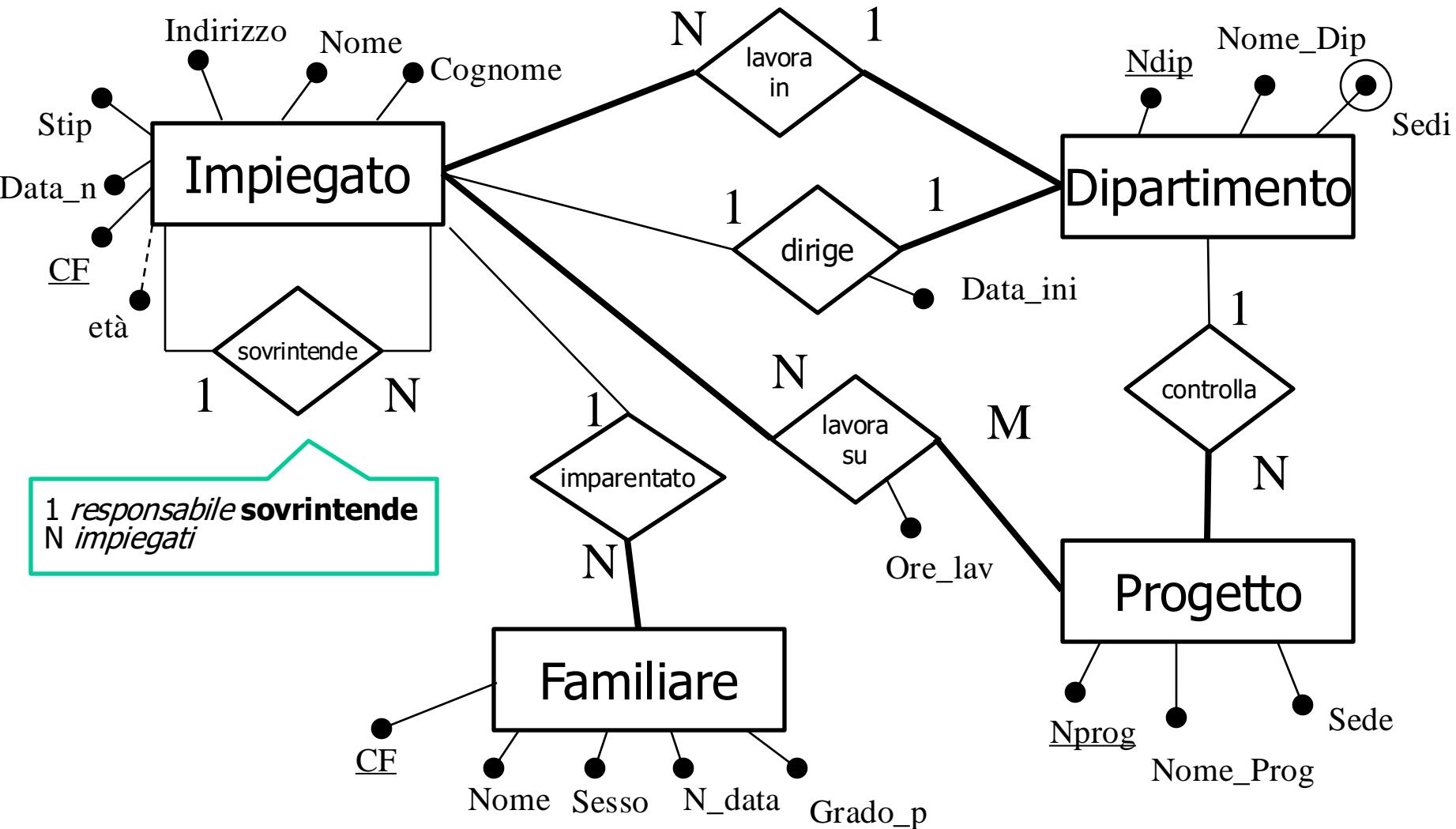
- Esistono notazioni alternative per la rappresentazione dei vincoli
- Si possono rappresentare per ogni entità il numero minimo e massimo di entità partecipanti.
- Se  $\text{min}=0 \rightarrow$  partecipazione **parziale**
- Se  $\text{min}>0 \rightarrow$  partecipazione **totale**



# Un esempio di progetto

- Vogliamo progettare il modello E-R per un database di una **azienda**
  - Supponiamo di aver raccolto ed analizzato i **requisiti** in una fase precedente. I principali requisiti sono qui elencati:
1. L'azienda è organizzata in **dipartimenti**. Ogni dipartimento ha un identificativo e un nome univoco; un **impiegato** gestisce il dipartimento. Il dipartimento può avere più **sedi** dislocate sul territorio.
  2. Un dipartimento gestisce un numero variabile di **attività** identificabili univocamente; ciascuna attività ha inoltre un nome e si svolge in un unico luogo.
  3. Per ciascun impiegato si desidera tenere traccia di varie informazioni anagrafiche. Si desidera tenere traccia dei **rapporti gerarchici** del personale. Da un punto di vista organizzativo ciascun impiegato è assegnato ad un dipartimento e può lavorare su vari **progetti**, non necessariamente gestiti dal suo dipartimento.
  4. E' necessario inoltre tenere traccia dei **familiari** di ciascun dipendente per motivi fiscali

# Schema E-R del database aziendale

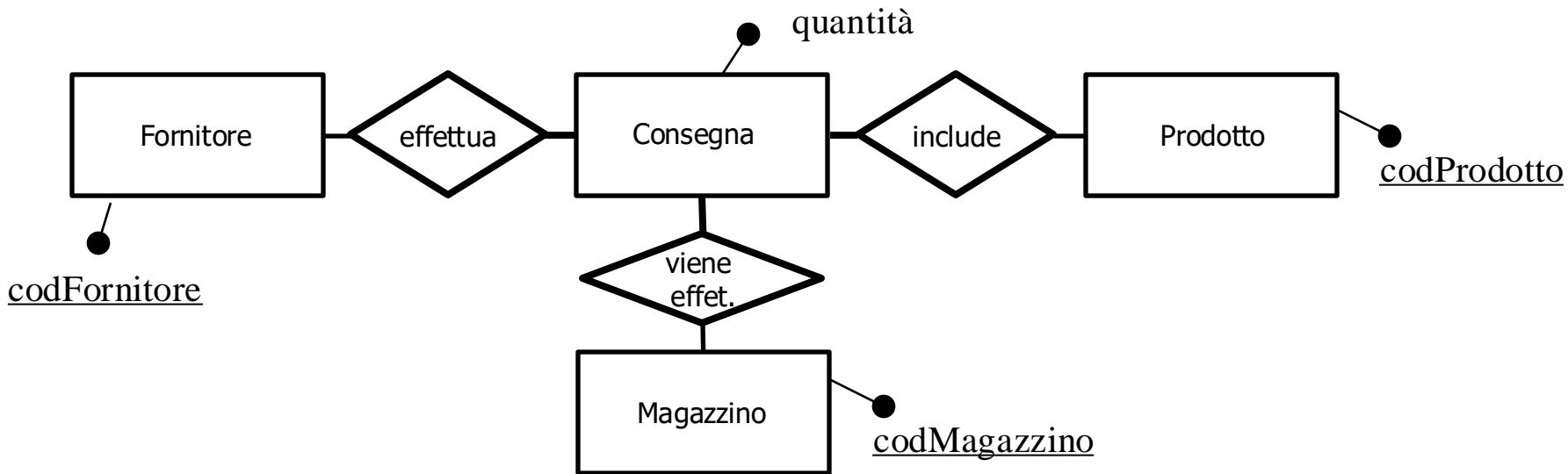
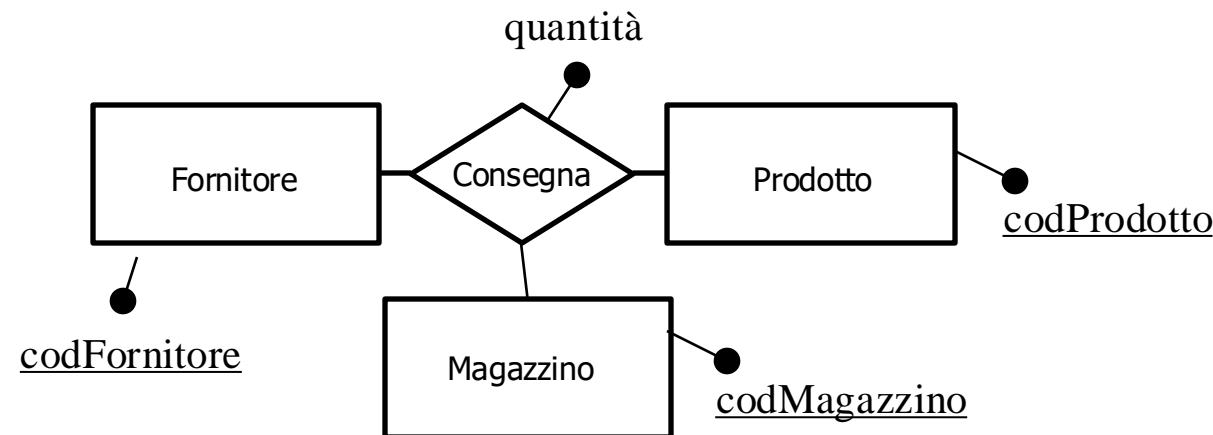




# Relazioni con grado > 2

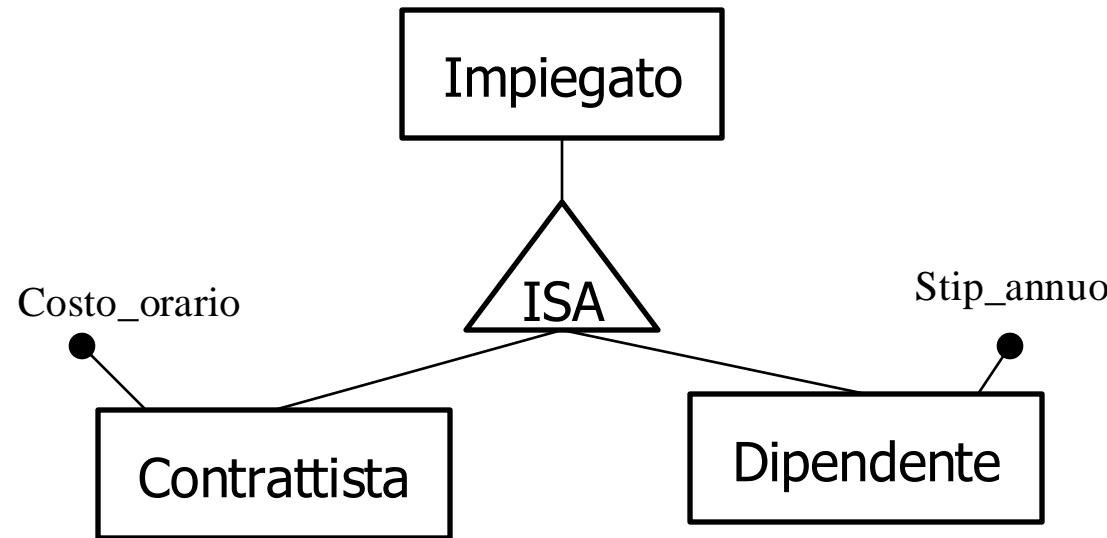
- Possono esistere relazioni ternarie ( $>3$  molto improbabili)
- Molti sistemi reali non consentono di “mappare” relazioni con grado  $> 2$ . E’ pertanto necessario rappresentare la relazione ternaria utilizzando relazioni binarie
- Questa operazione però può causare perdita di informazione se non condotta con attenzione

# Relazioni con grado > 2 (Risoluzione)

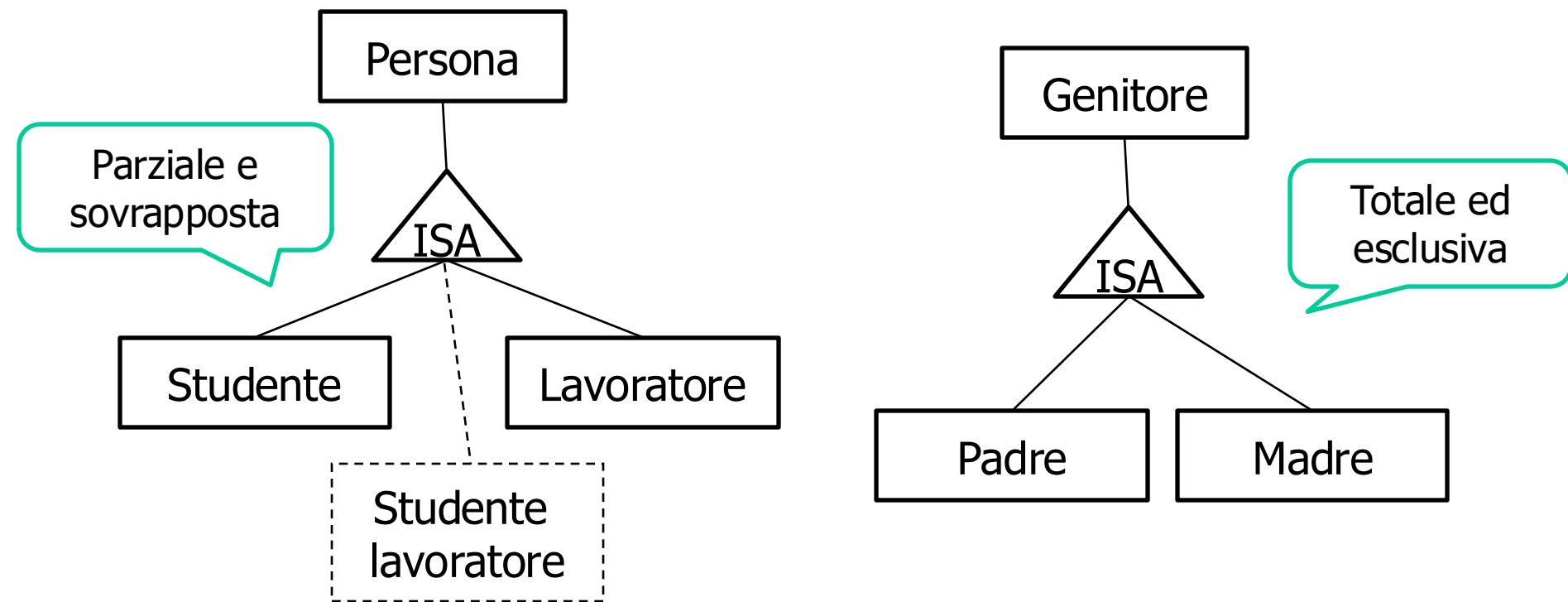


- Rappresentano legami logici tra una entità E (padre) e una o più entità E1, E2, ..., EN (figli). Il *padre* è più generale dei *figli* che sono considerabili specializzazioni

ISA : IS-A -> “è un” Es.: Un liceale *è uno* studente; un universitario *è uno* studente



- Generalizzazione totale → ogni occorrenza della classe **padre** è una occorrenza di almeno una delle **figlie** (in caso contrario si dice **parziale**)
- Generalizzazione **esclusiva** → ogni occorrenza della classe padre è al più una sola occorrenza di una delle classi figlie (in caso contrario si dice **sovraposta**)





# Considerazioni finali sul modello E-R

- Va costruito dopo una adeguata attività di raccolta di requisiti
- Consente una descrizione ad alto livello dei dati
- Aiuta a chiarire ulteriormente i requisiti
- Consente di esplicitare numerosi vincoli
- E' soggettivo: numerose scelte sono possibili e vanno adeguatamente ponderate
- Se costruito in modo adeguato consente la mappatura immediata nel modello logico relazionale
- E' comunque necessario procedere a raffinamenti e verifiche (normalizzazione)



# Feedback



- Qui puoi fornire il tuo feedback
- <https://forms.gle/9nAfH3hw9cXvt7BAA>



# Progettazione logica: Il modello relazionale





# Introduzione

- Basato sul lavoro di Codd (~1970)
- E' attualmente il modello più utilizzato nel mondo dei database (Oracle, Informix, IBM, Microsoft, etc.)
- E' basato su una struttura dati semplice ed uniforme, la **relazione**
- Ha solide basi teoriche

- Una base di dati è rappresentata come una collezione di relazioni
- Possiamo informalmente considerare una relazione come una tabella
- Ciascuna riga rappresenta una collezione di valori di dati tra loro collegati
- Il nome della relazione e quelli delle colonne consentono di comprendere il significato dei valori delle righe
- Tutti i valori di una colonna sono del medesimo tipo, appartengono cioè ad un medesimo dominio

Studente

Nome	Cognome	Matricola
Mario	Rossi	119310
Piero	Bianchi	514222



## Concetti base del modello (2/3)

- Dominio: un insieme di valori atomici, cioè indivisibili
  - Es.: voti\_università: valori tra 0 e 30
  - nomi\_propri: l'insieme dei nomi di persona
- A ciascun dominio si associa un tipo di dato o formato
  - Es: voti\_università: interi tra 0 e 30
  - nomi\_propri: stringhe di caratteri rappresentanti nomi propri



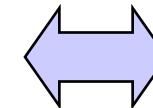
# Concetti base del modello (3/3)

- Schema (o componente intensionale) di una relazione  $R$ : è specificato come un nome  $R$  e una lista di attributi ( $A_1, A_2, \dots, A_n$ ). Ciascun attributo  $A_i$  corrisponde ad un dominio  $D(A_i)$  nello schema relazionale.  
Es.: Studente (nome, cognome, matricola, data\_nascita, data\_immatr)
- Grado della relazione: numero di attributi
- Istanza (o componente estensionale) di  $R$ :  $r(R)$ , è un insieme di tuple  $r=(t_1, t_2, \dots, t_m)$ . Ciascuna tupla  $t$  è una lista ordinata di valori  $t=<v_1, v_2, \dots, v_n>$ ; ciascun valore  $v_i \in \text{dom}(A_i) \cup \text{null}$  (dove null rappresenta uno speciale valore, sconosciuto, inapplicabile)
- Cardinalità: il numero di tuple nella estensione  $r(R)$

- Le tuple sono definite come insieme. Ciò implica l'assenza di un particolare ordinamento sulle tuple stesse

Studente

Nome	Cognome	Matricola
Mario	Rossi	119310
Piero	Bianchi	514222



Studente

Nome	Cognome	Matricola
Piero	Bianchi	514222
Mario	Rossi	119310

- I valori all'interno di una tupla sono stati definiti come "lista ordinata", quindi esiste un ordinamento
- N.B. Le relazioni sono definite a livello logico, quindi ha senso non avere ordinamento predefinito sulle tuple; è inoltre possibile, fornendo una diversa definizione di tupla, prescindere anche dall'ordinamento all'interno della tupla stessa. Intuitivamente è sufficiente considerare ogni valore all'interno della generica tupla direttamente associato al nome di attributo corrispondente.

- E' possibile interpretare uno schema di relazione come una dichiarazione o asserzione: ciascuna tupla può essere pertanto vista come un esempio di asserzione
- E' anche possibile interpretare uno schema di relazione come un predicato: i valori di ciascuna tupla soddisfano il predicato

- Vincolo di dominio: i valori di ciascun attributo sono valori **atomici** appartenenti al corrispondente dominio.
- Vincolo di chiave: per ogni relazione  $R$  esiste un sottoinsieme di attributi  $sk$  tale che  $t1[sk] \neq t2[sk]$ .
  - Il sottoinsieme  $sk$  dicesi **superchiave**
  - Dicesi **chiave** un sottoinsieme di attributi che gode della ulteriore proprietà di essere minimo
    - Es.: studente(nome, cognome, matricola, data\_nascita)  
Il sottoinsieme *nome, cognome, matricola* è superchiave, come pure *cognome, matricola*.  
*Matricola* è chiave
- In genere possono esserci più sottoinsiemi di attributi che verificano le due proprietà: **chiavi candidate**
  - La chiave designata è detta **chiave primaria (PK)**.



## Vincoli del modello (2/2)

- Integrità di entità: nessuna chiave primaria può assumere valore null.
- Integrità referenziale: una tupla in una relazione R1 che si riferisca ad un'altra relazione R2 deve riferirsi ad una tupla esistente in R2.
- Il precedente vincolo è normalmente espresso mediante il concetto di **chiave esterna** (foreign key).

Un sottoinsieme di attributi FK di una relazione R1 è una chiave esterna se:

1. Gli attributi di FK hanno lo stesso dominio degli attributi della chiave primaria PK di un'altra relazione R2
2. Un valore di FK in una tupla t1 di R1 o è presente identicamente come valore di PK di qualche tupla t2 in R2 o è nullo. Nel primo caso si avrà  $t1[FK]=t2[PK]$  e si dice che t1 **referenzia** t2.

# Vincoli del modello (esempio)

Infrazioni

	<u>Codice</u>	Data	Vigile	Prov	Numero
	34321	1/2/15	3987	MI	39548K
	53524	4/3/15	null	TO	E39548
	64521	5/4/16	3295	PR	839548
	73321	5/2/18	9345	PR	839548

Vigili

	<u>Matricola</u>	Cognome	Nome
	3987	Rossi	Luca
	3295	Neri	Piero
	9345	Neri	Mario
	7543	Mori	Gino

- Un vincolo di integrità referenziale ("foreign key") fra gli attributi X di una relazione R<sub>1</sub> e un'altra relazione R<sub>2</sub> impone ai valori su X in R<sub>1</sub> di comparire come valori della chiave primaria di R<sub>2</sub>

# Vincoli del modello (esempio foreign key)

## Infrazioni

<u>Codice</u>	Data	Vigile	Prov	Targa
34321	1/2/15	3987	MI	39548K
53524	4/3/15	3295	TO	E39548
64521	5/4/16	3295	PR	839548
73321	5/2/18	9345	PR	839548

Auto	<u>Prov</u>	<u>Targa</u>	Cognome	Nome
	MI	39548K	Rossi	Mario
	TO	E39548	Rossi	Mario
	PR	839548	Neri	Luca

- Creare una relazione per ogni entità con gli **attributi semplici della stessa entità**, scegliendo come chiave primaria una delle chiavi dell'entità
- Per le entità deboli:
  - creare una relazione con gli attributi dell'entità e includere, inoltre, come **chiave esterna**, la **chiave primaria dell'entità "proprietaria"**
  - la chiave primaria della nuova relazione sarà la combinazione della chiave esterna più la chiave parziale.

- Per ogni associazione binaria 1:1
  - identificare le entità partecipanti S e D
  - scegliere una entità S con partecipazione totale ed includere in essa, come chiave esterna, la PK di D.
  - includere in S anche eventuali attributi dell'associazione.
- Per ogni associazione binaria 1:N
  - identificare le entità partecipanti
  - scegliere l'entità S dal lato N ed includere in essa, come chiave esterna, la PK dell'altra
  - includere in S anche eventuali attributi dell'associazione.

- Per ogni associazione R di tipo M:N
  - creare una nuova relazione R
  - includere le chiavi esterne delle entità partecipanti
  - la loro combinazione formerà la PK di R
  - includere eventuali attributi di R
- Per ogni attributo multivalore A
  - creare una nuova relazione che include A e la chiave primaria della entità di partenza come chiave esterna
  - la PK sarà la combinazione della chiave esterna e di A
- Per ogni associazione con grado > 2
  - creare una nuova relazione
  - includere come chiavi esterne le PK delle entità partecipanti nonché eventuali attributi dell'associazione
  - la PK sarà la combinazione delle chiavi esterne.



# Algebra Relazionale





# Introduzione

- L'algebra relazionale è un linguaggio **procedurale**: le operazioni vengono specificate descrivendo il procedimento da seguire
- Essa è un'algebra **chiusa**. E' costituita da un insieme di operatori definiti su relazioni che producono relazioni
- Pertanto il risultato di una operazione può essere ulteriormente manipolato
- I principali operatori sono:
  - selezione, proiezione, ridenominazione (**unari**)
  - join (**binario**)
- Esistono le operazioni tipiche dell'insiemistica:
  - unione, intersezione, differenza e prodotto cartesiano



# Selezione (1/2)

- Consente di selezionare un *sottoinsieme delle tuple* di una relazione che soddisfino una **condizione di selezione**
- L'operazione è espressa in generale come:

$$\sigma_{<\text{condizione\_di\_selezione}>}(<\text{relazione}>)$$

- La condizione di selezione può prevedere comparazione tra attributi compatibili o con costanti, oltre a essere ottenuta collegando varie condizioni mediante connettivi logici (and, or, not)

- L'operazione viene applicata a ciascuna tupla individualmente
- La relazione risultante mantiene lo **schema** di quella di partenza, i.e. il **grado** rimane invariato
- La **cardinalità** risultante è invece  $\leq$  di quella di partenza.
- La selezione è commutativa:

$$\sigma_{\langle cond_1 \rangle}(\sigma_{\langle cond_2 \rangle} R) = \sigma_{\langle cond_2 \rangle}(\sigma_{\langle cond_1 \rangle} R)$$

$\sigma_{\text{dataNasc}>01-01-60 \wedge \text{dataAss}>05-03-93}$  Impiegato

## Impiegato

Cognome	Nome	Data di nascita	Data di assunzione
Davolio	Nancy	08-dic-48	01-mag-92
Fuller	Andrew	19-feb-52	14-ago-92
Leverling	Janet	30-ago-63	01-apr-92
Peacock	Margaret	19-set-37	03-mag-93
Buchanan	Steven	04-mar-55	17-ott-93
Suyama	Michael	02-lug-63	17-ott-93
King	Robert	29-mag-60	02-gen-94
Callahan	Laura	09-gen-58	05-mar-94
Dodsworth	Anne	27-gen-66	15-nov-94
Rossi	mario		

Cognome	Nome	Data di nascita	Data di assunzione
Suyama	Michael	02-lug-63	17-ott-93
King	Robert	29-mag-60	02-gen-94
Dodsworth	Anne	27-gen-66	15-nov-94



# Proiezione

- Consente di selezionare un *sottoinsieme delle colonne* (attributi) di una relazione elencati in una **lista di attributi** di proiezione
- L'operazione è espressa in generale come:

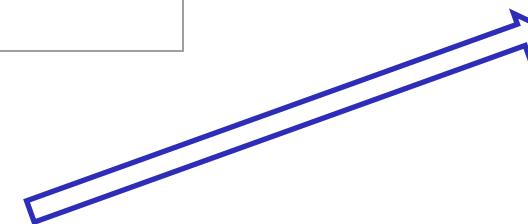
$$\pi_{<\text{lista\_di\_attributi}>}(<\text{relazione}>)$$

- L'operazione modifica, in generale, il **grado** (cioè il numero di attributi)
- L'operazione modifica, in generale, anche la **cardinalità** (NB: perché?)
- L'operazione non è commutativa

## Impiegato

Cognome	Nome	Posizione	Titolo	Data di nascita
Davolio	Nancy	Rappresentante	Dott.ssa	08-dic-48
Fuller	Andrew	Direttore vendite	Ing.	19-feb-52
Leverling	Janet	Funzionario commerciale	Dott.ssa	30-ago-63
Peacock	Margaret	Rappresentante	Dott.ssa	19-set-37
Buchanan	Steven	Direttore commerciale	Ing.	04-mar-55
Suyama	Michael	Rappresentante	Dott.	02-lug-63
King	Robert	Rappresentante	Dott.	29-mag-60
Callahan	Laura	Resp. comm. di zona	Dott.ssa	09-gen-58
Dodsworth	Anne	Rappresentante	Dott.ssa	27-gen-66
Rossi	mario			

Cognome	Nome
Davolio	Nancy
Fuller	Andrew
Leverling	Janet
Peacock	Margaret
Buchanan	Steven
Suyama	Michael
King	Robert
Callahan	Laura
Dodsworth	Anne
Rossi	mario



$\pi_{cognome, nome}(\text{impiegato})$



$\pi_{titolo}(\text{impiegato})$

Titolo
Dott.
Dott.ssa
Ing.

