



ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA

## Dispensa di "*Basi Di Dati*"

A.A. 2024/25

Ivan De Simone

# Indice

Introduzione ai database	3
Modello dati relazionale	17
Algebra relazionale e calcolo relazionale	32
SQL base	56
SQL avanzato	92
Modellazione concettuale dei dati	106
Progettazione concettuale	128
Progettazione logica	144
Normalizzazione	158
Database attivi	170

# Introduzione ai database

Verrà effettuata un'analisi dal punto di vista metodologico e tecnologico riguardante i seguenti contenuti:

- Modelli per organizzare i dati
- Linguaggi per utilizzare i dati
- Sistemi per gestire i dati
- Metodologie per la progettazione di database

## Sistema informativo

Un componente di un'istituzione che gestisce le porzioni interessanti delle informazioni (ad esempio, per perseguire gli obiettivi aziendali).

Ogni istituzione ha il suo personale sistema informativo, che potrebbe non essere esplicitamente riconoscibile all'interno dell'organizzazione.

Il sistema informativo supporta altri "sotto-sistemi". Per questa ragione dovrebbe essere studiato dentro il suo contesto operativo.

## Gestire le informazioni

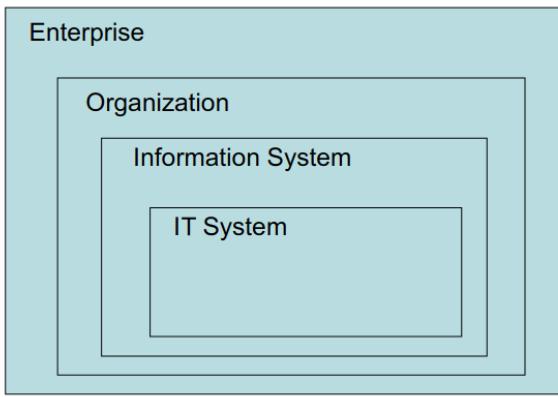
- Raccolta, acquisizione
- Memorizzazione, conservazione
- Elaborazione, trasformazione, produzione
- Distribuzione, comunicazione, scambio

## Automazione

L'idea di "sistema informativo" è indipendente da qualsiasi automazione informatica. Alcune organizzazioni hanno il solo scopo di gestire dati e sono esistite per secoli, ad esempio raccolta demografica e servizi bancari.

## Sistema IT

Una delle parti automatizzate di un sistema informativo è il componente che gestisce le informazioni usando la tecnologia dei computer.



## Informazioni e dati

Attività umane differenti possono rappresentare pezzi di informazione diversamente:

- idee informali
- linguaggio naturale (scritto o parlato, formale o conversazionale, in diverse lingue...)
- disegni, grafici, diagrammi
- numeri e codici

Tali informazioni hanno bisogno di diversi supporti di memorizzazione: cervello umano, carta, dispositivi elettronici.

## Dati

All'interno dei sistemi IT, l'informazione è approssimativamente espressa come **dati**.

*Dall'Oxford Dictionary:*

**Informazione:** fatto fornito o imparato riguardante qualcosa o qualcuno.

**Dato:** pezzo di informazione. Un punto di partenza fisso di un'operazione.



Cosa significano quei numeri? Segnali stradali in Finlandia, sono ore.

Qual è la differenza tra di essi? Il dato è inutile senza la sua interpretazione.

I dati sono il risultato del processo di organizzazione, programmazione e memorizzazione dell'informazione.

Ad esempio, all'interno dei registri demografici, quando ci si riferisce ai cittadini: descrizioni a parole, nome e cognome, informazioni personali, codice fiscale...

## Perché i dati?

È difficolto rappresentare precisamente grandi pezzi di informazione e conoscenza.  
I dati sono una risorsa strategica, poiché sono più stabili delle altre rappresentazioni (come processi aziendali, tecnologie, ruoli umani).

## Database

*Definizione generica:*

Un insieme organizzato di dati che supporta lo svolgimento di attività specifiche di un'entità (un'istituzione, un'impresa, un ufficio, un essere umano).

*Definizione formale:*

Un insieme di dati gestito da un DBMS.

## DataBase Management System

Un **DBMS** è un sistema che gestisce collezioni di dati grandi, persistenti e condivise, assicurando privacy, affidabilità, efficienza ed efficacia.

Software *as a product* e *as a service* (complessi) disponibili sul mercato: DB2, Oracle, SQLServer, MySQL, PostgreSQL, Access, BigQuery, SQLite.

## DBMS vs File System

Grandi raccolte di dati potrebbero essere gestite anche da sistemi più semplici, come il file system del sistema operativo. I file system forniscono un accesso grossolano ai dati: "tutto o niente". I DBMS estendono le funzionalità del file system, fornendo più servizi e in modo più integrato.

## Caratteristiche dei database

### Grandezza

La loro dimensione è (di molto) superiore alla memoria centrale di un sistema di elaborazione. Solo il limite fisico dei dispositivi è considerato.

Alcuni esempi di grandi dimensioni: 1-5 Terabyte (transactional data), 30-50 Terabyte (decision data), 500-800 Terabyte (scientific data), 100 billions of records.

## Persistenza

Il loro tempo di vita è indipendente dal tempo di vita dei processi sul computer che usano tali dati.

## Condivisione

Ogni grande organizzazione è divisa in differenti aree, che potrebbero svolgere attività diverse. Ciascuna area/attività ha un sotto-sistema informativo, che non è necessariamente scollegato da quello centrale.

Problemi: **ridondanza**, ovvero gli stessi dati appaiono più volte, la quale può causare **inconsistenza**, cioè descrizioni differenti per gli stessi dati.

Un database è una risorsa integrata e condivisa tra più applicazioni software. Come conseguenza:

- Differenti attività si basano su dati condivisi: serve un meccanismo di permessi.
- Numerosi utenti accedono ai dati condivisi: servono controlli di concorrenza.

## Privacy

Potremmo implementare un meccanismo di permessi del tipo:

- "L'utente A è autorizzato a leggere tutti i dati e a modificare i dati X"
- "L'utente B è autorizzato a leggere i dati X e a modificare i dati Y"

## Affidabilità

Intesa come tolleranza ai guasti sia hardware che software. Un database è una risorsa preziosa e di conseguenza dovrebbe essere preservata per lunghi tempi. Una tecnica cruciale è la gestione delle **transazioni**.

## Transazione

Un insieme di operazioni **atomiche** che sono corrette anche su un sistema **concorrente** con effetti **permanenti**.

**Atomicità**: una sequenza di operazioni collegate...

*"Spostare i soldi dal conto corrente A a B: o il prelievo viene effettuato su A e trasferito su B oppure non viene eseguita alcuna operazione"*

... viene effettuata interamente o non effettuata per nulla.

**Concorrenza:** transazioni concorrenti devono essere coerenti:

Se due assegni diversi vengono emessi su uno stesso conto bancario e vengono incassati contemporaneamente... non devi perderne uno.

Se due agenzie diverse vogliono prenotare lo stesso posto (disponibile) su un treno... non devi prenotarlo due volte.

**Permanenza:** il *commit* di una transazione richiede che il risultato finale sia registrato e tracciato, anche in un contesto concorrente o quando si verificano errori.

## Efficienza

Cercare di utilizzare in modo proficuo le risorse di memoria (*primaria* e *secondaria*) e tempo (esecuzione e risposta).

Poiché i DBMS forniscono molte operazioni, potrebbero essere implementate in modo inefficiente.

Per questo motivo ci sono enormi investimenti e concorrenza.

L'efficienza è anche un risultato della qualità del software.

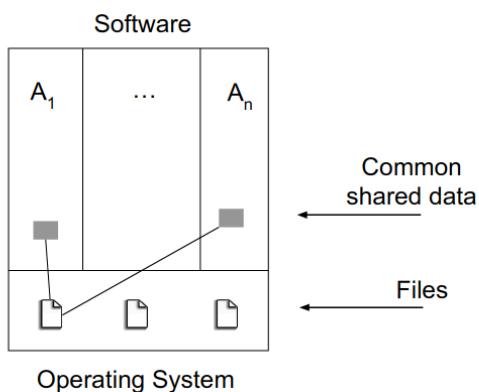
## Efficacia

Cercare di migliorare le attività degli utenti, fornendo funzionalità potenti e flessibili.

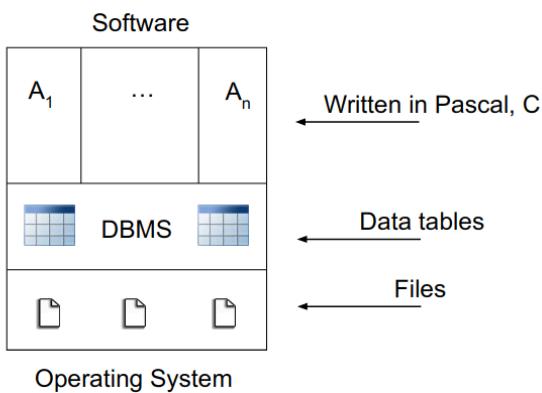
Un argomento frequente nella letteratura è come i DBMS persegono l'efficacia.

# Evoluzione della gestione dei dati

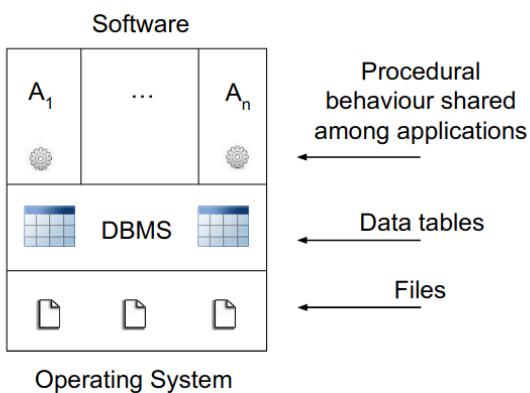
## Anni '70: nessun DBMS



## Anni '80: primi DBMS



## Anni '90: comportamento procedurale



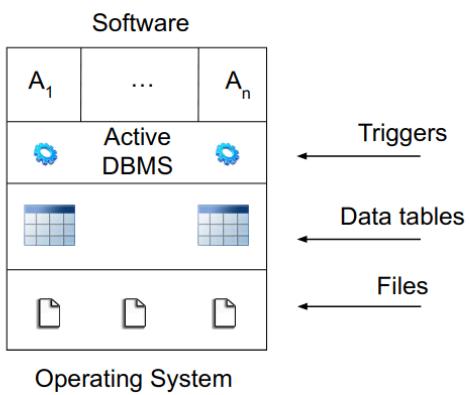
## Avanzamenti nei DBMS

Le stored procedure sono state introdotte per condividere i comportamenti procedurali comuni tra software diversi.

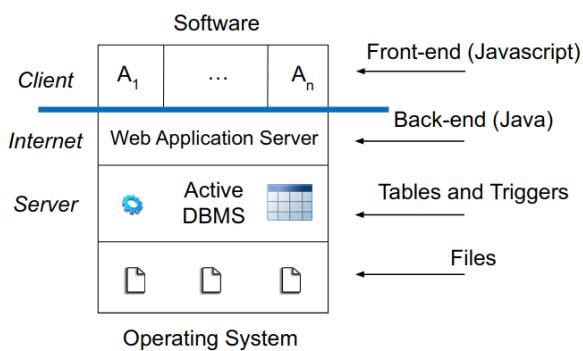
Le stored procedure non sono standardizzate e sono affette dal problema di "impedance mismatch" (vedremo più avanti) con il linguaggio utilizzato per esprimere tali procedure.

Di conseguenza sono state introdotte regole specifiche (*trigger*) per modellare il comportamento procedurale condiviso tra software diversi che sono gestite dal DBMS stesso.

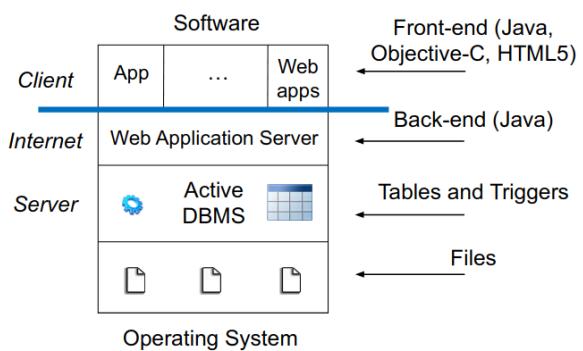
## Anni '90: DBMS attivi



## Anni 2000: web



## Anni 2010: applicazioni mobili



## Descrivere i dati

Ogni software (che non si basa su un DBMS) contiene una descrizione interna della struttura del file che verrà elaborato. Come svantaggio, più software potrebbero avere più descrizioni dei dati: questa situazione genera inconsistenza nelle rappresentazioni dei dati.

Nei DBMS, una parte del database (il catalogo o dizionario) contiene una descrizione centralizzata dei dati: la stessa descrizione è condivisa tra tutte le applicazioni software.

# Descrizione nei DBMS

I dati sono descritti a diversi livelli di astrazione.

I dati non sono dipendenti dalla loro rappresentazione fisica: i programmi utilizzano una rappresentazione di alto livello, indipendente dalle rappresentazioni dei dati a basso livello. Di conseguenza, un cambiamento nel livello inferiore non richiede cambiamenti nei programmi. Ciò è ottenuto più precisamente tramite il concetto di **modello di dati**.

## Modello di dati

Un insieme di costrutti che vengono utilizzati per organizzare e descrivere il comportamento dei dati. È un componente cruciale: **fornisce una struttura ai dati** (tramite un costruttore di tipo).

Il modello di dati fornisce alcuni costruttori di tipo di dati predefiniti, come di solito accade nei linguaggi di programmazione attuali.

Il **modello relazionale** fornisce il costruttore *relazione*, consentendo di definire un insieme di record omogenei.

## Schema e istanze

In ogni database esiste:

- lo **schema**: è invariante nel tempo e descrive la struttura dei dati (aspetto intensionale) → le intestazioni della tabella
- l'**istanza**: i valori effettivi che potrebbero cambiare rapidamente (aspetto estensionale) → il corpo della tabella

SCEDULING

Lectures	Lecturer	Room	Time
Calculus I	Luigi Neri	N1	8:00
Databases	Pier Rossi	N2	9:45
Chemistry	Nicola Mori	N1	9:45
Physics I	Mario Bruni	N1	11:45
Physics II	Mario Bruni	N3	9:45
IT Systems	Pier Rossi	N3	8:00

## Modellazione concettuale

I modelli concettuali rappresentano i dati indipendentemente dal sistema specifico. Descrivono concetti del mondo reale e sono utilizzati nella fase preparatoria di un progetto.

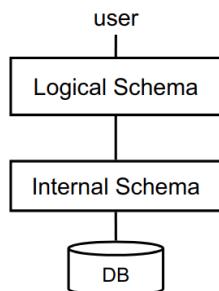
Il linguaggio di modelli di dati più diffuso è **Entità-Associazione (ER)**.

# Modelli logici

I DBMS utilizzano tali modelli per memorizzare e organizzare i dati. Sono utilizzati dal software a livello superiore e sono indipendenti dalla rappresentazione fisica.

Alcuni esempi sono **relazionale**, gerarchico, a oggetti, XML.

## Architettura semplificata di un DBMS



**Schema logico**: descrive la struttura del database attraverso il modello di dati logico (la struttura della tabella).

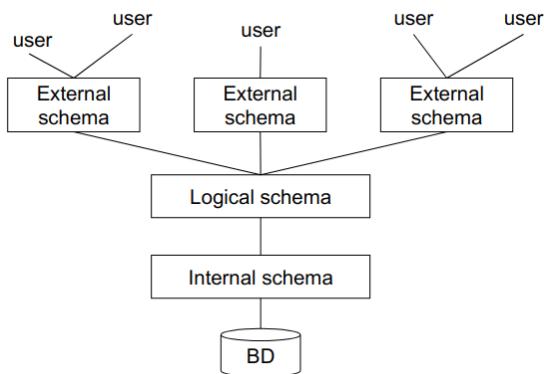
**Schema interno (o fisico)**: rappresenta lo schema logico a livello di archiviazione utilizzando specifiche strutture dati *raw* (come file, record e puntatori di dati).

## Indipendenza dei dati

La rappresentazione dei dati a livello logico è indipendente da quella a livello fisico. Una tabella viene gestita "così com'è", indipendentemente dalla sua rappresentazione sottostante (che potrebbe anche variare nel tempo).

Per tale motivo in questo corso vedremo solo il livello logico e non quello interno.

# Architettura standard a tre livelli ANSI/SPARC



Uno **schema interno** rappresenta il livello concettuale a livello di archiviazione utilizzando specifiche strutture dati *raw* (come file, record e puntatori di dati).

Uno **schema logico** descrive la struttura del database tramite il modello di dati logico "principale".

Uno **schema esterno** descrive parte del database in un modello logico (viste parziali, tabelle derivate, anche da modelli diversi).

## Viste

LECTURES

Lecture	Lecturer	Room
Databases	Rossi	DS3
Systems	Neri	N3
Networks	Bruni	N3
Controls	Bruni	G

ROOMS

Name	Building	Floor
DS1	OMI	Ground
N3	OMI	Ground
G	A2	First

LECTURESROOMS

Lecture	Room	Building	Floor
Systems	N3	OMI	Ground
Networks	N3	OMI	Ground
Controls	G	A2	First

## Indipendenza dei dati

È la conseguenza dell'avere diversi livelli. L'accesso ai dati è fornito dal livello esterno, che potrebbe coincidere con quello logico. Due tipi di indipendenza:

- Indipendenza **fisica** dei dati
- Indipendenza **logica** dei dati

### Indipendenza fisica dei dati

I livelli logico ed esterno hanno una rappresentazione indipendente dal livello fisico:

- una relazione è "gestita" alla stessa maniera indipendentemente dalla sua implementazione fisica
- l'implementazione fisica potrebbe subire alcune modifiche senza la necessità di cambiare il software

## Indipendenza logica dei dati

Il livello esterno è indipendente dal livello logico:

- le modifiche e gli aggiornamenti sulle viste non richiedono alcuna modifica a livello logico
- le modifiche "trasparenti" dello schema logico mantengono inalterato lo schema esterno

## Linguaggi per database

Un altro aspetto dell'efficacia dei database è dato dalla loro disponibilità attraverso diversi linguaggi ed interfacce:

- linguaggi testuali "interattivi" (**SQL**)
- istruzioni (SQL) iniettate in un linguaggio *host* (Pascal, Java, C, ...)
- istruzioni (SQL) iniettate in un linguaggio *ad hoc* con altre funzionalità (ad esempio grafici, tabelle di stampa)
- interfacce user friendly (senza interfaccia testuale)

## SQL

LECTURES			ROOMS		
Lecture	Lecturer	Room	Name	Building	Floor
Databases	Rossi	DS3	DS1	OMI	Ground
Systems	Neri	N3	N3	OMI	Ground
Networks	Bruni	N3	G	A2	First
Controls	Bruni	G			

Ritorna tutte le lezioni tenute al piano terra.

```
SELECT Lecture, Room, Floor  
FROM Rooms, Lectures  
WHERE Name = Room AND Floor = 'Ground'
```

Lecture	Room	Floor
Systems	N3	Ground
Networks	N3	Ground

## SQL iniettato in un linguaggio host

```
write('name of the city?'); readln(citta);
EXEC SQL DECLARE P CURSOR FOR
  SELECT NAME, INCOME
  FROM PEOPLE
  WHERE CITY = :city ;
EXEC SQL OPEN P ;
EXEC SQL FETCH P INTO :name, :income ;
while SQLCODE = 0 do begin
  write('name of the person:', name, 'rise?');
  readln(rise);
  EXEC SQL UPDATE PEOPLE
    SET INCOME = INCOME + :rise
    WHERE CURRENT OF P
  EXEC SQL FETCH P INTO :name, :income
end;
EXEC SQL CLOSE CURSOR P
```

## SQL ad hoc (Oracle PL/SQL)

```
declare Stip number;
begin
  SELECT SALARY INTO STIP FROM EMPLOYEE
  WHERE NUMBER = '575488' FOR UPDATE OF SALARY;
  if Stip > 30 then
    UPDATE EMPLOYEE SET SALARY = SALARY * 1.1
    WHERE NUMBER = '575488';
  else
    UPDATE EMPLOYEE SET SALARY = SALARY * 1.15
    WHERE NUMBER = '575488';
  end if;
  commit;
exception
  when no_data_found then
    INSERT INTO ERRORS
      VALUES( 'NUMBER DOES NOT EXIST', SYSDATE );
end;
```

# Interfaccia user friendly



## Separazione dei dati dal software

- **Data Definition Language (DDL)**: viene utilizzato per definire gli *schemi* (logici, esterni, fisici) ed altre operazioni.
- **Data Manipulation Language (DML)**: utilizzato per interrogare ed aggiornare (*istanze di*) database.

## Un'operazione DDL

Effettuata sullo schema.

```
CREATE TABLE hours (
    course CHAR(20),
    teacher CHAR(20),
    room CHAR(4),
    hour CHAR(5)
)
```

## Attori e utenti

- Progettisti e sviluppatori di DBMS
- Progettisti e sviluppatori di database
- Amministratori (DBA)
- Progettisti e sviluppatori software di applicazioni per utenti finali
- Utenti
  - utenti finali: eseguono insiemi predefiniti di operazioni (transazioni)
  - utenti occasionali: eseguono operazioni che non sono definite a priori, utilizzando linguaggi interattivi

# Amministratori di database (DBA)

Una persona specifica o un gruppo di persone che sono responsabili del controllo e della gestione del database in modo centralizzato (per efficienza, affidabilità, permessi).

Un **DBA** spesso progetta anche il database, tranne che per alcuni progetti complessi.

## Transazioni: due punti di vista

**Dell'utente:** qualsiasi software disponibile che implementa operazioni frequenti e ben note, definite a priori, con pochissime eccezioni previste.

Esempi:

- versare denaro sul proprio conto
- emettere un certificato di nascita
- una registrazione all'anagrafe
- prenotare un volo

Le transazioni sono implementate in uno specifico linguaggio host (noto oppure "ad hoc").

**Del sistema:** sequenza indistruttibile di operazioni.

## Pro e contro del DBMS

### PRO

- dati come risorsa condivisa, database che modellano l'ambiente
- gestione centralizzata dei dati standardizzabile e scalabile
- fornisce servizi integrati
- riduce ridondanze ed inconsistenze nei dati
- indipendenza dei dati (supportando lo sviluppo e la gestione delle applicazioni software)

### CONTRO

- i prodotti possono essere costosi così come l'adozione di tali soluzioni
- le funzionalità non sono separabili (l'efficienza è ridotta)

# Modello dati relazionale

## Introduzione

### Modelli logici

Ci sono tre modelli logici tradizionali:

- Gerarchico
- A grafo/a rete
- Relazionale

Più recenti sono il modello orientato agli oggetti (poco comune) e il modello basato su XML (complementare al modello relazionale).

### Caratteristiche

I modelli *gerarchico* e *a rete* usano riferimenti esplicativi tra i records (puntatori, esempio del web).

Il modello *relazionale* è "basato sui valori" (value-based). I riferimenti tra i dati memorizzati all'interno delle relazioni sono rappresentati tramite valori.

### Esempio di relazionale

STUDENT	Number	Surname	Name	BirthD
	6554	Rossi	Mario	1978/12/05
	8765	Neri	Paolo	1976/11/03
	9283	Verdi	Luisa	1979/11/12
	3456	Rossi	Maria	1978/02/01

EXAM	Student	Grade	Lecture
	3456	30	04
	3456	24	02
	9283	28	01
	6554	26	01

LECTURE	Code	Name	Lecturer
	01	Maths	Mario
	02	Chemistry	Bruni
	04	Chemistry	Verdi

# Il modello dati relazionale

Definito da E.F. Codd nel 1970 per consentire l'indipendenza dei dati.

Implementato in DBMS reali nel 1981 (non è facile implementare l'indipendenza dei dati in maniera sia efficiente che affidabile).

Si basa sulla definizione logica di "relazione" (con alcune differenze).

Le relazioni sono rappresentate tramite tabelle.

## Relazioni ed associazioni

- **Relazione logica** come nella Teoria degli insiemi
- **Relazione** come nel modello dati relazionale
- **Associazione** esprime una specifica classe di informazioni nel modello Entità-Associazione

## Relazione logica

$$D_1 = \{a,b\}$$

$$D_2 = \{x,y,z\}$$

$$\text{cartesian product } D_1 \times D_2$$

a	x
a	y
a	z
b	x
b	y
b	z

$$\text{a relation } r \subseteq D_1 \times D_2$$

a	x
a	z
b	y

Dati  $D_1, \dots, D_n$  n insiemi non necessariamente distinti:

**Prodotto cartesiano**  $D_1 \times \dots \times D_n$ :

l'insieme di tutte le tuple  $(d_1, \dots, d_n)$  tali che  $d_1 \in D_1, \dots, d_n \in D_n$ .

**Relazione logica** su  $D_1, \dots, D_n$ : un sottoinsieme di  $D_1 \times \dots \times D_n$ .

$D_1, \dots, D_n$  sono i domini della relazione.

## Proprietà

Una **relazione logica** è un insieme di tuple ordinate  $(d_1, \dots, d_n)$  tali che  $d_1 \in D_1, \dots, d_n \in D_n$

Una **relazione** è un insieme in cui:

- non c'è ordine tra le tuple
- le tuple sono tutte distinte
- ogni tupla n-esima è ordinata: l'i-esimo valore è preso dal i-esimo dominio

## Esempio di relazione logica

**Matches  $\subseteq$  string  $\times$  string  $\times$  int  $\times$  int**

Barca	Bayern	3	1
Bayern	Real	2	0
Barca	Psg	0	2
Psg	Real	0	1

Ogni dominio appare con due diversi ruoli, che possono essere riconosciuti in base alla loro posizione. Questa struttura è **posizionale**.

## Struttura dati non posizionale

Ciascun nome univoco nella tabella (attributo) è associato a un dominio. L'attributo fornisce il "ruolo" del dominio.

Home	Away	GoalsH	GoalsA
Barca	Bayern	3	1
Bayern	Real	2	0
Barca	Psg	0	2
Psg	Real	0	1

La posizione specifica di ogni attributo nello schema della tabella è irrilevante: la struttura dati è **non posizionale**.

## Esempio di struttura dati non posizionale

Home	Away	GoalsH	GoalsA
Barca	Bayern	3	1
Bayern	Real	2	0
Barca	Psg	0	2
Psg	Real	0	1

Away	Home	GoalsA	GoalsH
Bayern	Barca	1	3
Real	Bayern	0	2
Psg	Barca	2	0
Real	Psg	1	0

## Tabelle e relazioni

Una tabella rappresentante una relazione:

- ogni riga può assumere qualunque posizione
- ogni colonna può assumere qualunque posizione

Una tabella rappresenta una relazione se e solo se:

- tutte le righe sono differenti
- tutte le intestazioni delle colonne sono differenti
- i valori all'interno di una stessa colonna sono omogenei

## Il modello dati è basato sui valori

I riferimenti tra i dati memorizzati in relazioni diverse sono rappresentati tramite valori nelle tuple.

### Esempio

STUDENT	Number	Surname	Name	BirthD
	6554	Rossi	Mario	1978/12/05
	8765	Neri	Paolo	1976/11/03
	9283	Verdi	Luisa	1979/11/12
	3456	Rossi	Maria	1978/02/01

EXAM	Student	Grade	Lecture
	3456	30	04
	3456	24	02
	9283	28	01
	6554	26	01

LECTURE	Code	Name	Lecturer
	01	Maths	Mario
	02	Chemistry	Bruni
	04	Chemistry	Verdi

15

## Un'altra opzione

Alcuni modelli (come quello a rete, gerarchico, orientato agli oggetti) usano riferimenti esplicativi, gestiti dal sistema.

### Esempio value-based

STUDENT	Number	Surname	Name	BirthD
	6554	Rossi	Mario	1978/12/05
	8765	Neri	Paolo	1976/11/03
	9283	Verdi	Luisa	1979/11/12
	3456	Rossi	Maria	1978/02/01

EXAM	Student	Grade	Lecture
	3456	30	04
	3456	24	02
	9283	28	01
	6554	26	01

LECTURE	Code	Name	Lecturer
	01	Maths	Mario
	02	Chemistry	Bruni
	04	Chemistry	Verdi

### Esempio puntatori

STUDENT	Number	Surname	Name	BirthD
	6554	Rossi	Mario	1978/12/05
	8765	Neri	Paolo	1976/11/03
	9283	Verdi	Luisa	1979/11/12
	3456	Rossi	Maria	1978/02/01

EXAM	Student	Grade	Lecture
	3456	30	04
	3456	24	02
	9283	28	01
	6554	26	01

LECTURE	Code	Name	Lecturer
	01	Maths	Mario
	02	Chemistry	Bruni
	04	Chemistry	Verdi

17

18

# PRO di una struttura dati basata sui valori

- Indipendente dalla struttura fisica del dato che può cambiare dinamicamente (la stessa cosa potrebbe essere fornita con puntatori "ad alto livello").
- Gli unici dati che vengono memorizzati sono quelli rilevanti dal punto di vista dell'applicazione software.
- L'utente ed il programmatore vedono gli stessi dati.
- I dati possono essere facilmente condivisi tra più ambienti.
- I puntatori sono direzionali.

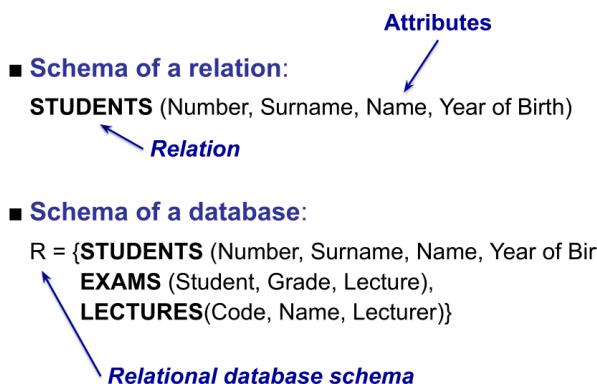
## Definizioni

### Schema di una relazione:

Una relazione  $R$  con un set di attributi  $A_1, \dots, A_n$ :  $R(A_1, \dots, A_n)$

### Schema di un database

Un insieme di schema (relazioni):  $R = \{R_1(X_1), \dots, R_k(X_k)\}$



Una **tupla** su un insieme di attributi  $X$  è una mappatura da ogni attributo  $A$  in  $X$  a un valore nel dominio di  $A$ .

$t[A]$  esprime il valore dell'attributo  $A$  in una tupla  $t$ .

Es. se  $t$  è la prima tupla in STUDENTS, allora  $t[Name] = Mario$

Una **istanza di relazione**  $r$  è un insieme finito di tuple aventi uno schema  $R(X)$ .

Una **istanza di database relazionale** con uno schema  $R = \{R_1(X_1), \dots, R_n(X_n)\}$  è un insieme di istanze di relazioni  $r = \{r_1, \dots, r_n\}$  (con  $r_i$  relazione su  $R_i$ ).

## Esempio di istanza di relazione

STUDENT	Number	Surname	Name	BirthD
6554	Rossi	Mario	1978/12/05	
8765	Neri	Paolo	1976/11/03	
9283	Verdi	Luisa	1979/11/12	
3456	Rossi	Maria	1978/02/01	

## Esempio di istanza di database relazionale

STUDENT	Number	Surname	Name	BirthD
	6554	Rossi	Mario	1978/12/05
	8765	Neri	Paolo	1976/11/03
	9283	Verdi	Luisa	1979/11/12
	3456	Rossi	Maria	1978/02/01

EXAM	Student	Grade	Lecture	
	3456	30	04	
	3456	24	02	
	9283	28	01	
	6554	26	01	

LECTURE	Code	Name	Lecturer	
	01	Maths	Mario	
	02	Chemistry	Bruni	
	04	Chemistry	Verdi	

4

## Relazioni con un solo attributo

STUDENT	Number	Surname	Name	BirthD
	6554	Rossi	Mario	1978/12/05
	8765	Neri	Paolo	1976/11/03
	9283	Verdi	Luisa	1979/11/12
	3456	Rossi	Maria	1978/02/01

WORKERSTU	Number
	6554
	3456

## Strutture dati annidate

Restaurant Da Filippo Via Roma 2, Roma		
Receipt 1235 at 2002/10/12		
3	Cover Charge	3,00
2	Appetizer	6,20
3	Soup	12,00
2	Meat Loaf	18,00
Total Sum		39,20

Restaurant Da Filippo Via Roma 2, Roma		
Receipt 1240 at 2002/10/13		
2	Cover Charge	2,00
2	Appetizer	7,00
2	Soup	8,00
2	Fish	20,00
2	Coffee	2,00
Total Sum		39,00

RECEIPT	Number	Data	Total
	1235	12/10/2002	39,20
	1240	13/10/2002	39,00

COURSE	Receipt	Qty	Description	Price
	1235	3	Cover Charge	3,00
	1235	2	Appetizer	6,20
	1235	3	Soup	12,00
	1235	2	Meat Loaf	18,00
	1240	2	Cover Charge	2,00
	...	...	...	...

Se dovessimo memorizzare le informazioni blu e rosse insieme avremmo delle ripetizioni. Per evitare di avere informazioni annidate potremmo operare una suddivisione in più tabelle.

Abbiamo realmente mappato tutti gli aspetti della ricevuta?

Dipende dal nostro obiettivo!

- Sono consentite linee ripetute nello scontrino?

- L'ordine delle righe è rilevante?

Ci sono molte rappresentazioni diverse possibili.

## Disannidare strutture dati annidate

Se volessimo imporre un ordine alle righe sarebbe necessario un attributo aggiuntivo.

RECEIPT	Number	Data	Total
	1235	12/10/2002	39,20
	1240	13/10/2002	39,00

COURSE	Receipt	Row	Qty	Description	Price
	1235	1	3	Cover Charge	3,00
	1235	2	2	Appetizer	6,20
	1235	3	3	Soup	12,00
	1235	4	2	Meat Loaf	18,00
	1240	1	2	Cover Charge	2,00
	...	...	...	...	...

## Informazioni parziali

Il modello dati relazionale ha una struttura rigida:

- l'informazione è espressa in tuple
  - non tutte le tuple sono ammesse: le tuple valide devono rispettare uno schema
- I dati potrebbero non combaciare con il formato atteso.

## Esempio di informazioni parziali

Name	MidName	Surname
Franklin	Delano	Roosevelt
Winston		Churchill
Charles		De Gaulle
Josip		Stalin

## Soluzioni percorribili

Non si dovrebbero usare specifici valori all'interno del dominio (0, stringa vuota, "99", ...). Potrebbe succedere che non ci siano valori "inutilizzati" disponibili, oppure questi valori potrebbero diventare utili ad un certo punto. Si dovrebbero segnare questi valori "non rappresentabili" nel software, per fornirgli il significato atteso.

## Informazioni parziali nel modello relazionale

Nelle basi di dati l'informazione parziale viene rappresentata tramite il **valore NULL**, il quale denota la mancanza di un valore atteso (questo valore non appartiene al dominio).

$t[A]$ , per ogni attributo A, mappa verso un valore in  $\text{dom}(A)$  oppure NULL.

Dobbiamo definire vincoli specifici per puntualizzare quando i valori NULL non sono ammessi.

## Diversi tipi di NULL

Ci sono almeno tre casi differenti di valori mancanti:

- valore sconosciuto (non vuol dire che non esiste)
- valore inesistente
- valore non informativo (valore che dovrebbe esserci ma al momento non c'è)

Non ci sono specifiche distinzioni tra questi valori nei moderni DBMS.

## Troppi NULL

STUDENT	Number	Surname	Name	BirthD
	6554	Rossi	Mario	1978/12/05
	8765	Neri	Paolo	NULL
	9283	Verdi	Luisa	1979/11/12
	NULL	Rossi	Maria	1978/02/01

EXAM	Student	Grade	Lecture
	3456	30	04
	NULL	24	02
	NULL	28	01
	6554	26	01

LECTURE	Code	Name	Lecturer
	01	Maths	Mario
	02	NULL	NULL
	04	Chemistry	Verdi

39

## Vincoli di integrità

Alcune istanze di database, seppur sintatticamente corrette, potrebbero non rappresentare alcuna informazione corretta per l'applicazione destinataria.

Informazione scritta correttamente, ma semanticamente non ha significato.

## Rappresentazione errata

EXAM	Student	Grade	Laude	Lecture
	276545	32		01
	276545	30	yes	02
	787643	27	yes	03
	739430	24		04

STUDENT	Number	Surname	Name
	276545	Rossi	Mario
	787643	Neri	Piero
	787643	Bianchi	Luca

# Vincolo di integrità

Proprietà che deve essere mantenuta nelle istanze per poter rappresentare informazioni corrette per l'applicazione destinataria.

Un vincolo può essere interpretato come una funzione booleana (predicato): associa ad ogni istanza un valore booleano true o false, che ci dice se tale istanza soddisfa o meno il vincolo di integrità.

## Perché averli?

- Forniscono una descrizione più accurata del nostro scenario reale
- Supportano la qualità dei dati
- Sono utili nella progettazione di database
- Vengono usati nella valutazione delle query

## Osservazioni

I DBMS non supportano tutti i tipi di vincoli.

Possiamo definire vincoli dei tipi messi a disposizione nel nostro database ed il DBMS preverrà la loro violazione.

Quando il DBMS non supporta specifici vincoli, è l'utente o il programmatore che deve codificare tali vincoli al di fuori del DBMS.

## Tipi di vincoli

Vincoli **intra-relazionali**, all'interno di una singola relazione o tabella:

- sui valori (o vincoli di dominio)
- sulle tuple

Vincoli **inter-relazionali**, a cavallo tra due o più tabelle.

## Esempio

		Intra-relational over values			
		Student	Grade	Laude	Lecture
Inter-relational	276545	32			01
	276545	30	yes		02
	787643	27	yes		03
	739430	24			04

		Intra-relational over tuples		
		Number	Surname	Name
EXAM	276545	Rossi	Mario	
	787643	Neri	Piero	
	787643	Bianchi	Luca	

# Vincoli sulle tuple

Esprimono regole sui valori di ciascuna tupla, indipendentemente dalle altre tuple.  
Caso specifico sono i vincoli di dominio, che coinvolgono un singolo attributo.

## Sintassi

Una possibile sintassi può essere un'espressione booleana formata da atomi. Ogni atomo può comparare:

- valori del dominio dell'attributo. ( $Grade \geq 18$ ) AND ( $Grade \leq 30$ )
- espressioni aritmetiche su quei valori.  $GrossPay = (Deductions + Net)$

## Esempio di vincolo sulle tuple

SALARY	Employee	GrossPay	Deductions	Net
	Rossi	55'000	12'500	42'500
	Neri	45'000	10'000	35'000
	Bruni	47'000	11'000	36'000

$$\text{GrossPay} = (\text{Deductions} + \text{Net})$$

## Violazione del vincolo sulle tuple

SALARY	Employee	GrossPay	Deductions	Net
	Rossi	55'000	12'500	42'500
	Neri	45'000	10'000	35'000
	Bruni	50'000	11'000	36'000

$$\text{GrossPay} = (\text{Deductions} + \text{Net})$$

## Altro esempio

STUDENT	Number	Surname	Name
	276545	Rossi	Mario
	787643	Neri	Piero
	787643	Bianchi	Luca

Non ci possono essere due studenti con lo stesso numero di matricola.

# Chiavi

## Identificare le tuple

Number	Surname	Name	Curriculum	Birth
27655	Rossi	Mario	CSE	78/12/05
78763	Rossi	Mario	BioTech	76/11/03
65432	Neri	Piero	Mech En	79/07/10
87654	Neri	Mario	CSE	76/11/03
67653	Rossi	Piero	Mech En	78/12/05

Ciascuna tupla ha un *Number ID* univoco.

Non ci sono tuple che hanno lo stesso *Surname*, *Name* e *Date of Birth*.

# Superchiave e chiave

Un insieme di attributi che identifica univocamente le tuple all'interno di una relazione.

Più formalmente:

- Una **superchiave** è un insieme di attributi  $K$  su  $r$ , se non ci sono due tuple distinte  $t_1$  e  $t_2$  in  $r$  tali che  $t_1[K] = t_2[K]$ .
- Una **chiave** è una superchiave minimale della relazione, ovvero che non contiene un'altra superchiave.

## Una chiave

Number	Surname	Name	Curriculum	Birth
27655	Rossi	Mario	CSE	78/12/05
78763	Rossi	Mario	BioTech	76/11/03
65432	Neri	Piero	Mech En	79/07/10
87654	Neri	Mario	CSE	76/11/03
67653	Rossi	Piero	Mech En	78/12/05

Number è una chiave:

- è una superchiave
- contiene un solo attributo, quindi è minimale

## Un'altra chiave

Number	Surname	Name	Curriculum	Birth
27655	Rossi	Mario	CSE	78/12/05
78763	Rossi	Mario	BioTech	76/11/03
65432	Neri	Piero	Mech En	79/07/10
87654	Neri	Mario	CSE	76/11/03
67653	Rossi	Piero	Mech En	78/12/05

L'insieme *Surname, Name, Birth* è un'altra chiave: è una superchiave ed è minimale.

## Un'altra chiave ancora?

Number	Surname	Name	Curriculum	Birth
27655	Rossi	Mario	CSE	78/12/05
78763	Rossi	Mario	BioTech	76/11/03
65432	Neri	Piero	Mech En	79/07/10
87654	Neri	Mario	CSE	76/11/03
67653	Rossi	Piero	Mech En	78/12/05

Non ci sono tuple identiche su *Surname* e *Curriculum*: *Surname* e *Curriculum* è una chiave.

Ma è sempre vero ciò?

Ci sono chiavi per caso e chiavi costruite appositamente.

# Vincoli, schema ed istanze

I vincoli corrispondono a proprietà del mondo reale, modellate all'interno del database.

Sono modellati basandosi su uno schema, e si applicano a tutte le tuple.

Ciascuno schema può avere un insieme di vincoli. Ogni tupla memorizzata è da considerarsi corretta (valida) perché soddisfa tutti i vincoli.

Un'istanza potrebbe soddisfare altri vincoli "per caso".

## Esempio di vincolo e schema

STUDENTS

Number	Surname	Name	Curriculum	Birth
--------	---------	------	------------	-------

Keys:

*Number*

*Surname, Name, Birth*

## Esempio di soddisfazione dei vincoli

Number	Surname	Name	Curriculum	Birth
27655	Rossi	Mario	CSE	78/12/05
78763	Rossi	Mario	BioTech	76/11/03
65432	Neri	Piero	Mech En	79/07/10
87654	Neri	Mario	CSE	76/11/03
67653	Rossi	Piero	Mech En	78/12/05

Corretto: tutti i vincoli sono rispettati.

Altri vincoli sono soddisfatti "per caso": *Surname, Curriculum* è una chiave.

## Esistenza delle chiavi

Nessuna relazione può contenere tuple con esattamente gli stessi valori.

Ogni relazione ha una superchiave che è l'insieme di tutti gli attributi.

Quindi ha per forza almeno una chiave.

## Importanza delle chiavi

L'esistenza di almeno una chiave assicura l'accessibilità di ogni tupla nel database.

Le chiavi consentono di correlare tuple attraverso relazioni diverse: il modello dati relazionale è basato sui valori.

# Chiavi e valori NULL

Quando la tupla contiene valori NULL, i valori della chiave non permettono di:

- identificare le tuple
- creare facilmente riferimenti esterni da altre relazioni

Le chiavi non possono MAI avere valori NULL.

## Esempio di chiavi e valori NULL

Number	Surname	Name	Curriculum	Birth
NULL	NULL	Mario	CSE	78/12/05
78763	Rossi	Mario	BioTech	76/11/03
65432	Neri	Piero	Mech En	79/07/10
87654	Neri	Mario	CSE	NULL
NULL	Rossi	Piero	NULL	78/12/05

L'uso di valori NULL nelle chiavi deve essere limitato.

## Chiave primaria

La **Chiave Primaria (PK)** non ammette valori NULL.

Viene rappresentata visivamente con una sottolineatura.

Number	Surname	Name	Curriculum	Birth
27655	NULL	Mario	CSE	78/12/05
78763	Rossi	Mario	BioTech	76/11/03
65432	Neri	Piero	Mech En	79/07/10
87654	Neri	Mario	CSE	NULL
67653	Rossi	Piero	NULL	78/12/05

## Integrità referenziale

Le informazioni tra relazioni diverse sono correlate tramite valori comuni (modello value-based).

In particolare, i valori delle chiavi (ad esempio le chiavi primarie).

Questi collegamenti devono essere coerenti.

## Esempi di integrità referenziale

INFRINGEMENT	Code	Date	Policeman	Country	Plate
	34321	95/02/01	3987	Italy	MI395BK
	53524	95/03/04	3295	Italy	AJ046EZ
	64521	96/04/05	3295	France	ET234RY
	73321	98/02/05	9345	Finland	MMG-418

INFRINGEMENT	Code	Date	Policeman	Country	Plate
	34321	95/02/01	3987	Italy	MI395BK
	53524	95/03/04	3295	Italy	AJ046EZ
	64521	96/04/05	3295	France	ET234RY
	73321	98/02/05	9345	Finland	MMG-418

POLICEMAN	Id	Surname	Name
	3987	Rossi	Luca
	3295	Neri	Piero
	9345	Neri	Mario
	7543	Mori	Gino

CAR	Country	Plate	Surname	Name
	Italy	MI395BK	Rossi	Mario
	Italy	AJ046EZ	Verdi	Giuseppe
	France	ET234RY	Debussy	Claude
	Finland	MMG-418	Sibelius	Jean

# Vincoli di integrità referenziale

Un vincolo di integrità referenziale, per esempio una **chiave esterna (FK)**, tra gli attributi X di una relazione  $R_1$  e un'altra relazione  $R_2$ , costringe i valori di X in  $R_1$  ad apparire come valori della chiave primaria di  $R_2$ . È un vincolo inter-relazionale.

Un attributo A in  $R_1$  è una chiave esterna che referenzia la relazione  $R_2$  se e solo se soddisfa i seguenti requisiti:

- l'attributo A ha lo stesso dominio dell'attributo chiave primaria di  $R_2$
- il valore di A in una tupla  $t_1$  in  $R_1$  è NULL oppure compare come valore di qualche tupla  $t_2$  in  $R_2$  tale che  $t_1[A] = t_2[PK]$  (o anche  $t_1[FK] = t_2[PK]$ )

## Nell'esempio precedente

Vincolo di integrità referenziale tra:

- L'attributo *Policeman* di INFRINGEMENT e la relazione POLICEMAN.
- Gli attributi *Country* e *Plate* di INFRINGEMENT e la relazione CAR.

## Violazione del vincolo

INFRINGEMENT	Code	Date	Policeman	Country	Plate
	34321	95/02/01	3987	Italy	MI395BK
	53524	95/03/04	3295	Italy	AJ046EZ
	64521	96/04/05	3295	France	ET234RY
	73321	98/02/05	9345	Finland	MMG-418

CAR	Country	Plate	Surname	Name
	Italy	MI395BK	Rossi	Mario
	France	AJ046EZ	Verdi	Giuseppe
	France	ET234RY	Debussy	Claude
	Finland	MMG-418	Sibelius	Jean

## Commenti sui vincoli di integrità referenziale

- Hanno un ruolo cruciale nel modello dati basato sui valori.
- Quando appaiono valori NULL, i vincoli possono essere rilassati.
- Potremmo gestire violazioni ai vincoli ("side effects") con azioni compensative.
- Fare attenzione quando i vincoli sono definiti su più attributi.

## Integrità referenziale e valori NULL

Potremmo usare il valore NULL per indicare la mancanza di corrispondenza dei valori.

EMPLOYEE	<u>Number</u>	Surname	Project
	34321	Rossi	IDEA
	53524	Neri	XYZ
	64521	Verdi	NULL
	73032	Bianchi	IDEA

PROJECT	<u>Code</u>	Begin	Span	Cost
	IDEA	01/2000	36	200
	XYZ	07/2001	24	120
	BOH	09/2001	24	150

## Esempi di azioni compensative

Una tupla viene eliminata, causando la violazione di un vincolo.