- Trattando operazioni in una algebra chiusa è ovviamente possibile scrivere espressioni come sequenza di operazioni

$$\pi_{cognome, nome}(\sigma_{titolo="dott."}(impiegato))$$

Cognome	Nome	Posizione	Titolo	Data di nascita
Davolio	Nancy	Rappresentante	Dott.ssa	08-dic-48
Fuller	Andrew	Direttore vendite	Ing.	19-feb-52
Leverling	Janet	Funzionario commerciale	Dott.ssa	30-ago-63
Peacock	Margaret	Rappresentante	Dott.ssa	19-set-37
Buchanan	Steven	Direttore commerciale	Ing.	04-mar-55
Suyama	Michael	Rappresentante	Dott.	02-lug-63
King	Robert	Rappresentante	Dott.	29-mag-60
Callahan	Laura	Resp. comm. di zona	Dott.ssa	09-gen-58
Dodsworth	Anne	Rappresentante	Dott.ssa	27-gen-66
Rossi	mario			



Nome	Cognome
Michael	Suyama
Robert	King

- Consente di **rinominare** uno o più attributi di una relazione per facilitare le precedenti operazioni insiemistiche
- L'operazione è espressa in generale come:

$$\rho_{<\text{lista\_attributi\_rinominati}>} \leftarrow <\text{lista\_attributi}> (<\text{relazione}>)$$

- Vengono modificati solo i nomi degli attributi, i valori **non cambiano**

## Paternità

<b>Padre</b>	<b>Figlio</b>
Adamò	Abele
Abramo	Isacco

## $\rho_{\text{Genitore} \leftarrow \text{Padre}} (\text{Paternità})$

<b>Genitore</b>	<b>Figlio</b>
Adamò	Abele
Abramo	Isacco



# Unione, Intersezione, Differenza (1/2)

- Operazioni **binarie** dell'insiemistica
- Hanno senso su relazioni **compatibili** (tuple omogenee)
- Identici attributi o almeno appartenenti al medesimo dominio (**union-compatible**, esiste una corrispondenza 1:1 fra gli attributi delle due relazioni, ossia se i valori dei loro rispettivi domini appartengono allo stesso tipo di dati)
- **Unione**:  $R \cup S$  produce una relazione che include tutte le tuple presenti in  $R$  o in  $S$  o in entrambe

$$\text{grado } (R \cup S) = \text{grado } (R) = \text{grado } (S)$$

$$\text{cardinalità } (R \cup S) \leq \text{cardinalità } (R) + \text{cardinalità } (S)$$



## Unione, Intersezione, Differenza (2/2)

- **Intersezione:**  $R \cap S$  produce una relazione che include tutte le tuple presenti sia in R che in S

$$\text{grado} (R \cap S) = \text{grado} (R) = \text{grado} (S)$$

$$\text{cardinalità} (R \cap S) \leq \min (\text{cardinalità} (R), \text{cardinalità} (S))$$

- **Differenza:**  $R - S$  produce una relazione che include tutte le tuple presenti in R, ma non in S

$$\text{grado} (R - S) = \text{grado} (R) = \text{grado} (S)$$

$$\text{cardinalità} (R - S) \leq \text{cardinalità} (R)$$

# Esempio

Studente (S)

matricola	cognome	città
119310	di sciascio	bari
125314	rossi	bari
199111	verdi	foggia

Laureato (L)

mat	cognome	città
119310	di sciascio	bari
125314	rossi	bari
501111	bianchi	lecce

Unione: S  $\cup$  L

mat	cognome	città
119310	di sciascio	bari
125314	rossi	bari
199111	verdi	foggia
501111	bianchi	lecce

Intersezione: S  $\cap$  L

mat	cognome	città
119310	di sciascio	bari
125314	rossi	bari

Differenza: L-S

mat	cognome	città
501111	bianchi	lecce

# Prodotto cartesiano

- E' una operazione **binaria** su insiemi, ma **non richiede la compatibilità** delle relazioni partecipanti
- Essa è denotata con il simbolo "X" ( $R \times S$ )
- Il grado della relazione risultante è pari alla somma dei gradi delle due relazioni partecipanti
- La cardinalità è pari al prodotto di quelle delle due relazioni partecipanti

$$\text{grado } (R \times S) = \text{grado } (R) + \text{grado } (S)$$

$$\text{cardinalità } (R \times S) \leq \text{cardinalità } (R) * \text{cardinalità } (S)$$

# Esempio

## Impiegato (I)

Matricola	Cognome	Progetto
M000	Rossi	A
M001	Bianchi	A
M002	Verdi	B

## Progetto (P)

Codice	Nome
A	Venere
B	Marte

## Impiegato – Progetto (I x P)

Matricola	Cognome	Progetto	Codice	Nome
M000	Rossi	A	A	Venere
M000	Rossi	A	B	Marte
M001	Bianchi	A	A	Venere
M001	Bianchi	A	B	Marte
M002	Verdi	B	A	Venere
M002	Verdi	B	B	Marte





# Join

- E' l'operazione utilizzata per combinare coppie di tuple, provenienti da relazioni collegate, in singole tuple
- Consente di elaborare le "associazioni" tra relazioni

$$R \triangleright\triangleleft_{<\text{condizione}>} S = \sigma_{<\text{condizione}>} (R \times S)$$

- *<condizione>* è dello stesso tipo di quella già vista per la selezione
- Il join con una condizione generica è detto **theta join**
- Quando la condizione è una uguaglianza viene invece detto **equi join**
  - Questa operazione includerà **due volte**, nella tabella risultante, gli attributi su cui è posta la condizione di uguaglianza

# Esempio

## Impiegato (I)

Matricola	Cognome	Progetto
M000	Rossi	A
M001	Bianchi	A
M002	Verdi	B

## Progetto (P)

Codice	Nome
A	Venere
B	Marte

$I \triangleright \triangleleft_{\text{Progetto} = \text{Codice}} P$

Matricola	Cognome	Progetto	Codice	Nome
M000	Rossi	A	A	Venere
M001	Bianchi	A	A	Venere
M002	Verdi	B	B	Marte

- Il join è detto **completo** se ciascuna tupla di ciascuna delle relazioni contribuisce ad almeno una tupla del risultato

- Quando l'operazione si limita a riportare **una volta** l'attributo di comparazione nella tabella risultante parleremo di Join naturale (**Natural Join**)
  - Utilizza una condizione di uguaglianza su attributi con lo **stesso nome**

## Impiegato (I)

Matricola	Cognome	CodProg
M000	Rossi	A
M001	Bianchi	A
M002	Verdi	B

## Progetto (P)

CodProg	Nome
A	Venere
B	Marte

$I \bowtie P$

Matricola	Cognome	CodProg	Nome
M000	Rossi	A	Venere
M001	Bianchi	A	Venere
M002	Verdi	B	Marte

- Inserisce nella relazione risultante le tuple che contribuiscono al risultato, eventualmente estese con valori nulli nel caso in cui non ci siano controparti opportune

- LEFT Outer Join       $R \triangleright\triangleleft_{LEFT} S$
- RIGHT Outer Join      $R \triangleright\triangleleft_{RIGHT} S$
- FULL Outer Join       $R \triangleright\triangleleft_{FULL} S$

**Impiegato (I)**

Matricola	Cognome	CodProg
M000	Rossi	A
M001	Bianchi	A
M002	Verdi	B
M003	Neri	NULL

**Progetto (P)**

CodProg	Nome
A	Venere
B	Marte
C	Giove

# Outer Join (2/2)

$I \triangleright\triangleleft P$

Matricola	Cognome	CodProg	Nome
M000	Rossi	A	Venere
M001	Bianchi	A	Venere
M002	Verdi	B	Marte

$I \triangleright\triangleleft_{LEFT} P$

Matricola	Cognome	CodProg	Nome
M000	Rossi	A	Venere
M001	Bianchi	A	Venere
M002	Verdi	B	Marte
M003	Neri	NULL	NULL

$I \triangleright\triangleleft_{RIGHT} P$

Matricola	Cognome	CodProg	Nome
M000	Rossi	A	Venere
M001	Bianchi	A	Venere
M002	Verdi	B	Marte
NULL	NULL	C	Giove

$I \triangleright\triangleleft_{FULL} P$

Matricola	Cognome	CodProg	Nome
M000	Rossi	A	Venere
M001	Bianchi	A	Venere
M002	Verdi	B	Marte
M003	Neri	NULL	NULL
NULL	NULL	C	Giove



# Evaluation Form

<https://forms.gle/otRdrvfpwDtiMEG9>



# Raffinamento dello schema e Normalizzazione nei database relazionali



# Introduzione

- La modellazione E-R ci ha consentito di descrivere schemi relazionali
- Lo strumento base per la modellazione è stato finora “la ragionevolezza”
- Esistono però metodi **formali** per assicurare la scelta di “buoni” schemi relazionali
- Tali metodi verranno essenzialmente utilizzati per verificare e raffinare lo schema

# Una relazione con anomalie

<u>Impiegato</u>	<u>Stipendio</u>	<u>Progetto</u>	<u>Bilancio</u>	<u>Funzione</u>
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

- Lo stipendio di ciascun impiegato è ripetuto in tutte le ennuple relative
  - ridondanza
- Se lo stipendio di un impiegato varia, è necessario andarne a modificare il valore in diverse ennuple
  - anomalia di aggiornamento
- Se un impiegato interrompe la partecipazione a tutti i progetti, dobbiamo cancellarlo
  - anomalia di cancellazione
- Un nuovo impiegato senza progetto non può essere inserito
  - anomalia di inserimento

- abbiamo usato un'unica relazione per rappresentare informazioni eterogenee
  - gli impiegati con i relativi stipendi
  - i progetti con i relativi bilanci
  - le partecipazioni degli impiegati ai progetti con le relative funzioni

- Scelta degli attributi:
  - Progettare lo schema in modo che sia **semplice** descrivere il suo significato, prestare cioè attenzione alla semantica degli attributi
  - Evitare di riunire in uno schema di relazione attributi propri di **entità diverse** del mondo reale
- Ridurre la **ridondanza**. Ha un duplice scopo:
  - ridurre la “storage area”
  - evitare le **anomalie** da aggiornamento
- Anomalie da aggiornamento
  1. Anomalie da inserimento
  2. Anomalie da modifica
  3. Anomalie da cancellazione



# Linee guida informali (2/3)

## Impiegato\_dipartimento

Nome	Cognome	CF	Data_n	indirizzo	StipAnnuo	<u><b>Id_imp</b></u>	<b>Id_dip</b>	DipNome	Dir_dip	Data_in
------	---------	----	--------	-----------	-----------	----------------------	---------------	---------	---------	---------

### Anomalie da inserimento

- Inserimento di un nuovo impiegato → inserimento dati dipartimento oppure null
- Inserimento dati nuovo dipartimento → almeno 1 impiegato (**Id\_imp** è PK, non null)

### Anomalie da cancellazione

- Eliminazione dell'ultimo dipendente di un dipartimento → perdita delle informazioni sul dipartimento

### Anomalie da modifica

- Nuovo direttore dipartimento → modifica di tutte le tuple relative a impiegati di quel dipartimento

- Ridurre il numero di valori **null** nelle tuple
- Valori null possono essere variamente interpretati: non applicabile, sconosciuto, non inserito
- In molti casi conviene creare **nuove relazioni** per ridurre i null

Esempio: Se solo il 5% degli impiegati ha un fax personale avremo il 95% di null

## Impiegato\_Fax

Nome	Cognome	CF	Data_n	indirizzo	StipAnnuo	<b><u>Id_Imp</u></b>	Id_dip	Tel.	Fax.
------	---------	----	--------	-----------	-----------	----------------------	--------	------	------

## Impiegato

Nome	Cognome	CF	Data_n	indirizzo	StipAnnuo	<b><u>Id_Imp</u></b>	Id_dip	Tel.	<b><u>Id_Imp</u></b>	Fax.
------	---------	----	--------	-----------	-----------	----------------------	--------	------	----------------------	------

## Fax\_Impiegato

- Evitare le tuple **spurie** (cioè con informazioni **non corrette**)
- Può essere generata a seguito di un join naturale
- E' necessario realizzare schemi che consentano di effettuare join con condizioni di uguaglianza su attributi che siano o chiavi **primarie** o chiavi **esterne**, così da garantire l'assenza di tuple spurie



# Tuple Spurie

Impiegato	Categoria	Stipendio
Rossi	2	1800
Verdi	3	1800
Bianchi	4	2500
Neri	6	3500
Bruni	2	1800

Impiegato	Stipendio	Categoria	Stipendio
Rossi	1800	2	1800
Verdi	1800	3	1800
Bianchi	2500	4	2500
Neri	3500	5	2500
Bruni	1800	6	3500

- Per studiare in maniera sistematica questi aspetti, è necessario introdurre un vincolo di integrità: la **dipendenza funzionale**
- Es: 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 (anche se può avere funzioni diverse in progetti diversi)

- Una dipendenza funzionale è un **vincolo** tra due **insiemi** di attributi di una base di dati e dipende dalla semantica dello schema
- Data uno schema relazionale R e due **sottoinsiemi** non vuoti di attributi X e Y esiste una dipendenza funzionale  $X \rightarrow Y$  se per ogni istanza r di R:

$$t1 \in r, t2 \in r, \Pi_x(t1) = \Pi_x(t2) \Rightarrow \Pi_y(t1) = \Pi_y(t2)$$

Date due tuple in r se i **valori su X** coincidono, anche i **valori su Y** coincidono

- Se un vincolo su uno schema R indica che X sia una **chiave candidata** di R ciò implica che  $X \rightarrow Y$  **per ogni** sottoinsieme Y di attributi di R
- **Non vale** la proprietà :  $X \rightarrow Y \Rightarrow Y \rightarrow X$



# Dipendenze funzionali

Y → Z

- Esempi:

Impiegato → Stipendio

Progetto → Bilancio

Impiegato Progetto → Funzione

# Dipendenze funzionali

<u>Impiegato</u>	<u>Stipendio</u>	<u>Progetto</u>	<u>Bilancio</u>	<u>Funzione</u>
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

Le anomalie sono legate ad alcune FD

- gli impiegati hanno un unico stipendio

Impiegato → Stipendio

- i progetti hanno un unico bilancio

Progetto → Bilancio

- Le anomalie sono causate dalla presenza di concetti eterogenei:
  - proprietà degli impiegati (lo stipendio)
  - proprietà di progetti (il bilancio)
  - proprietà della chiave Impiegato Progetto



# Dipendenze funzionali

- Sia  $F$  l'insieme delle FD che sono specificate su uno schema  $R$
- In generale ci saranno comunque altre FD non specificate esplicitamente
- Chiamiamo  $F^+$ , **chiusura** di  $F$ , l'insieme delle FD implicate da  $F$
- Esistono regole, corrette e complete, per determinare le FD, detti **assiomi di Armstrong**:
  - Riflessività:  $X \supseteq Y \Rightarrow X \rightarrow Y$
  - Incremento:  $X \rightarrow Y \Rightarrow XZ \rightarrow YZ$
  - Transitività:  $X \rightarrow Y, Y \rightarrow Z \Rightarrow X \rightarrow Z$
- Possono essere anche determinate altre utili regole:
  - Unione:  $X \rightarrow Y, X \rightarrow Z \Rightarrow X \rightarrow YZ$
  - Decomposizione:  $X \rightarrow YZ \Rightarrow X \rightarrow Y$



# Dipendenze funzionali

- Un insieme  $F$  di dipendenze funzionali è **minimo** se:
  - Ogni FD ha **1 solo** attributo alla sua destra
  - Non è possibile **rimuovere** alcuna FD da  $F$  ed avere ancora un insieme equivalente ad  $F$
  - La parte sinistra di ogni dipendenza funzionale è **minima**
- In tal modo otterremmo sicuramente schemi **privi** di **ridondanza**
- Problemi: il calcolo della  $F^+$  è computazionalmente **pesante** e in genere va accettata una certa quantità di ridondanza per motivi di **efficienza**

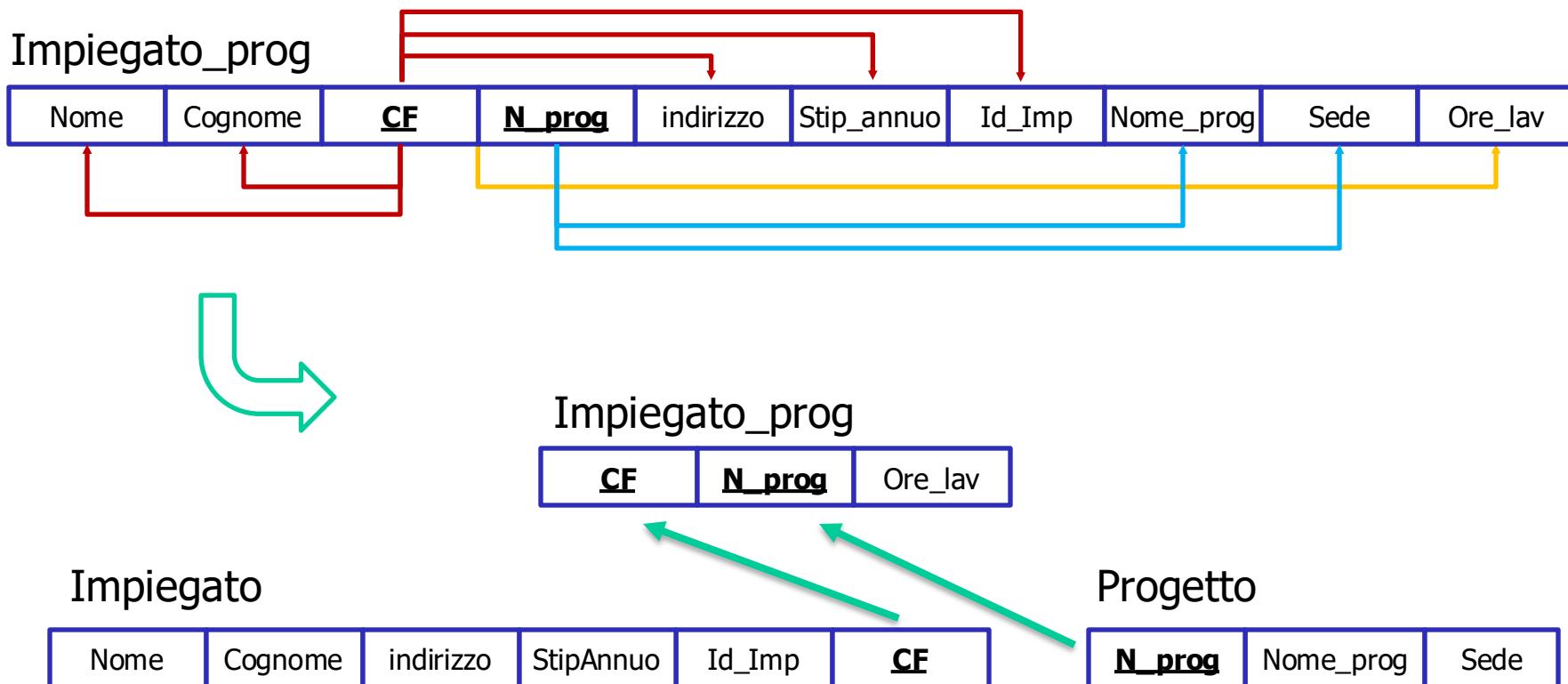
- Un metodo formale per analizzare le relazioni
- Consente una eventuale **decomposizione** di relazioni in schemi che godono di migliori proprietà
- E' in generale necessario che anche altre proprietà siano verificate per garantire un buon progetto:
  - Lossless join (join non additivo o decomposizione senza perdite)
  - Conservazione delle dipendenze



# Prima Forma Normale (1NF)

- Coincide in pratica con la definizione di **relazione**
- Il dominio di ciascun attributo deve consistere di **valori atomici** e il valore di un attributo in una tupla deve essere un **singolo** del dominio
- Esiste una **chiave primaria** che identifica in modo **univoco** ogni tupla della relazione
- Il metodo di decomposizione è quello già descritto per il mapping ER relazionale

- Uno schema R è in questa forma se è in 1FN e ogni attributo non chiave ha una FD **piena** dall'intera chiave primaria di R (non solo da una parte di essa)



- Uno schema R è in questa forma se è in 2FN e per ogni FD  $X \rightarrow Y$ , X è una **superchiave** di R oppure ogni attributo in Y è un attributo primo (fa parte di una chiave di R)
- Non esistono** attributi che dipendono *transitivamente* da altri attributi non-chiave

## Impiegato\_Dipartimento

Nome	Cognome	<b>CF</b>	Data_n	indirizzo	StipAnnuo	Id_Imp	Id_dip	DipNome	Dir_dip	Data_in
------	---------	-----------	--------	-----------	-----------	--------	--------	---------	---------	---------



## Impiegato

Nome	Cognome	<b>CF</b>	Data_n	indirizzo	StipAnnuo	Id_Imp	Id_dip
------	---------	-----------	--------	-----------	-----------	--------	--------

## Dipartimento

<b>Id_Dip</b>	DipNome	Dir_dip	Data_in
---------------	---------	---------	---------

- Uno schema R è in questa forma se è in 2FN e per ogni FD  $X \rightarrow Y$ , X è una **superchiave** di R oppure ogni attributo in Y è un attributo primo (fa parte di una chiave di R)
- Non esistono** attributi che dipendono *transitivamente* da altri attributi non-chiave

Dirigenza

	Dirigente	Progetto	Sede
Rossi	Marte	Roma	
Verdi	Giove	Milano	
Verdi	Marte	Milano	
Neri	Saturno	Milano	
Neri	Venere	Milano	

- ogni Dirigente opera presso una Sede
- ogni Progetto ha più Dirigenti, ma in Sedi diverse
- ogni Dirigente può essere responsabile di più progetti
- per ogni Sede, un Progetto ha un solo Dirigente
- Dirigente -> Sede**
- Progetto Sede → Dirigente**

credits: Prof.ssa D'Amato



# Forma Normale di Boyce-Codd

- Uno schema R è in questa forma se è in 3FN e per ogni FD  $X \rightarrow Y$ , X è una **superchiave** di R
- In altre parole R è in BCNF se le uniche FD non ovvie sono vincoli di chiave
- Se R è in BCNF è ovviamente anche in 3NF. Se una R è in 3NF, ma non in BCNF è ancora possibile la **ridondanza**
- 3NF è comunque accettabile ed è sempre possibile decomporre una R attraverso relazioni in 3NF che godano anche del lossless-join e preservino le dipendenze



# Problemi delle decomposizioni

- Interrogazioni più costose a causa dei join
- Perdita di informazione (tuple spurie) dovuta a join
- Il controllo delle dipendenze può richiedere l'utilizzo di join

- Data un relazione  $R(X)$  su un insieme di attributi

$$X = X_1 \cup X_2$$

Si parla di decomposizione senza perdite se il join delle proiezioni di  $R$  su  $X_1$  e  $X_2$  è uguale a  $R$  e **non contiene tuple spurie**

$$\pi_{X_1}(R) \bowtie \pi_{X_2}(R) = R$$

- $R$  si decomponete senza perdite su due relazioni se l'insieme degli attributi comuni è **chiave** per almeno una delle relazioni decomposte [condizione sufficiente ma non necessaria]

$$X_1 \cap X_2 = X_0 \quad X_0 \rightarrow X_1 \text{ or } X_0 \rightarrow X_2$$

# Decomposizione senza perdite (Esempio)

## Impiegato\_Progetto

<u>Impiegato</u>	<u>Progetto</u>	<u>Sede</u>
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Venere	Milano
Neri	Saturno	Milano
Neri	Venere	Milano



## Impiegato

<u>Impiegato</u>	<u>Sede</u>
Rossi	Roma
Verdi	Milano
Neri	Milano

## Impiegato *Natural JOIN* Progetto

<u>Impiegato</u>	<u>Progetto</u>	<u>Sede</u>
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Venere	Milano
Neri	Saturno	Milano
Neri	Venere	Milano
Verdi	Saturno	Milano
Neri	Giove	Milano



## Progetto

<u>Progetto</u>	<u>Sede</u>
Marte	Roma
Giove	Milano
Saturno	Milano
Venere	Milano

Associazioni *impiegato/progetto* non presenti nella tabella di partenza!

## Decomposizione senza perdite (2/2)

- Le relazioni in R sono **ricostruibili** con precisione
- La **conservazione delle dipendenze**, che garantisce il mantenimento dei vincoli di integrità originari
- Una decomposizione **conserva le dipendenze** se ciascuna delle dipendenze funzionali dello schema originario coinvolge attributi che compaiono tutti insieme in uno degli schemi decomposti
- La decomposizione senza perdita è garantita se gli attributi comuni contengono una **chiave** per almeno una delle relazioni decomposte
- Data R con FD F, se  $X \rightarrow Y$  viola la BCNF possiamo sempre decomporre in R-Y e XY ottenendo relazioni BCNF e lossless join, ma non è assicurata la conservazione delle dipendenze

## Decomposizione senza perdite (2/2)

- Data R con FD F, se  $X \rightarrow Y$  viola la BCNF possiamo sempre decomporre in R-Y e XY ottenendo relazioni BCNF e lossless join, ma non è assicurata la conservazione delle dipendenze

<b>Prefisso</b>	<b>Numero</b>	<b>Località</b>	<b>Abbonato</b>	<b>Indirizzo</b>
051	457856	Bologna	Rossi	Via Roma 8
059	452332	Modena	Verdi	Via Bari 16
051	987856	Bologna	Bianchi	Via Napoli 77
051	552346	Castenaso	Neri	Piazza Borsa 12
059	387654	Vignola	Mori	Via Piave 65

<b>Numero</b>	<b>Località</b>	<b>Abbonato</b>	<b>Indirizzo</b>	<b>Prefisso</b>	<b>Località</b>
457856	Bologna	Rossi	Via Roma 8	051	Bologna
452332	Modena	Verdi	Via Bari 16	059	Modena
987856	Bologna	Bianchi	Via Napoli 77	051	Castenaso
552346	Castenaso	Neri	Piazza Borsa 12	059	Vignola
387654	Vignola	Mori	Via Piave 65		



# Conservazione delle dipendenze

- Garantire sullo schema decomposto il soddisfacimento degli **stessi vincoli** presenti sullo schema originario (così da poterli verificare)
- Ciascuna delle dipendenze funzionali dello schema R di partenza, deve coinvolgere attributi che compaiono tutti **insieme** in uno degli schemi composti
- Le relazioni decomposte hanno le stesse capacità di R di rappresentare i vincoli di **integrità** e rilevare aggiornamenti **non consentiti**
- Esiste un algoritmo che garantisce il lossless join e la conservazione delle dipendenze con decomposizione in 3NF (non in BCNF)

# Conservazione delle dipendenze (Esempio)

## Impiegato\_Progetto

<u>Impiegato</u>	<u>Progetto</u>	<u>Sede</u>
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Venere	Milano
Neri	Saturno	Milano
Neri	Venere	Milano



## Impiegato

<u>Impiegato</u>	<u>Sede</u>
Rossi	Roma
Verdi	Milano
Neri	Milano

## Progetto

<u>Impiegato</u>	<u>Progetto</u>
Rossi	Marte
Verdi	Giove
Verdi	Venere
Neri	Saturno
Neri	Venere
Neri	Marte



Ogni impiegato lavora in una sede e può partecipare a più progetti ma solo se assegnati alla sede di afferenza

Se aggiungo in **Progetto** la tupla (Neri, Marte) vado a violare il vincolo sulla sede.

Nella tabella **Progetto** questa violazione non viene rilevata.



# Riassumendo

- Relazioni in BCNF sono **prive** di ridondanza
- Se non è possibile ottenere una decomposizione BCNF, lossless join e con conservazione delle dipendenze possiamo accontentarci di una 3NF
- Soprattutto considerare ciò che desideriamo, cioè tenere comunque in conto le **prestazioni** e le interrogazioni probabili

- Feedback
- <https://forms.gle/rjWKmUQhxcJCH1PHA>



## Structured Query Language



- SQL non è solo un linguaggio di **interrogazione**
- Linguaggio di **definizione** e **manipolazione** dati
- **DDL** (Data Definition Language): consente di definire lo schema: tabelle, vincoli, domini, viste, ecc.
- **DML** (Data Manipulation Language): permette la modifica e l'interrogazione dell'istanza di una base di dati
- Varie versioni con successivi miglioramenti:
  - SQL-1 o SQL-89
  - SQL-2 o SQL-92 (entry, intermediate e full)
  - SQL-3 o SQL-99



# Definizione dei domini elementari: Stringhe

- 6 gruppi di domini elementari. Da essi possono costruirsi nuovi domini.
- Carattere: consente di definire singoli caratteri oppure stringhe  
`character [varying][(num_char)][character set nome_set]`
- Es.(con uso della forma compatta):  
`Codice_fiscale char(13),`  
`prodotto_greco varchar(100) character set Greek`

- **Bit:** introdotto in SQL2 assume valori 0 e 1

bit [varying][(num\_bit)]

- Es.(con uso della forma compatta)  
Sequenza bit(5),  
Codice varbit(16)

- Tipi numerici esatti: consentono la rappresentazione di numeri interi o in virgola fissa

*integer*

*smallint*

- Differenze: *integer* e *smallint* non consentono di controllare la precision (dettata dal sistema di calcolo)  
precision di *integer*  $\geq$  precision di *smallint*

- Rappresentazione di numeri in virgola fissa  
`decimal [(precision[,scale])]`  
`numeric[(precision[,scale])]`
- **precision**: numero cifre significative
- **scale**: numero cifre dopo la virgola

`dato_vendita numeric(6,2)`

ES: consente di rappresentare i valori tra –9999,99 e +9999,99

- la precision di numeric è un valore **esatto**, il **minimo** per decimal

- Tipi numerici approssimati: consentono la rappresentazione di numeri in virgola mobile

Float[(precision)]

Real

Double precision

- A float può essere assegnata **esplicitamente** una precision (numero di cifre della mantissa)
- Per gli altri due la precisione dipende dal sistema di calcolo
  - precision di double precision  $\geq$  real (normalmente doppia)



# Definizione dei domini elementari: date/time

- Data e ora: consentono di rappresentare **istanti** di tempo

Date

Time [(precision)] [with time zone]

Timestamp [(precision)] [with time zone]

- Sono strutturati:
  - date: **year-month-day** (yyyy:mm:dd)
  - time: **hour-min-sec** (HH:MM:SS)
- Default precision: 0 (s) per time, 6 per timestamp ( $\mu$ s)
- Time zone fa riferimento all'ora di Greenwich

- **Time intervals:** consentono di rappresentare **intervalli** di tempo  
Interval FirstTimeUnit [to LastTimeUnit]
- Specifica un intervallo di tempo cioè un valore relativo utilizzabile per incrementare/decrementare un valore di date, timestamp.

durata **interval** year(5) to month

permette di rappresentare intervalli temporali fino a 99.999 anni e 11 mesi

durata **interval** day(4) to second(6)

permette di rappresentare intervalli temporali fino a 9.999 giorni, 23 ore, 59 minuti e 59,999999 secondi, con una precisione al milionesimo di secondo

- E' possibile definire **nuovi domini** a partire da quelli elementari, alternativamente è possibile dichiarare il dominio ed usarlo (peggiora la leggibilità e la modificabilità)

```
CREATE DOMAIN DomainName AS DataType [DefaultValue][Constraint]
```

### Esempio:

```
CREATE DOMAIN voto AS smallint DEFAULT 0 CHECK (VALUE >= 18 AND  
VALUE <= 30)
```

- Default: valore assunto da un attributo in assenza di specificazione  
Default <generic|user|null>
  - generic: valore scelto (purché nel dominio)
  - user: ID dell'utente che effettua l'update
  - null: default generico

- E' possibile definire lo **schema** di una base di dati.
- Lo schema sarà costituito da un insieme di domini, tabelle, indici, asserzioni, viste e autorizzazioni

```
CREATE SCHEMA [SchemaName] [[authorization] AuthorizedName] {DefiningElements}
```

- I DefiningElements possono essere definiti successivamente

Esempio:

```
CREATE SCHEMA azienda AUTHORIZATION antonio
```

- Una **tabella** viene definita specificando una collezione ordinata di **attributi** e un insieme di **vincoli**

CREATE TABLE RelationName(

    AttributeName Domain [DefaultValue][constraints]

    {, AttributeName Domain [DefaultValue][constraints] }

    FurtherConstraints )

Esempio:

CREATE TABLE Impiegato (

    cf char(16) primary key,

    nome varchar(30),

    cognome varchar(50) )

- Descrivono le condizioni che ciascuna istanza di una relazione deve sempre soddisfare
  - **intra-relazionali**: operano su una relazione
  - **inter-relazionali**: il predicato opera su più di una relazione
- **Vincoli intra-relazionali**: proprietà che devono essere verificate da ogni istanza del DB
  - **Vincoli di dominio**: i dati inseriti devono essere sempre del tipo corretto
  - **not null**: indica che il valore null non è ammesso per l'attributo che quindi va sempre specificato a meno che non sia previsto un default diverso da null

nome char(30) not null default pippo

Reazione alla violazione di vincoli: il sistema che rilevi **violazioni** di vincoli reagirà **rifiutando** il comando di aggiornamento e segnalando la violazione all'utente

- **Primary key:** specifica la chiave primaria. Può essere composta anche da **più attributi**, i cui valori non possono essere null

PRIMARY KEY (AttributeName{, AttributeName })

ESEMPIO:

CREATE TABLE aereo (  
    codAereo char(6),  
    codCompagnia char(8),  
    numPosti smallint,  
    **PRIMARY KEY** (codAereo, codCompagnia) )



CREATE TABLE aereo (  
    codAereo char(6) **PRIMARY KEY**,  
    codCompagnia char(8) **PRIMARY KEY**,  
    numPosti smallint )



- **unique**: impone l'unicità degli attributi cui il vincolo è applicato
- In pratica definisce una chiave candidata, ma non scelta come primaria

UNIQUE (AttributeName {,AttributeName })

Esempio:

```
CREATE TABLE Impiegato (
    nome varchar(30) not null,
    cognome varchar(30) not null,
    codFiscale char(13) primary key,
    matricola integer unique )
```

N.B.: unique(A,B) ≠ unique(A), unique(B)

- Stabiliscono i vincoli di integrità **referenziale**
- Crea un legame tra i valori di un attributo della tabella corrente R (**interna**) e i valori di un attributo di una tabella S (**esterna**)

```
FOREIGN KEY (AttributeName {,AttributeName}) REFERENCES  
TableName(AttributeName {,AttributeName})
```

- I campi devono essere una **chiave** (primaria, possibilmente) di S
- Nelle tuple di R i valori della FK devono corrispondere a valori **presenti** in S o essere null
- La corrispondenza tra attributi avviene in base all'**ordinamento espresso** nella dichiarazione.

```
CREATE TABLE Dipartimento (
    idDip char(8) primary key,
    nome varchar(50) )
```

```
CREATE TABLE Attivita (
    codProgetto char(8),
    numAttivita integer,
    primary key (codProgetto, numAttivita) )
```

```
CREATE TABLE Impiegato (
    codFiscale char(13) primary key
    nome varchar(30),
    cognome varchar(30),
    idDip char(8) references Dipartimento (idDip),
    codProgetto char(8),
    numAttivita integer,
    foreign key (codProgetto, numAttivita) references Attivita (codProgetto, numAttivita) )
```

- A seguito di violazioni nella tabella interna R la reazione è quella standard: l'azione viene rifiutata (**no action**)
- SQL consente però di definire tipi diversi di reazione a violazione di vincoli inter-relazionali
  - cancellazione di una tupla riferita
  - modifica di una primary key riferita
- A seguito di una azione nella tabella S esterna la reazione avviene nell'ambito della tabella R interna
- L'idea base è fornire un modo per **adeguare** una tabella che sia in associazione con un'altra a seguito di modifiche in quest'ultima

- Posso introdurre ulteriori specifiche di reazione che riducono la forza del vincolo
- La specifica di reazione va esplicitata subito dopo il vincolo di integrità
  - Aggiornamento      **ON UPDATE**
  - Cancellazione      **ON DELETE**
- Su un aggiornamento (on update) o cancellazione (on delete) posso specificare le **azioni** da intraprendere
  - cascade
  - set null
  - set default
  - no action

- Esplicitando le reazioni per la richiesta di cancellazione:  
ON DELETE <cascade | set null | set default | no action>
- **Cascade:** tutte le tuple della tabella R interna corrispondenti alla tupla cancellata in S vengono cancellate
- **Set null:** il valore cancellato nella tabella S viene sostituito con il valore null nell'attributo della tabella R
- **Set default:** il valore cancellato nella tabella S viene sostituito con un valore di default nell'attributo di R
- **No action:** la cancellazione viene inibita

- Esplicitando le reazioni per la richiesta di modifica:

ON UPDATE <cascade | set null | set default | no action>

- **Cascade**: il nuovo valore dell'attributo della tabella esterna S viene riportato su tutte le corrispondenti righe della tabella interna
- **Set null**: il valore modificato nella tabella S viene sostituito con il valore null nell'attributo di R
- **Set default**: il valore modificato nella tabella S viene sostituito con un valore di default nell'attributo di R
- **No action**: la modifica viene inibita



# Vincoli inter-relazionali (7/7)

```
CREATE TABLE Dipartimento (  
    idDip char(8) primary key,  
    nome varchar(50) )
```

```
CREATE TABLE Attivita (  
    codProgetto char(8),  
    numAttivita integer,  
primary key (codProgetto, numAttivita) )
```

```
CREATE TABLE Impiegato (  
    codFiscale char(13) primary key  
    nome varchar(30),  
    cognome varchar(30),  
    idDip char(8) references Dipartimento (idDip) ON DELETE set null ON UPDATE cascade ,  
    codProgetto char(8),  
    numAttivita integer,  
foreign key (codProgetto, numAttivita) references Attivita (codProgetto, numAttivita)  
ON DELETE set default ON UPDATE cascade )
```



# Clausola check

## CHECK (Condition)

- Condition corrisponde alle **condizioni** che possono essere specificate
- La condizione deve essere **sempre** verificata per mantenere **l'integrità** del DB

```
CREATE TABLE esami_sup (
    ...
    voto integer,
    check (voto >=18 and voto <= 30)
    ...
)
```

- Le modifiche possono consistere in alterazioni e cancellazioni di schemi e domini
- Per modifiche si usa il comando **Alter**

```
ALTER DOMAIN DomainName <
    set default DefaultValue | 
    drop default | 
    add constraint ConstraintDefinition | 
    drop constraint ConstraintName >
```

# Modifica di una tabella

```
ALTER TABLE TableName <  
    alter column ColumnName <  
        set default DefaultValue |  
        drop default > |  
    add constraint ConstraintDefinition |  
    drop constraint ConstraintName |  
    add column ColumnName |  
    drop column ColumnName >
```

Esempio: ALTER TABLE Impiegato ADD COLUMN telefono char(20)

ALTER TABLE Dipartimento DROP COLUMN città varchar(20)

- Per le cancellazioni si usa il comando Drop

DROP <schema|table|domain|view|assertion> ItemName [restrict|cascade]

Esempio: DROP TABLE Progetto CASCADE

- **Restrict** (opzione di default) specifica che il comando non deve essere eseguito in presenza di oggetti non vuoti
  - Per uno **schema**: richiede che lo schema sia vuoto
  - Per una **tabella**: richiede che sia vuota e non abbia vincoli esterni
  - Per un **dominio**: richiede che non sia presente in alcuna tabella
- **Cascade**: supera le limitazioni precedenti ed esegue una cancellazione in cascata, tutti gli oggetti specificati devono essere rimossi
  - Per uno **schema**: effettua una cancellazione completa
  - Per un **dominio**: cancella la definizione, ma gli attributi rimangono definiti secondo il dominio elementare di origine
  - Per una **tabella**: tutte le righe vengono perse e se la tabella compariva in qualche definizione di tabella o vista, anche queste vengono rimosse
  - Per una **vista**: elimina tutte le tabelle che compaiono nella definizione



# Feedback Form

<https://forms.gle/Z3mgYiv5iSKHH4mG6>



## SQL come linguaggio di manipolazione dei dati



- SQL è anche un linguaggio per la manipolazione dei dati (Data Manipulation Language)
  - Interrogazione
  - Inserimento
  - Aggiornamento
  - Cancellazione

- L'interrogazione è specificata in maniera **dichiarativa**: si specifica non il modo in cui l'interrogazione deve essere eseguita, ma le **caratteristiche del risultato** che deve fornire (obiettivo dell'interrogazione)

```
Select AttrExpr [[as] Alias]{, AttrExpr [[as] Alias]}
```

```
From TableName [[as] Alias]{, TableName [[as] Alias]}
```

```
[Where condition]
```

- Le tre parti componenti vengono chiamate **target list** (clausola select), **from clause** (clausola from) e **where clause** (clausola where)

- Select seleziona tra le righe che appartengono al prodotto cartesiano delle tabelle elencate nella **clausola from**, quelle che soddisfano le condizioni nella **clausola where**.
- Le **colonne** della tabella ottenuta dipenderanno dalla valutazione delle espressioni AttrEspr

IMPIEGATO (nome, cognome, dipartimento, stipendio\_mensile, extra\_mensile)

**Interrogazione 1 :** Selezionare nome e cognome di tutti i dipendenti

```
Select nome as nome_di_battesimo, cognome as cognome_di_battesimo  
From Impiegato
```

**Interrogazione 2 :** Estrarre le informazioni relative a tutti i dipendenti

```
Select *  
From Impiegato
```

**N.B.** L'uso dell'alias nella target list ha maggiore utilità quando si definiscono delle espressioni (vedere es. seguente) non corrispondenti a **nessun attributo** della base di dati a cui pertanto non è associato alcun nome



# Target List - Operatori Aritmetici

Le quattro operazioni aritmetiche (+, -, \*, /) si applicano ai seguenti operandi:

- Campi di una tabella
- Valori numerici
- Funzioni aggregate

**Interrogazione 3 :** Selezionare nome, cognome e stipendio annuale di tutti i dipendenti

```
Select nome, cognome, stipendio_mensile*12 as stipendio_annuale  
From Impiegato
```

**Interrogazione 4 :** Selezionare nome, cognome e totale mensile di tutti i dipendenti

```
Select nome, cognome, stipendio_mensile + extra_mensile as stipendio_tot  
From Impiegato
```

```
Select [ [ALL] | [DISTINCT] ] AttrExpr [[as] Alias],  
AttrExpr [[as] Alias]  
From TableName [[as] Alias]{, TableName [[as] Alias]}  
[Where condition]
```

**Interrogazione 5 :** Selezionare i cognomi di tutti i dipendenti facendo comparire ciascun cognome al più una volta

```
Select distinct cognome  
From Impiegato
```

Se i duplicati fossero rappresentati da più campi (cognome e stipendio\_mensile per esempio)?

## OPERATORI A VALORE SINGOLO

### OPERATORI DI CONFRONTO(OP) ( =, <>, >, >=, <, <= )

- Valori numerici o caratteri:

<condizione del where> ::= <espressione> OP <valore>

- NULL:

<condizione del where> ::= <espressione> IS [NOT] NULL

### OPERATORI LOGICI ( AND, OR )

- Valori numerici o caratteri:

<espressione logica> ::= <espressione> <OP> <valore>

<condizione del where> ::= <expressionelogica> {<AND|OR> <expressionelogica>}

### LIKE

- Solo caratteri:

<condizione del where> ::= <espressione> [NOT] LIKE <stringaModello>

La <stringaModello> contiene i caratteri *jolly* '%' e '\_'



# Clausola Where (2/2)

## OPERATORI A VALORI MULTIPLI

BETWEEN (Valori numerici o caratteri)

<condizione del where> ::= <espressione> [NOT] BETWEEN <valore infer> AND <valore super>

IN (Valori numerici o caratteri)

<condizione del where> ::= <espressione> [NOT] IN <valore1,valore2,..,valoreN>

**Interrogazione 6 :** Selezionare tutti i dati dei dipendenti di cognome De Paperis

Select \*

From Impiegato

Where cognome='De Paperis'

**Interrogazione 7 :** Selezionare nome e cognome dei dipendenti con uno stipendio annuo superiore a 30.000 euro

Select nome, cognome

From Impiegato

Where stipendio\_mensile\*12 > 30000

**Interrogazione 8 :** Selezionare nome e cognome dei dipendenti di cui non si conosce il valore dell'extra mensile

Select nome, cognome

From Impiegato

Where extra\_mensile IS NULL

**Interrogazione 9 :** Selezionare tutti i dati dei dipendenti con un cognome che contiene almeno due 'o' di cui la prima come terzo carattere

Select \*

From Impiegato

Where cognome LIKE '\_\_o%o%'

**Interrogazione 10 :** Selezionare nome e cognome dei dipendenti che lavorano al 'DEE' oppure al 'DIMEG'

Select nome, cognome

From Impiegato

Where dipartimento='DEE' OR dipartimento='DIMEG'

**Interrogazione 11 :** Selezionare nome e cognome dei dipendenti che lavorano al 'DEE' oppure al 'DIMEG' e che guadagnano più di 2550 euro

Select nome, cognome

From Impiegato

Where (dipartimento='DEE' OR dipartimento='DIMEG') AND stipendio\_mensile>2500



# IN e BETWEEN

**Interrogazione 12 :** Selezionare tutti i dati dei dipendenti con un nome che si trova nel seguente elenco: marco, maria, marta e francesco

Select \*

From Impiegato

Where nome IN ('marco','maria','marta','francesco')

**Interrogazione 13 :** Selezionare nome e cognome dei dipendenti che lavorano al DEE e percepiscono uno stipendio mensile inferiore a 1000 euro e superiore a 3000 euro

Select nome, cognome

From Impiegato

Where dipartimento='DEE' AND (stipendio\_mensile NOT BETWEEN 1000 AND 3000)

In che altro modo possono essere espresse le interrogazioni 12 e 13?

## SUBQUERY nella clausola WHERE

- **IN – per le subquery che restituiscono un insieme di valori**

<condizione del where> ::= <espressione> [NOT] IN <subquery>

- **Operatore di confronto – per le subquery che restituiscono un solo valore**

<condizione del where> ::= <espressione> OP <subquery>

- **ALL ed ANY – per le subquery che restituiscono un insieme di valori**

<condizione del where> ::= <espressione> OP <ALL | ANY> <subquery>

**ANY** – il valore dell'espressione deve essere uguale o diverso o maggiore o ... ad almeno uno dei valori restituiti dalla subquery

**ALL** – il valore dell'espressione deve essere uguale o diverso o maggiore o ... contemporaneamente a tutti i valori restituiti dalla subquery

IMPIEGATO (nome, cognome, dipartimento, ufficio, stipendio\_mensile)

ANAGRAFE\_AZIENDALE (nome, cognome, data\_nascita, data\_assunzione, tel)

**Interrogazione 14:** Selezionare i dati dell'anagrafe dei dipendenti che lavorano al DEE

Select \*

From Anagrafe\_aziendale

Where (nome, cognome) IN ( Select nome, cognome  
From Impiegato  
Where dipartimento='DEE' )

**Interrogazione 15 :** Selezionare i dati dei dipendenti che lavorano nel dipartimento in cui lavora Mario Rossi

Select \*

From Impiegato

Where dipartimento = ( Select dipartimento  
From Impiegato  
Where nome='Mario' AND cognome='Rossi' )

IMPIEGATO (nome, cognome, dipartimento, ufficio, stipendio\_mensile)

DIPARTIMENTO (nome, indirizzo, città, tel)

**Interrogazione 16 :** Selezionare i dati dei dipendenti che lavorano in dipartimenti situati a Firenze

Select \*

From Impiegato

Where dipartimento = ANY ( Select nome  
From Dipartimento  
Where città='Firenze' )

**Interrogazione 17 :** Selezionare i dati dei dipendenti del DEE che percepiscono lo stipendio maggiore

Select \*

From Impiegato

Where dipartimento='DEE' AND  
stipendio\_mensile >= ALL ( Select stipendio\_mensile  
From Impiegato  
Where dipartimento='DEE' )

# Clausola From: Combinazione Di Tabelle

TAB1

Col1	Col2
A	C
B	D

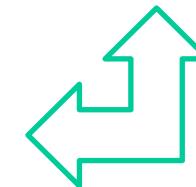
TAB2

Col1	Col2
1	3
2	4

Prodotto Cartesiano (TAB1 x TAB2)

Col1	Col2	Col1	Col2
A	C	1	3
A	C	2	4
B	D	1	3
B	D	2	4

Select \*  
From TAB1, TAB2



Come si accede alle colonne della tabella Prodotto Cartesiano che hanno lo stesso nome? → Utilizzare gli Alias

# Combinazione di tabelle con campi in comune

TAB1	
Col1	Col2
A	C
B	D

Prodotto Cartesiano ( $\text{Col2}=\text{Col3}$ )

Col1	Col2	Col3	Col4
B	D	D	4

Select \*  
From TAB1, TAB2  
Where Col2=Col3

TAB2	
Col3	Col4
A	3
D	4

Prodotto Cartesiano ( $\text{Col2}>\text{Col3}$ )

Col1	Col2	Col3	Col4
A	C	A	3
B	D	A	3

Select \*  
From TAB1, TAB2  
Where Col2>Col3



# Prodotto Cartesiano - Sintassi

<clausola *from*> ::= nomeTabella [[as] Alias], nomeTabella [[as] Alias]{,nomeTabella [[as] Alias]}

<condizione del where> ::= <condizione sui campi comuni>

{ <AND | OR> <condizione semplice o complessa>}

<condizione sui campi comuni> ::= <confronto campoTabella>{ <AND | OR> <confronto campoTebella>}

IMPIEGATO (nome, cognome, dipartimento, stipendio)

ANAGRAFE\_AZIENDALE (nome, cognome, data\_nascita, data\_assunzione, tel)

**Interrogazione 18 :** Selezionare tutte le informazioni disponibili per ciascun dipendente che lavora al DEE

Select I.nome, I.cognome, dipartimento, stipendio, data\_nascita, data\_assunzione, tel  
From Impiegato I, Anagrafe\_azendale A  
Where I.nome=A.nome **AND** I.cognome=A.cognome  
     **AND** dipartimento='DEE'

```
<clausola from> ::= nomeTabella [[as] Alias] { [<tipo di join>] join nomeTabella [[as] Alias] on
<condizione di join>
<tipo di join> ::= inner(valore di default che può essere omesso) | right outer | left outer | full outer
<condizione di join> ::= <confronto campoTabella>{ <AND|OR> <confronto campoTabella>}
<condizione del where> ::= = <condizione semplice o complessa>
                           { <AND|OR> <condizione semplice o complessa>}
```

- **Inner join** è equivalente alla combinazione di tabelle con **FROM** e **WHERE**
- **Outer join** (right o left) prende due tabelle e visualizza tutte le righe di una delle due (quella di destra o quella di sinistra) con solo i record dell'altra che soddisfano la condizione di join
  - NB: Il risultato dipende quindi da quale delle due tabelle compare per prima nella clausola FROM
- **Full join** preserva tutte le righe delle due tabelle e le righe della tabella che non soddisfano la condizione di join vengono riempite con **NULL**
- **Natural** davanti a join equivale a inner join senza condizione di join perché è implicita la condizione di uguaglianza su tutti gli attributi caratterizzati dallo stesso nome

# Left Outer Join

Impiegato (tabella di sin)

Matr	Nome	Cognome	Dip
A1	pio	rossi	1
A2	pia	rossi	1
B1	mario	bianchi	2
B2	maria	bianchi	NULL

Dipartimento (tabella di dx)

Codice	Nome	Indirizzo	Comune
1	DEE	Via Orabona,4	Bari
2	DIMEG	Viale Japigia, 182	Bari
3	DICA	Via Orabona,4	Bari

Select \*

From Impiegato left outer join Dipartimento  
on Dip=Codice

Matr	Nome	Cognome	Dip	Codice	Nome	Indirizzo	Comune
A1	pio	rossi	1	1	DEE	Via Orabona,4	Bari
A2	pia	rossi	1	1	DEE	Via Orabona,4	Bari
B1	mario	bianchi	2	2	DIMEG	Viale Japigia, 182	Bari
B2	maria	bianchi	NULL	NULL	NULL	NULL	NULL

# Right Outer Join

Impiegato (tabella di sin)

Matr	Nome	Cognome	Dip
A1	pio	rossi	1
A2	pia	rossi	1
B1	mario	bianchi	2
B2	maria	bianchi	NULL

Dipartimento (tabella di dx)

Codice	Nome	Indirizzo	Comune
1	DEE	Via Orabona,4	Bari
2	DIMEG	Viale Japigia, 182	Bari
3	DICA	Via Orabona,4	Bari

Select \*

From Impiegato right outer join Dipartimento  
on Dip=Codice

Matr	Nome	Cognome	Dip	Codice	Nome	Indirizzo	Comune
A1	pio	rossi	1	1	DEE	Via Orabona,4	Bari
A2	pia	rossi	1	1	DEE	Via Orabona,4	Bari
B1	mario	bianchi	2	2	DIMEG	Viale Japigia, 182	Bari
NULL	NULL	NULL	NULL	3	DICA	Via Orabona,4	Bari

# Full Join

Impiegato

Matr	Nome	Cognome	Dip
A1	pio	rossi	1
A2	pia	rossi	1
B1	mario	bianchi	2
B2	maria	bianchi	NULL

Dipartimento

Codice	Nome	Indirizzo	Comune
1	DEE	Via Orabona,4	Bari
2	DIMEG	Viale Japigia, 182	Bari
3	DICA	Via Orabona,4	Bari

Select \*  
 From Impiegato full join Dipartimento  
 on Dip=Codice

Matr	Nome	Cognome	Dip	Codice	Nome	Indirizzo	Comune
A1	pio	rossi	1	1	DEE	Via Orabona,4	Bari
A2	pia	rossi	1	1	DEE	Via Orabona,4	Bari
B1	mario	bianchi	2	2	DIMEG	Viale Japigia, 182	Bari
NULL	NULL	NULL	NULL	3	DICA	Via Orabona,4	Bari
B2	maria	bianchi	NULL	NULL	NULL	NULL	NULL

PERSONA (CF, nome, cognome, anno\_nascita, sesso)

MATRIMONIO (CF\_moglie, CF\_marito, anno\_matrimonio)

**Interrogazione 19 :** Trovare nome e cognome delle persone che si sono sposate entro i 20 anni di età

Select distinct nome, cognome

From Persona **join** Matrimonio **on** (CF = CF\_moglie **OR** CF = CF\_marito)

**Where** anno\_matrimonio <= anno\_nascita + 20

**Interrogazione 20:** Selezionare i dati anagrafici e l'anno di matrimonio di tutte le donne mantenendo nel risultato anche quelle non sposate.

Select nome, cognome, anno\_nascita, CF\_marito, anno\_matrimonio

From Persona **left join** Matrimonio **on** CF = CF\_moglie

**Where** sesso='F'

# Self Join e Alias (1/2)

Il **self join** è il join di una tabella con una copia di se stessa

**Problema:** se le righe della tabella originale **coincidono** con quelle identiche della tabella copia, bisogna eliminare le combinazioni di righe identiche (riga con se stessa).  
**CLIENTE** (CF, nome, cognome, età, sesso)

**PRANZO** (Data, CF\_cliente, Ristorante, num\_portate, costo\_tot)

**Interrogazione 21:** Selezionare i dati relativi ai clienti con lo stesso nome che hanno pranzato lo stesso giorno al prezzo di 10 euro.

Select \*

From cliente

Where CF IN ( Select c1.CF

From cliente as c1, cliente as c2, pranzo as p1, pranzo as p2

Where c1.CF=p1.CF\_cliente AND c2.CF=p2.CF\_cliente

AND p1.CF\_cliente<>p2.CF\_cliente

AND c1.nome=c2.nome

AND p1.Data=p2.Data

AND p1.costo\_tot=10 AND p2.costo\_tot=10 )

C1.CF	P1.CF_cliente	C2.cf	P2.cf_cliente	C1.nome	C2.nome	P1.data	P2.data	P1.costo_tot	P2.costo_tot
-------	---------------	-------	---------------	---------	---------	---------	---------	--------------	--------------

# Self Join e Alias (2/2)

PERSONA (CF, nome, cognome, data\_nascita, sesso, CF\_padre, CF\_madre)

**Interrogazione 22:** Estrarre gli uomini che hanno lo stesso nome del nonno paterno.

Select Figlio.\*

From (Persona as Figlio join Persona as Padre on Figlio.CF\_padre=Padre.CF)  
join Persona as Nonno on Padre.CF\_padre=Nonno.CF

Where Figlio.nome=Nonno.nome AND Figlio.sesso='M'

CF	Nome	Cognome	Data Nascita	Sesso	CF Padr e	CF Madr e
A	Claudio	Rossi	25/04/45	M	...	...
B	Carla	Verdi	15/03/48	F	...	...
C	Giorgio	Rossi	21/07/70	M	A	B
D	Claudio	Rossi	04/06/98	M	C	...

CF	Nome	Cognome	Data Nascita	Sesso	CF Padr e	CF Madr e
A	Claudio	Rossi	25/04/45	M	...	...
B	Carla	Verdi	15/03/48	F	...	...
C	Giorgio	Rossi	21/07/70	M	A	B
D	Claudio	Rossi	04/06/98	M	C	...

CF	Nome	Cognome	Data Nascita	Sesso
A	Claudio	Rossi	25/04/45	M
B	Carla	Verdi	15/03/48	F
C	Giorgio	Rossi	21/07/70	M
D	Claudio	Rossi	04/06/98	M

# Query con Raggruppamento (1/2)

```
Select AttrExpr [[as] Alias]{, AttrExpr [[as] Alias]}
```

```
From TableName [[as] Alias]{, TableName [[as] Alias]}
```

```
[Where condition]
```

```
[Group by Attr {, Attr}]
```

```
[Having condition]
```

```
[Order by AttrExpr [asc|desc] {, AttrExpr [asc|desc]}]
```

## Ordine di esecuzione:

1. Vengono scelte le righe in base alla clausola **where**
2. queste righe vengono raggruppate in base alla clausola **group by**
3. per ciascun gruppo vengono calcolati i risultati delle **funzioni aggregate**
4. vengono scelti ed eliminati i gruppi in base alla clausola **having**
5. I gruppi vengono ordinati sulla base della clausola **order by**

# Query con Raggruppamento (2/2)

1. La condizione del where viene valutata **sulle righe** della tabella mentre quella dell'having viene valutata sui **dati aggregati (gruppi)**
2. La condizione dell'having è, in genere, costituita da condizioni semplici o complesse (subquery) che coinvolgono funzioni aggregate
  - non usare mai la clausola having senza la clausola group by perché sarebbe una condizione esprimibile nel where
  - having senza group by equivale a considerare l'**intera tabella** come un unico gruppo
3. I campi della target list (con esclusione delle funzioni aggregate) devono essere **uguali o un sottoinsieme** dei campi della clausola group by
4. Nella clausola order by oltre ai campi della tabella è possibile trovare **alias** che rappresentano i risultati di funzioni aggregate

Cosa produce un group by su chiave primaria?

Considerando l'ordine di esecuzione di una query è possibile usare espressioni o funzioni aggregate nella clausola group by?

Per il conteggio dei valori della lista di attributi:

**count ( < \* | [all|distinct] lista di attributi > )**

count(\*) - conta il numero di righe restituite per raggruppamento

Per il calcolo, rispettivamente della media, del totale, del valore massimo e minimo:

**<avg | sum | max | min> ( [all|distinct] AttrEspr )**

# Funzioni Aggregate (2/2)

- La parola chiave di default è **ALL** ossia, se non si utilizza esplicitamente **DISTINCT**, tutte le funzioni aggregate, per il calcolo del risultato, considerano anche i valori duplicati mentre i valori **NULL** vengono, in ogni caso, **scartati**
- Le funzioni **sum** e **avg** si applicano solo a **valori numerici** mentre **max** e **min** a **valori numerici, stringhe di caratteri** (si considera l'ordine alfabetico) e date
- **Count** calcolato su una tabella vuota restituisce 0 mentre tutte le altre funzioni aggregate, calcolate su un insieme vuoto, restituiscono **NULL**
- NON è possibile **concatenare** le funzioni aggregate. Per es. per il calcolo della somma delle medie la seguente espressione **sum(avg(AttrEspr))** non è corretta!!!
- Nella target list e nella clausola **order by** è possibile utilizzare più funzioni aggregate
- È possibile utilizzare gli operatori aritmetici con le funzioni aggregate e con gli attributi a cui le funzioni si applicano



# Funzioni Aggregate nella Target List (1/2)

**IMPIEGATO** (CF, nome, cognome, dipartimento, stipendio\_mensile, extra\_mensile)

Select count(\*)  
From Impiegato

Calcola il numero di tuple della tabella Impiegato  
(la tabella Impiegato è vista come un unico raggruppamento)

Select count(\*)  
From Impiegato  
Group by dipartimento

Calcola per ciascun dipartimento il numero di impiegati

Select count(stipendio)  
From Impiegato  
Group by dipartimento

Calcola per ciascun dipartimento il numero di tuple che  
hanno un valore non nullo per l'attributo Stipendio  
(vengono scartati solo i NULL)

Select count(distinct stipendio)  
From Impiegato  
Group by dipartimento

Calcola per ciascun dipartimento il numero di diversi  
valori dell'attributo Stipendio  
(vengono scartati i NULL ed i valori duplicati)

Le funzioni aggregate sono calcolate su insiemi di valori:

```
Select avg(stipendio_mensile), sum(stipendio_mensile + extra_mensile)  
From Impiegato  
Where dipartimento='DEE'
```



Restituisce rispettivamente lo stipendio medio e la spesa mensile totale relativi solo al DEE

**Espressioni equivalenti**  
mentre i risultati possono discostarsi leggermente a causa degli arrotondamenti

```
Select dipartimento, sum(stipendio_mensile) + sum(extra_mensile) as tot  
From Impiegato  
Group by dipartimento  
Order by tot
```



Restituisce, per ciascun dipartimento, il nome del dipartimento e la relativa spesa mensile totale. I risultati sono ordinati per valori della spesa totale crescenti.



# Espressioni nelle Funzioni Aggregate

CONCERTO (codice, titolo, esecutore, durata)

SALA (codice, nome, indirizzo, citta, telefono)

PROGRAMMAZIONE (data\_programmazione, cod\_concerto, ora\_inizio, N\_posti\_prenotati, cod\_sala, prezzo\_biglietto)

**Interrogazione 23:** Calcolare l'incasso totalizzato da ciascun concerto.

```
Select cod_concerto, sum (N_posti_prenotati * prezzo_biglietto) AS incasso  
From Programmazione  
Group by cod_concerto
```

## Operatore di confronto a valore singolo

IMPIEGATO (nome, cognome, dipartimento, stipendio)

### Interrogazione 24 : Risoluzione equivalente all' Interrogazione 17 :

Selezionare i dati dei dipendenti del DEE che percepiscono lo stipendio maggiore

```
Select *
From Impiegato
Where dipartimento='DEE' AND
      stipendio = ( Select max(stipendio)
                      From Impiegato
                      Where dipartimento='DEE' )
```



# Funzioni aggregate nella clausola HAVING (1/5)

<Condizione della clausola having semplice> ::= <funzione aggregata> OP valore

CORSO (codice, nome, docente)

LEZIONE (codCorso, codPeriodo, aula)

PERIODO (codice, giorno, ora\_inizio)

## Interrogazione 25:

Determinare per ciascun docente, che insegna esattamente due corsi, il numero di lezioni che tiene tra Lunedì e Martedì in aula B o in aula A.