Comportamento "standard":

- rifiutare l'eliminazione

Possibili azioni compensative:

- rimozione a cascata
- introduzione di valori default o NULL

### Rimozione a cascata

EMPLOYEE	<u>Number</u>	Surname	Project
	34321	Rossi	IDEA
	64521	Verdi	NULL
	73032	Bianchi	IDEA

PROJECT	<u>Code</u>	Begin	Span	Cost
	IDEA	01/2000	36	200
	BOH	09/2001	24	150

### Porre a NULL

EMPLOYEE	<u>Number</u>	Surname	Project
	34321	Rossi	IDEA
	53524	Neri	NULL
	64521	Verdi	NULL
	73032	Bianchi	IDEA

PROJECT	<u>Code</u>	Begin	Span	Cost
	IDEA	01/2000	36	200
	BOH	09/2001	24	150

## Vincoli su più attributi

CRASH	<u>Code</u>	Date	StateA	PlateA	StateB	PlateB
	34321	01/02/1995	Italy	AJ046EZ	Italy	MI395BK
	53524	05/04/1996	Finland	MMG-418	France	ET234RY

CRASH	<u>Code</u>	Date	StateA	PlateA	StateB	PlateB
	34321	01/02/1995	Italy	AJ046EZ	Italy	MI395BK
	53524	05/04/1996	Spain	4189HEJ	France	ET234RY

CAR	<u>Country</u>	<u>Plate</u>	<u>Surname</u>	<u>Name</u>
	Italy	MI39548K	Rossi	Mario
	Italy	AJ046EZ	Verdi	Giuseppe
	France	ET234RY	Debussy	Claude
	Finland	MMG-418	Sibelius	Jean

CAR	<u>Country</u>	<u>Plate</u>	<u>Surname</u>	<u>Name</u>
	Italy	MI39548K	Rossi	Mario
	Italy	AJ046EZ	Verdi	Giuseppe
	France	ET234RY	Debussy	Claude
	Finland	MMG-418	Sibelius	Jean

Vincoli tra:

- Gli attributi *StateA* e *PlateA* di CRASH e la relazione CAR
- Gli attributi *StateB* e *PlateB* di CRASH e la relazione CAR

L'ordine tra gli attributi è rilevante.

# Algebra relazionale e calcolo relazionale

## Introduzione

### Linguaggi dei database

Operazioni sullo schema:

- DDL: Data Definition Language

Operazioni sui dati:

- DML: Data Manipulation Language
  - Query instructions, per estrarre i dati di interesse
  - Update instructions, per inserire nuovi dati o modificare quelli esistenti

### Linguaggi di query per DB relazionali

**Dichiarativi:** specificano quali sono i risultati che vogliamo ottenere.

**Procedurali:** specificano come il risultato è ottenuto.

### Linguaggi di query

**Algebra relazionale:** procedurale.

**Calcolo relazionale:** dichiarativo (teorico, non implementato).

**SQL (Structured Query Language):** parzialmente dichiarativo (implementato).

**QBE (Query by Example):** dichiarativo (implementato).

### Algebra relazionale

Definita da un insieme di **operatori**

- sulle relazioni
- che producono relazioni come risultato
- che possono essere componibili

# Operatori dell'algebra relazionale

- union, intersection, difference
- rename
- select
- project (proiezione)
- join (natural join, prodotto cartesiano, theta-join)

## Operatori insiemistici

Le relazioni sono insiemi. Anche i risultati devono essere insiemi. Perciò, possiamo usare **unione**, **intersezione e differenza** se le relazioni coinvolte hanno lo stesso schema.

### Unione

GRADUATED		
Number	Name	Age
7274	Rossi	42
7432	Neri	54
9824	Verdi	45

TECHNICIANS		
Number	Name	Age
9297	Neri	33
7432	Neri	54
9824	Verdi	45

### Intersezione

GRADUATED		
Number	Name	Age
7274	Rossi	42
7432	Neri	54
9824	Verdi	45

TECHNICIANS		
Number	Name	Age
9297	Neri	33
7432	Neri	54
9824	Verdi	45

### GRADUATED U TECHNICIANS

Number	Name	Age
7274	Rossi	42
7432	Neri	54
9824	Verdi	45
9297	Neri	33

### GRADUATED ∩ TECHNICIANS

Number	Name	Age
7432	Neri	54
9824	Verdi	45

## Differenza

GRADUATED		
Number	Name	Age
7274	Rossi	42
7432	Neri	54
9824	Verdi	45

TECHNICIANS		
Number	Name	Age
9297	Neri	33
7432	Neri	54
9824	Verdi	45

### FATHERHOOD

Father	Child
Adam	Abel
Adam	Cain
Abraham	Isaac

### MOTHERHOOD

Mother	Child
Eve	Abel
Eve	Seth
Sarah	Isaac

### GRADUATED - TECHNICIANS

Number	Name	Age
7274	Rossi	42

### FATHERHOOD U MOTHERHOOD

??

## Rinominazione

Operatore unario (un solo argomento).

Produce un risultato che "cambia lo schema" mantenendo i dati contenuti inalterati.

# Sintassi e semantica

**Sintassi:**

$$\rho_{NewName \leftarrow OldName}(RELATION)$$

**Semantica:** cambia il nome dell'attributo da "OldName" a "NewName".

## Esempi di renaming

FATHERHOOD

Father	Child
Adam	Abel
Adam	Cain
Abraham	Isaac

FATHERHOOD

Father	Child
Adam	Abel
Adam	Cain
Abraham	Isaac

$\rho_{Parent \leftarrow Father}$  (FATHERHOOD)

Parent	Child
Adam	Abel
Adam	Cain
Abraham	Isaac

$\rho_{Parent \leftarrow Father}$  (FATHERHOOD)

Parent	Child
Adam	Abel
Adam	Cain
Abraham	Isaac

MOTHERHOOD

Mother	Child
Eve	Abel
Eve	Seth
Sarah	Isaac

$\rho_{Parent \leftarrow Mother}$  (MOTHERHOOD)

Parent	Child
Eve	Abel
Eve	Seth
Sarah	Isaac

$\rho_{Parent \leftarrow Father}$  (FATHERHOOD)

Parent	Child
Adam	Abel
Adam	Cain
Abraham	Isaac

$\rho_{Parent \leftarrow Father}$  (FATHERHOOD) U  
 $\rho_{Parent \leftarrow Mother}$  (MOTHERHOOD)

Parent	Child
Adam	Abel
Adam	Cain
Abraham	Isaac
Eve	Abel
Eve	Seth
Sarah	Isaac

$\rho_{Parent \leftarrow Mother}$  (MOTHERHOOD)

Parent	Child
Eve	Abel
Eve	Seth
Sarah	Isaac

EMPLOYEE

Surname	Office	Salary
Rossi	Rome	55
Neri	Milan	64

WORKER

Surname	Factory	Wage
Bruni	Paris	45
Verdi	Berlin	55

$\rho_{Pay \leftarrow Salary}$  (EMPLOYEE) U

$\rho_{Office, Pay \leftarrow Factory, Wage}$  (WORKER)

Surname	Office	Pay
Rossi	Rome	55
Neri	Milan	64
Bruni	Paris	45
Verdi	Berlin	55

17

## Selezione

Operatore unario.

Come risultato:

- l'output ha lo stesso schema dell'input
- le tuple in output sono un sottoinsieme delle tuple in input
- le tuple in output soddisfano un predicato

# Sintassi e semantica

**Sintassi:**

$$\sigma_{predicate}(RELATION)$$

**Predicate:** la sua interpretazione è un'espressione booleana sulle tuple della relazione.

**Semantica:** un sottoinsieme della relazione che soddisfa il predicato fornito.

## Esempi di selection

EMPLOYEE

Number	Surname	Office	Salary
7309	Rossi	Rome	55
5998	Neri	Milan	64
9553	Milan	Milan	44
5698	Neri	Naple	64

Impiegati che guadagnano più di 50:

$$\sigma_{Salary > 50}(\text{EMPLOYEE})$$

Number	Surname	Office	Salary
7309	Rossi	Rome	55
5998	Neri	Milan	64
5698	Neri	Naple	64

Impiegati che guadagnano più di 50 e lavorano a Milano:

$$\sigma_{Salary > 50 \text{ AND } Office = 'Milan'}(\text{EMPLOYEE})$$

Number	Surname	Office	Salary
5998	Neri	Milan	64

Impiegati che hanno il cognome come la città in cui lavorano:

$$\sigma_{Surname = Office}(\text{EMPLOYEE})$$

Number	Surname	Office	Salary
9553	Milan	Milan	44

# Proiezione

Operatore unario.

Come risultato:

- lo schema in output è un sottoinsieme dello schema in input
- l'output è formato usando tutte le tuple dell'input

# Sintassi e semantica

**Sintassi:**

$$\pi_{AttributeList}(RELATION)$$

**Semantica:** il risultato contiene tutte le tuple in RELATION, ma solo gli attributi in *AttributeList*.

## Esempi di projection

**EMPLOYEE**

Number	Surname	Office	Salary
7309	Rossi	Rome	55
5998	Neri	Milan	64
9553	Milan	Milan	44
5698	Neri	Naple	64

Matricola e cognome degli impiegati:

$$\pi_{Number, Surname}(\text{EMPLOYEE})$$

Number	Surname
7309	Rossi
5998	Neri
9553	Milan
5698	Neri

Cognome ed ufficio di tutti gli impiegati:

$$\pi_{Surname, Office}(\text{EMPLOYEE})$$

Surname	Office
Rossi	Rome
Neri	Milan
Milan	Milan
Neri	Naple

## Cardinalità delle proiezioni

L'output di una proiezione:

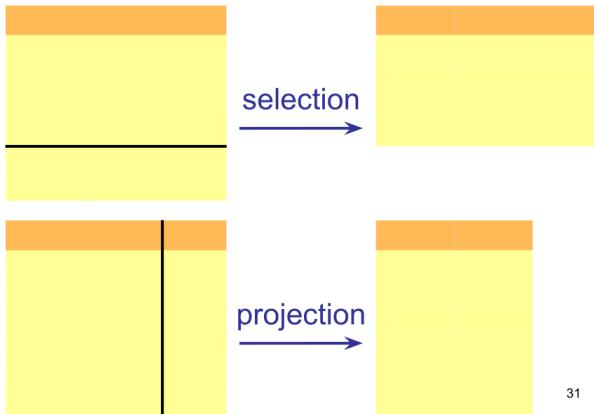
- Contiene al massimo lo stesso numero di tuple dell'input.
- Potrebbe contenere meno tuple: restringendo ad un sottoinsieme degli attributi, alcune tuple potrebbero essere ripetute. Le tuple ripetute sono scartate nelle relazioni.

Se X è una superchiave per R, allora  $\pi_X(R)$  contiene lo stesso numero di tuple di R.

# Selezione e proiezione

Operatori "ortogonali".

- **Selezione:** decomposizione orizzontale.
- **Proiezione:** decomposizione verticale.



31

Combinando selezione e proiezione, è possibile estrarre solo le informazioni interessanti da una relazione.

Ad esempio ritornare la matricola ed il cognome degli impiegati che hanno un salario maggiore di 50:

$\pi_{\text{Number}, \text{Surname}} (\sigma_{\text{Salary} > 50} (\text{EMPLOYEE}))$

Number	Surname
7309	Rossi
5998	Neri
5698	Neri

## Limiti della combo $\sigma + \pi$

Usando questi operatori possiamo estrarre informazioni solo da una singola relazione. In questo modo non possiamo correlare né tuple da relazioni diverse, né tuple differenti dalla stessa relazione.

# Join

Il join è l'operatore più interessante dell'algebra relazionale.

Consente la correlazione tra tuple in relazioni differenti.

## Esempio: esami durante un concorso pubblico

Ogni esame è anonimo e una busta chiusa, contenente il cognome del candidato, è associata ad esso. Ogni esame e la sua relativa busta hanno lo stesso ID.

Each exam is anonymous		A closed envelope ...		Id Grade		Id Candidate	
1	25	1	Jan Johansson	1	25	1	Jan Johansson
2	13	2	Jean B. Lully	2	13	2	Jean B. Lully
3	27	3	Johann S. Bach	3	27	3	Johann S. Bach
4	28	4	Giuseppe Verdi	4	28	4	Giuseppe Verdi

Jan Johansson	25	Id	Candidate	Grade
Jean B. Lully	13	1	Jan Johansson	25
Johann S. Bach	27	2	Jean B. Lully	13
Giuseppe Verdi	28	3	Johann S. Bach	27
		4	Giuseppe Verdi	28

## Join naturale

Operatore binario (generalizzabile tramite associatività).

Fornisce un risultato tale che:

- congiunge due tabelle basandosi su attributi con lo stesso nome
- il suo schema è l'unione degli attributi degli schema delle due relazioni
- ogni tupla è prodotta combinando due tuple: una da ognuna delle due relazioni

## Sintassi e semantica

**Sintassi:** date  $R_1(X_1), R_2(X_2)$

$$R_1 \bowtie R_2$$

è una relazione su  $X_1 X_2$ .

**Semantica:**

$$R_1 \bowtie R_2 = \{t \text{ su } X_1 X_2 \mid \exists t_1 \in R_1 \text{ and } \exists t_2 \in R_2 \text{ con } t[X_1] = t_1 \text{ and } t[X_2] = t_2\}$$

## Full join

Ogni tupla contribuisce al risultato finale.

Employee	Dept
Rossi	A
Neri	B
Bianchi	B



Dept	Chief
A	Mori
B	Bruni

## Non full join

Non tutte le tuple contribuiscono.

Employee	Dept
Rossi	A
Neri	B
Bianchi	B



Dept	Chief
B	Mori
C	Bruni

Employee	Dept	Chief
Rossi	A	Mori
Neri	B	Bruni
Bianchi	B	Bruni

Employee	Dept	Chief
Neri	B	Mori
Bianchi	B	Mori

## Join vuoto

Employee	Dept
Rossi	A
Neri	B
Bianchi	B



Dept	Chief
D	Mori
C	Bruni

## Full join avente n x m tuple

Employee	Dept	Chief
Rossi	B	Mori
Neri	B	Bruni



Employee	Dept	Chief
Rossi	B	Mori
Rossi	B	Bruni
Neri	B	Mori
Neri	B	Bruni

## Dimensione del risultato di un join

Il risultato del join tra  $R_1$  e  $R_2$  ha un numero di tuple compreso tra zero e  $|R_1| \times |R_2|$

$$0 \leq |R_1 \bowtie R_2| \leq |R_1| \times |R_2|$$

Se il join coinvolge una chiave da  $R_2$ , allora il numero di tuple risultanti è tra 0 e  $|R_1|$

$$0 \leq |R_1 \bowtie R_2| \leq |R_1|$$

Se il join coinvolge una chiave da  $R_2$  ed un vincolo di integrità referenziale, il numero di tuple è  $|R_1|$

$$|R_1 \bowtie R_2| = |R_1|$$

## Esempi di dimensioni

$|R_1|$  e  $|R_2|$  hanno dimensione 3.

La dimensione del join tra  $R_1$  e  $R_2$  è al massimo 9 (3x3).

$R_1$	Employee	Dept	$R_2$	Dept	Room
	Rossi	A		B	1
	Neri	B		B	2
	Bianchi	B		C	3

$$|R_1 \bowtie R_2| = 4$$

Il join tra  $R_1$  e  $R_2$  coinvolge la chiave di  $R_2$ , quindi la sua dimensione è tra 0 e  $|R_1|$ .  
 Essendo i valori della chiave unici, per ogni tupla di  $R_2$  ci possono essere più tuple di  $R_1$ , ma non viceversa.

$R_1$	Employee	Dept	$R_2$	Dept	Chief
	Rossi	A		B	Mori
	Neri	B		C	Bruni
	Bianchi	B			

$$|R_1 \bowtie R_2| \leq |R_1| = 2 \quad 47$$

Se esiste un vincolo di integrità referenziale tra *Dept* (chiave esterna in  $R_1$ ) e *Dept* (chiave in  $R_2$ ), ogni tupla in  $R_1$  è associata ad almeno una tupla in  $R_2$ .

$R_1$	Employee	Dept	$R_2$	Dept	Chief
	Rossi	A		A	Bruni
	Neri	B		B	Mori
	Bianchi	B		C	Neri

$$|R_1 \bowtie R_2| = |R_1| = 3 \quad 48$$

## Criticità del join

Alcune tuple non contribuiscono al risultato finale, sono "lasciate fuori".

Employee	Dept		Dept	Chief
Rossi	A		B	Mori
Neri	B		C	Bruni
Bianchi	B			

Employee	Dept	Chief
Neri	B	Mori
Bianchi	B	Mori

## Outer join

L'**outer** join riempie con valori NULL le tuple che normalmente sono scartate da un **inner** join.

Ci sono tre tipi di outer join:

- **left outer join**
- **right outer join**
- **full outer join**

## Sintassi e semantica

**Left outer join** ( $\bowtie$ ): mantiene tutte le tuple del primo operando, anche se non ci sono tuple che fanno match nel secondo operando. I buchi sono rimpiazzati da valori NULL.

**Right outer join** ( $\bowtie$ ):  $A \bowtie B$  è definito come  $B \bowtie A$ , ovvero tiene tutte le tuple di B.

**Full outer join** ( $\bowtie$ ): è definito come l'unione degli operandi che lo compongono, cioè mantiene tutte le tuple di entrambi gli operandi.

## Left outer join

EMPLOYEE	
Employee	Dept
Rossi	A
Neri	B
Bianchi	B

DEPARTMENT	
Dept	Chief
B	Mori
C	Bruni



## Right outer join

EMPLOYEE	
Employee	Dept
Rossi	A
Neri	B
Bianchi	B

DEPARTMENT	
Dept	Chief
B	Mori
C	Bruni



EMPLOYEE  $\bowtie$  DEPARTMENT

Employee	Dept	Chief
Neri	B	Mori
Bianchi	B	Mori
Rossi	A	NULL

EMPLOYEE  $\bowtie$  DEPARTMENT

Employee	Dept	Chief
Neri	B	Mori
Bianchi	B	Mori
NULL	C	Bruni

## Full outer join

EMPLOYEE	
Employee	Dept
Rossi	A
Neri	B
Bianchi	B

DEPARTMENT	
Dept	Chief
B	Mori
C	Bruni



EMPLOYEE  $\bowtie$  DEPARTMENT

Employee	Dept	Chief
Neri	B	Mori
Bianchi	B	Mori
Rossi	A	NULL
NULL	C	Bruni

54

## Prodotto cartesiano

Se non c'è coincidenza tra gli operandi si può fare il prodotto cartesiano.

EMPLOYEE	
Employee	Dept
Rossi	A
Neri	B
Bianchi	B

DEPARTMENT	
Code	Chief
A	Mori
B	Bruni

EMPLOYEE  $\bowtie$  DEPARTMENT

Employee	Dept	Code	Chief
Rossi	A	A	Mori
Rossi	A	B	Bruni
Neri	B	A	Mori
Neri	B	B	Bruni
Bianchi	B	A	Mori
Bianchi	B	B	Bruni

55

La dimensione del risultato è il prodotto della dimensione degli operandi.

Nella pratica, il prodotto cartesiano è utile solo se seguito da una selezione:

$$\sigma_{condition}(R_1 \bowtie R_2)$$

# Theta-join

Il prodotto cartesiano assume un significato specifico quando è seguito da un'operazione di selezione.

$$\sigma_{condition}(R_1 \bowtie R_2)$$

Tale operazione è definita come **theta-join** ( $\theta$ -join).

$$R_1 \bowtie_{condition} R_2$$

## Condizione

Il predicato *condition* è solitamente definito come una congiunzione (AND) di atomi che esprimono relazioni binarie  $A_1 \mathcal{R} A_2$ , dove  $\mathcal{R}$  è un operatore di confronto ( $>$ ,  $<$ ,  $=$ , ...).

## Equi-join

Un theta-join è chiamato "**equi-join**" se e solo se tutti gli atomi  $\mathcal{R}$  in theta sono relazioni di uguaglianza.

Nota che l'equi-join è usato molto più spesso del più generale theta-join: consente di specificare su quale attributo effettuare il join senza usare operazioni di rename.

## Esempi

EMPLOYEE		DEPARTMENT	
Employee	Dept	Code	Chief
Rossi	A	A	Mori
Neri	B	B	Bruni
Bianchi	B		

EMPLOYEE $\bowtie_{Dept=Code}$ DEPARTMENT			
Employee	Dept	Code	Chief
Rossi	A	A	Mori
Neri	B	B	Bruni
Bianchi	B	B	Bruni

Nota: l'equi-join fornisce un risultato simile al join tra EMPLOYEE e DEPARTMENT dove *Code* è rinominato come *Dept*.

EMPLOYEE $\bowtie_{Dept \leftarrow Code}$ (DEPARTMENT)		
Employee	Dept	Chief
Rossi	A	Mori
Neri	B	Bruni
Bianchi	B	Bruni

## Natural join ed equi-join

Dopo la rinominazione otteniamo i due seguenti schema, i quali possono essere congiunti con un join naturale. Procediamo poi con selezione e proiezione.

EMPLOYEE		DEPARTMENT	
Employee	Dept	Dept	Chief

EMPLOYEE  $\bowtie$  DEPARTMENT

$$\begin{aligned} & \Pi_{Employee,Dept,Chief} ( \\ & \sigma_{Dept=Code} ( EMPLOYEE \bowtie \rho_{Code \leftarrow Dept} (DEPARTMENT) ) ) \end{aligned}$$

## Algebra relazionale +

### Esempi di algebra relazionale

EMPLOYEE	Number	Name	Age	Wage
	7309	Rossi	34	45
	5998	Bianchi	37	38
	9553	Neri	42	35
	5698	Bruni	43	42
	4076	Mori	45	50
	8123	Lupi	46	60

SUPERVISOR	Employee	Chief
	7309	5698
	5998	5698
	9553	4076
	5698	4076
	4076	8123

63

1. Ritorna *number*, *name*, *age* e *wage* degli impiegati che guadagnano più di 40.

$$\sigma_{Wage > 40} (EMPLOYEE)$$

2. Ritorna *number*, *name* e *age* degli impiegati che guadagnano più di 40.

$$\Pi_{Number,Name,Age} (\sigma_{Wage > 40} (EMPLOYEE))$$

3. Ritorna i capi i cui impiegati guadagnano più di 40.

$$\begin{aligned} & \Pi_{Chief} (SUPERVISOR \bowtie_{Employee=Number} \\ & (\sigma_{Wage > 40} (EMPLOYEE))) \end{aligned}$$

4. Ritorna *name* e *wage* dei capi aventi impiegati che guadagnano più di 40.

$$\begin{aligned} & \Pi_{Name,Wage} (EMPLOYEE \bowtie_{Number=Chief} \\ & \Pi_{Chief} (SUPERVISOR \\ & \bowtie_{Employee=Number} \\ & \sigma_{Wage > 40} (EMPLOYEE) \\ & ) \\ & ) \end{aligned}$$

67

5. Ritorna gli impiegati che guadagnano più dei loro capi. Per entrambi impiegati e capi ritorna *number*, *name* e *wage*.

$$\begin{aligned} & \pi_{\text{Number}, \text{Name}, \text{Wage}, \text{NumC}, \text{NameC}, \text{WageC}} ( \\ & \sigma_{\text{Wage} > \text{WageC}} ( \\ & \rho_{\substack{\text{NumC}, \text{NameC}, \text{WageC}, \text{AgeC} \\ \text{Number}, \text{Name}, \text{Wage}, \text{Age}}} \leftarrow (\text{EMPLOYEE}) \bowtie_{\text{NumC} = \text{Chief}} \right. \\ & (\text{SUPERVISOR} \bowtie_{\text{Employee} = \text{Number}} \text{EMPLOYEE} \\ & ) \\ & ) \end{aligned}$$

68

6. Ritorna *number* dei capi aventi tutti gli impiegati che guadagnano più di 40.

$$\begin{aligned} & \pi_{\text{Chief}} (\text{SUPERVISOR}) - \\ & \pi_{\text{Chief}} (\text{SUPERVISOR} \\ & \quad \bowtie_{\text{Employee} = \text{Number}} \\ & \quad \sigma_{\text{Wage} \leq 40} (\text{EMPLOYEE}) \\ & ) \end{aligned}$$

## Espressioni dell'algebra relazionale equivalenti

Due espressioni dell'algebra relazionale  $E_1$  ed  $E_2$  sono **equivalenti** se e solo se forniscono lo stesso risultato indipendentemente dallo stato del database, ma possono dipendere dallo schema. Le regole di equivalenza sono molto importanti poiché i moderni DBMS cercano di riscrivere le query in espressioni equivalenti più efficienti (ad esempio che ci mettano meno tempo).

### Espressioni equivalenti

1. Combinare una cascata di selezioni

$$\sigma_{C1 \text{ AND } C2}(R) = \sigma_{C1}(\sigma_{C2}(R))$$

2. La proiezione è idempotente ( $X$  e  $Y$  appartengono allo schema di  $R$ )

$$\pi_X(R) = \pi_X(\pi_{XY}(R))$$

3. Ordine di selezione e proiezione

$$\pi_{XY}(\sigma_X(R)) = \sigma_X(\pi_{XY}(R))$$

4. "Spingere dentro" le selezioni

$$\sigma_C(R_1 \bowtie R_2) = R_1 \bowtie (\sigma_C(R_2))$$

-  $C$  coinvolge attributi che appartengono allo schema di  $R_2$

- Riduce drasticamente la dimensione del risultato intermedio (perciò anche il costo dell'operazione)

## 5. "Spingere dentro" le proiezioni

$$\pi_{X_1 Y_2}(R_1 \bowtie R_2) = R_1 \bowtie \pi_{Y_2}(R_2)$$

- $R_1$  ha schema  $X_1$ ,  $R_2$  ha schema  $X_2$
- Se  $Y_2 \subseteq (X_1 \cap X_2)$ , gli attributi in  $X_2 - Y_2$  non sono coinvolti nel join, quindi l'equivalenza è valida.