# Funzioni aggregate nella clausola HAVING (1/5)

<Condizione della clausola having semplice> ::= <funzione aggregata> OP valore

CORSO (codice, nome, docente)

LEZIONE (codCorso, codPeriodo, aula)

PERIODO (codice, giorno, ora\_inizio)

## Interrogazione 25:

Determinare per ciascun docente, che insegna esattamente due corsi, il numero di lezioni che tiene tra Lunedì e Martedì in aula B o in aula A.

Select docente, count(\*)

From Corso, Lezione, Periodo

Where Corso.codice=codCorso

AND Periodo.codice=codPeriodo

AND ( aula='A' OR aula='B')

AND giorno BETWEEN 'Lunedì' AND 'Martedì'

AND docente IN (Select docente

From Corso

Group by docente

Having count(\*)=2)

Group by docente



# Funzioni aggregate nella clausola HAVING (2/5)

<Condizione clausola having complessa> ::= <funzione aggregata> OP <subquery a valore singolo>

PERSONA (CF, nome, cognome, età, sesso, CF\_padre)  
MATRIMONIO (CF sposo, CF sposa, data, num\_invitati)

## Interrogazione 26:

Selezionare i padri che hanno *tutti i figli* sposati dopo il 1990

```
Select CF_padre
From Persona P
Where CF IN (
    Select CF_sposo
    From Matrimonio
    Where data > '31/01/1990' )
OR CF IN (
    Select CF_sposa
    From Matrimonio
    Where data > '31/01/1990' )
Group by CF_padre
Having count(*) = (
    Select count(*)
    From Persona
    Where CF_padre = P.CF_padre )
```



# Funzioni aggregate nella clausola HAVING (3/5)

AUTO(targa, modello, marca, alimentazione)

ACCESSO(targa, id garage, data accesso, costo)

GARAGE(ID, nome, città, indirizzo, capienza, costo orario)

## Interrogazione 27:

Selezionare i garage che hanno ospitato più macchine con alimentazione diesel che benzina

Select \*

From Garage

Where ID IN ( Select id\_garage

    From Accesso AS A, Auto

    Where A.targa = Auto.targa

    AND alimentazione = 'diesel'

    Group by id\_garage

    Having count(\*) > ( Select count(\*)

        From Accesso B, Auto

        Where B.targa = auto.targa

        AND alimentazione = 'benzina'

        AND B.id\_garage = A. id\_garage ))



# Funzioni aggregate nella clausola HAVING (4/5)

Impiegato(CF,Nome, cognome, ID\_DIP\_lav)  
Dipartimento(ID\_DIP, nome, indirizzo)  
Lavora\_su(CF\_imp,ID\_Prog\_lav,ore)  
Progetto(ID\_Prog,nome\_prog, ID\_DIP\_gestore)

## Interrogazione 28:

Si determinino nome e cognome di ciascun impiegato che lavora su tutti i progetti di cui è responsabile il dipartimento “produzione”.

### Ipotesi

*Consideriamo gli impiegati che lavorano su tutti i progetti del dipartimento “produzione” e su nessun progetto di altro dipartimento*

### Risoluzione

1. Trovare tutti i progetti di cui è responsabile il dipartimento ‘produzione’
2. Trovare gli impiegati che lavorano su un progetto che non è presente nell’insieme dei progetti del passo 1
3. Trovare gli impiegati che lavorano su un numero di progetti pari all’insieme dei progetti al passo 1
4. Estrarre gli impiegati che, contemporaneamente, non fanno parte degli impiegati del passo 2 ma che fanno parte degli impiegati del passo 3



# Funzioni aggregate nella clausola HAVING (5/5)

```
Select nome, cognome
From Impiegato
Where CF NOT IN ( Select CF_imp
    From lavora_su
    Where ID_Prog_lav IN ( Select ID_prog
        From Progetto
        Where ID_dip_gestore <> ( Select ID_Dip
            From Dipartimento
            Where nome='produzione' )))
AND CF IN ( Select CF_imp
    From lavora_su
    Group by CF_imp
    Having count(*) = ( Select count(*)
        From Progetto
        Where ID_dip_gestore = ( Select ID_Dip
            From Dipartimento
            Where nome='produzione' )))
```



# Clausola GROUP BY con più attributi

FILM (Codice, titolo, durata, regista)

PROGRAMMAZIONE (Data, Codice, ora\_inizio, rete)

## Interrogazione 29:

Per ciascun film, determinare il numero di volte che è stato trasmesso su ciascuna rete

```
Select codice, rete, count(*)  
From Programmazione  
Group By codice, rete
```

<querySQL o subquerySQL> ::=

<selectSQL> <UNION | EXCEPT (o MINUS) | INTERSECT [all]> <selectSQL>

- Union: restituisce i valori di due insiemi in un unico insieme
  - Except: restituisce solo i valori del primo insieme che non sono contenuti nel secondo
  - Intersect: restituisce solo i valori contenuti sia nel primo che nel secondo insieme
- 
- Gli operatori insiemistici per default **eliminano** i duplicati, usando la parola chiave **ALL** i duplicati si conservano
  - I nomi dei campi delle due selectSQL non devono coincidere necessariamente (l'insieme restituito ha i nomi dei campi della prima selectSQL) ma devono essere dello **stesso numero e stesso tipo**
  - Importanza dell'uso delle parentesi per definire l'ordine di esecuzione della subquery con operatori insiemistici rispetto alla query esterna

CLIENTE (CF, nome, cognome, età, sesso)

SVILUPPO (ID\_rullino, CF\_cliente, data, num\_foto)

FOTO (ID\_foto, formato, colore, ID\_rullino, tipo\_carta)

## Interrogazione 30 – Appello del 9 Novembre 2004 :

Il cognome ed il nome di chi ha sviluppato solo foto in bianco e nero nel 2003.

Select nome, cognome

From Cliente

Where CF IN ( ( Select CF\_cliente

From Sviluppo

Where data >= '01/01/2003' AND data<='31/12/2003' )

EXCEPT

( Select CF\_cliente

From Sviluppo Natural Join Foto

Where data >= '01/01/2003' AND data<='31/12/2003'

AND colore='a colori' ) )

# Il predicato EXISTS

<condizione del where> ::= = [NOT] EXISTS <subquery a valori multipli>

{ [AND | OR] < condizione semplice o complessa> }

- La subquery dopo l'exists restituisce sempre l'intera tabella della clausola *from* poiché il predicato si basa sulle righe e NON sulle colonne.
- Se la subquery restituisce almeno una riga il predicato è **true**
- Se la subquery è vuota il predicato è **false**

FILM (titolo, regista, anno, genere)

HARECITATOIN (attore, film)

**Interrogazione 31:** Selezionare gli attori che hanno recitato in tutti i film diretti da Clint Eastwood

Select distinct attore

From HaRecitatoIn H

Where NOT EXISTS ( Select \*

From Film

Where regista='Clint Eastwood'

AND titolo NOT IN (Select film

From HaRecitatoIn

Where attore=H.attore))

- La sintassi delle interrogazioni SQL è ben definita ma non esiste un metodo standard per risolvere una interrogazione
- Una stessa interrogazione può essere risolta **in modi differenti**
- Generalmente le interrogazioni risolte con l'operatore di sinistra possono risolversi con il corrispondente di destra:

IN	=ANY
NOT IN	<>ALL
BETWEEN val_inf AND val_sup	>=val_inf AND >=val_sup
EXCEPT	NOT IN
EXISTS (per le righe)	IN (per le colonne)

**Insert into NomeTabella [ ListaAttributi ] values (Lista di valori) | selectSQL**

IMPIEGATO (matricola, nome, cognome, età, salario)

LAVORA (matricola, codice, percentuale\_tempo)

DIPARTIMENTO (codice, nome, budget, matr\_direttore, matr\_vice\_direttore)

## Inserimento con lista di valori:

Insert into Impiegato values ('511AB', 'Mario', 'Rossi', 21, 3000)

Se fossero definiti dei valori di default per alcuni campi?

## Inserimento con selectSQL:

### Interrogazione 32:

Inserire tutti gli impiegati tra 25 e 30 anni nel dipartimento 'vendite' al 20% del loro tempo

Insert into Lavora (matricola, codice, percentuale\_tempo)

Select matricola, (Select codice From Dipartimento Where nome='Vendite'), 20

From Impiegato

Where età BETWEEN 25 AND 30



# Delete

**Delete From NomeTabella [Where condizione]**

IMPIEGATO (nome, cognome, dipartimento, ufficio, stipendio, extra)

DIPARTIMENTO (nome, indirizzo, città, tel)

**Delete senza clausola where:**

Delete From Impiegato (svuota completamente la tabella)

**Delete con clausola where semplice:**

Delete From Impiegato **Where** dipartimento='DEE'

**Delete con clausola where complessa:**

Delete From Impiegato

Where dipartimento **IN** (**Select** nome

From Dipartimento

**Where** città='Firenze')



# Update

**Update NomeTabella set attributo = < espressione | selectSQL | null | default >**  
{ , attributo = < espressione | selectSQL | null | default >}  
[ **Where** condizione ]

## Aggiornamento della chiave primaria:

**Update** Dipartimento **set** nome='D.E.E' **Where** nome='DEE'

## Aggiornamento mediante il risultato di un'espressione:

**Update** Impiegato **set** stipendio=stipendio\*1.1, **extra=null**  
**Where** dipartimento='DEE' **AND** stipendio>3000

## Aggiornamento mediante il risultato di una subquery a valore singolo:

**Update** Impiegato **set** stipendio= ( **Select** stipendio  
                 **From** Impiegato  
                 **Where** nome='Ugo' **AND** cognome='Rossi')  
**Where** dipartimento='DEE'



# Feedback

- <https://forms.gle/oKcQxFDsYYWsggxaA>



## Asserzioni, Viste e Trigger Basi di dati Attive



- Introdotte in SQL-2 rappresentano dei **vincoli** che non sono però associati a nessun attributo o tabella in particolare, ma appartengono direttamente allo schema.
- Permettono di esprimere tutti i vincoli d'integrità definiti nella definizione della tabella
  - vincoli di tupla
  - vincoli di tabella
  - vincoli su più tabelle
  - vincoli che richiedono una cardinalità minima su una tabella

- Le asserzioni possiedono un **nome**, tramite cui possono essere successivamente eliminate esplicitamente dallo schema. La sintassi per la loro definizione è la seguente:

```
create assertion NomeAsserzione check(condizione)
```

- Ad ogni vincolo di integrità (check o assertion) è associata una **politica di controllo**:

- i vincoli **immediati** sono verificati immediatamente dopo ogni modifica della base di dati (es. primary key, unique, not null, foreign key)  
se il vincolo non è soddisfatto viene eseguito un **rollback parziale**
- i vincoli **differiti** sono verificati solo al termine della transazione (serie di operazioni)  
se il vincolo non è soddisfatto viene “disfatta” l’intera sequenza di operazioni (**rollback**)

```
set constraint NomeVincolo immediate|deferred
```



# Vincolo a livello di TUPLA

**Esempio** – [...] può essere richiesto del “personale speciale” classificato come babysitter, clown, marionette e mimo. Ciascuna figura è caratterizzata da codice fiscale, nome e cognome [...]

```
Create table personale_speciale
( CF  char(16) primary key,
  nome  varchar(20) not null,
  ...
  ...
  tipo varchar(11) not null check(tipo IN
    ('babysitter','clown','marionette','mimo')) ,
  ...
  ...
  )
```

**Esempio** – [...] si verifichi che ciascun dipartimento abbia al massimo 5 professori associati [...]

```
Create table dipartimento
( codice  char(10) primary key,
  nome   varchar(20) not null,
  indirizzo  varchar(100) not null,
  ...
  ...
  check( 5 >= (Select count(*)
                  From Impiegato I
                 Where I.dipartimento = codice
                   AND I.ruolo = 'Prof. associato'))  
)
```

**Esempio** – [...] si verifichi che la quantità di un ordine non superi la quantità disponibile in magazzino per il prodotto ordinato[...]

PRODOTTO(codProd, qtDisp) - ORDINE(cliente, codProd, data, qt)

Create schema Magazzino authorization SisInfLAB

Create table Prodotto ( ... )

Create table Ordine ( ... )

Create assertion gestioneQta

check (

    NOT EXISTS ( Select \*

        From Prodotto P

        Where qtDisp < ( Select sum(qt)

            From Ordine

            Where codProd=P.codProd ))

)

**EXISTS** è un operatore che restituisce vero se la query SQL ha almeno una tupla

**Vista:** tabella **virtuale** il cui contenuto è definito a partire da altre tabelle (tabelle **base**) o viste nello schema, ma **non ricorsive**.

In pratica è una relazione non costituita da tuple, ma da una definizione.

Gli attributi nella vista devono essere in corrispondenza **1 a 1** con le colonne prodotte dalla query, altrimenti la vista li **eredita** direttamente dalla query.

Una vista è una query con un nome eseguita dinamicamente ma, a differenza di una semplice query, sono possibili **operazioni di modifica** come per le tabelle.

## Utilizzo:

- offrire **visioni diverse** degli stessi dati
- rendere più **semplici** alcune interrogazioni
- rendere **possibili** alcune interrogazioni

```
Create view NomeVista [(Lista di Attributi)] as  
selectSQL [with [cascaded | local] check option]
```



# Viste Aggiornabili

**Proprietà:** le modifiche si **propagano** dalla vista alla tabella base (lo standard SQL richiede che sia possibile determinare un **modo univoco** in cui la modifica possa propagarsi)

- SQL92 consente l'update solo per viste determinate a partire da **tabelle singole senza funzioni aggregate**. Ogni tupla della vista mappa una tupla della relazione di partenza.
- L'opzione **with check option** è necessaria quando si prevede l'update di una vista. Le tuple risultanti dovranno ancora **appartenere** alla vista (l'update non è in contrasto con i predicati di selezione).

```
Create view NomeVista [(Lista di Attributi)] as  
selectSQL [with [cascaded | local] check option]
```

- Per viste ottenute da altre viste, **local** e **cascaded** specificano, rispettivamente, se il controllo sarà effettuato solo al livello della vista che si sta definendo (local) o deve propagarsi (cascaded).
- Il default è cascaded

```
Create view NomeVista [(Lista di Attributi)] as  
selectSQL [with [cascaded | local] check option]
```

Sintassi della selectSQL affinchè la vista sia sicuramente aggiornabile:

SELECT senza DISTINCT e funzioni aggregate  
FROM singola tabella (senza join)  
WHERE senza subquery  
GROUP BY ed HAVING non presenti



# Vista senza "check option"

IMPIEGATO (codice, nome, cognome, stipendio, qualifica)

con qualifica IN ('dipendente', 'direttore', 'supervisore')

```
CREATE VIEW Supervisori
```

```
AS SELECT *
```

```
FROM Impiegato
```

```
WHERE qualifica = 'supervisore'
```

Vista senza clausola  
with check option

IMPIEGATO (TABELLA BASE)

codice	nome	cognome	stipendio	qualifica
000	Mario	Rossi	1000	dipendente
001	Giuseppe	Verdi	1500	supervisore
002	Claudio	Bianchi	3000	direttore

SUPERVISORI (VISTA)

codice	nome	cognome	stipendio	qualifica
001	Giuseppe	Verdi	1500	supervisore

# Vista senza “check option”

Dopo la creazione della vista eseguiamo il seguente comando:  
 UPDATE Supervisori SET qualifica = ‘direttore’

**SUPERVISORI (VISTA)**

codice	nome	cognome	stipendio	qualifica
001	Giuseppe	Verdi	1500	supervisore

direttore

**IMPIEGATO (TABELLA BASE)**

codice	nome	cognome	stipendio	qualifica
000	Mario	Rossi	1000	dipendente
001	Giuseppe	Verdi	1500	direttore
002	Claudio	Bianchi	3000	direttore



Se ora eseguiamo il seguente comando quale sarà il risultato?

SELECT \* FROM Supervisori

La vista è vuota!!!!



```
CREATE VIEW Supervisori
```

```
AS SELECT *
```

```
FROM Impiegato
```

```
WHERE qualifica = 'supervisor'
```

```
WITH CHECK OPTION
```



Vista con la clausola  
with check option

Dopo la creazione della vista eseguiamo il seguente comando:

```
UPDATE Supervisori SET qualifica = 'direttore'
```

- Il comando non viene eseguito questa volta poiché la vista deve essere aggiornabile quindi non deve perdere tuple
- La clausola WHERE della vista è in contrasto con l'aggiornamento del campo tipo richiesto nell'update ed impedisce l'aggiornamento
- Poiché per default la clausola check option è cascaded, il rispetto della clausola WHERE viene verificato per tutti gli “oggetti” che fanno riferimento alla vista

**DROP VIEW NomeVista [restrict|cascade]**

- **Restrict**: è l'opzione di default e specifica che il comando viene eseguito solo se la vista non è utilizzata nella definizione di altre tabelle o viste
- **Cascade**: specifica che eliminando una vista che compare nella definizione di altre viste, anche queste vengono rimosse

**DROP ASSERTION NomeAsserzione [restrict|cascade]**

- **Restrict**: è l'opzione di default e specifica che il comando viene eseguito solo se l'asserzione non è più "utilizzata"
- **Cascade**: specifica che l'asserzione possa sempre essere cancellata

IMPIEGATO (codice, nome, cognome, codDipart, ufficio, stipendio\_mensile)

DIPARTIMENTO (codice, nome, indirizzo, città, tel, ateneo)

**Query:** Trovare il numero medio di impiegati dei dipartimenti del 'Politecnico di Bari'

```
Select avg(count(*))
From Impiegato
Where codDipart IN ( Select codice
                      From Dipartimento
                      Where ateneo='Politecnico di Bari')
Group by codDipart
```

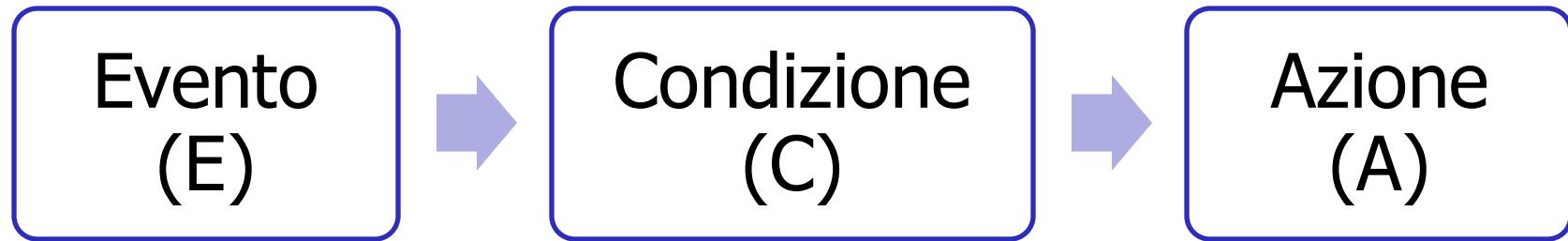


```
Create view NumImpiegatiDip (dipartimento, num_imp)
As Select codDipart, count(*)
      From Impiegato
      Where codDipart IN (Select codice
                           From Dipartimento
                           Where ateneo='Politecnico di Bari')
      Group by codDipart
```



```
Select avg(num_imp)
From NumImpiegatiDip
```

**Definizione:** Una base di dati si dice **attiva** quando dispone di un sottosistema integrato per definire e gestire regole di produzione (**regole attive**) che seguono il paradigma ECA:



Allo stato attuale molte basi di dati relazionali sia di tipo **commerciale** (Oracle, MS SQL server) che **open source** (PostgreSQL, MySQL) possono essere considerate basi di dati attive poiché mettono a disposizione semplici regole chiamate **trigger**

- **Evento:** primitiva per la manipolazione dei dati in SQL (DML) di tipo **insert, update o delete**
- **Condizione:** predicato booleano espresso in SQL (stessi operatori utilizzati per definire una condizione semplice o complessa della clausola WHERE di una selectSQL)
- **Azione:** sequenza di **statement** SQL (select, insert, update e delete) arricchita dai costrutti forniti da linguaggi di programmazione **proprietari** (PL/sql in Oracle, PL/pgsql in PostgreSQL)

Una regola attiva è definita su una sola tabella chiamata **target**. La regola è **attivata** a seguito di un evento su tale tabella, se la condizione è **verificata** allora viene **eseguita** l'azione.

## ■ Comportamento reattivo

- mediante l'attivazione di regole attive la base di dati è in grado di **reagire** autonomamente agli eventi, in generale alle modifiche delle istanze del DB

## ■ Processore delle regole (rule engine)

- **cattura** gli eventi
- **esegue** le regole attive
- determina un **alternarsi** tra l'esecuzione delle transazioni, lanciate dagli utenti, e quella delle regole, lanciate dal sistema

## ■ Indipendenza della conoscenza

- azioni sottratte ai programmi applicativi vengono codificate in regole attive
- tramite il DDL tali regole fanno **parte dello schema**
- le regole possono essere **condivise** da tutte le applicazioni che utilizzano la base di dati attiva
- le regole **non** devono essere **replicate** negli applicativi stessi
- modifiche alle regole non comportano modifiche nelle applicazioni

## ■ Gestione interna (DBMS)

- gestire **vincoli di integrità** predefiniti (per es. una politica di reazione di tipo cascade per una foreign key può essere implementata con una regola attiva)
- calcolare **attributi derivati**
- gestire **dati duplicati**
- gestire le **eccezioni** (per es. sollevate dalla violazione dei vincoli di integrità)

## ■ Gestione esterna

- codificare complesse **regole aziendali** (business rules) non rappresentabili in altro modo nello schema (per es. mediante check o assertion)
- non esistono schemi fissi per la codifica delle regole e ciascun problema applicativo va affrontato singolarmente

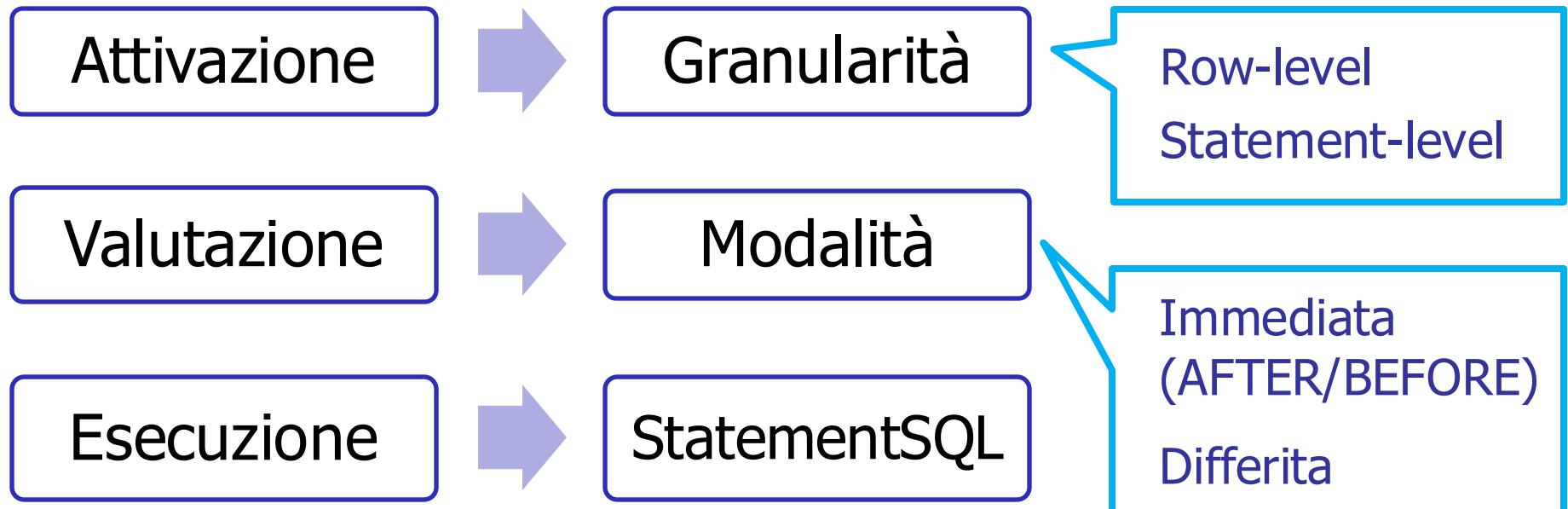


La sintassi dei trigger non è definita in SQL-92, quindi la loro definizione e gestione varia a seconda dello specifico DBMS utilizzato

Sintassi generale per la creazione di un trigger:

```
CREATE TRIGGER NomeTrigger
  modalità evento {, evento}
  on TabellaTarget
  [referencing referenza]
  [granularità]
  [when (condizione)]
  StatementsQL
```

Un trigger in quanto regola attiva è caratterizzato dalle seguenti fasi: **Attivazione**, **Valutazione** ed **Esecuzione**. Per ciascuna fase è possibile specificare proprietà diverse.



**StatementSQL**: può contenere la modifica delle tuple di una tabella su cui è definito un secondo trigger che a seguito della modifica viene attivato.  
L'azione di un trigger può quindi anche innescare un evento che attiva altri trigger (*trigger in cascata*)

Modalità **after**: la valutazione avviene **immediatamente dopo** l'evento (caso più frequente)

Modalità **before**: la valutazione del trigger **precede logicamente** l'evento a cui si riferisce

Modalità **differita**: la valutazione avviene **alla fine** della transazione, a seguito del **commit-work**

**CREATE TRIGGER** *NomeTrigger*

*AFTER|BEFORE* *evento* {, *evento*}

**on** *TabellaTarget* {*INITIALLY IMMEDIATE|INITIALLY DEFERRED*}

[**referencing** *referenza*]

[**granularità**]

[**when** (*condizione*)]

*StatementSQL*

I **trigger AFTER evento** vengono utilizzati:

- in applicazioni di audit, per **aggiornare** le tuple di una tabella B a seguito di una **modifica** nella tabella target A e, se esiste una clausola WHEN, solo se i nuovi dati verificano la condizione
  - ! la modalità after è necessaria poiché la modifica riuscita di una riga implica il superamento dei vincoli di integrità referenziale definiti per la tabella target e **solo in questo caso il trigger viene attivato**

- nel calcolo dei **dati derivati** (es., aggiornare la quantità di prodotto disponibile)
- nella gestione delle **politiche di reazione** dei vincoli di integrità referenziale

I **trigger BEFORE evento** vengono utilizzati:

- se si deve **impostare** il valore di una colonna in una riga inserita mediante un trigger, ossia se è necessario accedere ai valori "nuovi" e "vecchi" per poterli prima verificare
  - ! l'uso di un trigger AFTER INSERT non consentirebbe di impostare il valore inserito, in quanto la riga sarebbe già stata **inserita** nella tabella

- nella verifica di dati e chiavi **duplicati**
- nella gestione delle **eccezioni** impedendo, in caso di errore, l'esecuzione dell'evento che ha attivato il BEFORE trigger

- **row-level** (trigger a **livello di riga**), clausola **for each row**
  - il trigger viene attivato, verificato ed eseguito **per ogni tupla** della tabella target coinvolta dall'evento (comportamento orientato alle singole stanze)
- **statement-level** (trigger a **livello di primitiva**), clausola **for each statement**
  - il trigger viene attivato, verificato ed eseguito **una sola volta** per tutte le tuple della tabella target (comportamento orientato agli insiemi). E' il valore di default.

```
CREATE TRIGGER NomeTrigger
  modalità evento {, evento}
  on TabellaTarget
  [referencing referenza]
  [for each row | for each statement]
  [when (condizione)]
  StatementSQL
```



# Trigger: Granularità e Referenza

Per la definizione di un trigger è possibile fare riferimento ai valori “vecchi” e “nuovi” utilizzando due variabili predefinite:

**NEW**: rappresenta la **nuova tupla** (esiste quando l'evento è *insert* oppure *update*)

**OLD**: rappresenta la **vecchia tupla** (esiste quando l'evento è *delete* oppure *update*)

Per accedere ad un campo specifico di una tupla si usa la *dot notation*

	<b>NEW</b>	<b>OLD</b>
<b>INSERT</b>	Esiste	
<b>UPDATE</b>	Esiste	Esiste
<b>DELETE</b>		Esiste

La clausola **referencing** permette invece di rinominare tali variabili. Ad esempio è possibile scrivere:

**on TabellaTarget**

**referencing NEW AS NuovaVar, OLD AS VecchiaVar**

**for each row**

- Le **variabili NEW ed OLD** sono disponibili, per indicare la tupla nello stato precedente e successivo all'evento, **solo a livello di riga**
- La condizione del **WHEN** è valutata generalmente sui **valori assunti dalle tuple** della tabella target
- Se non è possibile utilizzare le variabili NEW e OLD, non è possibile esprimere neanche la condizione e pertanto essa viene solitamente implementata **solo a livello di riga**
- Segue che le clausole referencing e when rimangono sempre **opzionali**, ma possono essere utilizzate solo per trigger con granularità a livello di riga

I **trigger FOR EACH STATEMENT** vengono utilizzati:

- se l'azione del trigger deve essere **sempre** eseguita al verificarsi dell'evento su un'intera tabella

I **trigger FOR EACH ROW** vengono utilizzati:

- se l'attivazione, la valutazione o l'esecuzione di un trigger richiedono la **conoscenza dello stato** precedente e/o successivo all'evento

**Ordine di esecuzione:**

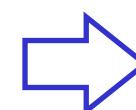
- BEFORE Statement-level
- BEFORE Row-level
- AFTER Row-level
- AFTER Statement-level



Si compone di due parti: parte **dichiarativa** e parte **esecutiva**.

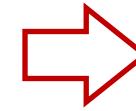
La sintassi varia a seconda del linguaggio procedurale utilizzato.

**DECLARE**  
parte dichiarativa  
(opzionale)



dichiarazione di variabili utilizzando i tipi definiti da SQL-92

**BEGIN**  
parte esecutiva  
(NO DDL)  
**END;**



sequenza di **istruzioni SQL** (select, insert, update, delete) arricchite con strutture tipiche dei linguaggi procedurali quali **IF [...] THEN [...] ELSE [...]**, cicli **FOR** e **WHILE**

Creare una regola che **riduca del 10%** lo stipendio di tutti gli impiegati quando la **media** dei salari **superà** i 5000 euro

## Come procedere:

1. L'aumento della media può avvenire a causa di un **inserimento, cancellazione o aggiornamento** nella tabella Impiegato. Gli eventi che attivano il trigger sono **insert, delete e update**. La **tabella target** è Impiegato
2. La regola va attivata **subito dopo** che è avvenuto l'evento che può aver aumentato la media. L'attivazione è quindi **after e immediata**
3. L'aggiornamento dello stipendio va fatto su **tutte le tuple** ed è attivato sulla base di una condizione (la media dei salari) che coinvolge tutte le tuple della tabella target (comportamento orientato agli insiemi). La granularità è a livello di **primitiva**
4. Con una granularità a livello di primitiva la clausola **when non esiste** quindi il valore della media va calcolato nello statementSQL come condizione della clausola **where**

# Esempio: Regola aziendale

**IMPIEGATO** (matricola, nome, cognome, stipendio)

Modalità

CREATE TRIGGER ControlloStipendio

Tabella  
target

AFTER insert, delete, update

Parte  
esecutiva

ON Impiegato

BEGIN

Update Impiegato

set stipendio = 0,9\*stipendio

where 5000 < ( SELECT avg(stipendio)  
FROM Impiegato )

END;

Eventi che  
attivano il trigger

Condizione



# Esempio: Integrità referenziale

**DIPARTIMENTO**(codDip, nome, sede, num\_dip, tel)

**IMPIEGATO**(codice, nome, cognome, dipartimento)

Creare una regola che reagisca alla cancellazione di un dipartimento ponendo a NULL il valore del campo “dipartimento” nella tabella Impiegato (vincolo ON DELETE SET NULL).

```
CREATE TRIGGER CancellaDipart
AFTER delete
ON Dipartimento
FOR EACH ROW
WHEN (exists (SELECT *
               FROM Impiegato
               WHERE dipartimento = OLD.codDip))
BEGIN
    Update Impiegato
    set dipartimento = NULL
    where dipartimento = OLD.codDip
END;
```

# Esempio: Integrità referenziale

## Evento che attiva il trigger:

```
DELETE FROM Dipartimento WHERE nome = 'DMMM'
```

### DIPARTIMENTO

codDip	nome	sede	num_dip	tel
D000	DMMM	Via Orabona 4	1	0805xxxxx
D001	DEI	Via Orabona 4	1	0805xxxxx

### Variabile OLD

D000	DMMM	Via Orabona 4	1	0805xxxxx
------	------	---------------	---	-----------

```
SET dipartimento = NULL
WHERE dipartimento = OLD.codDip
```

### IMPIEGATO

codice	nome	cognome	dipartimento
IMP000	Mario	Rossi	D000
IMP001	Carlo	Verdi	D001

**ORDINE**(Prod, Forn, Data, qt)

**TOTALE**(Prod, Forn, qtTot)

Creare una regola che ad ogni nuovo ordine aggiorni la quantità totale ordinata.

**CREATE TRIGGER** AggiornaTot

**AFTER insert**

**ON Ordine**

**FOR EACH ROW**

**BEGIN**

    Update Totale

        set qtTot = qtTot + NEW.qt

        where Prod = NEW.Prod AND Forn = NEW.Forn

**END;**

# Esempio: Dati derivati

## Evento che attiva il trigger:

```
INSERT INTO Ordine VALUES ('P000', 'F000', '28/11/2013', 10)
```

### ORDINE

Prod	Forn	Data	qt
P000	F000	10/11/2013	50
P000	F000	28/11/2013	10

### Variabile NEW

P000	F000	28/11/2013	10
------	------	------------	----

### TOTALE

```
SET qtTot = qtTot + NEW.qt
WHERE Prod=NEW.Prod AND Forn=NEW.Forn
```

Prod	Forn	qtTot
P000	F000	50
P001	F000	0

$50 + 10 = 60$

# Esempio: Regola aziendale

**LIBRO**(ISBN, titolo, editore, categoria, valutazione)

**GIUDIZIO** (ISBN, timestamp, nuova\_valutazione, vecchia\_valutazione)

Creare una regola che memorizzi necessariamente tutte le modifiche della valutazione di un libro nella tabella di audit ma solo nel caso si verifichi una riduzione del valore della valutazione.

```
CREATE TRIGGER AggiornaValutazione
BEFORE update
ON Libro
FOR EACH ROW
WHEN (NEW.valutazione < OLD.valutazione)
BEGIN
    Insert into Giudizio (ISBN, timestamp, nuova_valutazione,
                          vecchia_valutazione)
    Values(OLD.ISBN, current_date, NEW.valutazione, OLD.valutazione)
END;
```

La **transazione** eseguita sulla tabella BIBLIOTECA dipende dal **successo** dell'esecuzione del trigger

# Esempio: Regola aziendale

## Evento che attiva il trigger:

```
UPDATE Libro SET valutazione = 6 WHERE ISBN = 'L001'
```

### LIBRO

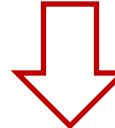
ISBN	Titolo	Editore	Categoria	Valutazione
L000	Il codice da Vinci	Mondadori	Romanzo	7
L001	Inferno	Mondadori	Romanzo	8

### Variabile OLD

L001	Inferno	Mondadori	Romanzo	8
------	---------	-----------	---------	---

### Variabile NEW

L001	Inferno	Mondadori	Romanzo	6
------	---------	-----------	---------	---



```
INSERT INTO Giudizio VALUES ('L001', current_time,
    NEW.valutazione, OLD.valutazione)
```

### GIUDIZIO

ISBN	Timestamp	Nuova_valutazione	Vecchia_valutazione
L001	2013-11-28 13:30:21	6	8

# Basi di Dati e Sistemi Informativi



## Le Transazioni



## Gestore delle interrogazioni

- Decide le strategie di accesso ai dati per rispondere alle interrogazioni

## Gestore dei metodi di accesso

- Esegue l'accesso fisico ai dati secondo la strategia definita dal gestore delle interrogazioni

## Gestore del buffer

- Gestisce il trasferimento delle pagine del DB dalla memoria di massa a quella centrale

## Gestore della memoria secondaria

- Controllore dell'affidabilità
- Controllore della concorrenza

## Memoria Secondaria

# Definizione di transazione

- Una unità elementare di lavoro effettuata da una applicazione dotata di specifiche caratteristiche in termini di correttezza, robustezza ed isolamento.
- Ciascuna transazione si considera encapsulata tra i due comandi:
  - Begin of Transaction (**bot**)
  - End of Transaction (**eot**)
- Uno (ed uno solo) dei due seguenti comandi viene eseguito –una volta- all'interno di una transazione:
  - Commit work (**commit**)
  - Roll Back work (**abort**)
- Sistema Transazionale: sistema che permette di definire ed eseguire transazioni

- Una transazione che inizia con `bot`, termina con `eot`, durante la propria esecuzione esegue uno ed uno solo tra `commit` e `abort` e dopo l'esecuzione di uno di questi non effettua alcuna altra operazione sui dati
- Proprieta ACID delle transazioni:
  - Atomicità
  - Consistenza
  - Isolamento
  - Durabilità (persistenza)



# Atomicità

- Una transazione è una unità atomica (**indivisibile**) di lavoro
- La sua esecuzione **non può** lasciare la base di dati in una situazione intermedia **indecidibile**
- Un errore (o un guasto) prima del commit provocheranno **l'UNDO** (il disfacimento) delle operazioni intermedie eventualmente effettuate → la base di dati si troverà nello stato **precedente** la transazione
- Un errore (o un guasto) dopo il commit provocheranno il **REDO** (il ri-fare) delle operazioni intermedie effettuate al fine di garantire che lo stato della base di dati rispetti gli effetti dell'esecuzione della transazione



# Consistenza

- Una transazione non può violare i vincoli di integrità della base di dati.
- Quando inizia una transazione il database si trova in uno stato **consistente** e quando la transazione termina il database deve trovarsi in un altro stato consistente
- Non devono verificarsi contraddizioni (inconsistenza) tra i dati archiviati nel DB



# Isolamento

- Ogni transazione è eseguita in maniera **indipendente** da qualsiasi altra
- In presenza di transazioni concorrenti, l'effetto di una collezione di transazioni è perfettamente equivalente a quello determinato da una arbitraria esecuzione in sequenza di ciascuna transazione
- L'eventuale fallimento di una transazione non deve interferire con le altre transazioni in esecuzione

- Gli effetti determinati dall'esecuzione di una transazione andata a buon fine su una base di dati permangono **indefinitamente nel tempo**
- Se una transazione richiede un **commit work**, i cambiamenti apportati non dovranno essere persi
- Per evitare che tra il termine della transazione e l'effettiva scrittura dei dati si verifichino perdite di informazioni dovuti a malfunzionamenti, vengono tenuti dei **registri di log** dove annotare tutte le operazioni sul DB
  - **Log:** file sequenziali gestiti dal sistema e scritti su memoria **stabile**

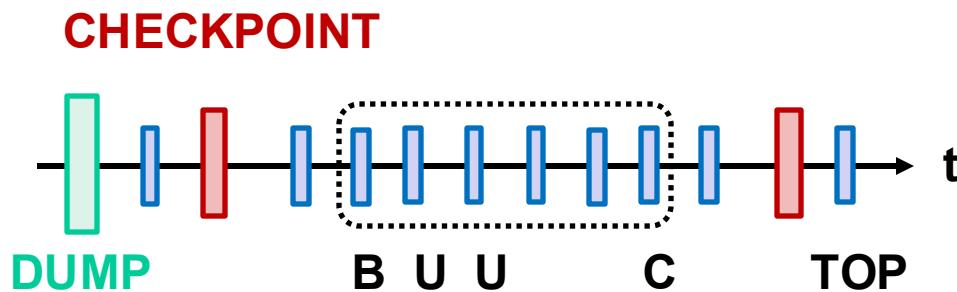


# Metodi per l'ottenimento delle proprietà ACID

- L'atomicità e la durabilità vengono garantite dal **Controllore dell'Affidabilità**
- L'isolamento viene garantito dal **Controllore della Concorrenza**
- La consistenza è garantita dal compilatore del Data Definition Language nel DBMS che si occupa di garantire il rispetto dei vincoli previsti sui dati

- Il controllore di affidabilità garantisce che:
  - le transazioni non vengano lasciate incomplete
  - gli effetti in caso di commit siano permanenti
  - sia possibile ripristinare lo stato del sistema in caso di guasti
- Realizza i comandi transazionali
  - Begin Transaction
  - Commit Work
  - Rollback Work
- Realizza le primitive per i ripristini
  - Ripresa a caldo
  - Ripresa a freddo
- Si occupa della scrittura dei file di log

- Informazioni ridondanti per la ricostruzione del DB a seguito di guasti
- Sequenza di operazioni svolte sul DB dalle transazioni, scritte in modo ordinato (temporalmente) e **sequenziale**
- L'ultimo blocco inserito nel log è chiamato **top** (blocco corrente)
- Record di log
  - Log di **transazione**: registrano le operazioni effettuate dalle transazioni sulla base di dati
    - BEGIN (B), COMMIT (C), ABORT (A) → ID Transazione
    - UPDATE (U) → ID Transazione, ID Oggetto, BS, AS
    - INSERT (I) → ID Transazione, ID Oggetto, AS
    - DELETE (D) → ID Transazione, ID Oggetto, BS
  - Log di **sistema**: registrano operazioni eseguite dal sistema, in questo caso dal controllore di affidabilità
    - CHECKPOINT, DUMP



- Undo: disfare un'azione su un oggetto O
- Redo: rifare un'azione su un oggetto O
- Proprietà di **idempotenza**
  - $\text{UNDO}(\text{UNDO}(A)) = \text{UNDO}(A)$
  - $\text{REDO}(\text{REDO}(A)) = \text{REDO}(A)$

Utili in caso di errori  
durante le operazioni di  
ripristino



# Log di sistema: Checkpoint

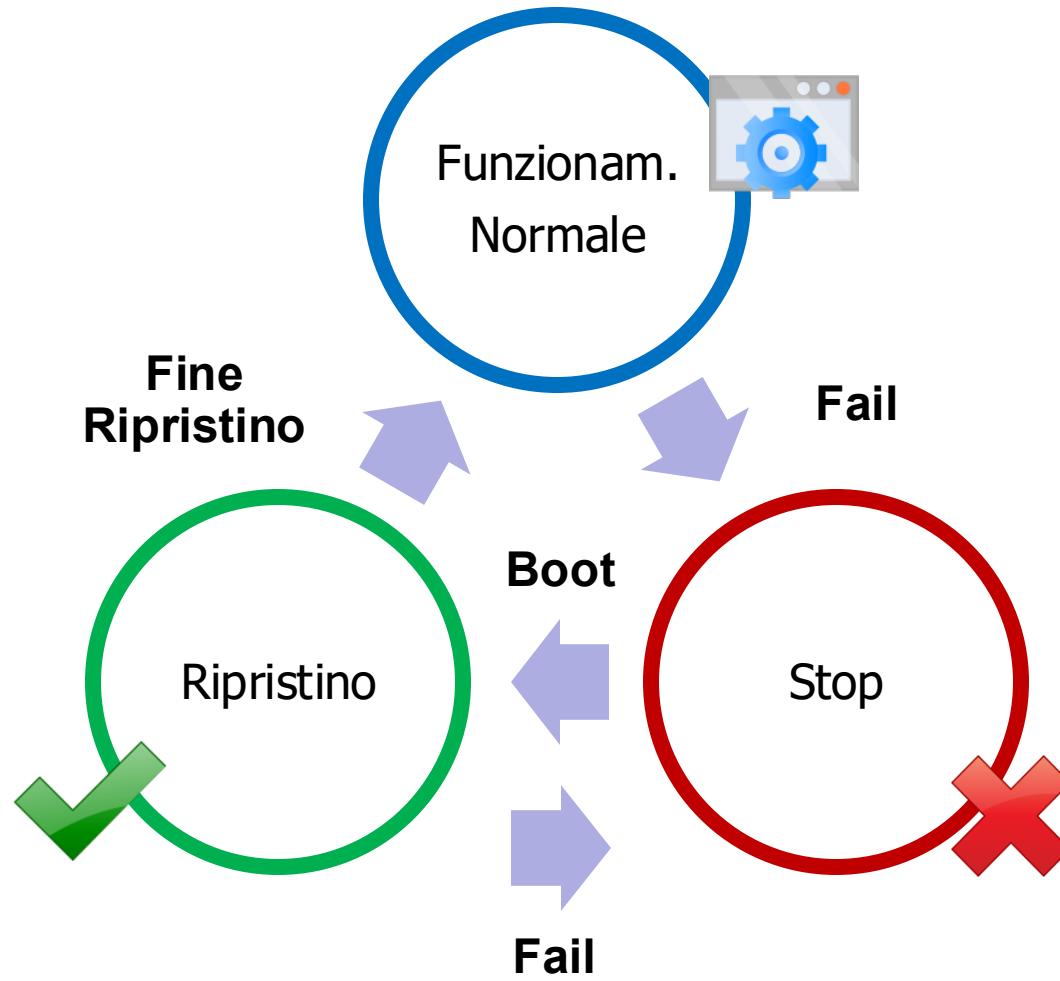
- Operazione svolta **periodicamente** dal gestore di affidabilità per registrare lo stato **attuale** delle transizioni **attive**  
 $\text{CHECK}(T_1, T_2, \dots, T_N)$
- Velocizzare e semplificare le operazioni di ripristino a seguito di un guasto
- Step da eseguire:
  - sospendere tutte le operazioni di scrittura, commit e abort di tutte le transazioni
  - trasferire in memoria di massa le pagine modificate da transazione che hanno già fatto il commit
  - scrivere in modo sincrono (invocando il comando FORCE) nel log il record di checkpoint
  - riprendere le transazioni sospese

- Copia completa e consistente (backup) dell'intero DB su supporti considerati stabili
  - DUMP()
- Effettuata solo quando il sistema è inattivo (nel fine settimana o di notte)
- Successivamente viene scritto in modo sincrono nel log il record corrispondente

- Guasto di sistema (soft failure)
  - perdita di dati in memoria centrale dovuta a un problema software (bug, crash del sistema, ...) oppure da interruzione di dispositivi hardware (ad esempio cali di tensione)
  - portano la base di dati in uno stato **inconsistente** (perdita dati buffer), mantenendo però validi i dati in memoria di massa (DB)
- Guasto di dispositivo (hard failure)
  - perdita di dati in memoria di massa e in memoria centrale
  - l'analisi del log (scritto su memoria stabile) permette di ricostruire la base di dati in memoria di massa prima del riavvio del servizio DBMS

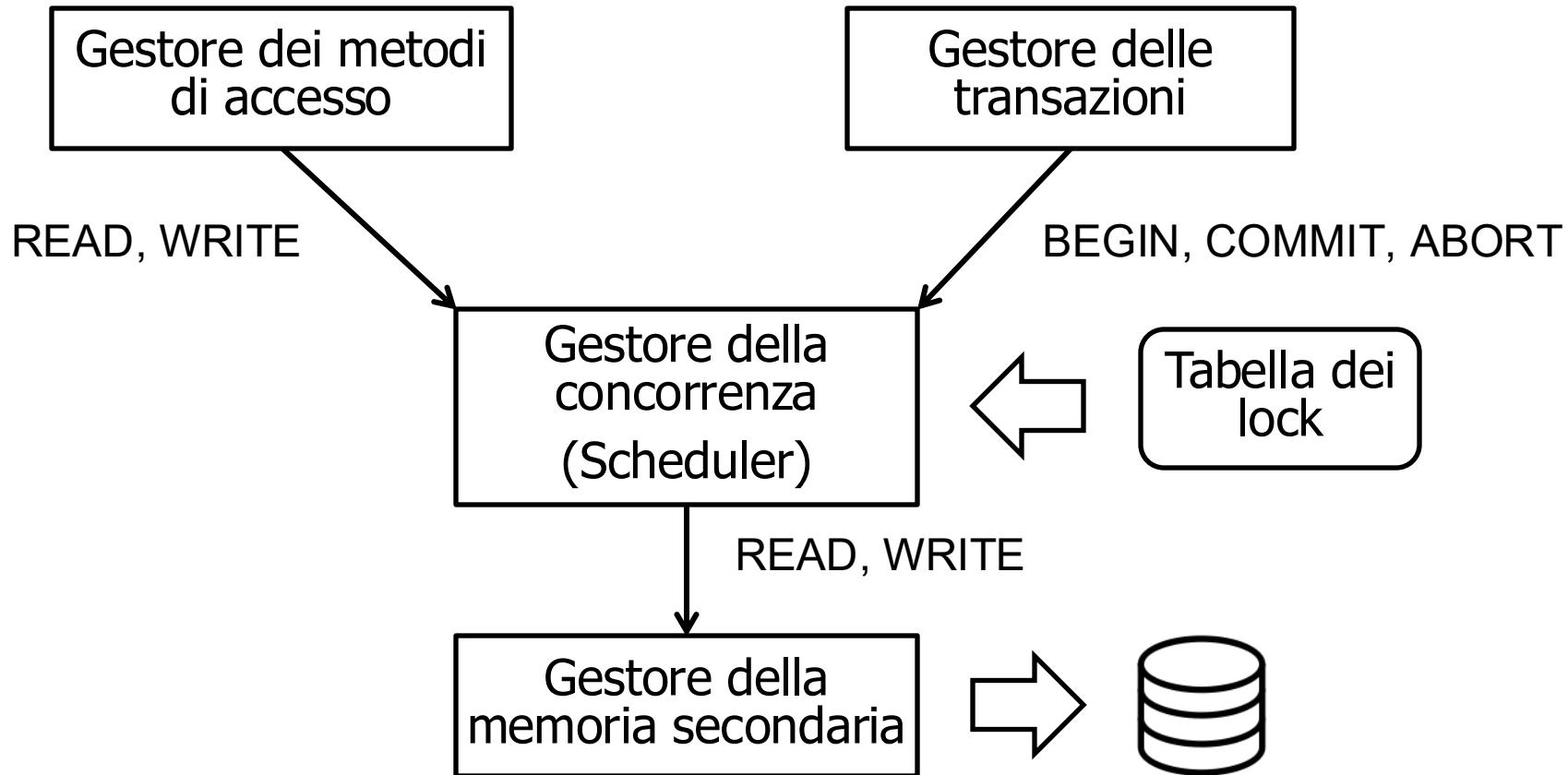
N.B.: Si suppone che il file di log sia sempre disponibile in una memoria stabile quindi la perdita del suo contenuto è considerato un evento “catastrofico” e irrimediabile

# Modello Fail-Stop



- Avviene a seguito di un guasto di sistema
- E' articolata in quattro fasi:
  1. Checkpoint-search: si cerca **l'ultimo checkpoint** inserito nel log ripercorrendolo all'indietro a partire dal blocco top (ultimo prima del guasto)
  2. Si costruiscono due insiemi: **REDO-set** e **UNDO-set**, i quali contengono le transazioni che hanno già superato il commit e quelle che non lo hanno superato. Queste ultime saranno da annullare, mentre le prime da rifare
  3. Si applicano le azioni per l'**UNDO** delle transazioni nell'omonimo set, scorrendo il log all'indietro  
Si noti che il log può essere analizzato anche prima del checkpoint nel caso la transazione attiva "più vecchia" sia iniziata prima del checkpoint
  4. Si applicano le azioni per il **REDO** delle transazioni da rifare

- Avviene a seguito di un guasto di dispositivo
- E' articolata in tre fasi:
  1. Accesso al dump: si accede all'ultimo dump della base di dati e si **ricopia** la parte dei dati danneggiati
  2. Correzione dei dati: si ripercorre in avanti il log, applicando per la parte danneggiata le operazioni effettuate  
Si **ricostruisce** lo stato dei dati, riportandoli all'ultimo valore antecedente il guasto
  3. Si esegue una **ripresa a caldo**



- Per motivi di efficienza non può il DBMS eseguire una transazione alla volta in modo seriale
- Eseguire transazioni in modo **seriale equivalente**, cioè con gli stessi effetti che avrebbero se fossero eseguite in modo seriale
- Più transazioni corrette eseguite e minor tempo di risposta
- L'esecuzione in concorrenza può portare a diverse **anomalie**:
  - Perdita di aggiornamento (lost update)
  - Lettura sporca (dirty read)
  - Lettura inconsistente (unrepeatable read)
  - Aggiornamento fantasma (phantom update)
  - Inserimento fantasma (phantom insert)

- Perdita di aggiornamento (lost update)
  - una transazione **scrive dopo la lettura** da parte di un'altra transazione e quest'ultima utilizza ancora il dato in questione

$$R1(x) \rightarrow R2(x) \rightarrow W2(x) \rightarrow W1(x)$$

in questo caso la scrittura  $W2(x)$  è persa e non può avere effetto sul DB perchè sovrascritta da  $W1(x)$

- Lettura sporca (dirty read)
  - una transazione effettua una modifica su una risorsa e poi effettua il **rollback**

$$R1(x) \rightarrow W1(x) \rightarrow R2(x) \rightarrow \text{rollback1} \rightarrow W2(x)$$

in questo esempio la transazione 2 ha letto un valore che viene successivamente eliminato dalla base di dati (per effetto del rollback)

- Lettura inconsistente (unrepeatable read)

ad ogni transazione bisogna garantire di leggere lo **stesso valore** della stessa risorsa per **tutta** la sua durata

$$R1(x) \rightarrow R2(x) \rightarrow W2(x) \rightarrow R1(x)$$

- Aggiornamento fantasma (phantom update)

questa anomalia porta il database (visto da una transazione) in uno stato **inconsistente**

Vincolo:  $x+y+z=10$  ( $x=1, y=5, z=4$ )

$$R1(x) \rightarrow R1(y) \rightarrow R2(x) \rightarrow R2(z) \rightarrow W2(x) \rightarrow W2(z) \rightarrow R1(z)$$

$W2(x)=x+2, W2(z)=z-2$  (formalmente corretto a livello di consistenza)

T1 vedrà uno stato della base di dati inconsistente ( $x=1, y=5, z=2$ )

<b>T1</b>	X=1	Y=5			Z=2		<b>TOT=8</b>
	R1(x)	R1(y)	R2(x)	R2(z)	W2(x)	W2(z)	R1(z)
<b>T2</b>		X=1	Z=4	X=X+2=3	Z=Z-2=2		<b>TOT=10</b>

- Inserimento fantasma (phantom insert)

si crea quando una transazione effettua delle operazioni con dati aggregati (media, somma, ...) e durante la sua esecuzione un'altra transazione effettua una **insert**

- Sono previsti quattro livelli di isolamento:
  - **read uncommitted**
    - consente transazioni in sola lettura, senza bloccare i dati
    - permette le anomalie dovute a transazioni concorrenti
  - **read committed**
    - prevede il rilascio immediato dei dati in lettura, ritardando quelli in scrittura
    - evita la lettura sporca, ma non le altre anomalie
  - **repeatable read**
    - vengono bloccati sia i dati in lettura che quelli in scrittura ma solo sulle tuple della tabella coinvolte
    - non evita l'inserimento fantasma
  - **serializable**
    - garantisce l'effettiva serializzabilità del codice bloccando gli accessi alle tabelle utilizzate
    - evita tutte le anomalie

(Transazione)  $T_1: R1(X) \rightarrow R1(y) \rightarrow W1(x) \rightarrow W1(y)$

(Schedule)  $S_1: R1(X) \rightarrow R2(Z) \rightarrow W1(x) \rightarrow W2(y)$

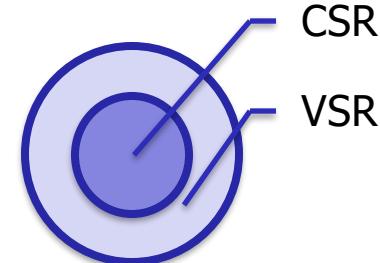
- Sequenza di operazioni di lettura/scrittura presentate da transazioni concorrenti
- Seguono l'ordine con cui sono eseguite sul DB
- **Schedule Seriale:** le azioni di ciascuna transazione compaiono in sequenza, senza essere inframmezzate da istruzioni di altre transazioni  
 $S_i: R_0(X) \rightarrow W_0(X) \rightarrow R_1(X) \rightarrow R_1(Y) \rightarrow W_2(X) \rightarrow R_2(Z)$
- **Schedule Serializzabile:** l'esecuzione dello schedule produce lo stesso risultato di uno schedule seriale

- $R_i(X)$  legge da  $W_j(X)$  se l'operazione di scrittura precede quella di lettura e non ci sono altre scritture comprese tra le due operazioni

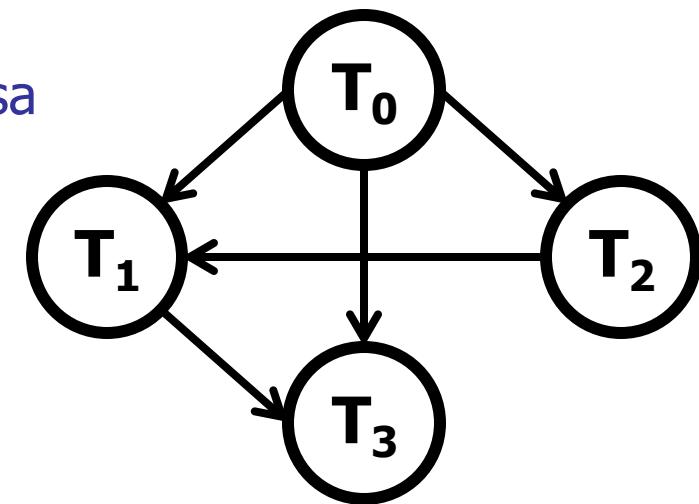
$$W_j(X) \rightarrow R_i(X)$$

- $W_j(X)$  viene detta **scrittura finale** se è l'ultima scrittura dell'oggetto  $X$  nello schedule
- Due schedule sono **view-equivivalenti** ( $S_i \approx_v S_j$ ) se:
  - possiedono la stessa relazione "legge da"
  - possiedono le stesse scritture finali
- Uno schedule è **view-serializzabile** se esiste uno schedule seriale view-eqvivalent
- Determinare se uno schedule è view-serializzabile è un problema NP-hard (poco pratico)

- **Conflitto:** l'azione  $a_i$  è in conflitto con  $a_j$  se operano sullo **stesso oggetto** e almeno una di esse è una **scrittura**
  - conflitti Lettura-Scrittura (rw - wr)
  - conflitti Scrittura-Scrittura (ww)
- Due schedule sono **conflict-equivalenti** ( $S_i \approx_C S_j$ ) se:
  - presentano le stesse operazioni
  - ogni coppia in conflitto è nello stesso ordine nei due schedule
- Uno schedule è **conflict-serializzabile** se esiste uno schedule seriale conflict-equivalente
- CSR: insieme degli schedule conflict-serializzabili
- VSR: insieme degli schedule view-serializzabili
- La classe degli schedule CSR è **strettamente inclusa** in quella degli schedule VSR



- E' possibile verificare se uno schedule è in CSR attraverso il **grafo dei conflitti**
- Ogni **nodo** è una transazione
- Si traccia un **arco** da  $T_i$  a  $T_j$  se:
  - esiste almeno un **conflitto** tra  $a_i$  e  $a_j$
  - $a_i$  precede  $a_j$
- Si può dimostrare che uno schedule è in CSR se il suo grafo dei conflitti è **acidico**
- Complessità lineare, ma ancora troppo onerosa
- Non utilizzabile in contesti distribuiti



- Le operazioni di lettura/scrittura sono protette attraverso 3 primitive
  - r\_lock: ogni operazione di lettura è preceduta da un *r\_lock* e seguita da un *unlock* lock **condiviso** (più lock contemporanei di tipo read sulla stessa risorsa)
  - w\_lock: ogni operazione di scrittura è preceduta da un *w\_lock* e seguita da un *unlock* lock **esclusivo** (unico lock sulla stessa risorsa)
  - unlock

## Transazione ben formata rispetto al locking

- Richiesta di lock concessa → risorsa **acquisita**
  - Richiesta di unlock concessa → risorsa **rilasciata**
  - Richiesta di lock non concessa → transazione in stato di **attesa**
  - lock condiviso → lock esclusivo (**lock upgrade**)
- 
- I lock concessi sono memorizzati nella *tabella dei lock*
- ID Transaz  
+  
ID Risorsa

# Tabella dei conflitti

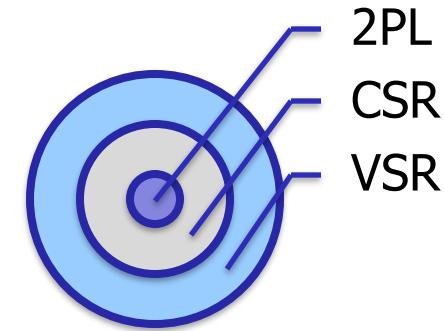
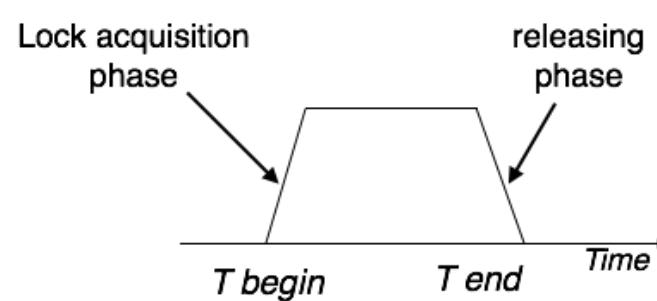
- Memorizza i lock concessi

Richiesta	Stato Risorsa		
	libero	r_locked	w_locked
r_lock	OK / r_locked	OK / r_locked (*)	NO / w_locked
w_lock	OK / w_locked	NO / r_locked	NO / w_locked
unlock	error	OK / dipende (#)	OK / libero

- (\*) incremento il **contatore** dei lock in lettura
- (#) decremento il contatore e solo se è pari a 0 la risorsa torna libera

# Locking a due fasi (Two Phase Locking, 2PL)

- Una transazione dopo aver **rilasciato** un lock **non può** acquisirne altri
- Fase **crescente**: si acquisiscono i lock per le risorse
- Fase **calante**: si rilasciano i lock acquisiti



- Transazioni ben formate + protocollo 2PL = serializzabile rispetto alla conflict-equivalenza
- La classe 2PL è strettamente contenuta nella classe CSR
- Non evita la lettura sporca!