## 6. Usare la definizione del theta-joint

$$\sigma_C(R_1 \bowtie R_2) = R_1 \bowtie_C R_2$$

Ad esempio:

$$\begin{aligned} \pi_{\text{Chief}}(\sigma_{\text{Age}<30 \text{ AND } \text{Number}=\text{Employee}}(\text{EMPLOYEE} \bowtie \text{SUPERVISOR})) &= \\ \pi_{\text{Chief}}(\sigma_{\text{Number}=\text{Employee}}(\sigma_{\text{Age}<30}(\text{EMPLOYEE} \bowtie \text{SUPERVISOR}))) &= \\ \pi_{\text{Chief}}(\sigma_{\text{Age}<30}(\text{EMPLOYEE}) \bowtie_{\text{Number}=\text{Employee}} \text{SUPERVISOR}) &= \\ \pi_{\text{Chief}}(\pi_{\text{Number}}(\sigma_{\text{Age}<30}(\text{EMPLOYEE})) \bowtie_{\text{Number}=\text{Employee}} \text{SUPERVISOR}) \end{aligned}$$

Alcune proprietà distributive.

Nota che la proiezione non è distributiva sulla differenza.

7.  $\sigma_C(R_1 \cup R_2) = \sigma_C(R_1) \cup \sigma_C(R_2)$
8.  $\sigma_C(R_1 - R_2) = \sigma_C(R_1) - \sigma_C(R_2)$
9.  $\pi_X(R_1 \cup R_2) = \pi_X(R_1) \cup \pi_X(R_2)$

Alcune proprietà riguardo insiemi e selezione.

10.  $\sigma_{C1 \text{ OR } C2}(R) = \sigma_{C1}(R) \cup \sigma_{C2}(R)$
11.  $\sigma_{C1 \text{ AND } C2}(R) = \sigma_{C1}(R) \cap \sigma_{C2}(R)$
12.  $\sigma_{C1 \text{ AND } \neg C2}(R) = \sigma_{C1}(R) - \sigma_{C2}(R)$

13. Proprietà distributiva del join rispetto all'unione

$$R_1 \bowtie (R_2 \cup R_3) = (R_1 \bowtie R_2) \cup (R_1 \bowtie R_3)$$

Inoltre, le proprietà associativa e commutativa si applicano a tutti gli operatori binari eccetto la differenza.

**Nota:** non è necessario imparare a scrivere query in algebra relazionale efficienti, è meglio averle corrette e comprensibili. Questo perché i moderni DBMS svolgono questo compito al posto nostro.

# Selezione con valori NULL

$\sigma_{Age > 40} (PEOPLE)$

PEOPLE

Number	Surname	Agency	Age
7309	Rossi	Rome	32
5998	Neri	Milan	45
9553	Bruni	Milan	NULL

Nessun valore NULL soddisfa la specifica condizione atomica.

## Un risultato indesiderabile

Le selezioni sono valutate separatamente.

$\sigma_{Age > 30} (PEOPLE) \cup \sigma_{Age \leq 30} (PEOPLE) \neq PEOPLE$

Le condizioni atomiche sono valutate separatamente.

$\sigma_{Age > 30 \text{ OR } Age \leq 30} (PEOPLE) \neq PEOPLE$

## Una soluzione

La seguente condizione atomica è soddisfatta solo da valori non NULL:  $\sigma_{Age > 40} (PEOPLE)$

Degli specifici statement atomici sono utilizzati per riferirsi a valori NULL: `IS NULL` e

`IS NOT NULL`.

Potremmo anche definire una logica a tre valori (basata su true, false, unknown) ma non è necessario.

Perciò:

$$\begin{aligned}\sigma_{Age > 30} (PEOPLE) \cup \\ \sigma_{Age \leq 30} (PEOPLE) \cup \\ \sigma_{Age \text{ IS NULL}} (PEOPLE)\end{aligned}$$

=

$$\sigma_{Age > 30 \text{ OR } Age \leq 30 \text{ OR } Age \text{ IS NULL}} (PEOPLE)$$

=

PEOPLE

$$\sigma_{(Age > 40) \text{ OR } (Age \text{ IS NULL})} (PEOPLE)$$

PEOPLE

Number	Surname	Agency	Age
7309	Rossi	Rome	32
5998	Neri	Milan	45
9553	Bruni	Milan	NULL

# Viste (views)

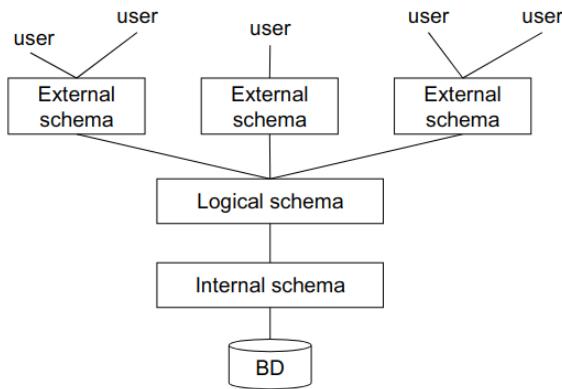
Rappresentazioni differenti per gli stessi dati.

- **Tabelle derivate**: relazioni create da query.

- **Tabelle base**: contenuti originali.

Le relazioni derivate possono essere formate a partire da altre relazioni derivate.

## Architettura standard a tre livelli ANSI/SPARC



## Esempio di vista

AFFILIATION		MANAGEMENT	
Employee	Dept	Dept	Chief
Rossi	A	A	Mori
Neri	B	B	Bruni
Bianchi	B		

- A view:

$\text{SUPERVISOR} := \pi_{\text{Employee}, \text{Chief}} (\text{AFFILIATION} \bowtie \text{MANAGEMENT})$

## Usare le viste

Le viste possono essere usate in due modi:

- **Materialized views**
- **Virtual relations (o views)**

## View materialization

Una vista temporanea o permanente viene memorizzata fisicamente nel database.

Pro:

- i dati sono prontamente disponibili per future query

Contro:

- ridondanza dei dati
- gli aggiornamenti sono rallentati
- raramente supportata dai DBMS

## Virtual relations (o views)

Tutti i DBMS le supportano.

La query sulla view viene trasformata in una query sul database sottostante.

Il nome della vista è rimpiazzato dalla sua query associata.

$\sigma_{Chief='Mori'} (\text{SUPERVISOR})$

Is run as:

$\sigma_{Chief='Mori'} (\pi_{Employee, Chief} (\text{AFFILIATION} \bowtie \text{MANAGEMENT}))$

## Perché averle?

**Schema esterno:** ogni utente può vedere

- quello a cui è interessato nella maniera che preferisce, senza ulteriori distrazioni
- solo quello che ha il permesso di vedere

**Strumento di programmazione:** potremmo semplificare la scrittura di query complesse quando delle sotto-espressioni sono ripetute, usando software già esistente su schema ri-fattorizzati.

Le viste non influenzano l'efficienza delle query.

## Esempio di vista come strumento di programmazione

Ritorna gli impiegati aventi il capo di Jones.

■ *Without views:*

```
 $\pi_{Employee} ((\text{AFFILIATION} \bowtie \text{MANAGEMENT}) \bowtie$ 
 $\rho_{EmpR,RepR \leftarrow Employee,Dept} ($ 
 $\sigma_{Employee='Jones'} (\text{AFFILIATION} \bowtie \text{MANAGEMENT}))$ 
)
```

■ *Using views:*

```
 $\pi_{Employee} (\text{SUPERVISOR} \bowtie$ 
 $\rho_{EmpR \leftarrow Employee} (\sigma_{Employee='Jones'} (\text{SUPERVISOR}))$ 
)
```

# Aggiornare le viste

AFFILIATION		MANAGEMENT	
Employee	Dept	Dept	Chief
Rossi	A	A	Mori
Neri	B	B	Bruni
Verdi	A	C	Bruni

SUPERVISOR	
Employee	Dept
Rossi	Mori
Neri	Bruni
Verdi	Mori

Come potremmo aggiornare i dati in modo che Bruni sia il capo di Lupi e che Falchi sia il capo di Belli?

## Aggiornamenti incrementali

"Aggiornare una vista" significa cambiare le tabelle base in modo che la vista aggiornata rifletta i cambiamenti.

Ad ogni aggiornamento sulla vista ne deve corrispondere uno sulle tabelle base.

- Tale aggiornamento potrebbe essere ambiguo.
- Solo pochi aggiornamenti possibili sono ammessi sulle viste.

## Notazione alternativa per il join

Nota: tale approccio è solitamente adottato per le implementazioni SQL.

Ignoriamo il join naturale: non assumiamo implicitamente condizioni sugli attributi con lo stesso nome.

Disambiguiamo gli attributi con lo stesso nome tra relazioni diverse usando la sintassi

`RELATION.Attribute`.

Assegniamo un nuovo nome alle relazioni creando delle viste, e rinominiamo gli attributi solo quando è necessario per l'operazione di unione.

## Esempio

EMPLOYEE	Number	Name	Age	Wage
	7309	Rossi	34	45
	5998	Bianchi	37	38
	9553	Neri	42	35
	5698	Brani	43	42
	4076	Mori	45	50
	8123	Lupi	46	60

SUPERVISOR	Employee	Chief
	7309	5698
	5998	5698
	9553	4076
	5698	4076
	4076	8123

Trova gli impiegati che guadagnano più dei loro capi, mostrando *number*, *name* e *wage* sia degli impiegati che dei capi.

```

ΠNumber,Name,Wage (
  σWage > WageC (
    ρNumC,NameC,AgeC,WageC ← Number,Name,Age,Wage (EMPLOYEE)
      ↗NumC=Chief
    (SUPERVISOR ↗Employee=Number EMPLOYEE)
  )
)

```

Assegna Employee a Chief (vista): `CHIEF := EMPLOYEE`

Usiamo il nome delle relazioni come prefisso per differenziare tra gli attributi condivisi, per esempio `EMPLOYEE.Wage` e `CHIEF.Wage`

```

ΠEMPLOYEE.Number,EMPLOYEE.Name,EMPLOYEE.Wage,
CHIEF.Number,CHIEF.Name,CHIEF.Wage (
  σEMPLOYEE.Wage > CHIEF.Wage (CHIEF ↗CHIEF.Number=Chief
    (SUPERVISOR ↗Employee=EMPLOYEE.Number EMPLOYEE)
  )
)

```

99

## Calcolo relazionale

Una famiglia di linguaggi dichiarativi basati sulla Logica del Prim'ordine.

Due definizioni differenti:

- calcolo relazionale sui domini
- calcolo relazionale su tuple con dichiarazioni di range

## Calcolo relazionale sui domini

**Sintassi:** un'espressione è nella forma

$$\{A_1 : x_1, \dots, A_k : x_k \mid f\}$$

$A_1 : x_1, \dots, A_k : x_k$  viene detta target list:

- $A_1, \dots, A_k$  attributi differenti (possono essere in database diversi)
- $x_1, \dots, x_k$  variabili differenti

$f$  è una formula, con operatori booleani e quantificatori.

**Semantica:** il risultato è una relazione su  $A_1, \dots, A_k$  contenente tuple di valori per  $x_1, \dots, x_k$  che soddisfano la formula  $f$ .

## Esempi

$\text{EMPLOYEE}(\underline{\text{Number}}, \text{Name}, \text{Age}, \text{Wage})$

$\text{SUPERVISOR}(\text{Chief}, \underline{\text{Employee}})$

1. Ritorna *number*, *name*, *age* e *wage* degli impiegati che guadagnano più di 40.

$\sigma_{\text{Wage}>40} (\text{EMPLOYEE})$

{ Number: *m*, Name: *n*, Age: *a*, Wage: *w* |  
 $\text{EMPLOYEE}(\text{Number: } m, \text{Name: } n, \text{Age: } a,$   
Wage: *w*)  $\wedge$  *w* > 40 }

2. Ritorna *number*, *name* e *age* di tutti gli impiegati.

$\Pi_{\text{Number}, \text{Name}, \text{Age}} (\sigma_{\text{Wage}>40} (\text{EMPLOYEE}))$

{ Number: *m*, Name: *n*, Age: *a* |  
 $\text{EMPLOYEE}(\text{Number: } m, \text{Name: } n, \text{Age: } a,$   
Wage: *w*)  $\wedge$  *w* > 40 }

La variabile libera potrebbe essere anche NULL, tuttavia nel calcolo relazionale non si considera.

3. Ritorna *number*, *name* e *age* degli impiegati che guadagnano più di 40.

$\sigma_{\text{Wage}>40} (\text{EMPLOYEE})$

{ Number: *m*, Name: *n*, Age: *a*, Wage: *w* |  
 $\text{EMPLOYEE}(\text{Number: } m, \text{Name: } n, \text{Age: } a,$   
Wage: *w*)  $\wedge$  *w* > 40 }

4. Ritorna il *number* che identifica i capi degli impiegati che guadagnano più di 40.

$\Pi_{\text{Chief}} (\text{SUPERVISOR} \bowtie_{\text{Employee}=\text{Number}} \sigma_{\text{Wage}>40} (\text{EMPLOYEE}))$

{ Chief: *c* | SUPERVISOR(Chief: *c*, Employee: *e*)  $\wedge$   
 $\text{EMPLOYEE}(\text{Number: } e, \text{Name: } n, \text{Age: } a, \text{Wage: } w) \wedge$   
*w* > 40 }

5. Ritorna *name* e *salary* dei capi aventi impiegati che guadagnano più di 40.

$\Pi_{\text{NameC}, \text{WageC}} (\rho_{\text{NumC}, \text{NameC}, \text{WageC}, \text{AgeC} \leftarrow \text{Number}, \text{Name}, \text{Wage}, \text{Age}} (\text{EMPLOYEE})$

$\bowtie_{\text{NumC}=\text{Chief}}$

$(\text{SUPERVISOR} \bowtie_{\text{Employee}=\text{Number}} (\sigma_{\text{Wage}>40} (\text{EMPLOYEE})))$

{ NameC: *nc*, WageC: *wc* |  
 $\text{EMPLOYEE}(\text{Number: } m, \text{Name: } n, \text{Age: } a, \text{Wage: } w) \wedge$   
*w* > 40  $\wedge$  SUPERVISOR(Chief: *c*, Employee: *m*)  $\wedge$   
 $\text{EMPLOYEE}(\text{Number: } c, \text{Name: } nc, \text{Age: } ac, \text{Wage: } wc)$  }

6. Ritorna gli impiegati che guadagnano più dei loro capi. Per entrambi, impiegati e capi, ritorna *number*, *name* e *salary*.

$$\begin{aligned} & \text{TT}_{\text{Number}, \text{Name}, \text{Wage}, \text{NumC}, \text{NameC}, \text{WageC}} (\sigma_{\text{Wage} > \text{WageC}} ( \\ & \rho_{\text{NumC}, \text{NameC}, \text{WageC}, \text{AgeC} \leftarrow \text{Number}, \text{Name}, \text{Wage}, \text{Age}} (\text{EMPLOYEE}) \\ & \bowtie_{\text{NumC} = \text{Chief}} \\ & (\text{SUPERVISOR} \bowtie_{\text{Employee} = \text{Number}} \text{EMPLOYEE}))) \\ & \{ \text{Number: } m, \text{Name: } n, \text{Wage: } w, \text{NumC: } c, \text{NameC: } nc, \text{WageC: } wc \mid \\ & \text{EMPLOYEE}(\text{Number: } m, \text{Name: } n, \text{Age: } a, \text{Wage: } w) \wedge \\ & \text{SUPERVISOR}(\text{Chief: } c, \text{Employee: } m) \wedge \\ & \text{EMPLOYEE}(\text{Number: } c, \text{Name: } nc, \text{Age: } ac, \text{Wage: } wc) \wedge w > sc \} \end{aligned}$$

7. Ritorna *number* e *name* dei capi aventi tutti gli impiegati che guadagnano più di 40.

$$\begin{aligned} & \text{TT}_{\text{Number}, \text{Name}} (\text{EMPLOYEE} \bowtie_{\text{Number} = \text{Chief}} \\ & (\pi_{\text{Chief}} (\text{SUPERVISOR}) - \\ & \text{TT}_{\text{Chief}} (\text{SUPERVISOR} \bowtie_{\text{Employee} = \text{Number}} \sigma_{\text{Wages} \leq 40} (\text{EMPLOYEE}))) \\ & \{ \text{Number: } c, \text{Name: } n \mid \\ & \text{EMPLOYEE}(\text{Number: } c, \text{Name: } n, \text{Age: } a, \text{Wage: } w) \wedge \\ & \text{SUPERVISOR}(\text{Chief: } c, \text{Employee: } m) \wedge \\ & \neg \exists m' (\exists n' (\exists a' (\exists w' (\text{EMPLOYEE}(\text{Number: } m', \text{Name: } n', \text{Age: } a', \text{Wage: } w') \wedge \\ & w' \leq 40 \wedge \text{SUPERVISOR}(\text{Chief: } c, \text{Employee: } m'))))) \} \end{aligned}$$

## Recap

- De Morgan rules:
  - $\neg(A \wedge B) = (\neg A) \vee (\neg B)$
  - $\neg(A \vee B) = (\neg A) \wedge (\neg B)$
- Moreover:
  - $\neg \forall x A = \exists x \neg A$
  - $\neg \exists x A = \forall x \neg A$
  - $\forall x A = \neg \exists x \neg A$
  - $\exists x A = \neg \forall x \neg A$
- Bonus:
  - $\neg A \vee B \rightarrow \text{if } A \text{ then } B$

## Quale quantificatore dovremmo usare?

Esistenziale o universale? Sono interscambiabili per le leggi di De Morgan.

$$\begin{aligned} & \{ \text{Number: } c, \text{Name: } n \mid \\ & \text{EMPLOYEE}(\text{Number: } c, \text{Name: } n, \text{Age: } a, \text{Wage: } w) \wedge \\ & \text{SUPERVISOR}(\text{Chief: } c, \text{Employee: } m) \wedge \\ & \neg \exists m' (\exists n' (\exists a' (\exists w' (\text{EMPLOYEE}(\text{Number: } m', \text{Name: } n', \text{Age: } a', \text{Wage: } w') \wedge \\ & w' \leq 40) \wedge \text{SUPERVISOR}(\text{Chief: } c, \text{Employee: } m'))) \} \\ & \{ \text{Number: } c, \text{Name: } n \mid \\ & \text{EMPLOYEE}(\text{Number: } c, \text{Name: } n, \text{Age: } a, \text{Wage: } w) \wedge \\ & \text{SUPERVISOR}(\text{Chief: } c, \text{Employee: } m) \wedge \\ & \forall m' (\forall n' (\forall a' (\forall w' (\neg \text{EMPLOYEE}(\text{Number: } m', \text{Name: } n', \text{Age: } a', \text{Wage: } w') \wedge \\ & \text{SUPERVISOR}(\text{Chief: } c, \text{Employee: } m') \vee w' > 40))) \} \end{aligned}$$

## Riguardo il calcolo relazionale sui domini

Pro:

- è dichiarativo

Contro:

- è verboso, troppe variabili
- permette di scrivere interrogazioni senza significato

$$\begin{aligned}\{A : x \mid \neg R(A : x)\} \\ \{A : x, B : y \mid R(A : x)\} \\ \{A : x, B : y \mid R(A : x) \wedge y = y\}\end{aligned}$$

Tali espressioni sono dipendenti dal dominio e dovremmo evitarle. Non possiamo esprimere questi statement nell'algebra relazionale perché essa è indipendente dal dominio.

## Calcolo sui domini vs algebra

Il calcolo relazionale sui domini (DRC, Domain Relational Calculus) e l'algebra relazionale (RA, Relational Algebra) sono **equivalenti**:

- > Per ogni espressione DRC indipendente dal dominio, esiste un'espressione RA equivalente.
- > Per ogni espressione RA, esiste un'espressione DRC indipendente dal dominio equivalente.

# Calcolo relazionale su tuple con dichiarazioni di range

Per superare le limitazioni del calcolo sui domini:

- Dobbiamo ridurre il numero di variabili. Un buon modo per farlo è restringere le variabili per le tuple, una variabile per ogni tupla.

- Tutti i valori devono arrivare dal database.

Il calcolo su tuple con dichiarazioni di range soddisfa entrambi i bisogni.

## Sintassi

Le espressioni hanno la seguente sintassi:

$$\{TargetList \mid RangeList \mid Formula\}$$

- *TargetList* ha elementi del tipo  $Y : x.Z$  (oppure  $x.Z$  altrimenti  $x.*$ )
- *RangeList* mostra le variabili libere in *Formula* specificando da quale relazione arrivano.
- *Formula* ha:
  - atomi di confronto  $x.A \mathcal{R} c, x.A \mathcal{R} y.B$
  - connettivi logici
  - quantificatori che associano un range alle variabili  $\exists x(R)(f), \forall x(R)(f)$

## Esempi

1. Ritorna *number*, *name*, *age* e *salary* degli impiegati che guadagnano più di 40.

$\sigma_{\text{Wage}>40}(\text{EMPLOYEE})$

{ Number:  $m$ , Name:  $n$ , Age:  $a$ , Wage:  $w$  |  
EMPLOYEE(Number:  $m$ , Name:  $n$ , Age:  $a$ , Wage:  $w$ )  $\wedge w > 40$  }

DRC

{  $e.* \mid e(\text{EMPLOYEE}) \mid e.\text{Wage} > 40$  }

TRC-RD

117

2. Ritorna *number*, *name* e *age* di tutti gli impiegati.

$\Pi_{\text{Number},\text{Name},\text{Age}}(\text{EMPLOYEE})$

{ Number:  $m$ , Name:  $n$ , Age:  $a$  |  
EMPLOYEE(Number:  $m$ , Name:  $n$ , Age:  $a$ , Wage:  $w$ ) }

{  $e.(\text{Number},\text{Name},\text{Age}) \mid e(\text{EMPLOYEE}) \mid \{\}$  }

La formula  $f$  qui è True, significa tutto quanto.

3. Ritorna *number*, *name* e *age* degli impiegati che guadagnano più di 40.

$\Pi_{\text{Number}, \text{Name}, \text{Age}} (\sigma_{\text{Wage} > 40} (\text{EMPLOYEE}))$

{ Number: *m*, Name: *n*, Age: *a* |  
 $\text{EMPLOYEE}(\text{Number: } m, \text{Name: } n, \text{Age: } a, \text{Wage: } w) \wedge w > 40 \}$

{ *e*.(*Number*,*Name*,*Age*) | *e*(EMPLOYEE) | *e*.*Wage* > 40 }

4. Ritorna il *number* che identifica i capi degli impiegati che guadagnano più di 40.

{ Chief: *c* | SUPERVISOR(Chief: *c*, Employee: *e*)  $\wedge$   
 $\text{EMPLOYEE}(\text{Number: } e, \text{Name: } n, \text{Age: } a, \text{Wage: } w) \wedge w > 40 \}$

{ *s*.Chief | *e*(EMPLOYEE), *s*(SUPERVISOR) |  
 $\iota. \text{Number} = s. \text{Employee} \wedge e. \text{Wage} > 40 \}$

5. Ritorna *name* e *salary* dei capi aventi impiegati che guadagnano più di 40.

{ NameC: *nc*, WageC: *wc* |  
 $\text{EMPLOYEE}(\text{Number: } m, \text{Name: } n, \text{Age: } a, \text{Wage: } w) \wedge w > 40 \wedge \text{SUPERVISOR}(\text{Chief: } c, \text{Employee: } m) \wedge \text{EMPLOYEE}(\text{Number: } c, \text{Name: } nc, \text{Age: } ac, \text{Wage: } wc) \}$

{ NameC, WageC: *e'*.(*Name*,*Wage*) |  
 $e'(\text{EMPLOYEE}), s(\text{SUPERVISOR}), e(\text{EMPLOYEE}) | e'. \text{Number} = s. \text{Chief} \wedge s. \text{Employee} = e. \text{Number} \wedge e. \text{Wage} > 40 \}$

6. Ritorna gli impiegati che guadagnano più dei loro capi. Per entrambi, impiegati e capi, ritorna *number*, *name* e *salary*.

{ Number: *m*, Name: *n*, Wage: *w*, NumC: *c*, NameC: *nc*, WageC: *wc* |  
 $\text{EMPLOYEE}(\text{Number: } m, \text{Name: } n, \text{Age: } a, \text{Wage: } w) \wedge \text{SUPERVISOR}(\text{Chief: } c, \text{Employee: } m) \wedge \text{EMPLOYEE}(\text{Number: } c, \text{Name: } nc, \text{Age: } ac, \text{Wage: } wc) \wedge w > wc \}$

{ *e*.(*Name*,*Number*,*Wage*), NameC, NumC, WageC: *e'*.(*Name*,*Number*,*Wage*) |  
 $e'(\text{EMPLOYEE}), s(\text{SUPERVISOR}), e(\text{EMPLOYEE}) | e'. \text{Number} = s. \text{Chief} \wedge s. \text{Employee} = e. \text{Number} \wedge e. \text{Wage} > e'. \text{Wage} \}$

7. Ritorna *number* e *name* dei capi aventi tutti gli impiegati che guadagnano più di 40.

{ Number: *c*, Name: *n* |  
 $\text{EMPLOYEE}(\text{Number: } c, \text{Name: } n, \text{Age: } a, \text{Wage: } w) \wedge \text{SUPERVISOR}(\text{Chief: } c, \text{Employee: } m) \wedge \neg \exists m' (\exists n' (\exists a' (\exists w' (\text{EMPLOYEE}(\text{Number: } m', \text{Name: } n', \text{Age: } a', \text{Wage: } w') \wedge \text{SUPERVISOR}(\text{Chief: } c, \text{Employee: } m') \wedge w' \leq 40)))) \}$   
{ *e*.(*Number*, *Name*) | *s*(SUPERVISOR), *e*(EMPLOYEE) |  
 $s. \text{Chief} = e. \text{Number} \wedge \neg (\exists e'(\text{EMPLOYEE}) (\exists s'(\text{SUPERVISOR}) (s. \text{Chief} = s'. \text{Chief} \wedge s'. \text{Employee} = e'. \text{Number} \wedge e'. \text{Wage} \leq 40))) \}$

# SQL base

## Introduzione

### SQL

Inizialmente era un acronimo per "Structured Query Language", ora è un nome proprio.

Linguaggio con molte funzionalità: implementa sia DDL che DML.

C'è un linguaggio standard ISO, però diversi DBMS hanno la loro grammatica del linguaggio.

Per il momento andremo a vedere le basi di questo linguaggio.

### Storia

Il suo predecessore era SEQUEL (1974).

Le prime implementazioni erano SQL/DS e Oracle (1981).

SQL diventa uno standard "de facto" a partire dal 1983.

Tanti aggiornamenti proposti (1986 poi 1989, 1992, 1999, 2003, 2006, 2008, ...) ma tutt'ora i DBMS hanno la loro grammatica personale (vedi dei confronti su <http://troels.arvin.dk/db/rdbms/>).

### Miglioramenti

- **SQL-86:** primo standard proposto. Aveva molte delle clausole per esprimere le interrogazioni, ma offriva un supporto limitato per creare ed aggiornare sia gli schema che i dati.
- **SQL-89:** aggiunti i vincoli di integrità referenziale.
- **SQL-92:** perlopiù compatibile con le versioni precedenti, ha delle novità:
  - Nuove funzioni (es. **COALESCE**, **NULLIF**, **CASE**)
  - 3 livelli di utilizzo: entry, intermediate, full

### SQL-3

Diverse sotto-versioni:

- **SQL:1999:** propone l'oggetto-relazione, i trigger e le funzioni esterne.
- **SQL:2003:** estende il modello orientato agli oggetti e consente di eseguire interrogazioni in Java e su dati semi-strutturati (XML).
- **SQL:2006:** SQL è esteso con altri linguaggi (es. XQuery) per interrogare dati XML.
- **SQL:2008:** alcune modifiche leggere alla sintassi (es. trigger con *instead of*).

## Riassunto miglioramenti

Unofficial Name	Official Name	Features
SQL-Base	SQL-86	Basic keywords
	SQL-89	Referential Integrity
SQL-2	SQL-92	Modello relazionale New keywords 3 levels: entry, intermediate, full
SQL-3	SQL:1999	Relational model with object-oriented Structured in different parts Trigger, external functions, ...
	SQL:2003	The support of the Object-Oriented model is extended The no-longer used keywords were removed Extensions: SQL/JRT, SQL/XML, ...
	SQL:2006	Extended support for XML data
	SQL:2008	Slight edits (e.g., trigger instead of)

## Data Definition

### CREATE DATABASE

Ciascun database appena creato contiene tabelle, views, triggers e altre cose.

```
CREATE DATABASE db_name
```

Nota bene:

- In SQLite `sqlite3 db_name.db sqlite3_open_v2(db_name)`
- In Mimer `CREATE DATABANK db_name`

### CREATE SCHEMA

Uno schema SQL è identificato da un nome e descrive gli elementi appartenenti ad esso (tabelle, tipi, vincoli, views, domini, ...). Lo schema apparirà all'utente che ha digitato lo statement.

```
CREATE SCHEMA schema_name
```

Questo statement può essere seguito dalla keyword **AUTHORIZATION**, per indicare uno specifico utente che possiede lo schema.

```
CREATE SCHEMA schema_name AUTHORIZATION 'user_name'
```

Da MySQL 8.0 Reference Manual: ... "CREATE SCHEMA is a synonym for CREATE DATABASE".

# CREATE TABLE

Specifica una nuova relazione e crea una sua istanza vuota.  
Specifica i suoi attributi (con i loro tipi) ed i suoi vincoli iniziali.

```
CREATE TABLE EMPLOYEE (
    Number CHARACTER(6) PRIMARY KEY,
    Name CHARACTER(20) NOT NULL,
    Surname CHARACTER(20) NOT NULL,
    Dept CHARACTER(15),
    Wage NUMERIC(9) DEFAULT 0,
    FOREIGN KEY(Dept) REFERENCES DEPARTMENT(Dept),
    UNIQUE (Surname, Name)
)
```

## Data Types (attributi)

I Data Types (attributi) in SQL corrispondono ai domini nel calcolo relazionale.

- **Basic** data types (subito disponibili)
- **Custom** data types (chiamati "domini" semplici e riutilizzabili)

### Basic Data Types

**Character-string:** i data type possono essere a lunghezza fissa o variabile.

**Numeric:** include numeri interi e diverse notazioni in virgola mobile.

**DATE, TIME, INTERVAL:** per rappresentare date, tempi o intervalli di tempo.

Introdotti in SQL-3 (SQL:1999):

- **Boolean:** valori booleani.
- **BLOB, CLOB:** Binary/Character Large Object, rappresentano enormi collezioni di dati (sia testuali che non).

### Custom Data Types

#### CREATE DOMAIN:

Ogni custom data type può essere utilizzato quando si definiscono nuove relazioni, esprimono dei vincoli o valori di default.

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

# Vincoli sulle tabelle

**NOT NULL**: impossibilità di assumere un valore NULL.

**UNIQUE**: definire chiavi (solo valori univoci).

**PRIMARY KEY**: solo una, implica il **NOT NULL**. DB2 ha un comportamento non standard.

**CHECK**: specifica vincoli sulle tuple, lo vedremo dopo.

## UNIQUE e PRIMARY KEY

Può essere usato:

- quando definiamo un attributo che identifica la chiave,
- come elemento a sè stante.

## Esempio di CREATE TABLE

```
CREATE TABLE EMPLOYEE (
    Number CHARACTER(6) PRIMARY KEY,    -- !!! (1)
    Name CHARACTER(20) NOT NULL,        -- !!! (2)
    Surname CHARACTER(20) NOT NULL,      -- !!! (2)
    Dept CHARACTER(15),
    Wage NUMERIC(9) DEFAULT 0,
    FOREIGN KEY(Dept) REFERENCES DEPARTMENT(Dept),
    UNIQUE (Surname, Name)    -- !!! (2)
)
```

## Alternative per PRIMARY KEY

```
Number CHARACTER(6) PRIMARY KEY,
-- oppure

Number CHARACTER(6),
...
PRIMARY KEY (Number)
```

## Attenzione!

```
Name CHARACTER(20) NOT NULL,  
Surname CHARACTER(20) NOT NULL,  
UNIQUE (Surname, Name)
```

e

```
Name CHARACTER(20) NOT NULL UNIQUE,  
Surname CHARACTER(20) NOT NULL UNIQUE,
```

non sono la stessa cosa!

Il primo dice che le coppie (Surname, Name) devono essere univoche, il secondo che i valori di Name e Surname singolarmente devono essere univoci.

## Chiave e vincoli di integrità referenziale

**CHECK** lo vedremo dopo.

**REFERENCES** e **FOREIGN KEY** definiscono vincoli di integrità referenziale.

Possono essere definiti:

- su un singolo attributo
- su attributi multipli

Possiamo definire azioni da effettuare quando tali vincoli vengono violati.

## Vincolo di integrità referenziale

OFFENSES	Code	Date	Officer	State	Number
	34321	95/02/01	3987	IT	AG548UK
	53524	95/03/04	3295	IT	TE395AB
	64521	96/04/05	3295	FR	ZT395AB
	73321	98/02/05	9345	FR	ZT395AB

OFFENSES	Code	Date	Officer	State	Number
	34321	95/02/01	3987	IT	AG548UK
	53524	95/03/04	3295	IT	TE395AB
	64521	96/04/05	3295	FR	ZT395AB
	73321	98/02/05	9345	FR	ZT395AB

OFFICER	Id	Surname	Name
	3987	Rossi	Luca
	3295	Neri	Piero
	9345	Neri	Mario
	7543	Mori	Gino

CAR	State	Number	Surname	Name
	IT	AG548UK	Verdi	Giuseppe
	IT	TE395AB	Verdi	Giuseppe
	FR	ZT395AB	Quinault	Philippe

## Esempio di CREATE TABLE

```
CREATE TABLE OFFENSES (
    Code CHARACTER(6) PRIMARY KEY,
    Day DATE NOT NULL,
    Officer INTEGER NOT NULL REFERENCES OFFICER(Id), -- !!!
    State CHARACTER(2),
    Number CHARACTER(6),
    FOREIGN KEY(State, Number) REFERENCES CAR(State, Number) -- !!!
)
```

## Referential Triggered Action

Dopo ogni vincolo referenziale, possiamo specificare una triggered action (delete, update) da invocare se la chiave esterna subisce variazioni nella relazione principale.

```
ON < DELETE | UPDATE > < CASCADE | SET NULL | SET DEFAULT | NO ACTION >
```

Per poter usare **SET NULL** la chiave esterna non deve avere il vincolo **NOT NULL**, per poter usare **SET DEFAULT** la chiave esterna deve avere definito un valore **DEFAULT**.

### Azioni ON DELETE

- **CASCADE**: elimina le tuple che avevano il riferimento.
- **SET NULL**: il valore dell'attributo eliminato che faceva riferimento viene posto a NULL.
- **SET DEFAULT**: il valore dell'attributo eliminato che faceva riferimento viene posto a uno specifico valore default.
- **NO ACTION**: non viene eseguita nessuna ulteriore cancellazione.

### Azioni ON UPDATE

- **CASCADE**: il valore dell'attributo chiave esterna che aveva il riferimento viene aggiornato al nuovo valore.
- **SET NULL**: il valore dell'attributo modificato che faceva riferimento viene posto a NULL.
- **SET DEFAULT**: il valore dell'attributo modificato che faceva riferimento viene posto a uno specifico valore default.
- **NO ACTION**: non viene eseguito nessun ulteriore aggiornamento.

# Statements di modifica allo schema

- ALTER DOMAIN
- ALTER TABLE
- DROP DOMAIN
- DROP TABLE

## ALTER DOMAIN

Permette di modificare domini definiti in precedenza.

Tale statement è da usare insieme ad uno dei seguenti: **SET DEFAULT**, **DROP DEFAULT**, **ADD CONSTRAINT** o **DROP CONSTRAINT**.

```
ALTER DOMAIN Grade SET DEFAULT 30
```

Imposta il valore default per *Grade* a 30.

Il default viene utilizzato quando si invoca un comando in cui vengono trovati valori di *Grade* mancanti.

```
ALTER DOMAIN Grade DROP DEFAULT
```

Rimuove il valore default di *Grade*.

```
ALTER DOMAIN Grade ADD CONSTRAINT isValid CHECK (value >=18 AND value <=30)
```

Aggiunge il vincolo *isValid* al tipo di dato *Grade*.

```
ALTER DOMAIN Grade DROP CONSTRAINT isValid
```

Rimuove il vincolo *isValid* associato al tipo di dato *Grade*.

## ALTER TABLE

Effettua modifiche su tabelle definite precedentemente.

Tale statement è da usare assieme ad uno di questi parametri: **ALTER COLUMN**, **ADD COLUMN**, **DROP COLUMN**, **DROP CONSTRAINT** o **ADD CONSTRAINT**.

```
ALTER TABLE EMPLOYEE ALTER COLUMN Number SET NOT NULL
```

Number della tabella EMPLOYEE non può avere valori nulli.

```
ALTER TABLE EMPLOYEE ADD COLUMN Level CHARACTER(10)
```

Un attributo *Level* è aggiunto alla tabella EMPLOYEE.

```
ALTER TABLE EMPLOYEE DROP COLUMN Level RESTRICT
```

Rimuove l'attributo *Level* da EMPLOYEE solo se non contiene valori.

```
ALTER TABLE EMPLOYEE DROP COLUMN Level CASCADE
```

Rimuove l'attributo *Level* da EMPLOYEE insieme a tutti i suoi valori.

```
ALTER TABLE EMPLOYEE ADD CONSTRAINT validNum CHECK (char_length(Number) = 10)
```

Aggiunge il vincolo *validNum* all'attributo *Number* di EMPLOYEE.

```
ALTER TABLE EMPLOYEE DROP CONSTRAINT validNum
```

Rimuove il vincolo *validNum* definito in precedenza su EMPLOYEE.

## DROP DOMAIN

Rimuove un tipo di dato definito dall'utente.

```
DROP DOMAIN Grade
```

## DROP TABLE

Rimuove un'intera tabella con il suo schema ed i suoi dati.

```
DROP TABLE OFFENSES
```

**Usare con enorme cautela**, a volte è addirittura bloccato come comando.

# Definire indici

Soltamente migliorano il tempo di interrogazione, rilevante per l'efficienza della computazione.  
Sono definiti a livello fisico, non logico.

In passato questo era l'unico modo per definire le chiavi.

## CREATE INDEX

```
CREATE INDEX idx_Surname ON OFFICER (Surname)
```

Crea l'indice *idx\_Surname* sull'attributo *Surname* della tabella OFFICER.

## DDL in pratica

In molti sistemi e progetti, per definire uno schema di database vengono utilizzati diversi strumenti invece di statements SQL puri (es. strumenti con un'interfaccia utente grafica).

# Data Manipulation

## Operazioni sui dati

Interrogazioni:

- **SELECT**

Modifiche:

- **INSERT**

- **DELETE**

- **UPDATE**

## SELECT statement base

```
SELECT <AttributeList>
FROM <TableList>
[ WHERE <Condition> ]
```

- Target list

- **FROM** statement

- **WHERE** statement

## Interpretazione

```
3 SELECT Number, Name  
1 FROM OFFICER  
2 WHERE Surname = 'Jones'
```

1 Dalla tabella OFFICER

2 Recupera tutti gli agenti che hanno 'Jones' come attributo Surname

3 Mostra per ogni tupla sia Number che Name

## Esempio di database

PEOPLE			MOTHERHOOD		FATHERHOOD	
Name	Age	Income	Mother	Child	Father	Child
Jim	27	21	Abby	Alice	Steve	Frank
James	25	15	Abby	Louis	Louis	Olga
Alice	55	42	Jesse	Olga	Louis	Phil
Jesse	50	35	Jesse	Phil	Frank	Jim
Phil	26	30	Alice	Jim	Frank	James
Louis	50	40	Alice	James		
Frank	60	20				
Olga	30	41				
Steve	85	35				
Abby	75	87				

## Scorciatoie (1)

```
SELECT *  
FROM PEOPLE  
WHERE Age < 30
```

Asterisco significa tutti gli attributi:

```
SELECT Name, Age, Income  
FROM PEOPLE  
WHERE Age < 30
```

## Selezione e proiezione

Return name and income of people under 30 yo

 $\Pi_{Name, Income} (\sigma_{Age<30} (PEOPLE))$ 

```
SELECT Name, Income  
FROM PEOPLE  
WHERE Age < 30
```

PEOPLE		
Name	Age	Income
Jim	27	21
James	25	15
Alice	55	42
Jesse	50	35
Phil	26	30
Louis	50	40
Frank	60	20
Olga	30	41
Steve	85	35
Abby	75	87

Return name and income of people under 30 yo

 $\Pi_{Name, Income} (\sigma_{Age<30} (PEOPLE))$ 

```
SELECT Name, Income  
FROM PEOPLE  
WHERE Age < 30
```

Name	Income
Jim	21
James	15
Phil	30

## Rinominazione attributi

```
SELECT P.Name AS GivenName, P.Income AS Revenue
FROM PEOPLE AS P
WHERE P.Age < 30
```

Name	Income
Jim	21
James	15
Phil	30



GivenName	Revenue
Jim	21
James	15
Phil	30

## Selezione pura (senza proiezione)

Provide the Name, Age and Income of people under 30 yo

 $\sigma_{Age<30} (\text{PEOPLE})$ 

```
SELECT *
FROM PEOPLE
WHERE Age < 30
```

PEOPLE		
Name	Age	Income
Jim	27	21
James	25	15
Alice	55	42
Jesse	50	35
Phil	26	30
Louis	50	40
Frank	60	20
Olga	30	41
Steve	85	35
Abby	75	87

Provide the Name, Age and Income of people under 30 yo

Name	Age	Income
Jim	27	21
James	25	15
Phil	26	30

 $\sigma_{Age<30} (\text{PEOPLE})$ 

```
SELECT *
FROM PEOPLE
WHERE Age < 30
```

## Proiezione senza selezione

Return the peoples' name and income

 $\pi_{Name,Income} (\text{PEOPLE})$ 

```
SELECT Name, Income
FROM PEOPLE
```

PEOPLE		
Name	Age	Income
Jim	27	21
James	25	15
Alice	55	42
Jesse	50	35
Phil	26	30
Louis	50	40
Frank	60	20
Olga	30	41
Steve	85	35
Abby	75	87

Return the peoples' name and income

 $\pi_{Name,Income} (\text{PEOPLE})$ 

```
SELECT Name, Income
FROM PEOPLE
```

Name	Income
Jim	21
James	15
Alice	42
Jesse	35
Phil	30
Louis	40
Frank	20
Olga	41
Steve	35
Abby	87

## Scorciatoie (2)

Data una relazione R(A,B)

```
SELECT *
FROM R
```

Corrisponde a:

```
SELECT X.A AS A, X.B AS B
FROM R AS X
WHERE true
```

## Condizioni composte

```
SELECT *
FROM PEOPLE
WHERE Income>25 AND (Age<30 OR Age>60)
```

PEOPLE

Name	Age	Income
Phil	26	30
Frank	60	20
Olga	30	41
Steve	85	35

Name	Age	Income
Phil	26	30
Steve	85	35

## Predicato LIKE

Il predicato **LIKE** si usa per esprimere delle espressioni regolari (semplificate) nella clausola WHERE. Le wildcard utilizzabili sono solo due:

- \_ per indicare un singolo carattere
- % per indicare zero o più caratteri

Ritorna le persone aventi un nome che inizia per 'J' e una 'm' come terza lettera:

```
SELECT *
FROM PEOPLE
WHERE Name LIKE 'J_m%'
```

```
SELECT *
FROM PEOPLE
WHERE Name LIKE 'J_m%'
```

PEOPLE

Name	Age	Income
Jim	27	21
James	25	15
Alice	55	42
Jesse	50	35
Phil	26	30
Louis	50	40
Frank	60	20

## Gestire valori NULL

Per controllare se un attributo ha valore NULL si usa **IS NULL**.

EMPLOYEE

Number	Surname	Agency	Age
5998	Neri	Milan	45
9553	Bruni	Milan	NULL

Ritorna gli impiegati che hanno più di 40 anni oppure hanno valore NULL.

$$\sigma_{(Age > 40) \text{ OR } (Age \text{ IS NULL})}(\text{EMPLOYEE})$$

```
SELECT *
FROM EMPLOYEE
WHERE Age > 40 OR Age IS NULL
```

## Proiezione con DISTINCT

EMPLOYEE

Number	Surname	Agency	Age
7309	Neri	Naples	55
5998	Neri	Milan	64
9553	Rossi	Rome	44
5698	Rossi	Rome	64

Ritorna cognome ed agenzia per tutti gli impiegati.

$$\pi_{Surname, Agency}(\text{EMPLOYEE})$$

```
SELECT
    Surname, Agency
FROM EMPLOYEE
```

```
SELECT DISTINCT
    Surname, Agency
FROM EMPLOYEE
```

Surname	Agency
Neri	Naples
Neri	Milan
Rossi	Rome
Rossi	Rome

Surname	Agency
Neri	Naples
Neri	Milan
Rossi	Rome

## Considerazioni

Usando una sola relazione nella clausola FROM, una singola query SQL può esprimere *selezione, proiezione e rinominazione*.

Usando più relazioni nella clausola FROM abbiamo i *join* (e il prodotto cartesiano).

## Alias e rinominazione

La rinominazione potrebbe essere richiesta nel prodotto cartesiano e/o nella target list.

```
SELECT X.A1 AS B1, ...
FROM R1 AS X, R2 AS Y, R1 AS Z
WHERE X.A2 = Y.A3 AND ...
```

Nel FROM non è rinominazione bensì aliasing, è rinominazione nel SELECT.

## SQL vs Algebra Relazionale

Date R1(A1,A2) e R2(A3,A4)

```
SELECT DISTINCT R1.A1, R2.A4
FROM R1, R2
WHERE R1.A2 = R2.A3
```

- Prodotto cartesiano (FROM)
- Selezione (WHERE)
- Proiezione (SELECT)

$$\pi_{A1,A4}(\sigma_{A2=A3}(R1 \bowtie R2))$$

## SQL vs Algebra Relazionale (2)

```
SELECT DISTINCT X.A1 AS B1, Y.A4 AS B2
FROM R1 AS X, R2 AS Y, R1 AS Z
WHERE X.A2 = Y.A3 AND Y.A4 = Z.A1
```

```

 $\rho_{B1,B2 \leftarrow A1,A4} ($ 
     $\pi_{A1,A4} (\sigma_{A2 = A3 \text{ AND } A4 = C1} ($ 
         $R1 \bowtie R2 \bowtie \rho_{C1,C2 \leftarrow A1,A2} (R1)$ 
    )
)
)
```

## Usare espressioni nella Target List

```
SELECT Income/2 AS halvedIncome  
FROM PEOPLE  
WHERE Name = 'Louis'
```

PEOPLE		
Name	Age	Income
Jim	27	21
James	25	15
Alice	55	42
Louis	50	40

halvedIncome

20

SQL ha un potere espressivo superiore rispetto all'algebra e al calcolo relazionale.

## Formulare e valutare le query

SQL è un linguaggio dichiarativo. Stiamo fornendo la semantica tramite degli esempi.

I DBMS hanno dei *piani di esecuzione query* per valutazioni efficienti:

- Le selezioni sono eseguite il prima possibile.
- Quando possibile, i join sono eseguiti al posto dei prodotti cartesiani.

Non dobbiamo necessariamente scrivere query efficienti in quanto i DBMS comprendono degli ottimizzatori di query. Per questo è più importante che le query fornite siano facili da capire (evitando errori nella formulazione).

## Esempio

PEOPLE			MOTHERHOOD		FATHERHOOD	
Name	Age	Income	Mother	Child	Father	Child
Jim	27	21	Abby	Alice	Steve	Frank
James	25	15	Abby	Louis	Louis	Olga
Alice	55	42	Jesse	Olga	Louis	Phil
Jesse	50	35	Jesse	Phil	Frank	Jim
Phil	26	30	Alice	Jim	Frank	James
Louis	50	40	Alice	James		
Frank	60	20				
Olga	30	41				
Steve	85	35				
Abby	75	87				

1) Padri che guadagnano più di 20:

$$\pi_{\text{Father}} (\text{FATHERHOOD} \bowtie_{\text{Child}=\text{Name}} \sigma_{\text{Income}>20} (\text{PEOPLE}))$$

La stessa query in SQL:

```
SELECT DISTINCT Father  
FROM PEOPLE, FATHERHOOD  
WHERE Name=Child AND Income > 20
```

2) Nome e reddito delle persone, reddito del loro padre, dove tali persone guadagnano più del padre:

```
πName, Income, IF (σIncome>IF (ρNF,AF,IF ← Name,Age,Income  
(PEOPLE))  
⊗NF=Father (FATHERHOOD ⊗Son=Name PEOPLE)  
)
```

```
SELECT C.Name, C.Income, F.Income
```

```
SELECT C.Name, C.Income, F.Income  
FROM PEOPLE F, FATHERHOOD, PEOPLE C  
WHERE F.Name = Father AND Child = C.Name AND C.Income > F.Income
```

Stesso SELECT con rinominazione:

```
SELECT C.Name AS Name, C.Income AS Income, F.Income AS fatherIncome  
FROM PEOPLE F, FATHERHOOD, PEOPLE C  
WHERE F.Name = Father AND Child = C.Name AND C.Income > F.Income
```

## Statement di JOIN

```
SELECT ...  
FROM LeftTable { ... JOIN RightTable ON JoinCondition }, ...  
[ WHERE otherPredicate ]
```

## JOIN implicito o esplicito

Ritorna madre e padre di ogni persona.