- Una transazione può rilasciare i lock solo dopo aver **correttamente effettuato** le operazioni di **commit/abort** (solo al termine della transazione)
- Lock di **predicato**: lock definiti con riferimento a condizioni (predicati) impendendo l'accesso e la scrittura a dati che rispettano il predicato
- Possono essere realizzati diversi livelli di isolamento nel 2PL (per transazione di sola lettura):
  - **read uncommitted**
    - la transazione non chiede lock e non osserva i lock esclusivi posti da altre transazioni
  - **read committed**
    - richiede lock in lettura ma li rilascia subito (quindi senza 2PL, si evitano letture sporche, ma non altre anomalie delle letture)
  - **repeatable read**
    - è applicato il 2PL stretto ma solo a livello di tupla (sono evitate tutte le anomalie, ma non l'inserimento fantasma perché non si impedisce l'inserimento di nuove tuple)
  - **serializable**
    - applica il 2PL stretto e i lock di predicato (evita tutte le anomalie)



# Controllo basato su timestamp (Metodo TS)

- Ad ogni transazione viene associato un **timestamp** (ts) che ne definisce l'inizio
- Lo schedule è accettato solo se riflette l'**ordinamento seriale** sulla base dei ts di ciascuna transazione
- $WTM(x)$  = ts della transazione che ha eseguito l'ultima scrittura su  $x$
- $RTM(x)$  = ts maggiore tra quelli delle transazioni che hanno letto  $x$

$r_t(x)$ :  $t < WTM(x) \rightarrow$  la transazione viene uccisa

$t \geq WTM(x) \rightarrow OK, RTM(x) = \max\{RTM(X), t\}$

$w_t(x)$ :  $t < WTM(x)$  o  $t < RTM(x) \rightarrow$  la transazione viene uccisa

$t \geq WTM(x) \rightarrow WTM(x) = t$

- Vengono uccise **molte** transazioni  $\rightarrow$  Variante *Multiversione*

	<b>2PL</b>	<b>Metodo TS</b>
<b>Transazioni Rifiutate</b>	Attesa	Uccise e riavviate
<b>Ordine</b>	Imposto dai conflitti	Imposto da TS
<b>Attesa esito</b>	Incremento tempo di blocco	Condizioni di attesa
<b>Problemi</b>	Deadlock	Restart molto lento ( > tempo attesa 2PL)

- Blocco Critico (Deadlock): due (o più) transazioni sono in attesa l'una dell'altra
- La probabilità di conflitto cresce **linearmente** col numero K di transazioni presenti nel sistema e **quadraticamente** col numero medio m di risorse richieste

- Timeout
  - le transazioni restano in attesa per un tempo **prefissato**, allo scadere del timeout il lock è rifiutato (transazione tolta da attesa e abortita)
  - semplice ma occorre definire **correttamente** il timeout
  - timeout troppo elevato: deadlock rilevato in ritardo (risolve tardi blocchi critici)
  - timeout troppo basso: “falsi” deadlock identificati (uccide inutilmente transazione e spreca lavoro già svolto dalla transazione)
- Prevenzione (deadlock prevention)
  - allocazione **preliminare** dei lock (transazione chiede il lock di tutte le risorse, non sempre possibile perché non tutte le risorse possono essere conosciute)
  - uso di **timestamp**, una transazione attende solo se esiste una precedenza tra i ts
  - uccisione transazioni
    - politiche preemptive (interrompente): uccide la transazione che possiede la risorsa
    - politiche non preemptive: uccide la transazione che richiede la risorsa
  - starvation: transazione uccisa ripetutamente, si verifica se uccido la transazione più giovane → mantenere lo stesso ts anche dopo il riavvio

- Rilevamento (deadlock detection)
  - nessun vincolo sul sistema
  - controllo delle tabelle di lock ad intervalli predefiniti o attraverso timeout
  - analisi del grafo delle attese tra transazioni (WAIT-FOR)



## Organizzazione fisica dei dati



## Gestore delle interrogazioni

- Decide le strategie di accesso ai dati per rispondere alle interrogazioni

## Gestore dei metodi di accesso

- Esegue l'accesso fisico ai dati secondo la strategia definita dal gestore delle interrogazioni

## Gestore del buffer

- Gestisce il trasferimento delle pagine del DB dalla memoria di massa a quella centrale

## Gestore della memoria secondaria

- Controllore dell'affidabilità
- Controllore della concorrenza

## Memoria Secondaria

- Area in **memoria centrale** preallocata al DBMS e condivisa tra le varie transazioni
- Suddiviso in **pagine** di dimensione pari al blocco di I/O usato dal S.O.
- Princípio di **località dei dati**: i dati referenziati di recente hanno maggiore probabilità di essere referenziati nuovamente
- Primitive di accesso alle pagine:
  - FIX
  - SET DIRTY
  - USE
  - UNFIX
  - FLUSH
  - FORCE



# Primitive del Buffer Manager (1/2)

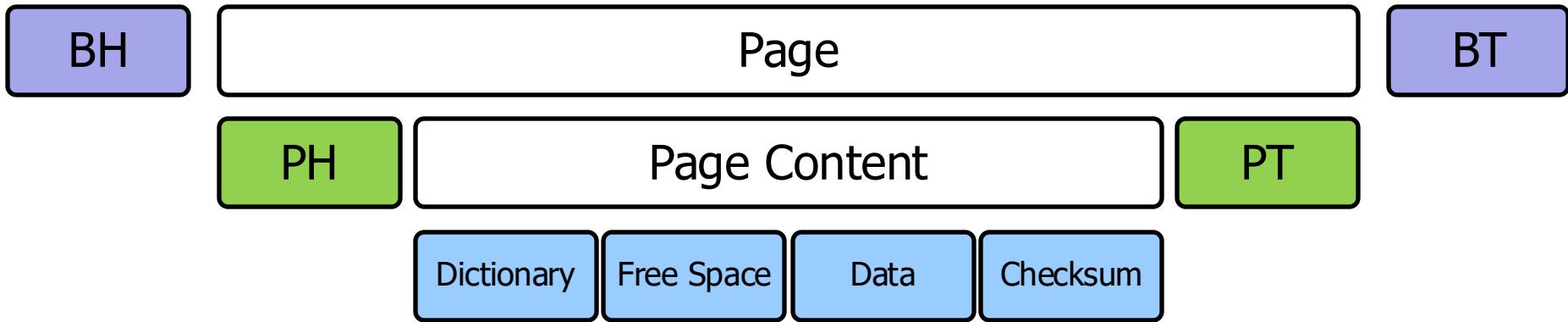
## FIX

- Usata per richiedere accesso a una pagina e **caricarla** nel buffer
- Restituisce il **puntatore** alla pagina, in modo che possa accedere effettivamente ai dati. La pagina sarà considerata **allocata** in una transazione attiva
- Funzionamento:
  1. si cerca la pagina tra quelle già **presenti** nel buffer, se presente l'operazione termina
  2. si cerca una pagina **libera** nel buffer (non allocate), se contiene modifiche queste vengono riportate in memoria di massa (FLUSH), infine avviene l'operazione di lettura
    - LRU (Least Recent Used – usata meno di recente)
    - FIFO (First In First Out – caricata da più tempo)
  3. se non esistono pagine libere (tutte allocate)
    - politica **Steal**: seleziona una pagina "vittima" sottratta ad una transazione e scaricata in memoria di massa (FLUSH)
    - politica **No Steal**: transazione sospesa in attesa di pagine libere (entra in coda di transazione), appena si libera una pagina il buffer manager opera come al punto 2
  4. **si incrementa** di uno il contatore di utilizzo della pagina

# Primitive del Buffer Manager (2/2)

- **SET DIRTY**: indica che una pagina è stata **modificata** (bit di stato)
- **USE**: accede alla pagina caricata in memoria
- **UNFIX**: indica al buffer manager che si è **terminato** di usare la pagina (decremento contatore)
- **FORCE**: trasferisce in modo **sincrono** (all'interno dell'applicazione che la richiede) una pagina dal buffer alla memoria secondaria (richiesta fatta da gestore affidabilità per garantire che alcuni dati non vengano persi)
- **FLUSH**: trasferisce pagine in modo **asincrono** (in modo indipendente dall'applicazione) indipendentemente dalle transazioni attive (decisione del gestore del buffer per recupero spazio od ottimizzazione)

- Trasforma un piano d'accesso (fornito dall'ottimizzatore) in una **sequenza di accessi alle pagine**
- Metodi d'accesso: moduli SW per l'accesso e la manipolazione dei dati
- Si individuano i **blocchi** da caricare e si indicano al buffer manager
- Riescono ad individuare specifici valori all'interno di una pagina
- Primitive fornite:
  - Accesso ad una particolare tupla in base alla chiave o per offset
  - Inserimento, Aggiornamento e Cancellazione di una tupla
  - Accesso ad un campo della tupla (tupla + offset + lunghezza campo)



- Block Header/Trailer: informazioni di controllo usate dal file system
- Page Header/Trailer: informazioni di controllo riferite alla specifica struttura (ID oggetto, puntatori a pagine successive/precedenti, ...)
- Dizionario: puntatori ai singoli elementi nella pagina
- Data: insieme di tuple
- Checksum: bit di parità

Tuple organizzate in blocchi (memoria secondaria) in modo sequenziale

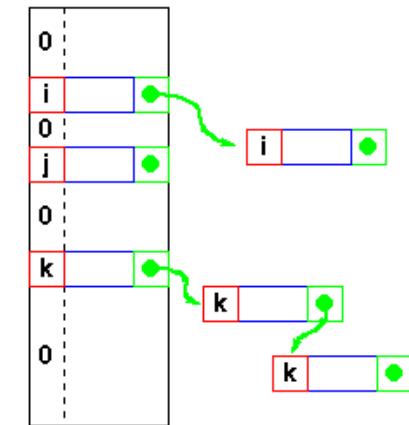
- Entry-Sequenced (seriale): in ordine di **immissione**
  - semplice per gli inserimenti
  - inefficiente per le ricerche (scansione seriale), usata con strutture di supporto
  - cancellazione: si marca la tupla come eliminata
  - modifica: locale se non varia la dimensione altrimenti cancellazione e riscrittura in coda
- Array: la posizione nell'array dipende da un campo **indice**
  - solo con tuple di dimensione fissa ( $n$  blocchi \*  $m$  posizioni disponibili)
  - le primitive utilizzano il valore dell'indice
  - poco utilizzate
- Sequenziale ordinata: ordinate in base al campo **chiave**
  - efficienti per interrogazioni su intervalli e con raggruppamenti (es: cognomi che iniziano con una certa sequenza di lettere oppure data compresa in intervallo)
  - difficoltà negli inserimenti e cancellazioni (per mantenere aggiornamento)
  - riorganizzazioni periodiche necessarie

Accesso **associativo** ai dati, la posizione dipende dal campo chiave

- Tuple anche non in ordine in memoria di massa
- Sfruttare l'accesso **diretto** tipico degli array
- Si allocano per il file B blocchi contigui

$$\text{Hash(fileID, key)} = \text{blockID } \{0, B-1\}$$

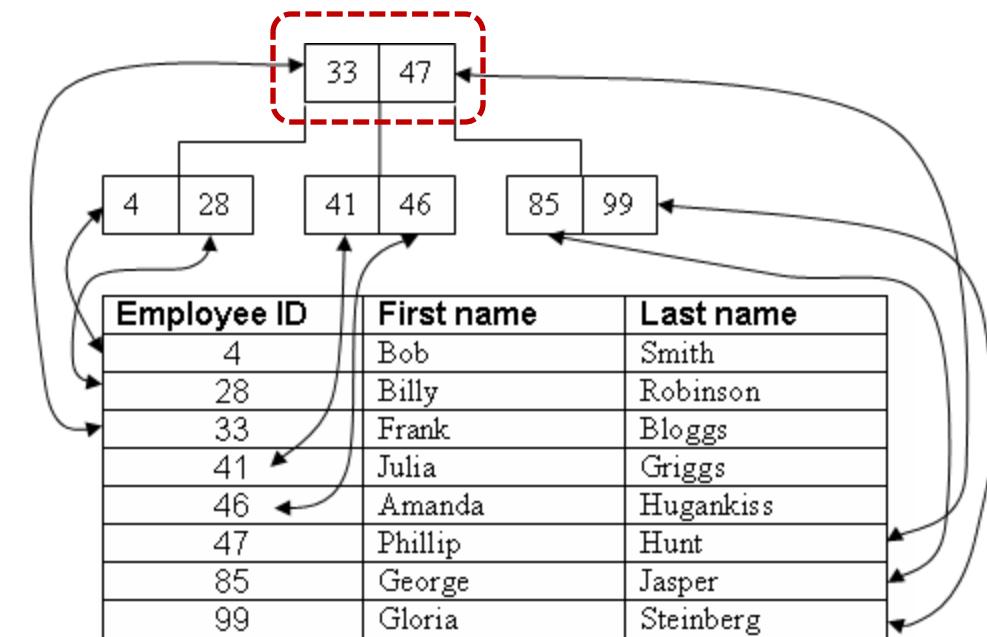
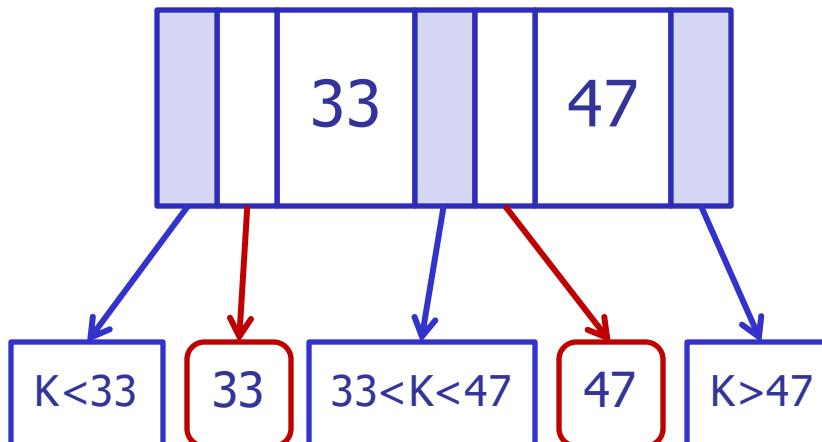
- Folding: key → valore intero positivo (suddivisione chiave in parti per ottenere un valore intero positivo)
- Hashing: valore intero → indirizzo di blocco tra 0 e B-1
- Problema delle **collisioni**: stesso blockID per 2 chiavi diverse (record vanno nello stesso blocco fino a esaurimento)
- Catene di **overflow**: allocazioni di nuovi blocchi collegati al precedente (rallentano la ricerca)



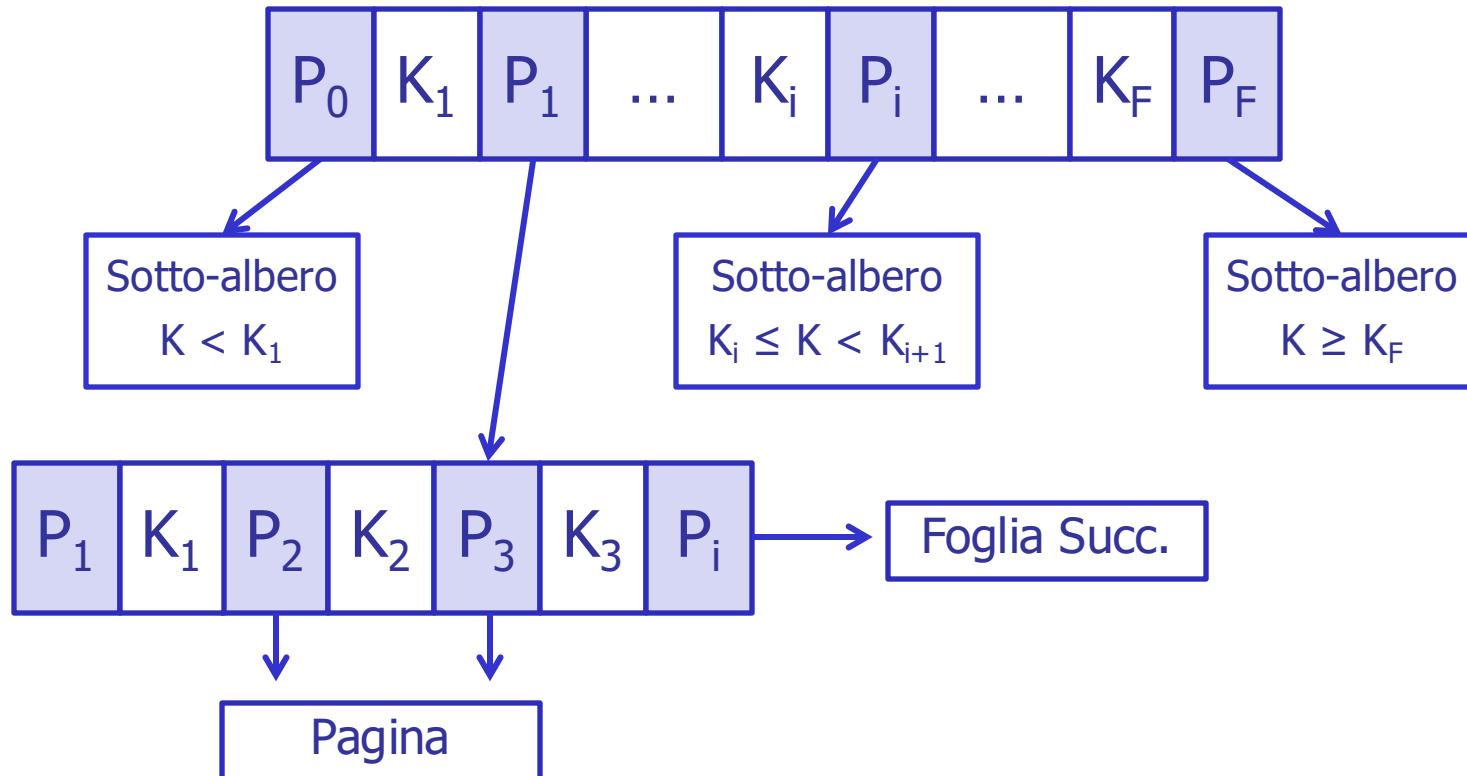
Accesso associativo ai dati, la posizione dipende da uno o più campi chiave

- Indici primari: contenenti i dati o riferito ad un file ordinato in base allo stesso campo dell'indice
- Indici secondari (non contengono dati): di supporto alle operazioni sui dati, ogni elemento contiene un valore di chiave  $k$  e l'indirizzo (o gli indirizzi) del record associato alla chiave
- Strutture ad albero dinamiche, efficienti anche per inserimenti (obiettivo: tempo medio di accesso costante)
  - B-tree (B)
  - B+ tree (B+)
- Ogni albero contiene dei nodi collegati attraverso puntatori
  - nodo radice
  - nodi intermedi
  - nodi foglia
- Ogni nodo coincide con una pagina (o blocco) del file system
- Alberi bilanciati: cammino di lunghezza costante tra radice e foglie

- I nodi intermedi (non foglia) sono composti da:
  - valore di chiave  $K_i$
  - un puntatore diretto ai blocchi riferiti a  $K_i$
  - un puntatore al sotto-albero riferito a blocchi con  $K_i < K < K_{i+1}$
- Non c'è collegamento tra le foglie



- I nodi foglia memorizzano **tutti i valori** e sono collegati tra loro da puntatori nell'ordine stabilito dalla chiave
- I nodi intermedi sono composti da:



- Ogni nodo contiene  $F$  valori di chiave e  $F+1$  puntatori (fan-out)
- $F$  dipende dalla dimensione della pagina, si sceglie il valore massimo in modo da **ridurre** i livelli dell'albero
- Accesso ad una tupla con chiave  $V$ 
  - $V < K_1$  si segue il puntatore  $P_0$
  - $V \geq K_F$  si segue il puntatore  $P_F$
  - $K_i \leq V < K_{i+1}$  si segue il puntatore  $P_i$
  - si procede sino a giungere ad un nodo foglia
    - index-sequential: le foglie contengono la **tupla**
    - indiretto: le foglie contengono **puntatori** alle tuple
- Inserimento
  - si inserisce la nuova pagina in uno slot libero (nelle foglie)
  - se non ci sono slot liberi si opera uno **split** sulla foglia
  - si inserisce un **nuovo** puntatore nel nodo di livello superiore (può richiedere un ulteriore **split**)

- Effettuata “in loco” segnando come **invalido** lo spazio allocato
- Se la cancellazione interessa una **chiave** dell’albero
  - recupero il valore di chiave successivo
  - aggiorno la chiave nell’albero
  - tutti i valori nell’albero corrispondono al DB (**consigliato** ma non strettamente necessario)
- Se sono libere due pagine contigue **nelle foglie**
  - **merge**: si unisce l’informazione disponibile in un’unica pagina
  - occorre modificare i puntatori del livello superiore (si **elimina** un puntatore)
  - può portare ad un ulteriore merge



## Architetture Distribuite per Basi di Dati



- **Parallelismo:** usato per ottimizzare le **prestazioni** di componenti/sistemi OLTP e OLAP
- **Replicazione dei dati:** capacità di costruire copie di collezioni di dati, esportabili da un nodo all'altro di un sistema distribuito, per massimizzare:
  - disponibilità dei dati
  - affidabilità dei dati
- **Interazione** fra sistemi e prodotti diversi
  - **portabilità:** trasportare programmi da un ambiente ad un altro (tempo di compilazione)
  - **interoperabilità:** far interagire sistemi eterogenei (tempo di esecuzione)
- **Importanza degli standard**
  - la portabilità dipende dagli standard relativi ai **linguaggi** (SQL)
  - l'interoperabilità dipende dagli standard relativi ai **protocolli** di accesso ai dati (ODBC, JDBC)

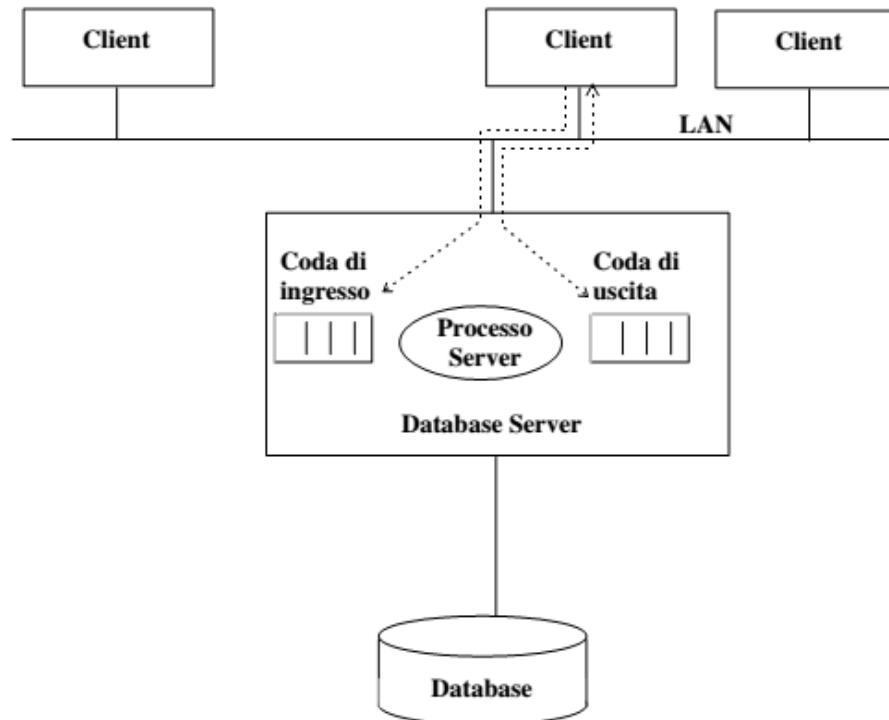


# Paradigma Client-Server

Caratteristica	Client	Server
Ruolo	Software Applicativo	DBMS che supporta più applicazioni
Hardware	Personal Computer	Sistema dimensionato rispetto al carico transazionale
Operazioni SQL	Interrogazioni	Invio risultati

- Interrogazioni compilate staticamente (**compile and store**)
  - interrogazioni sottomesse una sola volta e richiamate molte volte
  - chiamate a procedure e/o servizi remoti (cursori)
- Interrogazioni con SQL dinamico (**compile and go**)
  - invio dell'interrogazione sotto forma di stringhe di caratteri
- Interrogazioni **parametriche**
  - assegnazione di alcuni parametri
  - esecuzione di una interrogazione o procedura

- Server multi-threaded: un unico processo opera per conto di differenti transazioni
- Dispatcher: distribuisce le richieste ai server e restituisce le risposte ai client (gestione delle code)



- Omogenea: tutti i server usano lo stesso DBMS
- Eterogenea: i server utilizzano DBMS diversi
- Una transazione coinvolge più server
- Gestione distribuita dei dati
  - flessibilità, modularità e resistenza ai guasti
  - complessità strutturale
- Frammentazione dei dati
  - orizzontale
    - $R_i$  è un insieme di tuple con lo **stesso schema** di  $R$
    - ogni  $R_i$  è (logicamente) il risultato di una **selezione** su  $R$
  - verticale
    - lo schema di  $R_i$  è **sottoinsieme** dello schema di  $R$
    - ogni  $R_i$  è (logicamente) il risultato di una **proiezione** su  $R$



# Frammentazione Orizzontale: Esempio

ISBN	Titolo	Anno	Pagine	Genere
8817068691	La nave di Teseo	2014	456	Fantasy
8804596945	Il simbolo perduto	2009	814	Thriller
880464317X	Il trono di spade	2014	1495	Fantasy
8817005010	Il ritratto di Dorian Gray	2005	245	Classici



ISBN	Titolo	Anno	Pagine	Genere
8817068691	La nave di Teseo	2014	456	Fantasy
880464317X	Il trono di spade	2014	1495	Fantasy

ISBN	Titolo	Anno	Pagine	Genere
8804596945	Il simbolo perduto	2009	814	Thriller
8817005010	Il ritratto di Dorian Gray	2005	245	Classici

# Frammentazione Verticale: Esempio

<u>ISBN</u>	<u>Titolo</u>	<u>Anno</u>	<u>Pagine</u>	<u>Genere</u>
8817068691	La nave di Teseo	2014	456	Fantasy
8804596945	Il simbolo perduto	2009	814	Thriller
880464317X	Il trono di spade	2014	1495	Fantasy
8817005010	Il ritratto di Dorian Gray	2005	245	Classici



<u>ISBN</u>	<u>Titolo</u>	<u>Anno</u>
8817068691	La nave di Teseo	2014
8804596945	Il simbolo perduto	2009
880464317X	Il trono di spade	2014
8817005010	Il ritratto di Dorian Gray	2005

<u>ISBN</u>	<u>Pagine</u>	<u>Genere</u>
8817068691	456	Fantasy
8804596945	814	Thriller
880464317X	1495	Fantasy
8817005010	245	Classici

- **Correttezza** della frammentazione

- **completezza**: ogni dato in R deve essere presente in un qualche suo frammento  $R_i$
- **ricostruibilità**: la relazione deve essere interamente ricostruibile a partire dai suoi frammenti
- Ogni frammento  $R_i$  è implementato attraverso un **file fisico** su uno specifico server (allocazione)
- Schema di **allocazione**: mapping dai frammenti (o dalle relazioni) ai server che li memorizzano
  - mapping **non ridondante**: ogni frammento allocato su un unico server
  - mapping **ridondante**: stesso frammento allocato su più server

## ▪ Trasparenza di frammentazione

- interrogazione identica al caso di DB non frammentato
- non occorre sapere se il DB sia distribuito o frammentato

```
select Titolo  
from Libro  
where ISBN = :isbn;
```

## ▪ Trasparenza di allocazione

- il programmatore conosce la struttura dei frammenti
- non occorre indicarne l'allocazione

```
select Titolo  
from Libro1  
where ISBN = :isbn;  
if :empty then  
    select Titolo  
    from Libro2  
    where ISBN = :isbn;
```

- Trasparenza di linguaggio

- occorre indicare sia la struttura che l'allocazione dei frammenti

```
select Titolo
from Libro1@libreria.bari.it
where ISBN = :isbn;
if :empty then
    select Titolo
    from Libro2@libreria.roma.it
    where ISBN = :isbn;
```

- Assenza di trasparenza

- DBMS eterogenei
  - ogni DBMS accetta una propria sintassi SQL
  - occorre indicare sia la struttura che l'allocazione dei frammenti

- **Richieste remote**

- transazioni di **sola lettura** ad un solo DBMS remoto

- **Transazioni remote**

- transazioni con comandi SQL generici ad **un solo** DBMS remoto

- **Transazioni distribuite**

- transazioni rivolte a più DBMS, ma ogni comando SQL fa riferimento ai dati di **un solo** DBMS

- **Richieste distribuite**

- transazioni arbitrarie, dove ogni comando SQL può far riferimento a dati distribuiti su un **qualsiasi** DBMS

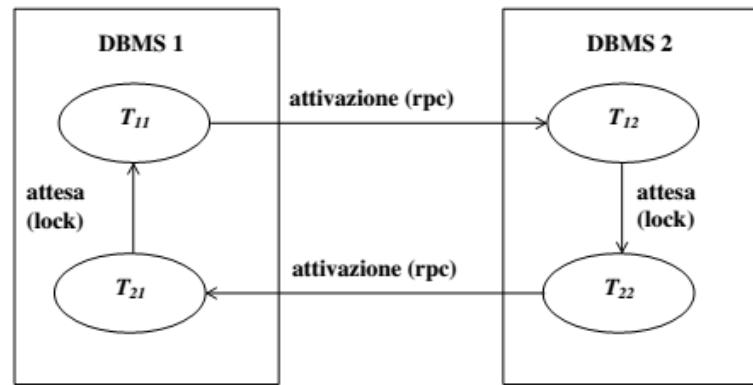


# Tecnologia delle Basi di Dati distribuite

La distribuzione dei dati incide sulle proprietà ACID ?