- JOIN implicito

```
SELECT F.Child, Father, Mother  
FROM MOTHERHOOD M, FATHERHOOD F  
WHERE M.Child = F.Child
```

- JOIN esplicito

```
SELECT Mother, FATHERHOOD.child, Father  
FROM MOTHERHOOD JOIN FATHERHOOD ON FATHERHOOD.Child = MOTHERHOOD.Child
```

## Esempio

Ritorna nome, reddito e reddito del padre di quelle persone aventi un reddito maggiore di quello del loro padre.

```
-- implicito
SELECT C.Name, C.Income, F.Income
FROM PEOPLE F, FATHERHOOD, PEOPLE C
WHERE F.Name = Father AND Child = C.Name AND C.Income > F.Income

-- esplicito
SELECT C.Name, C.Income, F.Income
FROM (PEOPLE F JOIN FATHERHOOD ON F.Name = Father) JOIN PEOPLE C ON Child = C.Name
WHERE C.Income > F.Income
```

Il join esplicito è più conveniente quando se ne deve fare più di uno.

## Join naturale

Conviene effettuare un join naturale quando possibile, ovvero quando gli attributi su cui fare join hanno lo stesso nome.

$$\pi_{\text{Child}, \text{Father}, \text{Mother}}(\text{FATHERHOOD} \bowtie_{\text{Child}=\text{Name}} \rho_{\text{Name}=\text{Child}}(\text{MOTHERHOOD}))$$

FATHERHOOD  $\bowtie$  MOTHERHOOD

```
-- join classico
SELECT Mother, FATHERHOOD.Child, Father
FROM MOTHERHOOD JOIN FATHERHOOD ON FATHERHOOD.Child = MOTHERHOOD.Child

-- join naturale
SELECT Mother, Child, Father
FROM MOTHERHOOD NATURAL JOIN FATHERHOOD
```

## Outer join

Con i precedenti join, anche chiamati **inner join**, alcune delle tuple potrebbero essere scartate dal risultato finale: questo avviene se non hanno una tupla corrispondente nell'altra tabella.

Per evitare questa perdita di informazioni, possiamo usare **LEFT/RIGHT/FULL OUTER JOIN**

Quando tale join è sinistro o destro, la keyword **OUTER** potrebbe essere omessa in quanto il left join ed il right join sono "outer" per definizione.

## Esempio di LEFT JOIN

Ritorna il padre e la madre, se noti.

```
SELECT FATHERHOOD.Child, Father, Mother  
FROM FATHERHOOD LEFT [OUTER] JOIN MOTHERHOOD ON FATHERHOOD.Child = MOTHERHOOD.Child
```

FATHERHOOD.Child	Father	Mother
Frank	Steve	NULL
Olga	Louis	Jesse
Phil	Louis	Jesse
Jim	Frank	Alice
James	Frank	Alice

## Esempi di OUTER JOIN

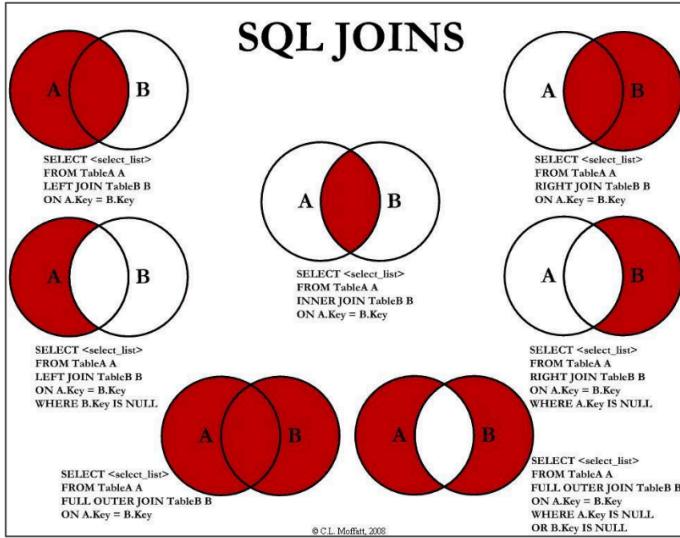
```
-- join classico  
SELECT FATHERHOOD.Child, Father, Mother  
FROM MOTHERHOOD JOIN FATHERHOOD ON MOTHERHOOD.Child = FATHERHOOD.Child  
  
-- left outer join  
SELECT FATHERHOOD.Child, Father, Mother  
FROM MOTHERHOOD LEFT OUTER JOIN FATHERHOOD ON MOTHERHOOD.Child = FATHERHOOD.Child  
  
-- full outer join  
SELECT FATHERHOOD.Child, Father, Mother  
FROM MOTHERHOOD FULL OUTER JOIN FATHERHOOD ON MOTHERHOOD.Child = FATHERHOOD.Child
```

Cosa ritorna l'ultima query?

FATHERHOOD.Child	Father	Mother
NULL	NULL	Abby
NULL	NULL	Abby
Olga	Louis	Jesse
Phil	Louis	Jesse
Jim	Frank	Alice
James	Frank	Alice
Frank	Steve	NULL

Il full outer join ritorna tutte le tuple che erano state escluse sia dall'operando destro che da quello sinistro.

## Recap



## Ordinare i risultati

Fornire nome e reddito delle persone con meno di 30 anni ordinate in ordine alfabetico.

```
SELECT Name, Income  
FROM PEOPLE  
WHERE Age < 30  
ORDER BY Name ASC
```

Clausola **ORDER BY** per ordinare le righe del risultato. Due ordinamenti possibili:

- **ASC:** ordine ascendente (crescente), default
- **DESC:** ordine discendente (decrescente)

## Esempio di ordinamento

PEOPLE

Name	Age	Income
Jim	27	21
James	25	15
Alice	55	42
Jesse	50	35
Phil	26	30
Louis	50	40
Frank	60	20
Olga	30	41
Steve	85	35
Abby	75	87

```
SELECT Name, Income  
FROM PEOPLE  
WHERE Age <= 30
```

Name	Income
Jim	21
James	15
Phil	30

```
SELECT Name, Income  
FROM PEOPLE  
WHERE Age <= 30  
ORDER BY Name
```

Name	Income
James	15
Jim	21
Phil	30

```
SELECT Name, Income  
FROM PEOPLE  
WHERE Age <= 30  
ORDER BY Name ASC
```

Name	Income
James	15
Jim	21
Phil	30

```
SELECT Name, Income  
FROM PEOPLE  
WHERE Age <= 30  
ORDER BY Name DESC
```

Name	Income
Phil	30
Jim	21
James	15

## Unione, intersezione, differenza

Il **SELECT** richiede uno statement specifico per effettuare unioni:

```
SELECT ...  
UNION [ALL]  
SELECT ...
```

Nel risultato le righe sono uniche (ad eccezione di quando viene usato **ALL**. In questo caso abbiamo un'unione di multi-set).

### Unione di insiemi

MOTHERHOOD		FATHERHOOD	
Mother	Child	Father	Child
Abby	Alice	Steve	Frank
Abby	Louis	Louis	Olga
Jesse	Olga	Louis	Phil

```
SELECT Child  
FROM MOTHERHOOD  
UNION  
SELECT Child  
FROM FATHERHOOD
```

Child
Alice
Louis
Olga
Frank
Phil

### Unione di multi-set

MOTHERHOOD		FATHERHOOD	
Mother	Child	Father	Child
Abby	Alice	Steve	Frank
Abby	Louis	Louis	Olga
Jesse	Olga	Louis	Phil

```
SELECT Child  
FROM MOTHERHOOD  
UNION ALL  
SELECT Child  
FROM FATHERHOOD
```

Child
Alice
Louis
Olga
Frank
Olga
Phil

Olga appears twice

92

### Notazione posizionale

```
SELECT Father, Child  
FROM FATHERHOOD  
UNION  
SELECT Mother, Child  
FROM MOTHERHOOD
```

Quando due tabelle hanno schema diversi, come possiamo risolvere i conflitti tramite rinominazione?

1. Usando nomi fintizi (alias) o nessuna rinominazione
  2. Adottando i nomi delle colonne del primo operando (il primo SELECT)
  3. Facendo il merge degli attributi che confliggono in una singola colonna
- Per convenzione si usano i nomi del primo operando.

Father	Child
Steve	Frank
Louis	Olge
Louis	Phil
Frank	Jim
Frank	James
Abby	Alice
Abby	Louis
Jesse	Olga
Jesse	Phil
Alice	Jim
Alice	James

```
SELECT Father, Child    SELECT Father, Child
FROM FATHERHOOD        FROM FATHERHOOD
UNION
SELECT Child, Mother   SELECT Mother, Child
FROM MOTHERHOOD        FROM MOTHERHOOD
```

In entrambi i casi lo schema della tabella risultante è (Father, Child).

## Differenza

```
SELECT Name
FROM EMPLOYEE
EXCEPT
SELECT Surname AS Name
FROM EMPLOYEE
```

Più avanti potremmo esprimere tale operatore tramite query SELECT annidate.

## Intersezione

```
SELECT Name
FROM EMPLOYEE
INTERSECT
SELECT Surname AS Name
FROM EMPLOYEE
```

È la stessa cosa di

```
SELECT E.Name
FROM EMPLOYEE E, EMPLOYEE F
WHERE E.Name = F.Surname
```

# Query annidate

I predicati permettono di:

- Confrontare un attributo (o più, come vedremo in seguito) con il risultato di una "sotto-query" annidata.
- Usare il quantificatore esistenziale (esiste,  $\exists$ ).

## Database esempio

PEOPLE			MOTHERHOOD		FATHERHOOD	
Name	Age	Income	Mother	Child	Father	Child
Jim	27	21	Abby	Alice	Steve	Frank
James	25	15	Abby	Louis	Louis	Olga
Alice	55	42	Jesse	Olga	Louis	Phil
Jesse	50	35	Jesse	Phil	Frank	Jim
Phil	26	30	Alice	Jim	Frank	James
Louis	50	40	Alice	James		
Frank	60	20				
Olga	30	41				
Steve	85	35				
Abby	75	87				

## Esempio di query annidate

Fornire il nome ed il reddito del padre di Frank.

```
SELECT Name, Income  
FROM PEOPLE, FATHERHOOD  
WHERE Name=Father AND Child='Frank'
```

Cartesian product and  
WHERE (equi-join)

```
SELECT Name, Income  
FROM PEOPLE  
WHERE Name=(SELECT Father  
            FROM FATHERHOOD  
            WHERE Child='Frank')
```

WHERE clause is true when  
subquery result is equal to  
Name. Moreover, only one  
tuple is produced by the  
subquery

Quando si parla di query annidata nel linguaggio comune si intende l'intera query, non solo la "sotto-query". Le "sotto-query" sono sempre e solo nella clausola WHERE.

## Commenti

Le query annidate sono "meno dichiarative", ma alcune volte più leggibili dal momento che richiedono meno variabili.

Query annidate e non annidate possono essere combinate.

Le "sotto-query" all'interno di una query annidata non possono esprimere operazioni insiemistiche ("l'unione può essere effettuata all'interno della query esterna"). Questa limitazione non è significativa.

Gli operatori di confronto richiedono valori singoli come operandi. Serve una soluzione per confrontare un valore con il risultato di una query (es. una relazione).

## ANY e ALL

Le query annidate possono essere formulate tramite un predicato usando **ANY** o **ALL** insieme ad un operatore di confronto ( $>$ ,  $<$ ,  $=$ ,  $\geq$ , ...), risolvendo il problema dell'omogeneità.

Attribute operator **ANY**( Expr )

Una tupla della query esterna viene selezionata se soddisfa il predicato rispetto almeno una tupla contenuta in *Expr*.

Attribute operator **ALL**( Expr )

Una tupla della query esterna viene selezionata se soddisfa il predicato rispetto tutte le tuple contenute in *Expr*.

## IN

Attribute **IN**( Expr )

Una tupla della query esterna viene selezionata se il suo valore per *Attribute* è presente tra gli elementi ritornati da *Expr*.

**ANY**, **ALL** e **IN** possono essere negati usando davanti la parola **NOT**.

Alcune equivalenze interessanti:

- A **IN**(Expr)  $\equiv$  A = **ANY**(Expr)
- A **NOT IN**(Expr)  $\equiv$  A  $\neq$  **ALL**(Expr)

## Esempi di query annidate

Fornire nome e reddito dei padri aventi figli che guadagnano più di 20.

```
SELECT DISTINCT F.Name, F.Income  
FROM PEOPLE F, FATHERHOOD, PEOPLE C  
WHERE F.Name = FATHERHOOD.Father AND FATHERHOOD.Child = C.Name AND C.Income > 20
```

Possiamo riscriverla senza **DISTINCT**, perché non faremo il join sulle tabelle quindi il nome dei padri non verrà ripetuto per ogni figlio:

Fornire nome e reddito dei padri aventi figli che guadagnano più di 20.

```
SELECT DISTINCT F.Name, F.Income  
FROM PEOPLE F, FATHERHOOD, PEOPLE C  
WHERE F.Name = Father AND Child = C.Name AND C.Income > 20
```

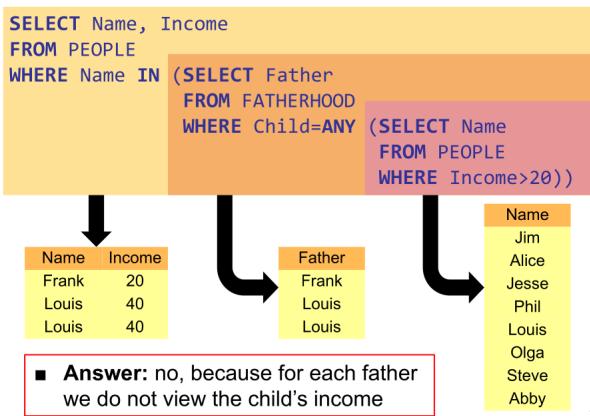
Possiamo riscriverla senza DISTINCT:

```
SELECT Name, Income  
FROM PEOPLE  
WHERE Name IN (SELECT Father  
                FROM FATHERHOOD, PEOPLE  
                WHERE Child=Name AND Income>20)
```

Fornire nome e reddito dei padri aventi figli che guadagnano più di 20, e fornire anche il reddito di tali figli.

```
SELECT DISTINCT F.Name, F.Income, C.Income  
FROM PEOPLE F, FATHERHOOD, PEOPLE C  
WHERE F.Name = Father AND Child = C.Name AND C.Income > 20
```

La seguente query fornisce gli stessi risultati?



## Visibilità

### Regole di visibilità:

- Non è possibile fare riferimento a variabili dichiarate in blocchi interni.
- Se il nome di una variabile è omesso, assumiamo di prendere quella dichiarata "più vicino".

Una sotto-query può fare riferimento a una variabile definita:

- all'interno del suo scope
- all'interno di uno qualunque degli scope esterni (outer-query)

## Semantica delle query annidate con variabili

La sotto-query viene effettuata una volta per ogni tupla della query esterna. L'unico modo per evitare ciò è creare una vista, che comunque modifica lo schema del database.

## Quantificazione esistenziale

**EXISTS ( Expr )**

Il predicato è true se e solo se *Expr* ritorna almeno una tupla, ovvero ritorna un insieme non-vuoto.

Tipicamente *Expr* è una query annidata.

È utile con una variabile che lega la query esterna e la sotto-query.

## Esempi di quantificazione esistenziale

Persone che hanno almeno un figlio.

```
SELECT *  
FROM PEOPLE  
WHERE EXISTS (SELECT *  
               FROM FATHERHOOD  
               WHERE Father = Name)  
  
OR  
EXISTS (SELECT *  
        FROM MOTHERHOOD  
        WHERE Mother = Name)
```

Name column is taken from the relation declared in the outer block

Padri aventi tutti i figli che guadagnano più di 20.

```
SELECT DISTINCT Father  
FROM FATHERHOOD Z  
WHERE NOT EXISTS (SELECT *  
                   FROM FATHERHOOD W,  
                   PEOPLE  
                   WHERE W.Father=Z.Father  
                         AND W.Child=Name  
                         AND Income<=20)
```

Z.Father is taken from the relation obtained in the outer block

Persone con la stessa età e reddito.

### Errore

```
SELECT DISTINCT *  
FROM PEOPLE  
WHERE EXISTS (SELECT *  
               FROM PEOPLE  
               WHERE Age = Age AND Income = Income)
```

Per le regole di scope, *Age* e *Income*, senza riferimento alla tabella, fanno implicitamente riferimento alla clausola *FROM* più vicina.

### Correzione

```
SELECT DISTINCT *  
FROM PEOPLE P  
WHERE EXISTS (SELECT *  
               FROM PEOPLE  
               WHERE P.Name = Name AND P.Age = Age AND P.Income = Income)
```

## Errore di visibilità

```
SELECT *
FROM EMPLOYEE
WHERE Dept IN (SELECT Name
                FROM DEPARTMENT D1
                WHERE Name = 'Production')
      OR
Dept IN (SELECT Name
                FROM DEPARTMENT D2
                WHERE D2.City = D1.City)
```

Errato perché nell'ultima selezione *D1.City* non è visibile.

## Differenza insiemistica e query annidate

```
SELECT Name
FROM EMPLOYEE
EXCEPT
SELECT Surname AS Name
FROM EMPLOYEE
```

È la stessa cosa che scrivere:

```
SELECT E.Name
FROM EMPLOYEE E
WHERE NOT EXISTS (SELECT *
                   FROM EMPLOYEE
                   WHERE Surname = E.Name)
```

## Posizione delle query annidate

- Nella clausola **WHERE**: uso standard, abbiamo visto numerosi esempi.
- Nella clausola **FROM**: è richiesta una nuova sorgente di dati (es. una relazione), l'alternativa è di creare una vista che, nonostante tutto, modifica lo schema del database.
- Nella clausola **SELECT**: uso non comune, è equivalente ad un join. Richiede necessariamente una tupla come risultato.

## Query annidate nella clausola FROM

Fornire nome e reddito dei figli di Jim.

```
SELECT Name, Income  
FROM PEOPLE P, (SELECT Child  
                  FROM FATHERHOOD  
                 WHERE Father = 'Jim') AS JIMCHILD  
WHERE Name = JIMCHILD.Child
```

**! NON È STANDARD !**

## Query annidate nella clausola SELECT

Fornire il totale delle spese di spedizione per ogni cliente nella tabella CUSTOMER.

```
SELECT CUSTOMER.Num,  
       (SELECT SUM(ShipCharge)  
        FROM ORDERS  
       WHERE CUSTOMER.Num=ORDERS.Num) AS TotalShipCharge  
  FROM CUSTOMER
```

**! NON È STANDARD !**

La query può essere riscritta usando semplicemente il join tra le due tabelle.

## Funzioni aggregate

Nella target list, possiamo inserire espressioni che calcolano valori a partire da un insieme di tuple, tramite funzioni aggregate.

Aggr: **COUNT | MIN | MAX | AVG | SUM**

Sintassi base:

```
Aggr([DISTINCT] *)  
Aggr([DISTINCT] Attribute)
```

## COUNT

Il numero di figli di Frank.

```
SELECT COUNT(*) AS NumFrankChildren  
FROM FATHERHOOD  
WHERE Father = 'Frank'
```

La funzione aggregata (**COUNT**) è applicata alle tuple del seguente risultato:

```
SELECT *  
FROM FATHERHOOD  
WHERE Father = 'Frank'
```

FATHERHOOD

Father	Chils
Steve	Frank
Louis	Olga
Louis	Phil
Frank	Jim
Frank	James

NumFrankChildren
2

## COUNT DISTINCT

PEOPLE

Name	Age	Income
Jim	27	30
James	25	24
Alice	55	36
Jesse	50	36

```
SELECT COUNT(*)          COUNT(*)  
FROM PEOPLE             4
```

```
SELECT COUNT(DISTINCT Income)    COUNT(DISTINCT Income)  
FROM PEOPLE                  3
```

Se non si specifica nulla, il nome della tabella risultante sarà il nome della funzione.

## Altre funzioni aggregate

### SUM, AVG, MAX, MIN

Media del reddito dei figli di Frank.

```
SELECT AVG(Income)
FROM PEOPLE JOIN FATHERHOOD ON Name=Child
WHERE Father='Frank'
```

## COUNT con valori NULL

PEOPLE

Name	Age	Income
Jim	27	30
James	25	NULL
Alice	55	36
Jesse	50	36

```
SELECT COUNT(*)
FROM PEOPLE
```

COUNT(*)
4

```
SELECT COUNT(Income)
FROM PEOPLE
```

COUNT(Income)
3

Nel primo caso sono 4 tuple distinte, anche se c'è un valore NULL.

Nel secondo caso sono 3 perché il NULL non viene contato.

PEOPLE

Name	Age	Income
Jim	27	21
James	25	NULL
Alice	55	21
Jesse	50	35

```
SELECT COUNT(DISTINCT Income)
FROM PEOPLE
```

COUNT(DISTINCT Income)
2

## Funzioni aggregate e NULL

PEOPLE

Name	Age	Income
Jim	27	21
James	25	NULL
Alice	55	21
Jesse	50	35

```
SELECT AVG(Income) AS AvgInc
FROM PEOPLE
```

AvgInc
25.6

Il valore NULL non viene considerato.

## Funzioni aggregate e target list

Una query errata:

```
SELECT Name, MAX(Income)  
FROM PEOPLE
```

Di chi è il nome? Non possiamo estrarre il nome avendo il reddito massimo. La *Target List* deve avere tutti attributi dello stesso tipo.

```
SELECT MIN(Age), MAX(Income)  
FROM PEOPLE
```

## Massimo e query annidate

Ritorna le persone aventi (lo stesso) reddito massimo.

```
SELECT *  
FROM PEOPLE  
WHERE Income = (SELECT MAX(Income)  
                 FROM PEOPLE)
```

## Raggruppamenti

Le funzioni aggregate possono operare su gruppi di relazioni tramite lo statement **GROUP BY**.

```
GROUP BY AttrList
```

## Esempio

Il numero di figli dei padri.

```
SELECT Father, COUNT(*) AS NumberOfChildren  
FROM FATHERHOOD  
GROUP BY Father
```

## FATHERHOOD

Father	Child
Steve	Frank
Louis	Olga
Louis	Phil
Frank	Jim
Frank	James

Father	NumberOfChildren
Steve	1
Louis	2
Frank	2

## Semantica

- Effettua l'interrogazione senza funzioni aggregate e senza operatori aggregati.

```
SELECT *
FROM FATHERHOOD
```

- Effettua il raggruppamento ed applica la funzione aggregata su ciascun gruppo.

## Raggruppamento e target list

Per poter selezionare un campo nella clausola SELECT, esso deve comparire nel GROUP BY.

**Wrong:**

```
SELECT Father, AVG(C.Income), F.Income
FROM PEOPLE C JOIN FATHERHOOD ON C.Name=Child
    JOIN PEOPLE F ON Father=F.Name
GROUP BY Father
```

We also need to  
group by F.Income

**Correct:**

```
SELECT Father, AVG(C.Income), F.Income
FROM PEOPLE C JOIN FATHERHOOD ON C.Name=Child
    JOIN PEOPLE F ON Father=F.Name
GROUP BY Father, F.Income
```

## Condizioni sui gruppi

Fornire quei padri i cui figli hanno un reddito medio superiore a 25; ritorna il padre e il reddito medio dei suoi figli.

```
SELECT Father, AVG(C.Income)
FROM PEOPLE C JOIN FATHERHOOD ON C.Name=Child
GROUP BY Father
HAVING AVG(C.Income) > 25
```

## WHERE vs HAVING

Fornire i padri i cui figli sotto i 30 anni hanno un reddito medio superiore a 20.

```
SELECT Father, AVG(C.Income)
FROM PEOPLE C JOIN FATHERHOOD ON C.Name=Child
WHERE C.Age < 30
GROUP BY Father
HAVING AVG(C.Income) > 20
```

La clausola **WHERE** si usa per fare una selezione sulle righe prima del raggruppamento, mentre la clausola **HAVING** si usa per selezionare solo i gruppi che soddisfano determinate condizioni aggregate.

## Raggruppamento e NULL

Le funzioni aggregate si comportano alla stessa maniera con e senza raggruppamento.

R	A	B
1	11	
2	11	
3	NULL	
4	NULL	

SELECT B, COUNT(*)	B	COUNT(*)
FROM R GROUP BY B	11	2
	NULL	2

SELECT A, COUNT(*)	A	COUNT(*)
FROM R GROUP BY A	1	1
	2	1
	3	1
	4	1

SELECT A, COUNT(B)	A	COUNT(B)
FROM R GROUP BY A	1	1
	2	1
	3	0
	4	0

## Sintassi del SELECT: sommario

```
SELECT AttList1 + Exprs
FROM TableList + Joins
[ WHERE Condition ]
[ GROUP BY AttList2 ]
[ HAVING AggrCondition ]
[ ORDER BY OrderingAttr1 ]
```

# Operazioni di aggiornamento

Tali operazioni sono:

- **INSERT**
- **DELETE**
- **UPDATE**

... di una o più tuple all'interno di una tabella, sulla base di un predicato che può coinvolgere altre relazioni.

## INSERT

```
INSERT INTO Table [(AttList)]
VALUES( Vals )
```

oppure

```
INSERT INTO Table [(AttList)]
SELECT ...
```

## Esempi

```
INSERT INTO PEOPLE(Name, Age, Income)
VALUES('Jack', 25, 52)

INSERT INTO PEOPLE VALUES('John', 25, 52)

INSERT INTO PEOPLE(Name, Income)
VALUES('Robert', 55)

INSERT INTO PEOPLE(Name)
SELECT Father
FROM FATHERHOOD
WHERE Father NOT IN (SELECT Name FROM PEOPLE)
```

## Commenti

L'ordine degli attributi e dei valori **è rilevante**.

Entrambe le liste dovrebbero avere lo stesso numero di argomenti.

Se la lista degli attributi viene omessa, assumiamo che tutti gli attributi vengano considerati e che ogni valore corrisponda ad uno specifico attributo come dichiarato nello schema della relazione.

Se la lista degli attributi non contiene tutti gli attributi della relazione, agli attributi mancanti viene assegnato il valore default o NULL.

## DELETE

```
DELETE FROM Table  
[ WHERE Condition ]
```

## Esempi

```
DELETE FROM PEOPLE  
WHERE Age < 35  
  
DELETE FROM FATHERHOOD  
WHERE Child NOT IN (SELECT Name FROM PEOPLE)  
  
DELETE FROM FATHERHOOD -- tutti i dati
```

## Commenti

Rimuove le tuple che soddisfano una determinata condizione.

- Potrebbe causare la rimozione di ulteriori tuple (se i vincoli sono definiti con CASCADE).
- Se non viene fornita nessuna condizione, essa è da intendersi come WHERE TRUE.

## UPDATE

```
UPDATE TableName  
SET Attribute = < Expr | SELECT ... | NULL | DEFAULT >  
[ WHERE Condition ]
```

## Esempi

BEFORE →

PEOPLE		
Name	Age	Income
Jim	27	30
James	25	15
Bob	55	36

UPDATE PEOPLE

SET Income = 45

WHERE Name = Bob

AFTER →

PEOPLE		
Name	Age	Income
Jim	27	30
James	25	15
Bob	55	45

BEFORE →

PEOPLE		
Name	Age	Income
Jim	27	30
James	25	15
Bob	55	36

UPDATE PEOPLE

SET Income = Income\*1.1

WHERE Age < 30

AFTER →

PEOPLE		
Name	Age	Income
Jim	27	33
James	25	16.5
Bob	55	36

UPDATE PEOPLE

SET Income =  
(SELECT Income FROM  
PEOPLE WHERE Name=Jim)

WHERE Name = Bob

AFTER →

PEOPLE		
Name	Age	Income
Jim	27	30
James	25	15
Bob	55	30

BEFORE →

PEOPLE		
Name	Age	Income
Jim	27	30
James	25	15
Bob	55	36

UPDATE PEOPLE

SET Income = NULL

WHERE Age < 30

AFTER →

PEOPLE		
Name	Age	Income
Jim	27	NULL
James	25	NULL
Bob	55	36

PEOPLE

BEFORE →

Name	Age	Income
Jim	27	30
James	25	15
Bob	55	36

UPDATE PEOPLE

SET Income = DEFAULT  
WHERE Age < 30

AFTER →

PEOPLE		
Name	Age	Income
Jim	27	0
James	25	0
Bob	55	36

Assuming that in CREATE TABLE  
we specified 0 as the DEFAULT  
value for Income

1!

# SQL avanzato

## Altri vincoli

### CHECK

Il **CHECK** specifica vincoli sulle tuple (e possibilmente, anche altri più complessi che non sempre sono supportati in tutte le implementazioni di SQL).

```
CHECK ( Predicate )
```

Ha un costo molto elevato in quanto viene richiamato ad ogni inserimento.

### Esempi

```
create table EMPLOYEE (
    Number      integer primary key,
    Surname     character(20),
    Name        character(20),
    Gender      character not null
        CHECK (Gender in ('M','F')),
    Salary      integer
        CHECK (Salary >= 0),
    Supervisor  integer,
        CHECK (Salary <= (select Salary
                            from EMPLOYEE J
                           where Supervisor = J.Number))
)
```

Una clausola SELECT all'interno del vincolo **CHECK** non è completamente supportata in tutte le implementazioni di SQL, poiché è molto oneroso fare un'interrogazione ad ogni richiamo del check.

```
create table EMPLOYEE (
    Number      integer primary key,
    Surname     character(20),
    Name        character(20),
    Gender      character not null
        CHECK (Gender in ('M','F')),
    Salary      integer,
    Withholding integer,
    Net         integer,
    Supervisor  character(6),
        CHECK (Net = Salary - Withholding)
)
```

```
INSERT INTO EMPLOYEE VALUES
(1 , 'Doe', 'John', '' , 100, 20, 80);
```

Errore: *Gender* non è né 'M' né 'F'.

```
INSERT INTO EMPLOYEE VALUES  
(2 , 'Lee', 'Jim', 'M', 100, 10, 80);
```

Errore: Net (80) non è 100-10.

```
INSERT INTO EMPLOYEE VALUES  
(3, 'Hill', 'Sam', 'M', 70, 20, 50);
```

Questo aggiornamento è permesso.

## ASSERTION

Un'asserzione definisce vincoli a livello di schema. Si può definire solo dopo aver creato la tabella.

```
CREATE ASSERTION NameAs CHECK (Predicate)  
  
CREATE ASSERTION AtLeastOneEmployee  
CHECK (1 <= (SELECT COUNT(*) FROM EMPLOYEE))
```

Una clausola SELECT all'interno del vincolo **CHECK** non è completamente supportata in tutte le implementazioni di SQL.

## View

### Creare una vista

```
CREATE VIEW ViewName [ ( AttrList ) ] AS  
SELECT ...  
[ WITH [ LOCAL | CASCDED ] CHECK OPTION ]
```

### Esempio

```
CREATE VIEW ADMINEMPLOYEES (Name, Surname, Salary) AS  
SELECT Name, Surname, Salary  
FROM EMPLOYEE  
WHERE Dept = 'Administration' AND Salary > 10
```

# Aggiornare una vista

Solitamente, tali aggiornamenti sono consentiti solo per le viste generate a partire da una sola relazione (tabella). Possiamo forzare il database ad effettuare alcuni controlli.

## CHECK OPTION

```
CREATE VIEW POORADMINEMPLOYEES AS
  SELECT *
    FROM ADMINEMPLOYEES
   WHERE Salary < 50
 WITH CHECK OPTION
```

**CHECK OPTION** permette di aggiornare la vista solamente se la tupla inserita appartiene alla vista stessa (l'utente non può avere un salario superiore a 50).

## Esempio di operazione non permessa

```
CREATE VIEW POORADMINEMPLOYEES AS
  SELECT *
    FROM ADMINEMPLOYEES
   WHERE Salary < 50
 WITH CHECK OPTION

UPDATE POORADMINEMPLOYEES
  SET Salary = 60
 WHERE Name = 'Ann'
```

## LOCAL e CASCADED

**LOCAL** (nel caso di viste generate da viste): l'aggiornamento delle tuple deve essere effettuato solo sulla vista all'ultimo livello.

**CASCADED** (nel caso di viste generate da viste): l'aggiornamento delle tuple deve essere effettuato su tutte le viste e relazioni sottostanti.

## Interrogare una vista

Le viste possono essere interrogate come ogni altra relazione all'interno del database.

```
SELECT * FROM ADMINEMPLOYEES
```

è come effettuare la seguente query (ed è eseguita come):

```
SELECT Name, Surname, Salary  
FROM EMPLOYEE  
WHERE Dept = 'Administration' AND Salary > 10
```

## Query SQL errata

Fornire il numero medio di uffici per dipartimento.

```
SELECT AVG(COUNT(DISTINCT Office))  
FROM EMPLOYEE  
GROUP BY Dept
```

Questa query è errata perché la sintassi SQL non permette di annidare operatori di aggregazione.

## Query SQL corretta

Fornire il numero medio di uffici per dipartimento.

Usando una vista intermedia:

```
CREATE VIEW DEPTOFFICES(NameDept,OffNum) AS  
SELECT Dept, COUNT(DISTINCT Office)  
FROM EMPLOYEE  
GROUP BY Dept;  
  
SELECT AVG(OffNum)  
FROM DEPTOFFICES;
```

## Altra query SQL errata

Fornire i dipartimenti che hanno il più alto monte ingaggi.

Una soluzione errata per alcuni sistemi:

```
SELECT Dept
FROM EMPLOYEE
GROUP BY Dept
HAVING SUM(Salary)>=ALL(SELECT SUM(Salary)
                           FROM EMPLOYEE
                           GROUP BY Dept)
```

Una clausola **HAVING** annidata non è consentita in alcune implementazioni SQL.

## Altra query SQL corretta

Fornire i dipartimenti che hanno il più alto monte ingaggi.

```
CREATE VIEW BUDGETSALARY(Dept,TotalSalary) AS
  SELECT Dept, SUM(Salary)
  FROM EMPLOYEE
  GROUP BY Dept;

  SELECT Dept
  FROM BUDGETSALARY
  WHERE TotalSalary = (SELECT MAX(TotalSalary)
                        FROM BUDGETSALARY)
```

## Query ricorsive

Per ogni persona fornire i suoi antenati, avendo:

**FATHERHOOD(Father, Child)**

FATHERHOOD

Father	Child
Carl	Frank
Louis	Olga
Louis	Bob
Frank	Alex
Frank	Alfred

# Datalog

Dobbiamo usare la ricorsione, in Datalog (linguaggio formale):

```
ANCESTORS(Anccestor: f, Descendant: c) ← FATHERHOOD(Father: f, Child: c)
ANCESTORS(Anccestor: a, Descendant: d) ← FATHERHOOD(Father: a, Child: c),
ANCESTORS(Anccestor: c, Descendant: d)
```

## SQL:1999

```
WITH RECURSIVE ANCESTORS(Anccestor, Descendant) AS
(
    SELECT Father, Son
    FROM FATHERHOOD
    UNION ALL
    SELECT Ancestor, Son
    FROM ANCESTORS, FATHERHOOD
    WHERE Descendant = Father
)

SELECT *
FROM ANCESTORS
```

**WITH** definisce la vista ANCESTORS, che è costruita ricorsivamente usando FATHERHOOD.

## Esempio

Ritorna tutti i supervisori di John Doe.

```
WITH RECURSIVE INCHARGE(Num, Supervisor) AS (
    SELECT Num, Supervisor
    FROM EMPLOYEE
    UNION
    SELECT Employee.Num, INCHARGE.Supervisor
    FROM EMPLOYEE, INCHARGE
    WHERE EMPLOYEE.Supervisor = INCHARGE.Num
)

SELECT Name, Surname, INCHARGE.Supervisor
FROM EMPLOYEE JOIN INCHARGE ON (EMPLOYEE.Num = INCHARGE.Num)
WHERE Name = 'John' AND Surname = 'Doe'
```

# Nuove espressioni

## Funzioni scalari

Funzioni a livello di tuple che forniscono un singolo valore per tupla.

- Temporali
  - **CURRENT\_DATE()** ritorna la data attuale.
  - **EXTRACT(*yearExpression*)** estrae parte di una data a partire da un'espressione (es. mese, giorno, ora, ...)
- Manipolazione stringhe
  - **CHAR\_LENGTH()** ritorna la lunghezza della stringa.
  - **LOWER()** converte la stringa in lowercase.
  - **UPPER()** converte la stringa in uppercase.
- Casting
  - **CAST()** permette il casting di un valore verso un altro dominio.
- Condizionali
  - **COALESCE, NULLIF, CASE**

## Esempio

Ritorna l'anno dell'ordine degli ordini emessi oggi.

```
SELECT EXTRACT(YEAR FROM OrderDate) AS OrderYear  
FROM ORDERS  
WHERE DATE(OrderDate)=CURRENT_DATE()
```

## COALESCE

**COALESCE** prende numerose espressioni come input e ritorna la prima di esse che non è NULL.

Data la relazione EMPLOYEE(Number, Dept, Mobile, PhoneHome),  
per ogni impiegato ritorna un numero di telefono valido, altrimenti il numero di casa.

```
SELECT Number, COALESCE(Mobile, PhoneHome)  
FROM EMPLOYEE
```

**COALESCE** potrebbe essere usata per fornire un valore default come rimpiazzo per valori NULL. Nel seguente esempio, un dipartimento NULL viene rimpiazzato con "None".

```
SELECT Name, Surname, COALESCE(Dept, "None")
FROM EMPLOYEE
```

## NULLIF

**NULLIF** confronta il primo argomento (es. un attributo di una relazione) con il secondo (es. un valore costante). Ritorna NULL se i due argomenti combaciano, altrimenti ritorna il valore del primo argomento.

Estrai i cognomi e i dipartimenti a cui gli impiegati appartengono, ritornando un valore NULL per il dipartimento quando l'attributo *Dept* assume il valore 'Unknown'.

```
SELECT Surname, NULLIF(Dept, "Unknown")
FROM EMPLOYEE
```

## CASE

La funzione **CASE** permette di definire strutture condizionali, il cui risultato dipende dalla valutazione del contenuto della tabella.

È usata per fornire una logica del tipo "if-then-else" al linguaggio SQL.

### Esempio

Calcola il bollo dei veicoli secondo il tipo e l'anno di registrazione (dal 1975 in poi).

VEHICLE(PlateNum, Type, Year, KWatt, Length)

```
SELECT PlateNum,
(CASE Type
    WHEN 'Car' THEN 2.58 * KWatt
    WHEN 'Moto' THEN (22.00 + 1.00 * KWatt)
    ELSE NULL
END) AS Tax
FROM VEHICLE
WHERE Year > 1975
```

**ELSE NULL** non è obbligatorio da mettere, però chiarifica che se non trova i valori cercati non fa nulla.

# Sicurezza dei database

I DBMS solitamente lavorano con dati in chiaro, quindi la sicurezza si basa su chi può fare cosa. SQL permette di concedere privilegi (lettura, scrittura, ...) per ogni specifico utente, per l'intero database o per una parte di esso.

I privilegi possono essere garantiti su relazioni, attributi, viste o domini.

C'è almeno un utente amministratore default (es. `_system`) per il quale tutti i privilegi sono concessi.

Qualunque utente crei una specifica risorsa ottiene automaticamente tutti i privilegi su di essa.

## Privilegi

Un privilegio è descritto da:

- una specifica **risorsa**
- l'**utente che concede** il privilegio
- l'**utente al quale viene concesso** il privilegio
- una specifica **operazione**
- la possibilità per l'utente di **propagare** il privilegio o meno

## Tipi di privilegio

**insert** consente di inserire nuove tuple.

**update** consente di modificare tuple esistenti.

**delete** consente di eliminare tuple.

**select** consente di leggere una risorsa.

**references** consente di definire vincoli di integrità referenziale.

**usage** consente di usare una definizione (es. un custom data type).

## Concedere privilegi

Concessione di privilegi:

```
GRANT < PRIVILEGES | ALL PRIVILEGES >
    ON Resource
    TO Users [ WITH GRANT OPTION ]
```

**GRANT OPTION** permette all'utente di propagare i suoi privilegi ad altri utenti.

```
GRANT SELECT ON DEPARTMENT TO Jack
```

# Revocare privilegi

Revocazione di privilegi:

```
REVOKE PRIVILEGES ON Resource  
FROM Users [ RESTRICT | CASCADE ]
```

I privilegi devono essere revocati dallo stesso utente che li ha concessi in primo luogo.

- **RESTRICT** (default): la revoca non coinvolge altri utenti che hanno ricevuto la concessione dall'utente in questione.
- **CASCADE**: la revoca è estesa agli altri utenti. Attenzione, causa una "reazione a catena".

## Commenti

L'implementazione di SQL deve nascondere (senza lasciare indizi) le parti del database che non sono accessibili agli utenti.

Per esempio, sia nel caso in cui la tabella non esiste che nel caso in cui la tabella esiste ma l'utente non può accedervi, l'utente deve ricevere lo stesso messaggio dal sistema.

Potremmo usare una view per mostrare solo specifiche tuple ad un utente.

- La vista è definita con un predicato di selezione.
- Il privilegio è concesso per la specifica vista.

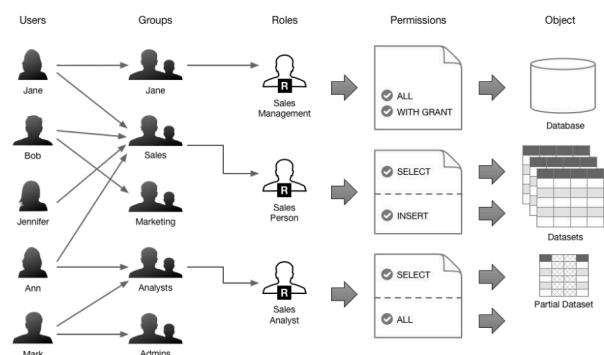
## Autorizzazioni

SQL-3 ha introdotto **RBAC** per la prima volta (Role-Based Access Control).

Ogni **ruolo** agisce come un contenitore di numerosi privilegi che possono essere concessi tramite il comando **GRANT**.

In ogni momento ciascun utente ha:

- privilegi concessi individualmente, associati direttamente a sé,
- privilegi concessi al suo ruolo tramite RBAC.



Il seguente comando crea un nuovo ruolo *Name*:

```
CREATE ROLE Name
```

Questo comando assegna il ruolo *Name* all'utente attuale:

```
SET ROLE Name
```

## Esempio

Concedere il privilegio **CREATE TABLE** ad uno specifico utente tramite il ruolo *Employee*.

1. Creare il nuovo ruolo:

```
CREATE ROLE Employee;
```

2. Concedere il privilegio al ruolo appena creato:

```
GRANT CREATE TABLE TO Employee;
```

3. Concedere il ruolo allo specifico utente:

```
GRANT Employee TO user;
```

4. Revocare il privilegio concesso precedentemente:

```
REVOKE CREATE TABLE FROM Employee;
```

## Transazioni

Una transazione è un programma in esecuzione che forma un'unità logica di elaborazione del database (operazione atomica).

Proprietà:

- **A**tomicity
- **C**onsistency preservation
- **I**solation
- **D**urability (permanenza)

## Atomicity

Una transazione è un'unità atomica di elaborazione: deve essere processata per intero oppure non processata per niente.

*Trasferimento di denaro da un conto bancario A a B: il denaro viene prelevato da A e depositato su B, altrimenti nessuna operazione viene effettuata.*

## Consistency

Una transazione deve preservare la consistenza: se viene completamente eseguita dall'inizio alla fine senza interferenze, il database si muove tra due stati consistenti.

Durante l'esecuzione di una transazione potrebbero verificarsi alcune violazioni, ma esse non possono rimanere al termine della transazione. Se alla conclusione di una transazione ci sono ancora violazioni, la transazione deve essere cancellata completamente (abort).

## Isolation

Anche se numerose transazioni sono eseguite concorrentemente, l'esecuzione della transazione attuale non deve interferire con le altre.

*Ad esempio, il caso di due trasferimenti di denaro dallo stesso conto bancario che avvengono "allo stesso momento".*

## Durability

I cambiamenti applicati al database da una transazione corretta (*committed*) devono persistere e non essere persi, anche quando avvengono guasti (hardware o software) e nelle esecuzioni parallele.

## Supporto in SQL

Una volta che viene stabilita la connessione al database, la prima transazione comincia con **START TRANSACTION**, sia prima della prima operazione che dopo l'ultima (il comando è opzionale nella maggior parte degli RDBMS).

La transazione può terminare con:

- **COMMIT [WORK]**: le operazioni vengono salvate nel database.
- **ROLLBACK [WORK]**: le operazioni vengono scartate e il database ritorna al precedente stato "sicuro".

La maggior parte degli RDBMS ha **AUTOCOMMIT**, ovvero ogni statement è una transazione differente.

## Esempio

Preleva 10€ dal conto 42177 e trasferiscili al conto 12202.

```
START TRANSACTION
UPDATE BANKACCOUNT
SET Balance = Balance - 10
WHERE AccountNumber = 42177;
UPDATE BANKACCOUNT
SET Balance = Balance + 10
WHERE AccountNumber = 12202;
COMMIT WORK;
```

# Modellazione concettuale dei dati

## Introduzione

Proviamo a costruire un database relazionale partendo direttamente dal modello logico:

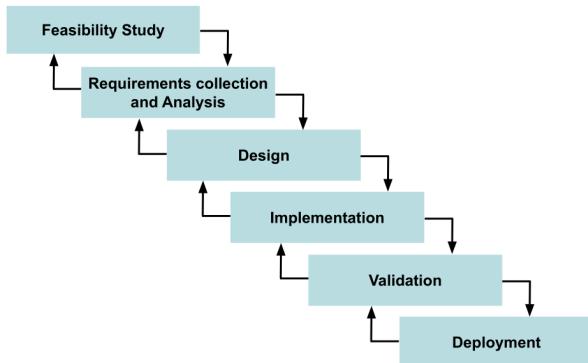
- Da dove dovremmo partire?
- È rischioso andare ulteriormente in dettaglio.
- Dobbiamo definire relazioni tra le tabelle.
- Il modello relazionale è troppo "rigido" per modellare.

## Progettazione di database

La progettazione è una delle attività appartenenti al processo di sviluppo di un Sistema Informativo. Bisogna vederlo in un contesto più generale:

**Ciclo di vita di un Sistema Informativo (Information System, IS)**: l'insieme delle attività effettuate sequenzialmente da analisti, architetti del software, utenti, ... durante lo sviluppo ed utilizzo del sistema informativo. Attività iterative → "cicli".

## Ciclo di vita del software



Lo sviluppo segue una modalità a cascata poiché l'output di una fase diventa l'input della fase successiva.

### Fasi:

- *Studio di fattibilità*: analisi di cosa fa un'azienda, la sua filosofia... Definizione di costi e priorità. L'output evidenzia cosa è necessario fare, i rischi, i tempi. Sarebbe da far eseguire ad un attore esterno.
- *Raccolta e analisi dei requisiti*: analisi delle proprietà del sistema da realizzare. Deve essere dettagliata e approvata con il committente. Va controfirmata da sviluppatore e cliente.

- *Progettazione*: analisi dei dati da memorizzare e sviluppo dei metodi da realizzare. L'output è un progetto esecutivo (implementabile). Anche questo documento va approvato e firmato.
  - *Implementazione*: realizzazione del progetto.
  - *Validazione*: check del modello e collaudo (debug).
  - *Deployment*: messa in esecuzione del sistema. Comprende la formazione del personale per essere sicuri che il sistema venga utilizzato correttamente. Da subito dopo il deployment il prodotto entra in *manutenzione*.

Dover passare da una determinata fase ad una più indietro è un indicatore di lavoro svolto male ai piani superiori.

# Progettazione

La progettazione di un Sistema Informativo coinvolge due aspetti:

- Progettazione dei **dati**
  - Progettazione dei **programmi applicativi**

Ma i dati hanno un ruolo chiave, sono più stabili.