- Consistenza e Durabilità delle transazioni **non dipendono** dalla distribuzione dei dati (proprietà locali al DBMS)
- Ottimizzazione delle **interrogazioni**
  - solo per richieste distribuite
  - effettuata dal DBMS che genera la richiesta
  - decomponere la richiesta in sotto-interrogazioni rivolte a più DBMS
- Controllo di concorrenza (isolamento)
- Controllo di affidabilità (atomicità)

- Una transazione  $t_i$  può essere suddivisa in più sottotransazioni  $t_{ij}$  ( $j = \text{indice nodo}$ )
- La serializzabilità **locale** presso gli scheduler non è garanzia **sufficiente** per la serializzabilità
- Serializzabilità **globale**
  - unico schedule seriale  $S$ , che coinvolge **tutte** le transazioni del sistema, equivalente a tutti gli schedule locali  $S_i$
  - per ogni nodo  $i$ , la proiezione  $S[i]$  di  $S$ , contenente le sole operazioni svolte su  $i$ , deve essere **equivalente** a  $S_i$
- Facile da garantire se gli scheduler locali usano il 2PL o il metodo dei timestamp
  - se ciascun nodo applica il 2PL ed effettua la commit in modo atomico in un istante in cui le sotto-transazioni ai vari nodi detengono **tutte le risorse**, gli schedule risultanti sono globalmente serializzabili rispetto ai conflitti
  - se un insieme di sotto-transazioni distribuite acquisisce un **unico timestamp**, gli schedule risultanti sono globalmente seriali in base all'ordinamento indotto dai timestamp



## Rilevazione distribuita dei deadlock

- impiego dei **timeout**
- algoritmo di rilevazione dei deadlock
  - due sotto-transazioni della stessa transazione in attesa su **DBMS distinti** (una attende la fine dell'altra)
  - due sotto-transazioni di diverse transazioni sono in attesa sullo **stesso DBMS** (una blocca i dati a cui vuole accedere l'altra)

Algoritmo attivato **periodicamente**



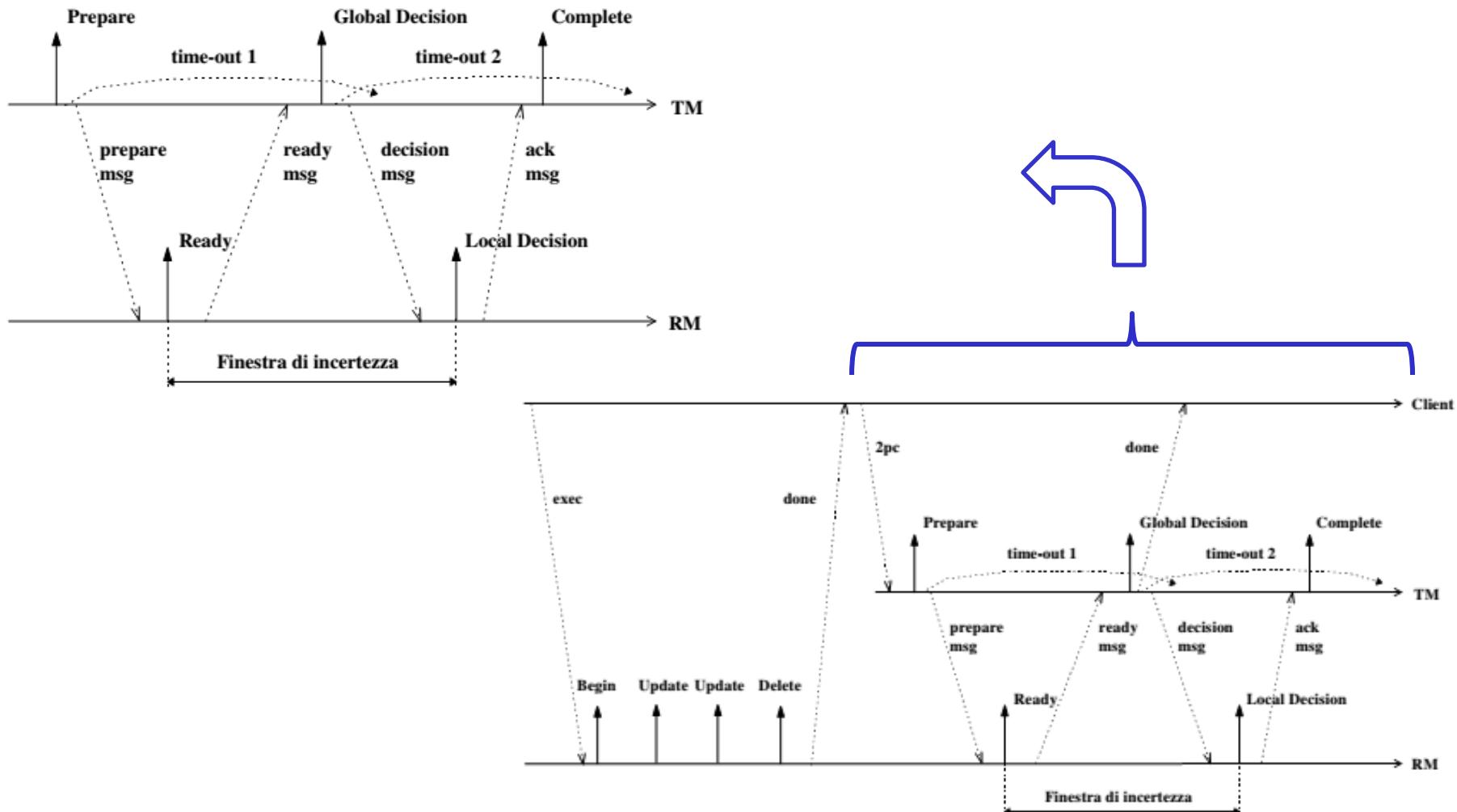
- **Atomicità** di transazioni distribuite: tutti i nodi che partecipano alla transazione devono pervenire allo **stesso risultato** (commit o abort)
- Protocollo di **commit a due fasi**
  - scambio di messaggi fra Transaction Manager (TM) e Resource Manager (RM)
  - protocollo **resistente** ai guasti: TM e RM scrivono alcuni nuovi record nei loro file di log
- Il TM scrive:
  - record di **prepare** con l'identità dei processi RM (ID di nodo + ID processo)
  - record di **global commit** o di **global abort** (determina l'esito finale dell'intera transazione, su tutti i nodi in cui opera)
  - record di **complete**, scritto alla conclusione del protocollo di commit a due fasi
- Ogni RM rappresenta una sotto-transazione e scrive i record di *begin*, *insert*, *delete*, *update* (come nel caso centralizzato)
  - record di **ready**: disponibilità (irrevocabile) a partecipare al protocollo di commit a due fasi; contiene anche l'identificatore (nodo e processo) del TM

- TM può usare sia meccanismi di broadcast che di comunicazione **seriale** per comunicare con diversi RM
- Prima Fase
  1. il TM scrive il record di **prepare** ed invia un messaggio di prepare per informare dell'inizio del protocollo (un timeout predefinito indica il ritardo massimo nella ricezione dei messaggi di risposta da RM)
  2. gli RM, che sono in uno stato **affidabile**, ricevuto il messaggio scrivono il record di **ready** (scelta favorevole di partecipazione) e mandano il relativo messaggio al TM (oppure **not-ready** nel caso di guasto di transazione)
  3. il TM colleziona i messaggi di risposta:
    - se tutti sono positivi, scrive sul suo log un record di **global commit**
    - in caso contrario scrive un record di **global abort**

## ▪ Seconda Fase

1. il TM invia la sua decisione **globale** agli RM ed imposta un time-out per la ricezione dei messaggi di risposta dagli RM
2. gli RM, che sono in uno stato di ready, ricevono il messaggio e scrivono il record di commit o abort (questa volta **locale**)
3. gli RM mandano il messaggio di **acknowledgement** (ack) al TM
4. Implementazione di commit/abort procede su ciascun server
5. il TM colleziona i messaggi di ack:
  - se tutti i messaggi arrivano, scrive sul suo log un record di **complete**
  - in caso contrario reimposta un **nuovo timeout** e ripete la trasmissione verso gli RM che non hanno risposto
  - l'operazione si ripete finché tutti gli RM non hanno risposto

# Commit a due fasi





# Protocolli di ripristino: Caduta di un partecipante

- Perdita del contenuto del buffer
- Può quindi lasciare la base di dati in uno stato **inconsistente**
- Stato del partecipante recuperato attraverso il relativo file di log
- Ripresa a caldo:
  - se l'ultimo record nel log è relativo ad un'azione o ad un **abort**, le azioni vanno **disfatte**
  - se l'ultimo record è un **commit**, le azioni vanno **rifatte**
- Caso **critico**: l'ultimo record nel log è di **ready**
- Per tutte le transazioni in dubbio (collezionate nell'insieme **READY**), si richiede l'esito finale
  - richiesta diretta dal nodo RM al nodo TM (remote recovery)
  - ripetizione della seconda fase del protocollo (trasferimento da TM a RM)
  - esplicita richiesta di effettuare la recovery



# Protocolli di ripristino: Caduta del coordinatore

- Comporta la perdita di messaggi (avviene durante la loro trasmissione)
- Dal log è possibile ricavare solo informazioni sullo stato di avanzamento del protocollo (non sui messaggi inviati)
- ultimo record nel log = **prepare**, alcuni RM possono essere in situazione di attesa (blocco)
  - il TM decide un **global abort** e poi svolge la seconda fase del protocollo
  - il TM ripete anche la **prima fase** (RM in attesa nello stato di ready) al fine di poter decidere un global commit
- ultimo record nel log = **global commit** o **global abort**, alcuni RM possono essere in attesa (il TM non è riuscito a comunicare la decisione finale a tutti gli RM)
  - il TM ripete la **seconda fase** del protocollo
- ultimo record nel log = **complete**, la caduta del coordinatore **non ha effetto sulla transazione**

- Protocollo oneroso: assunzione scritture nel log **sincrone** (tramite una force) per garantire la persistenza
- Esito di **default** in assenza di informazione circa alcuni partecipanti
  - protocollo di abort presunto
    - ad ogni richiesta di *remote recovery* → abort
    - devono essere scritti in modo sincrono solo i record di **ready**, **commit** (RM) e **global commit** (TM) (non sono critici *prepare*, *global abort*, *complete*)
  - protocollo di commit presunto
  - sola lettura
    - in caso di operazioni di sola lettura, il RM non deve **influenzare** l'esito finale della transazione
    - al messaggio di *prepare*, ogni partecipante "solo lettura" avvisa il TM con messaggio **read-only**, che lo **ignorerà** nella seconda fase del protocollo



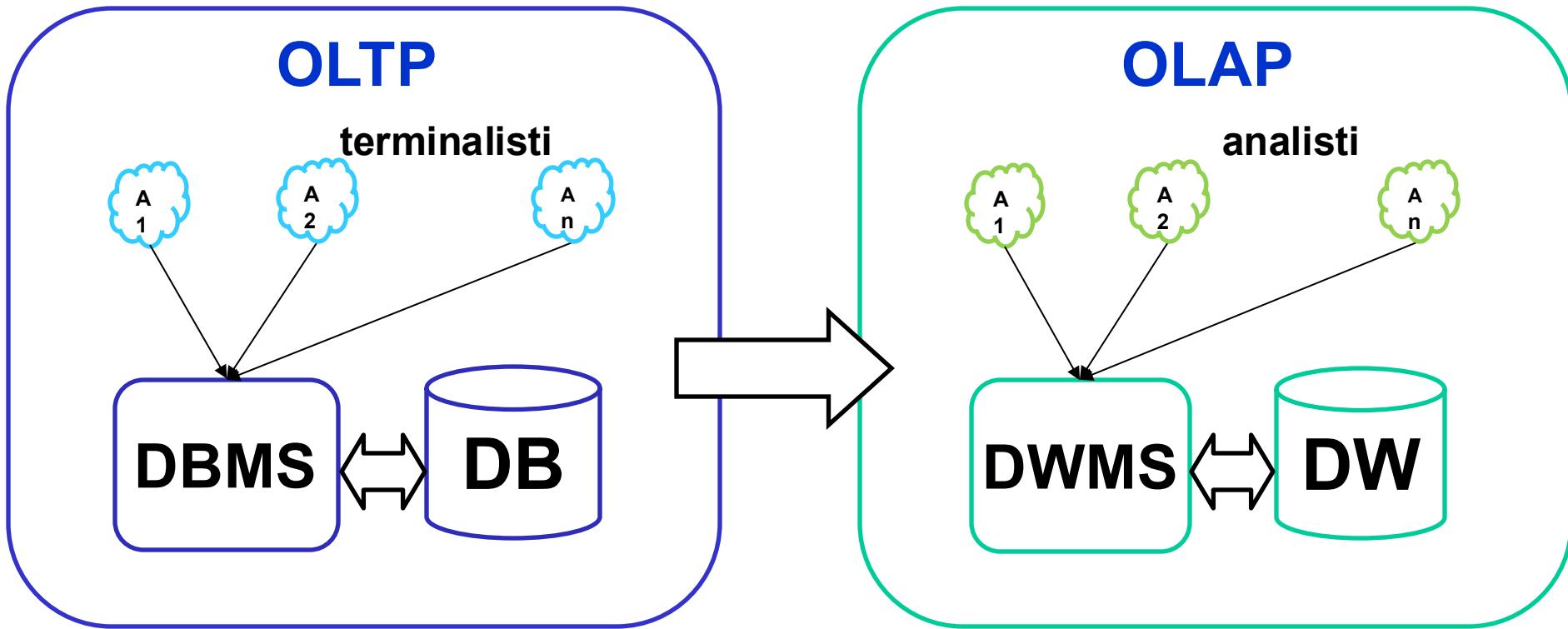
## Analisi dei Dati: OLAP, Data Warehousing, Data Mining





# Dall'OLTP all'OLAP

- La tecnologia delle basi di dati è finalizzata prevalentemente alla gestione dei dati **in linea**, si parla di *OnLine Transaction Processing* (**OLTP**)
- I dati disponibili possono essere utilizzati anche nella pianificazione
- Un'analisi dei dati passati e presenti può essere utile per la **programmazione** delle attività future dell'impresa
- Si parla in questo caso di *On Line Analytical Processing* (**OLAP**)
- **Data warehouse** (magazzino dei dati), in cui sono contenuti dati che, opportunamente analizzati possono fornire un supporto alle decisioni
- I sistemi OLTP forniscono i dati per l'ambiente OLAP, sono cioè una **sorgente di dati** (data source) per tale ambiente
- Tra i due sistemi cambia la tipologia di utente:
  - terminalisti (OLTP)
  - analisti (OLAP)



**Terminalisti:** utenti finali. Possono eseguire operazioni di lettura e di scrittura

**Analisti:** Pochi utenti, occupano posizioni di **alto livello** nell'impresa e svolgono attività di **supporto alle decisioni**.

	OLTP	OLAP
<b>Finalità</b>	Gestione dei dati	Analisi dei dati
<b>Operazioni</b>	Set ben definito	Operazioni non previste nella progettazione del DB (sistemi di supporto alle decisioni)
<b>Dati</b>	Limitata quantità di dati coinvolti, bassa complessità	Grosse moli di dati
<b>Sorgenti Dati</b>	DB singolo	DB eterogenei e distribuiti
<b>Variabilità</b>	Continuo aggiornamento dei dati, stato del sistema in tempo reale	Dati storici aggiornati ad intervalli regolari
<b>Proprietà ACID</b>	Rispettate	Non rilevanti, operazioni di sola lettura

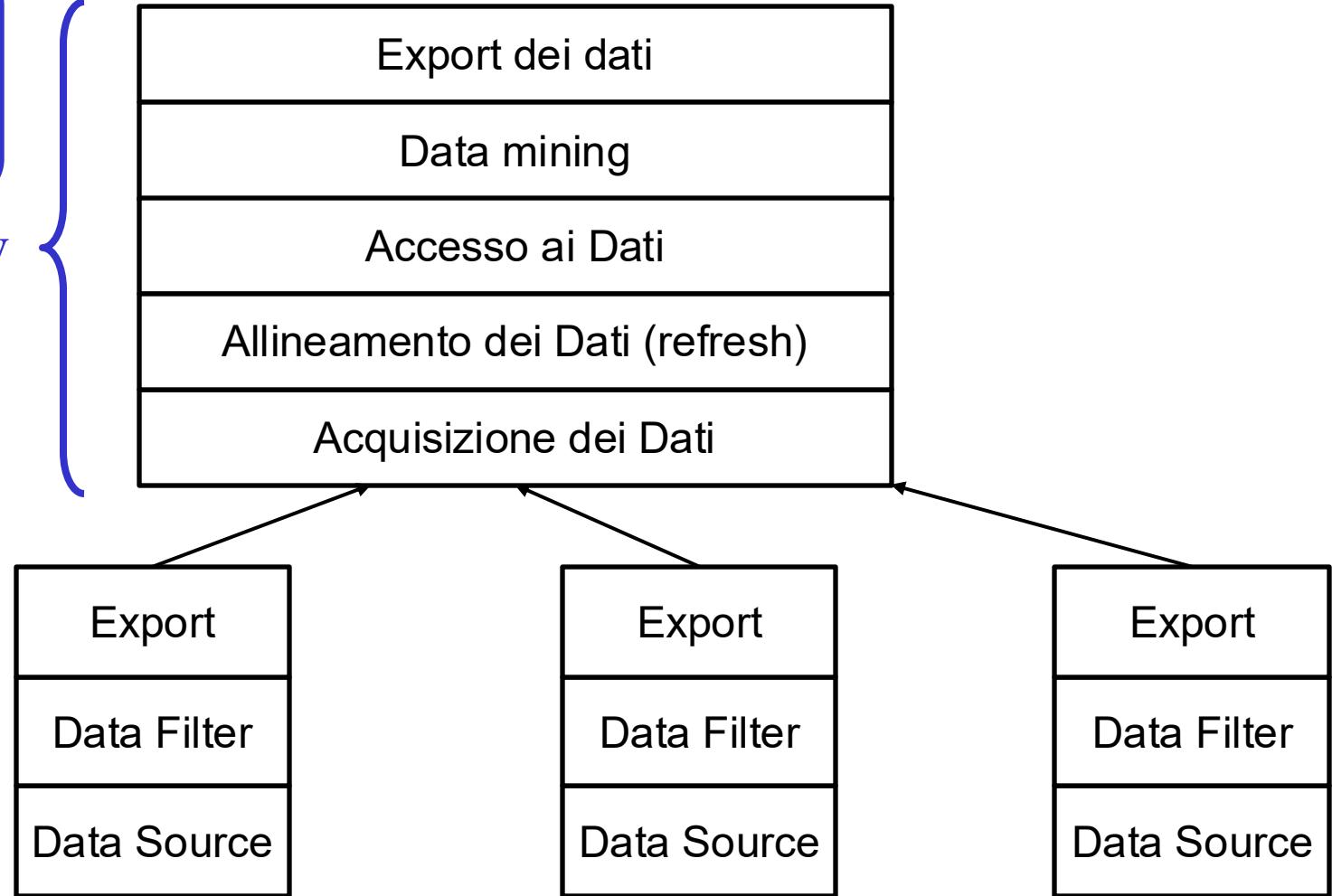
- Utilizzano dati provenienti da più DB eterogenei
- I meccanismi di importazione sono di tipo **asincrono e periodico**
- Non vengono penalizzate le prestazioni delle data source
- La warehouse **non** contiene dati perfettamente **allineati** con il flusso di transazioni negli OLTP
- Problema legato alla qualità dei dati: la semplice raccolta di dati può non essere sufficiente per una **corretta analisi**
- I dati possono contenere inesattezze, errori, omissioni

# Architettura di un DataWarehouse (1/6)

Gli altri cinque componenti operano nella DW

DW

Prime due componenti operano nelle data source



## Data Source

Possono essere di **qualsiasi tipo**, anche raccolte di dati non gestite tramite DBMS oppure gestite da DBMS di vecchia generazione (legacy system).

## Data Filter

Controlla la **correttezza** dei dati prima dell'inserimento nella warehouse.

Può eliminare dati scorretti e rilevare o correggere eventuali **inconsistenze** tra dati provenienti da più data source.

Viene fatta la pulizia dei dati (**data cleaning**) necessaria ad assicurare un buon livello di qualità.

## Export

L'esportazione dei dati avviene in maniera **incrementale**: il sistema colleziona solo le modifiche (inserzioni o cancellazioni) delle data source.



# Architettura di un DataWarehouse (3/6)

## Acquisizione dei Dati (*loader*)

- È responsabile del **caricamento** iniziale dei dati nella DW
- Predisponde i dati all'uso operativo, svolge operazioni di ordinamento, aggregazione e costruisce le strutture dati della warehouse
- Le operazioni di acquisizione vengono svolte a lotti (in **batch**), quando la DW non è utilizzata
- In applicazioni con pochi dati il modulo è invocato periodicamente per acquisire tutto il contenuto della DW
- In genere, invece, i dati vengono allineati in modo **incrementale**, utilizzando il modulo di allineamento dei dati



# Architettura di un DataWarehouse (4/6)

## Allineamento dei Dati (*refresh*)

- Propaga incrementalmente le modifiche della data source in modo da aggiornare il contenuto della DW
- L'aggiornamento può essere effettuato tramite:
  - invio dei dati (**data shipping** – aggiornamenti in archivi varazionali)
  - invio delle transazioni (**transaction shipping** – utilizza log di transazione)
- Nel primo caso all'interno delle data source vengono inseriti dei **trigger** che registrano cancellazioni, inserimenti e modifiche (coppie inserimento-cancellazione) in archivi **varazionali**
- Nel secondo caso viene usato il **log** delle transazioni per costruire gli archivi varazionali

## Accesso ai Dati

- E' il modulo che si occupa dell'**analisi** dei dati
- Realizza in maniera efficiente interrogazioni complesse, caratterizzate da join tra tabelle, ordinamenti e aggregazioni complesse
- Consente nuove operazioni sui dati:
  - roll up
  - drill down
  - data cube

## Data mining

- Tecniche algoritmiche che consentono di fare deduzioni sui dati
- Consente di svolgere ricerche sofisticate sui dati e di esplicitare relazioni "nascoste" tra i dati

## Export dei dati

- Consente l'esportazione dei dati da una DW ad un'altra (architettura gerarchica)

Moduli di **ausilio** alla progettazione e gestione di una DW:

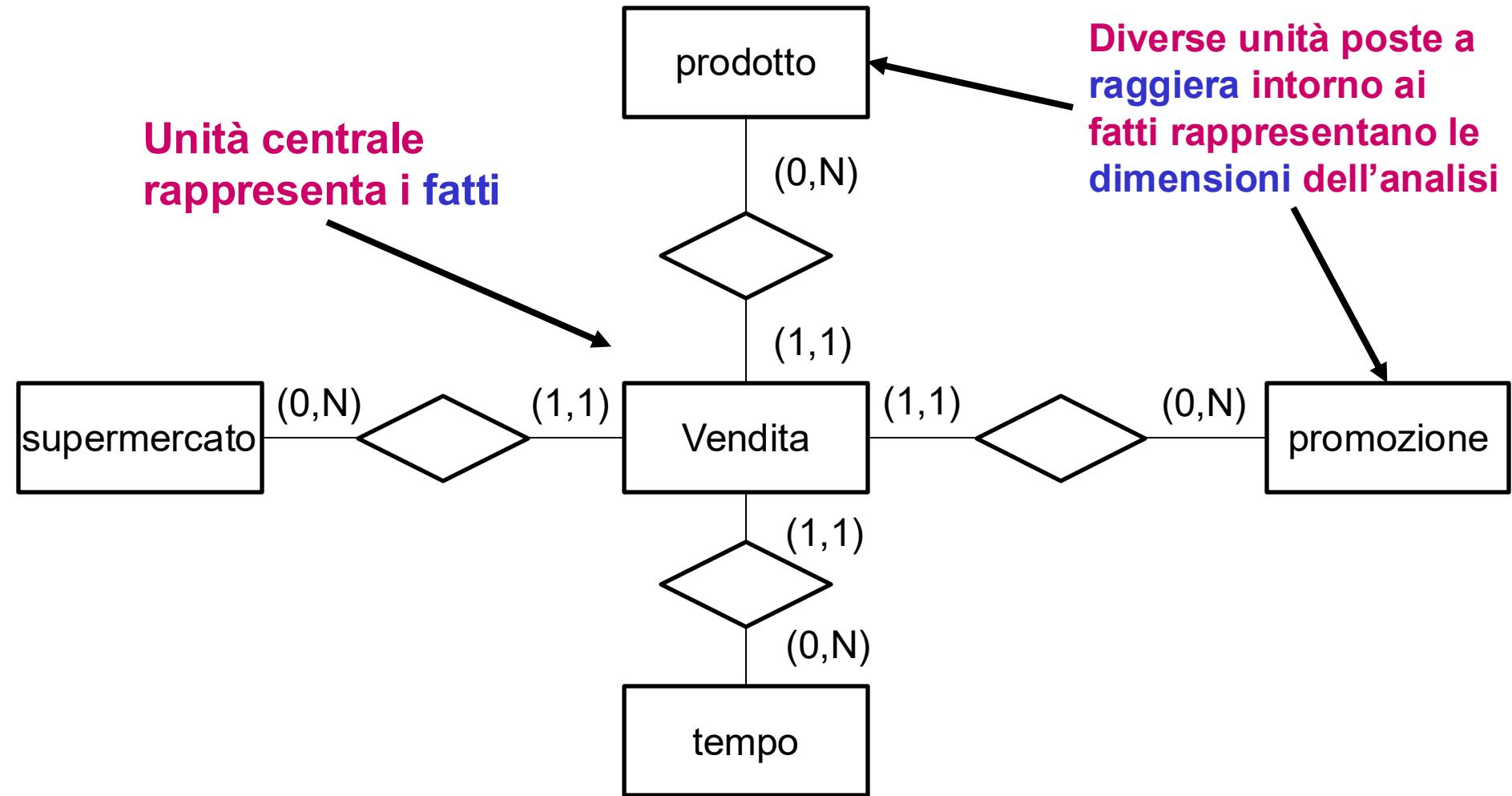
- un componente per l'assistenza allo **sviluppo** della DW, che permette di facilitare le definizione dello schema dei dati e i meccanismi per l'importazione dei dati
- un **dizionario** dei dati, che descrive il contenuto della DW, utile per comprendere quali analisi dei dati possono essere eseguite (glossario)

- Nel costruire una DW aziendale ci si concentra su sottoinsiemi molto semplici dei dati aziendali che si vogliono analizzare (**dati dipartimentali**)
- Ogni schema semplificato dei dati dipartimentali prende il nome di **data mart**
- L'organizzazione dei dati di un data mart avviene secondo uno schema multidimensionale (**schema a stella**)

# Schema a stella

**Unità centrale  
rappresenta i fatti**

**Diverse unità poste a  
raggiera intorno ai  
fatti rappresentano le  
dimensioni dell'analisi**



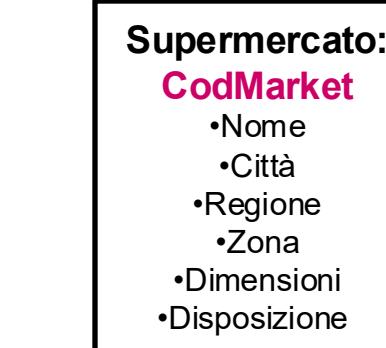
- Varie relazioni **uno a molti** collegano ciascuna occorrenza di fatto con **una ed una sola** occorrenza di ciascuna delle dimensioni
- Il fatto ha una chiave composta da attributi chiave delle dimensioni, gli altri attributi rappresentano le misure dei fatti e sono solitamente numerici
- La struttura è regolare e indipendente dal problema considerato
- Occorrono almeno due dimensioni altrimenti il problema degenera in una semplice gerarchia uno-molti
- Un numero elevato di dimensioni è **sconsigliato** perché la gestione dei fatti e l'analisi si complicano

# Schema a stella

Ciascuna occorrenza di vendita ha per identificatore i quattro codici:

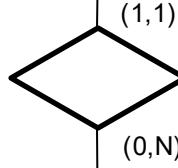
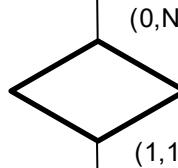
**CodProd**  
**CodMarket**  
**CodPromo**  
**CodTempo**

Gli attributi non chiave sono Amm e Qta.



**Prodotto: CodProd**

- Nome
- Categoria
- Marca
- Peso
- Fornitore



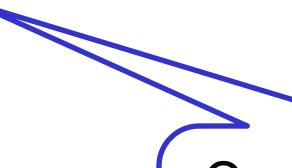
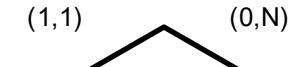
**Promozione:**  
**CodPromo**

- Nome
- Tipo
- Percentuale
- FlagCoupon
- DataInizio
- DataFine
- Costo
- Agenzia

**Tempo: CodTempo**

- GiornoSett
- GiornoMese
- GiornoAnno
- SettimanaMese
- SettimanaAnno
- MeseAnno...

(1,1)



Ogni occorrenza di vendita è un dato **aggregato**

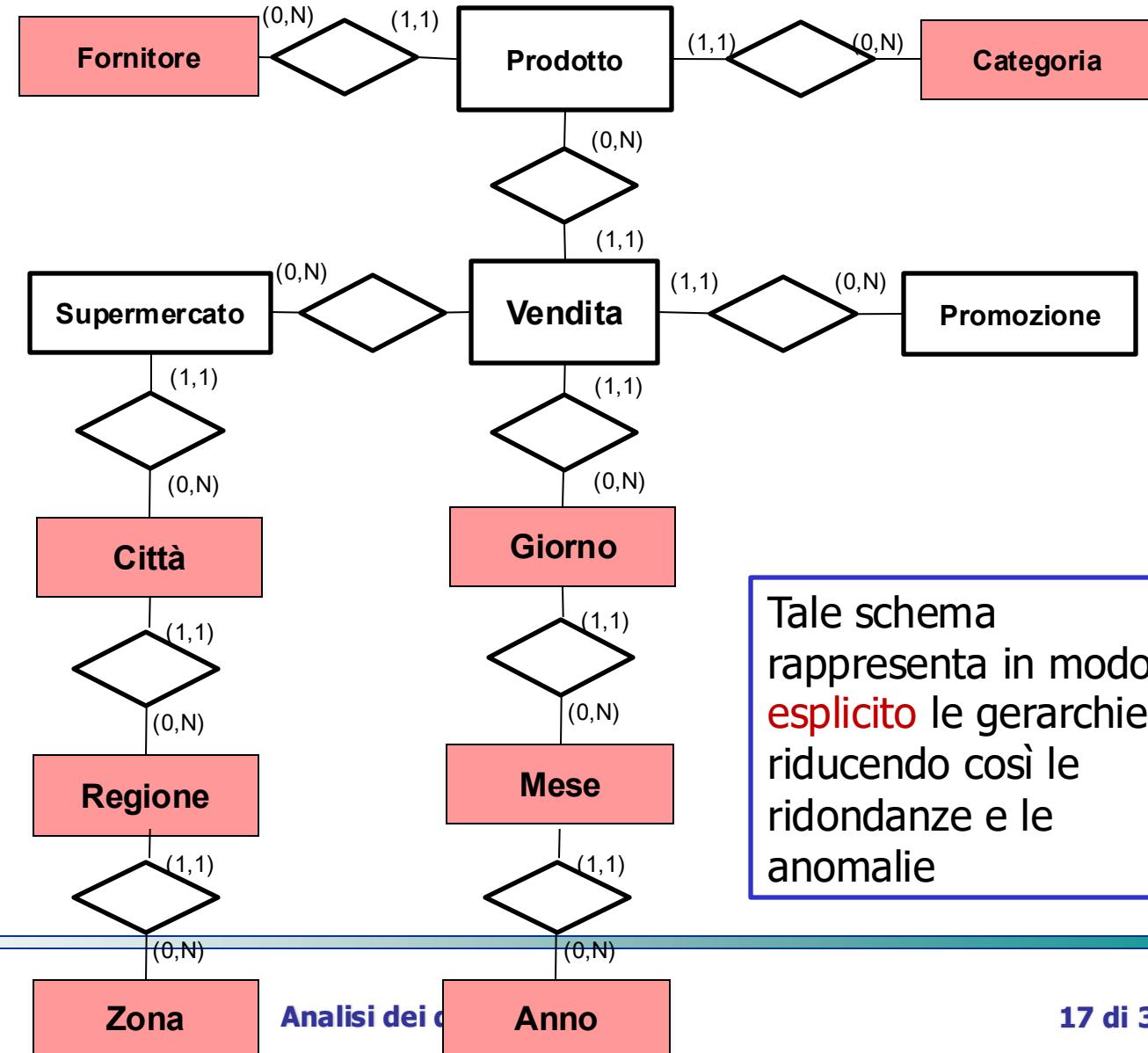


# Schema a stella

- Nella dimensione del tempo sono presenti dati **derivati** e **ridondanze**
- Le ridondanze servono per **facilitare** le operazioni di analisi dei dati
- I fatti sono in forma normale di **Boyce-Codd** in quanto ogni attributo non chiave dipende funzionalmente dalla sua unica chiave
- Le dimensioni sono in genere relazioni **non normalizzate**, si evitano operazioni di join

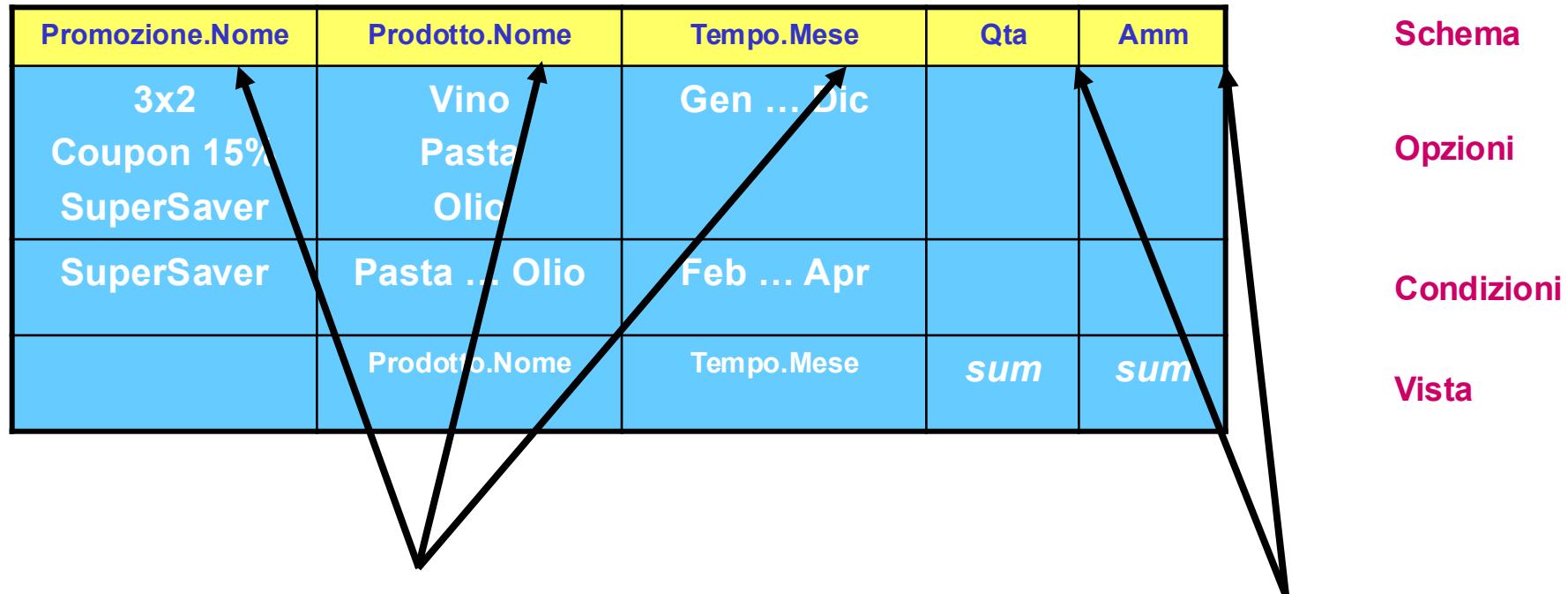
# Schema a fiocco di neve

Evoluzione dello schema a stella, introdotta per strutturare gerarchicamente le dimensioni **non normalizzate**



Tale schema rappresenta in modo **esplicito** le gerarchie, riducendo così le ridondanze e le anomalie

- Interfaccia standard di formulazione delle query
  - Drill down
  - Roll up
  - Data Cube (Slice-and-dice)
- L'analisi dei dati di un data mart organizzato a stella richiede l'estrazione di un **sottoinsieme** dei fatti e delle dimensioni
- Le **dimensioni** vengono usate per selezionare i dati e per raggrupparli
- I **fatti** vengono tipicamente aggregati
- È possibile costruire moduli predefiniti per estrarre i dati in cui si offrono scelte definite (selezioni, aggregazioni, valutazioni di funzioni aggregate)



## Attributi delle dimensioni:

- Promozione
- Prodotto
- Tempo

## Attributi dei Fatti:

- Aggregati (SUM)

# Interfaccia Standard di formulazione delle Query

Promozione.Nome	Prodotto.Nome	Tempo.Mese	Qta	Amm
3x2 Coupon 15% SuperSaver	Vino Pasta Olio	Gen ... Dic		
SuperSaver	Pasta ... Olio	Feb ... Apr		
	Prodotto.Nome	Tempo.Mese	<i>sum</i>	<i>sum</i>

Schema

Opzioni

Condizioni

Vista

```
select Tempo.Mese, Prodotto.Nome, sum(Amm), sum(Qta)
from Vendite, Tempo, Prodotto
where Vendite.CodTempo = Tempo.CodTempo
and Vendite.CodProdotto = Prodotto.CodProdotto
and (Prodotto.Nome = 'Pasta' or Prodotto.Nome = 'Olio')
and Tempo.Mese between 'Feb' and 'Apr'
and Promozione.Nome = 'SuperSaver'
group by Tempo.Mese, Prodotto.Nome
order by Tempo.Mese, Prodotto.Nome
```

Tempo.mese	Prodotto.nome	sum(Amm)	sum(Qta)

# Drill-down e Roll-up

- Il **drill down** permette di **aggiungere** una dimensione di analisi disaggregando i dati
- Il **roll up** dualmente consente di **eliminare** una dimensione di analisi
- L'operazione di **roll up** può essere fatta agendo sui risultati dell'interrogazione
- L'operazione di **drill down** richiede la riformulazione dell'interrogazione (servono dati non presenti nell'interrogazione)

# Drill-down e Roll-up: Esempio

Prodotto.Nome	Tempo.Mese	Qta
Vino	Gen ... Dic	
Pasta	Feb ... Apr	
Olio		
Pasta	Feb ... Apr	
Prodotto.Nome	Tempo.Mese	sum

Schema

Opzioni

Condizioni

Vista

somma delle  
quantità vendute  
di pasta nel  
trimestre Feb-Apr

Tempo.mese	Prodotto.Nome	Sum(Qta)
Feb	Pasta	46 Kg
Mar	Pasta	50 Kg
Apr	Pasta	51 Kg

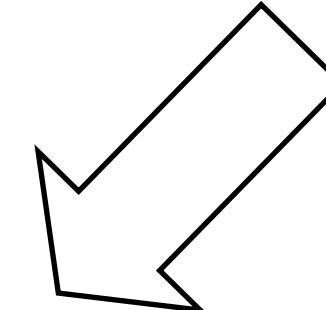
# Drill-down: Esempio

Il manager è interessato alle vendite per zona:

*Drill down on Zona*

Tempo.mese	Prodotto.Nome	Sum(Qta)
Feb	Pasta	46 Kg
Mar	Pasta	50 Kg
Apr	Pasta	51 Kg

Tempo.mese	Prodotto.Nome	Zona	Sum(Qta)
Feb	Pasta	Nord	18
Feb	Pasta	Centro	15
Feb	Pasta	Sud	13
Mar	Pasta	Nord	18
Mar	Pasta	Centro	18
Mar	Pasta	Sud	14
Apr	Pasta	Nord	18
Apr	Pasta	Centro	17
Apr	Pasta	Sud	16

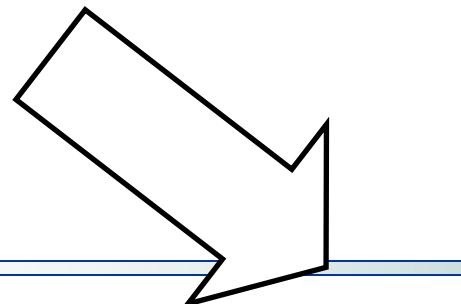


# Roll-up: Esempio

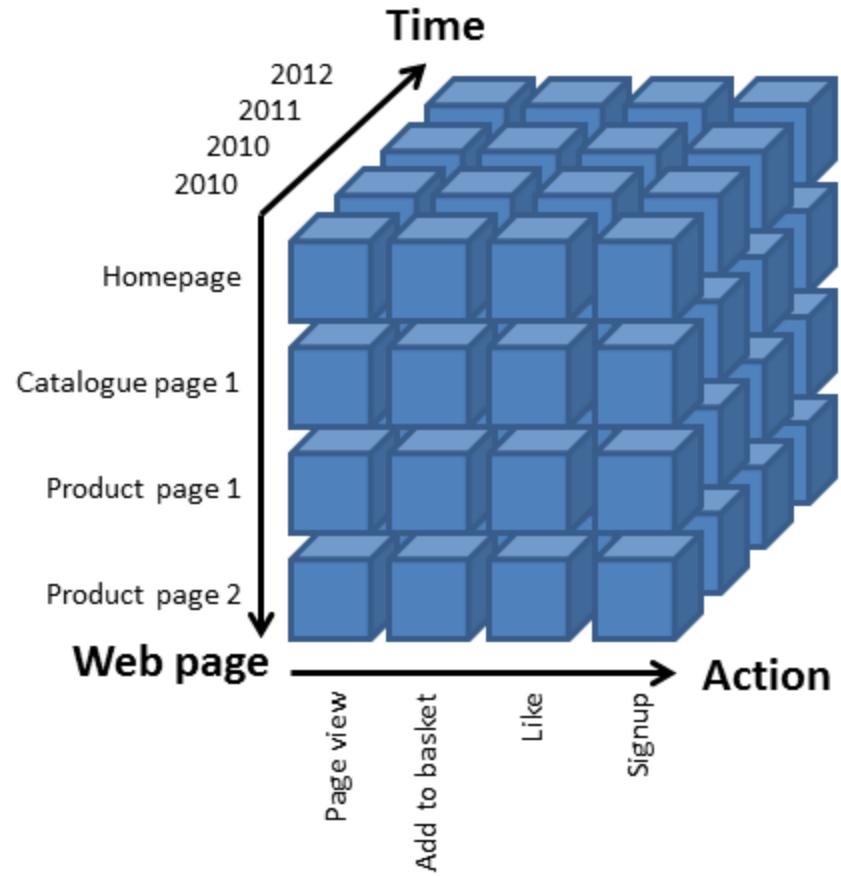
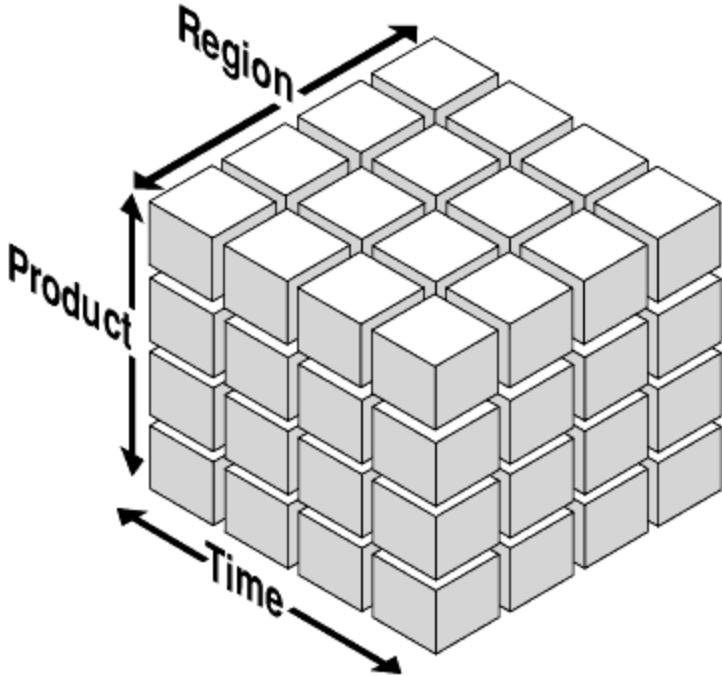
Tempo.mese	Prodotto.Nome	Zona	Sum(Qta)
Feb	Pasta	Nord	18
Feb	Pasta	Centro	15
Feb	Pasta	Sud	13
Mar	Pasta	Nord	18
Mar	Pasta	Centro	18
Mar	Pasta	Sud	14
Apr	Pasta	Nord	18
Apr	Pasta	Centro	17
Apr	Pasta	Sud	16

Il manager è interessato solo alla suddivisione delle vendite per zona:

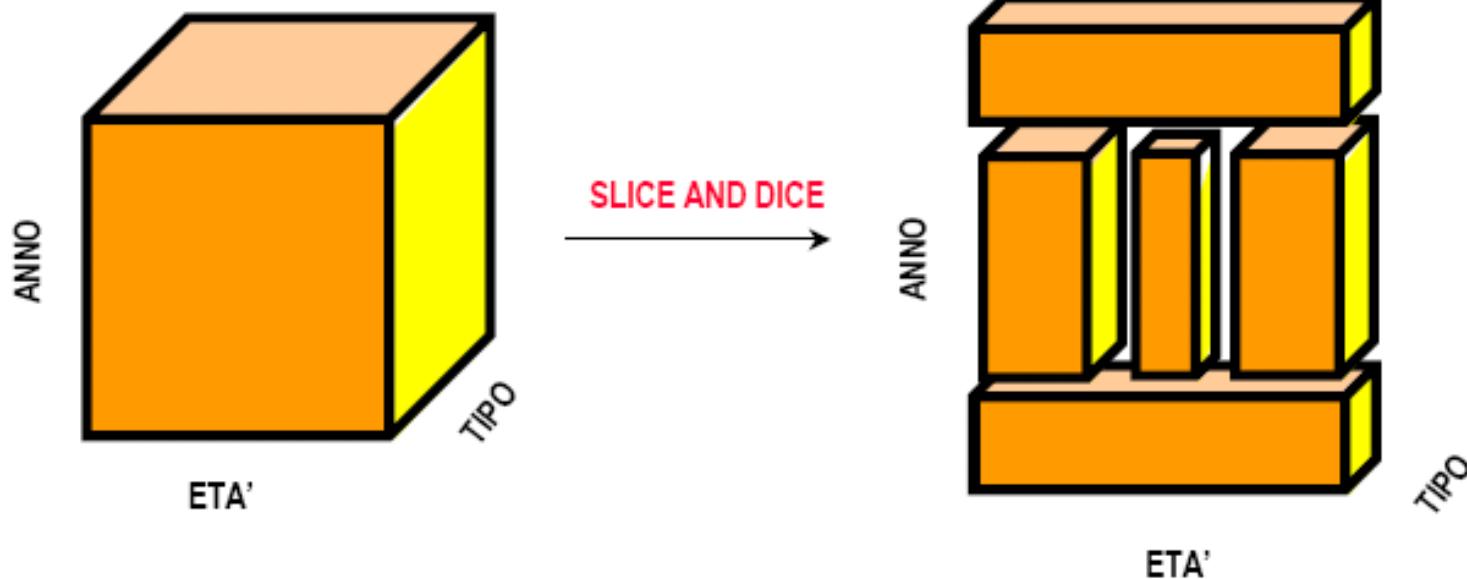
*Roll up on Mese*



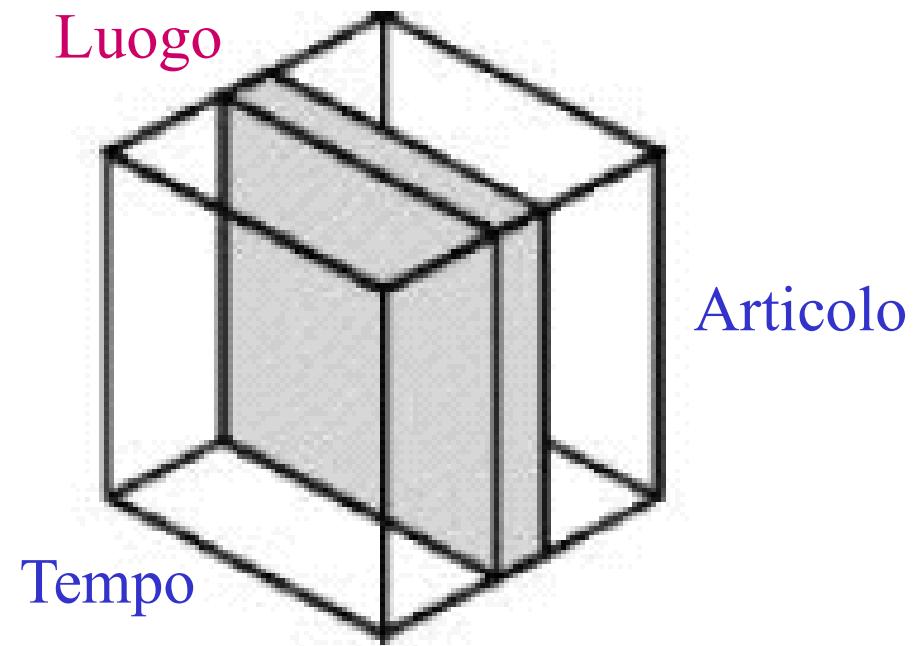
Zona	Prodotto.Nome	Sum(Qta)
Nord	Pasta	54 Kg
Centro	Pasta	50 Kg
Sud	Pasta	43Kg



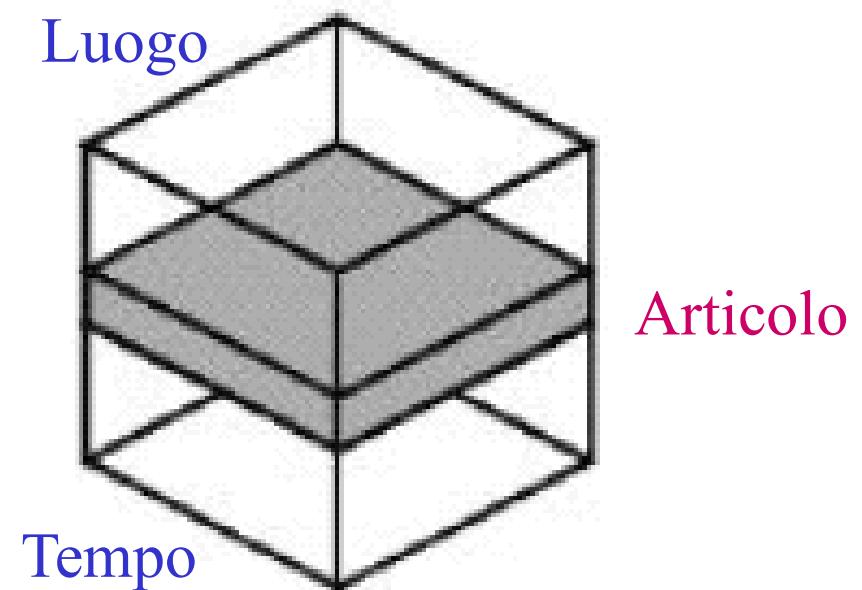
- Seleziona un **sottoinsieme** delle celle del un cubo, ottenuta “affettando e tagliando” a cubetti il cubo stesso.
- Seleziona e proietta riducendo la **dimensionalità** dei dati



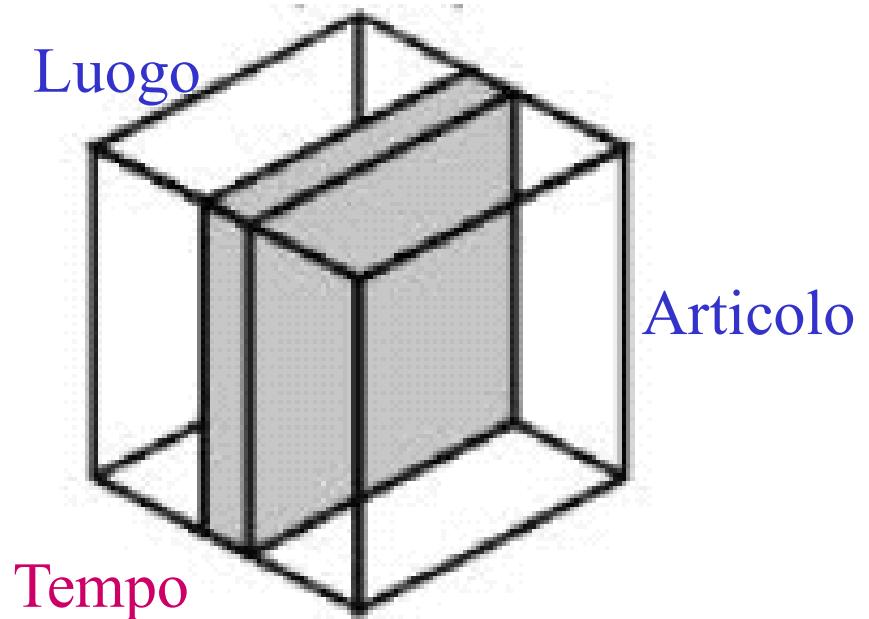
Il manager vuole effettuare un'analisi relativa alle vendite in **tutti** i periodi nella zona **Roma-2**



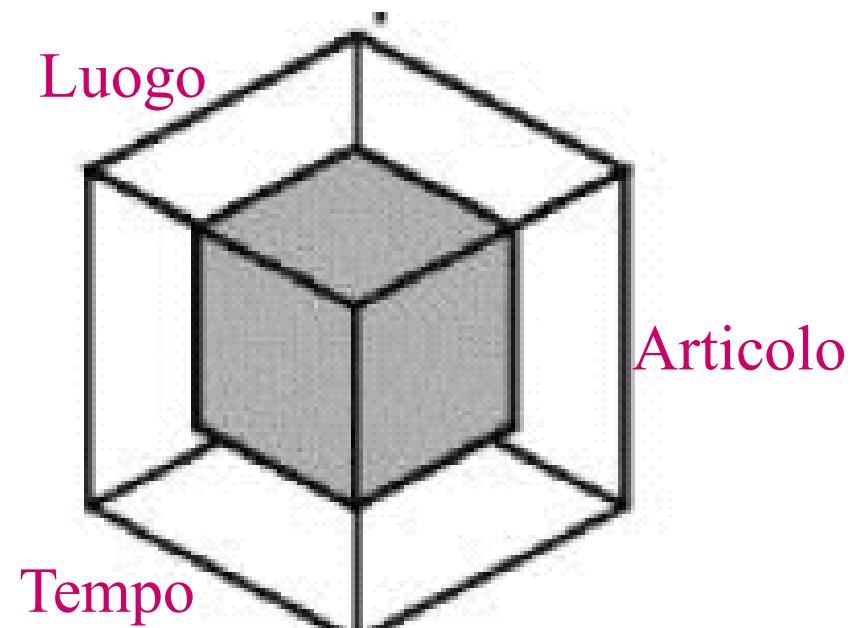
Il manager di prodotto esamina la vendita di **un particolare** prodotto in **tutti** i periodi e in **tutti** i mercati



Il manager finanziario esamina la vendita di **tutti** i prodotti in **tutti** i mercati relativamente ad un particolare **periodo**



Il manager strategico si concentra su **una** categoria di prodotti, **una area** regionale e **un orizzonte** temporale medio



- Ricerca di informazioni “nascoste” all’interno delle DW
- Esempi di utilizzo:
  - analisi di mercato (individuazione di oggetti acquistati assieme o in sequenza)
  - analisi di comportamento (frodi o usi illeciti delle carte di credito)
  - analisi di previsione (costo futuro delle cure mediche)

- 1. Comprensione** del dominio
- 2. Preparazione** sul set di dati: individuazione di un sottoinsieme dei dati della DW su cui effettuare il **mining** e loro codifica (input algoritmo)
- 3. Scoperta** dei pattern: ricerca e individuazione di **pattern** ripetitivi tra i dati
- 4. Valutazione** dei pattern: partendo dai pattern scoperti si valutano quali esperimenti compiere successivamente e quali ipotesi formulare o quali **conseguenze** trarre
- 5. Utilizzo** dei risultati: prendere **decisioni operative** a seguito del processo di data mining (allocazione merci, concessione credito)

- Scoprire associazioni di tipo causa-effetto
- **Basket Analysis:** analisi di dati raggruppati per transazioni di acquisto
- Una regola associativa consta di una premessa e di una conseguenza (Es: Pannolini → Birra )<sup>1</sup>
- E' possibile definire in modo preciso le probabilità associate alle regole di associazione
  - **supporto:** probabilità che in una osservazione sia presente sia la premessa che la conseguenza di una regola
  - **confidenza:** probabilità che in una osservazione sia presente la conseguenza di una regola essendo già presente la premessa

<sup>1</sup> <https://www.forbes.com/forbes/1998/0406/6107128a.html?sh=7daf50036260>

# Regole di associazione: Esempio

Nazionalità	Età	Stipendio
Italiana	50	Basso
Italiana	40	Alto
Francese	30	Alto
Italiana	50	Medio
Francese	45	Alto
Francese	35	Alto

1) Francese → Stip. Alto  
 supporto = 3/6 = 0.5  
 confidenza = 3/3 = 1

2) Stip. Alto → Francese  
 supporto = 3/6 = 0.5  
 confidenza = 3/4 = 0.75

3)  $\geq 40$  anni → Stip. Basso  
 supporto = 1/6 = 0.16  
 confidenza = 1/4 = 0.25

- Il problema di data mining relativo alla scoperta delle regole di associazione viene quindi enunciato come segue:

*Trovare tutte le regole di associazione con supporto e confidenza superiori a valori prefissati*



# Discretizzazione

- Consente di rappresentare un intervallo **continuo** di dati tramite pochi valori **discreti**
- Rendere più evidente il fenomeno sottoposto ad osservazione
- Esempio
  - $\text{stipendio} < 1000 \rightarrow \text{Basso}$
  - $1000 \leq \text{stipendo} < 2500 \rightarrow \text{Medio}$
  - $\text{stipendio} \geq 2500 \rightarrow \text{Alto}$

- Catalogazione di un fenomeno in una **classe predefinita** attraverso algoritmi di classificazione (es. alberi decisionali)
- Quando i fenomeni sono descritti da un gran numero di attributi i classificatori si occupano di determinare gli attributi **significativi**, separandoli da quelli irrilevanti