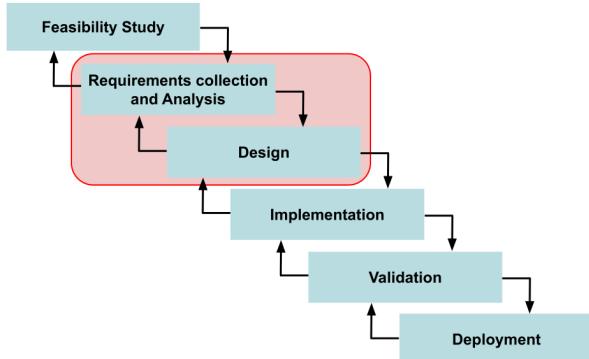
## **Metodologia di progettazione**

I progetti di buona qualità seguono sistematicamente una **metodologia di progettazione**, che consiste di:

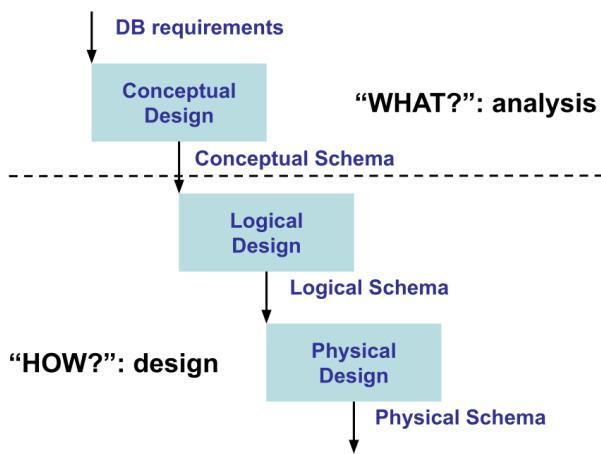
- Definire attività separate, quando si fa una cosa si pensa solo a quella
  - Criteri di selezione, scegliere cosa fare
  - Modelli di rappresentazione, abbastanza generali e facili da utilizzare
  - Soluzioni generali e user-friendly

# Requisiti e progettazione

Noi ci concentreremo principalmente su analisi dei requisiti e progettazione.



La fase di progettazione è in realtà composta da più sotto-fasi: prima si capisce **cosa** va fatto e poi **come** farlo. La progettazione fisica non esiste praticamente più.



## Schemi concettuali, logici e fisici

Il risultato di ciascuna fase di progettazione corrisponde ad uno schema che è utilizzato nella fase successiva:

- Schema concettuale
- Schema logico
- Schema fisico

## Modello di dati

Una collezione di componenti usata per categorizzare dati rilevanti e descrivere operazioni su di essi.

Il componente fondamentale è il costruttore, che genera i costrutti.

I costruttori giocano lo stesso ruolo delle definizioni dei tipi di dato nei linguaggi di programmazione. Ad esempio, il modello relazionale definisce il costruttore di relazioni per uniformare insiemi di tuple.

## Schema e istanze

Per ogni database ci sono:

- Uno **schema**, indipendente dal tempo, che descrive la struttura dei dati (aspetto intensionale). Nel modello dati relazionale sono gli attributi delle relazioni.
- Un'**istanza**, i valori attuali, che potrebbero cambiare nel tempo anche molto velocemente (aspetto estensionale). Nel modello dati relazionale è l'insieme di tuple uniformi.

## Due tipi di modelli

- **Modelli logici:** organizzano i dati all'interno dei DBMS.
  - livello di astrazione dei dati utilizzato dal software (framework di persistenza)
  - indipendente dalla progettazione fisica
  - es: relazionale, a grafo, gerarchico, ad oggetti
- **Modelli concettuali:** consentono di rappresentare i dati indipendentemente dal sistema.
  - provano a descrivere concetti del mondo reale
  - sono usati nelle fasi preliminari della progettazione software
  - il più utilizzato è il modello **Entity-Relationship** (entità-associazione).

## Perché usare i modelli concettuali?

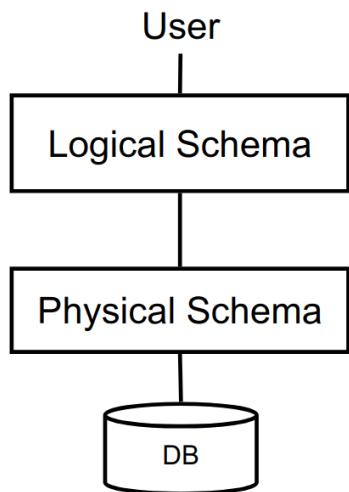
Consentono di ragionare sulla realtà di interesse definendo un modello indipendente dall'implementazione. Utile quando si deve progettare un database poiché aiuta a rappresentare i dati in modo astratto.

Permettono di definire classi di oggetti di interesse e le loro relazioni.

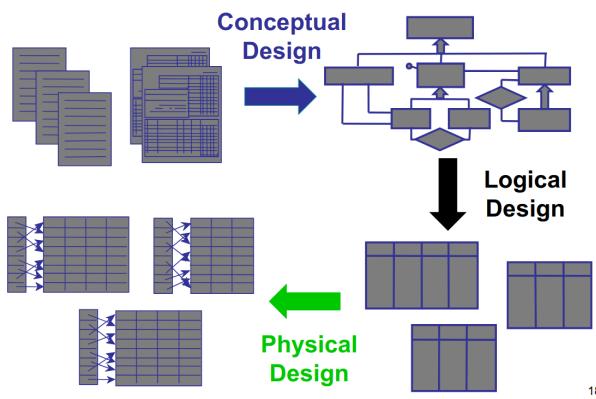
Forniscono rappresentazioni visive efficienti (utile a scopo di documentazione).

Sono utili inoltre per evitare di affrontare dettagli che ancora non servono, per forzare la separazione delle fasi di sviluppo.

## Architettura del DBMS

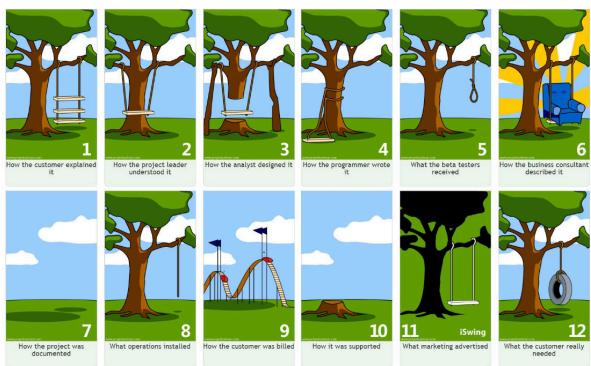


# Fasi della progettazione



18

## Gag time



# Modello Entity-Relationship (ER)

Il modello dati concettuale più utilizzato. Ci sono molte versioni più o meno simili.

## Costruttori ER

- Entity
- Relationship
- Attribute
- Key attribute
- Generalization
- ...

## Entità

**Classe di "oggetti"** (cose, persone, luoghi) che appartengono alla realtà di interesse, condividono alcune proprietà in comune e hanno un'esistenza autonoma.

Esempio: *Employee, City, BankAccount, SalesOrder, Bill*.

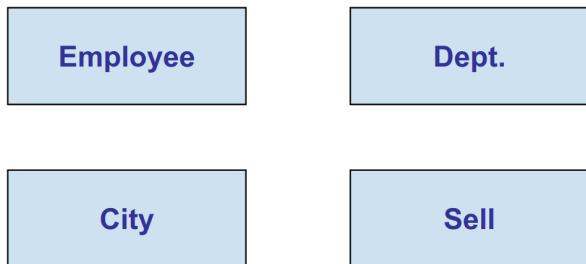
## Schema ed istanza

**Schema:** classe uniforme di oggetti.

**Istanza:** un elemento all'interno della classe (un'istanza, non il singolo dato).

Le entità degli schemi ER non rappresentano ogni possibile istanza (astrazione).

## Rappresentazione



## Commenti

Ciascuna entità ha un nome unico all'interno dello schema. Usare nomi significativi ed al singolare.

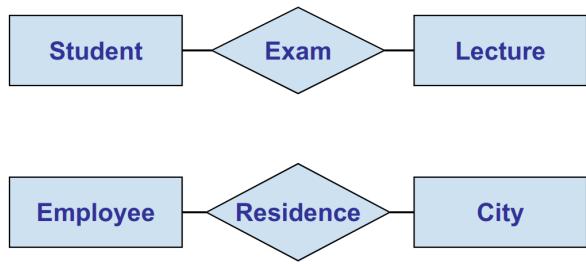
## Associazione

Stabilisce **associazioni** tra due o più tipi di entità all'interno del modello.

Esempio: *Residence* (tra *Employee* e *City*) o *Exam* (tra *Student* e *Lecture*).

È anche chiamata, un po' impropriamente, correlazione.

## Rappresentazione

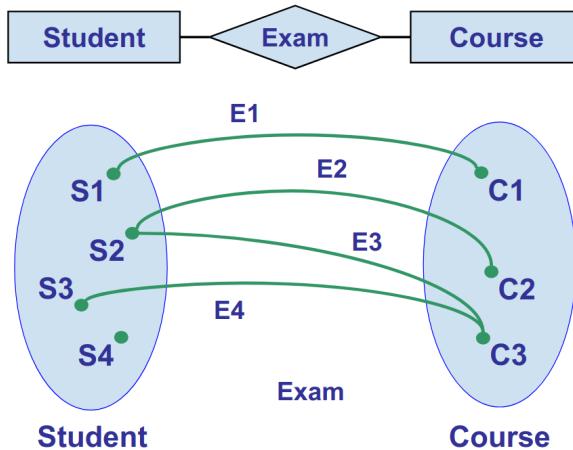


## Commenti

Ciascuna associazione ha un nome unico all'interno dello schema. Usare nomi significativi e nomi singolari al posto dei verbi (quando possibile).

## Schema vs istanza

Lo schema serve per generalizzare ciò che rappresentiamo, le istanze determinano l'associazione tra elementi specifici.



## Tipi di associazione

Un'istanza di associazione binaria è una coppia di istanze di entità, una per ogni entità coinvolta.

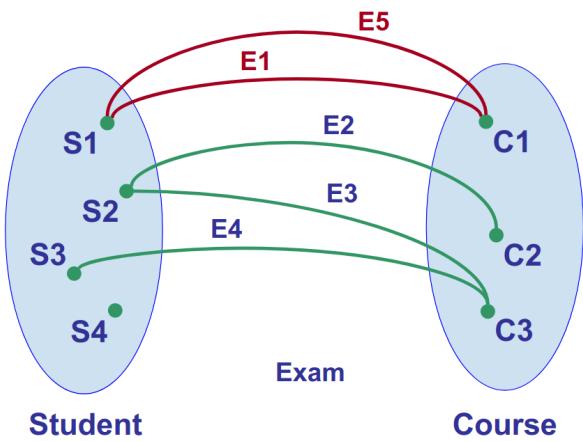
Un'istanza di associazione n-aria è una tupla di istanze di entità, una per ogni entità coinvolta.

Non ci possono essere istanze ripetute (coppie o tuple) in un'associazione.

## Sono corrette queste associazioni?



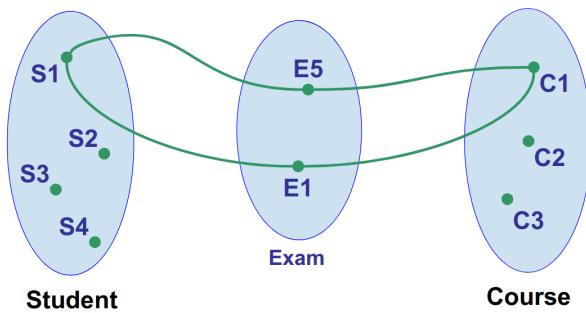
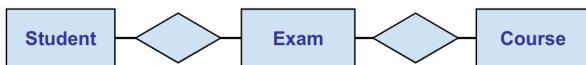
Se lo studente deve fare più esami, l'associazione non va bene.



## Promuovere associazioni ad entità

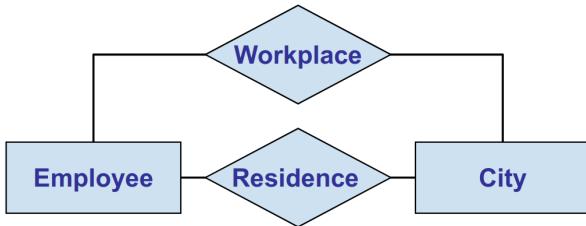


*Exam* non cattura il requisito "uno studente può sostenere lo stesso esame più volte per uno stesso corso". La soluzione è rappresentare *Exam* come un'entità, connessa tramite due associazioni a *Student* e *Course*.



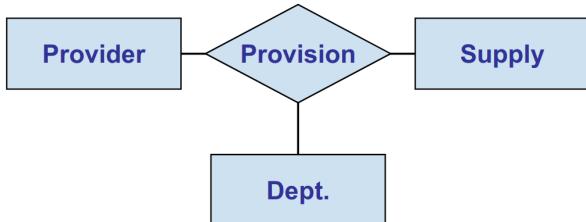
## Più associazioni tra due entità

Disponendo di due entità, può avere senso collegarle con associazioni differenti.

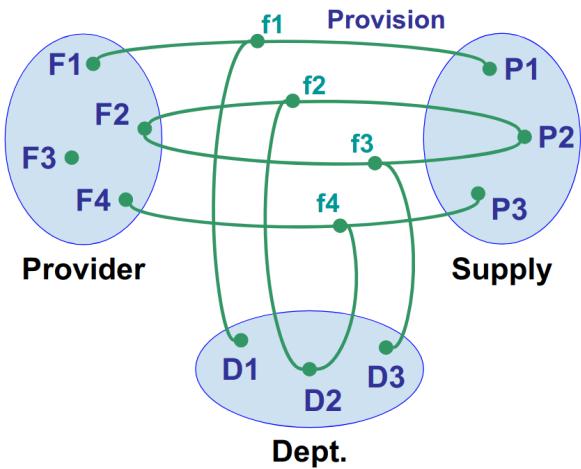


## Associazione n-aria

Possiamo avere associazioni che collegano più di due entità. In questo caso ternaria:

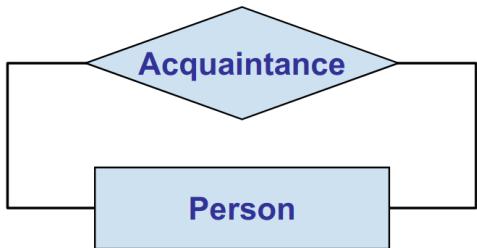


Esempio d'istanza:



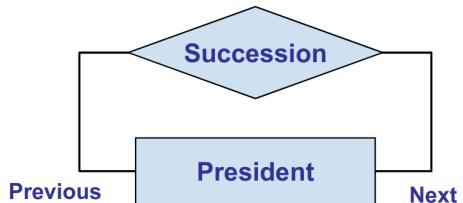
## Associazione ricorsiva

Coinvolge la stessa entità due volte.

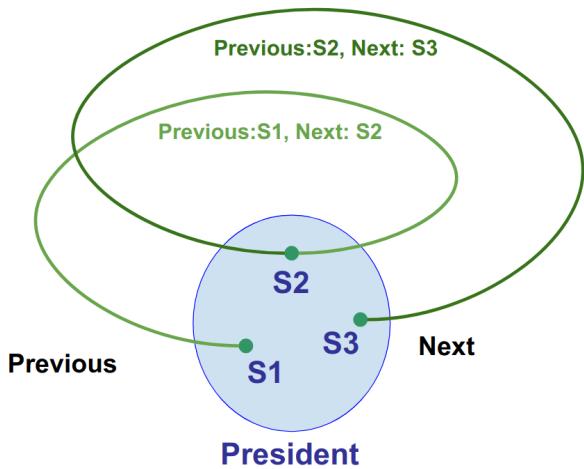


## Associazione con ruoli

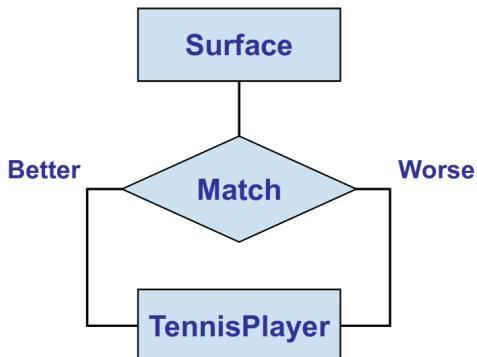
Possiamo definire un verso dell'associazione assegnando dei ruoli.



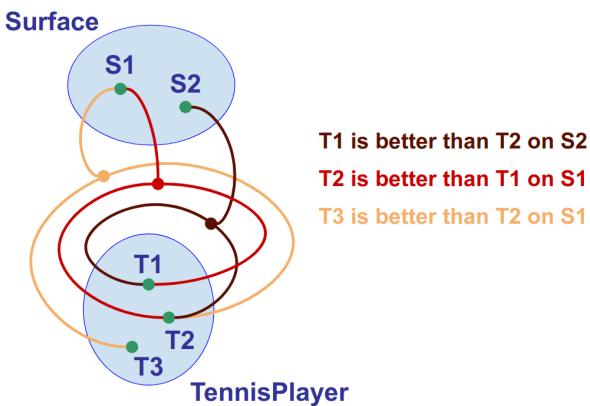
Esempio d'istanza:



## Associazione ternaria ricorsiva



Esempio d'istanza:

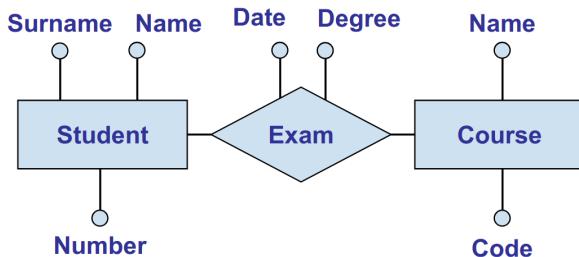


## Attributo

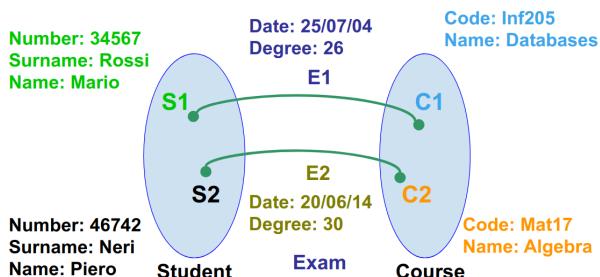
Un attributo è una **proprietà** che appartiene ad un'entità o ad un'associazione.

Esso lega ciascuna entità o associazione ad un valore appartenente ad un insieme (di valori) detto dominio.

## Rappresentazione



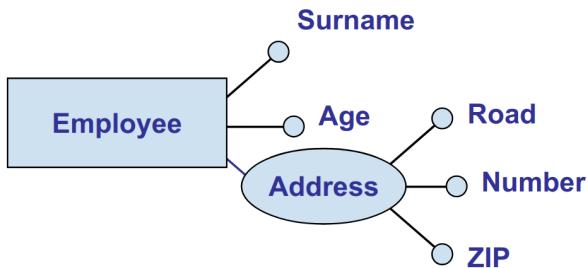
Esempio d'istanza:



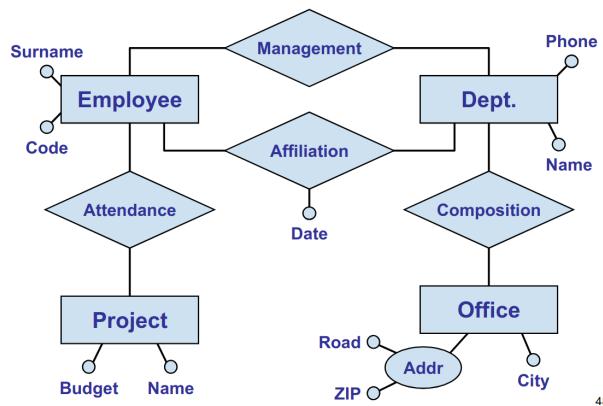
## Attributi composti

Gli attributi composti raggruppano insieme attributi della stessa entità o associazione che mostrano similarità nei loro significati o usi.

Esempio: *Road*, *Number* e *ZIP* definiscono *Address*.



## Esempio di schema ER



48

## Altri costruttori

- Cardinalità
  - per associazioni
  - per attributi
- Chiave
  - interna (all'entità)
  - esterna (all'entità)
- Generalizzazione

## Cardinalità delle associazioni

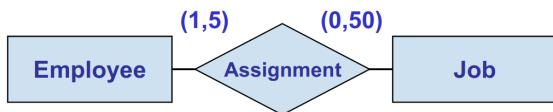
È una coppia di valori legata ad ogni entità coinvolta nell'associazione.

Specifica il numero minimo e massimo di occorrenze dell'associazione a cui ogni istanza di entità può partecipare. In soldoni è il numero minimo e massimo di entità dall'altro lato con cui può relazionarsi la prima entità.

## Esempio

Un impiegato ha almeno un lavoro e al massimo 5 lavori.

Ogni lavoro può essere assegnato al massimo a 50 impiegati, ma potrebbe non essere assegnato a nessun impiegato.

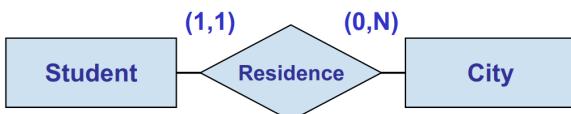
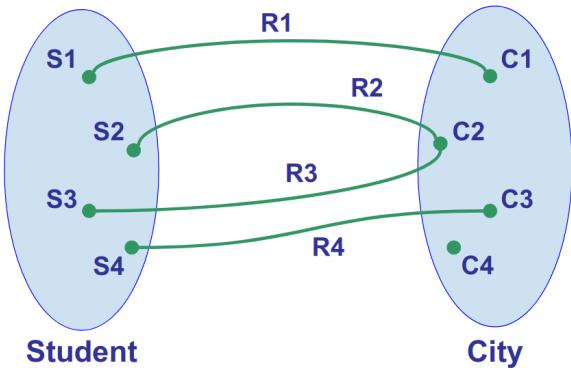


## Simbologia

Per semplicità utilizzeremo solo tre simboli:

- 0 e 1 per la cardinalità minima
  - 0 indica partecipazione opzionale
  - 1 indica partecipazione obbligatoria
- 1 e N per la cardinalità massima
  - 1 indica al massimo uno
  - N indica un qualsiasi numero senza restrizioni

## Esempio

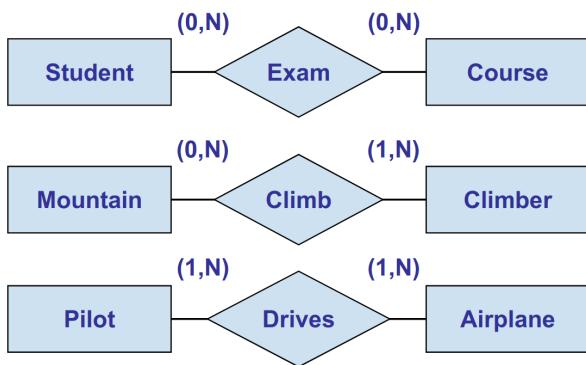


## Tipi di associazione

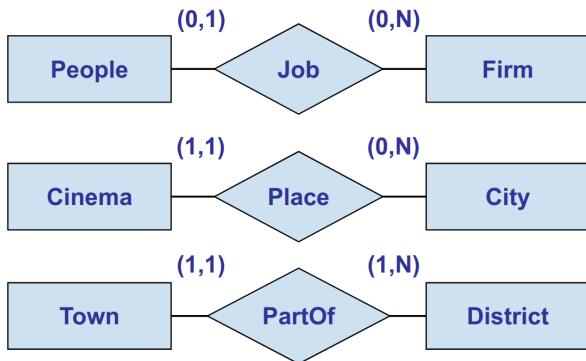
Facendo riferimento alle cardinalità massime, abbiamo:

- Uno a uno
- Uno a molti
- Molti a molti

## Associazioni molti a molti



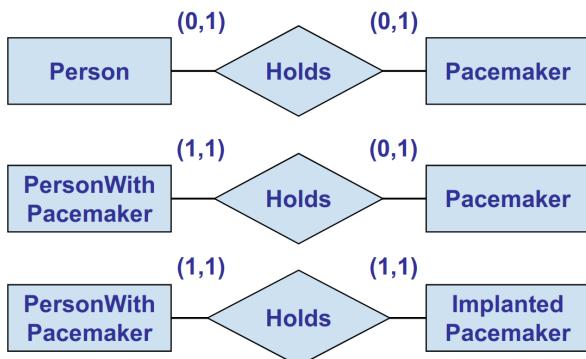
## Associazioni uno a molti



Fare attenzione alla direzione delle associazioni uno a molti.

Le associazioni con partecipazione obbligatoria da entrambi i lati non sono molto frequenti.

## Associazioni uno a uno

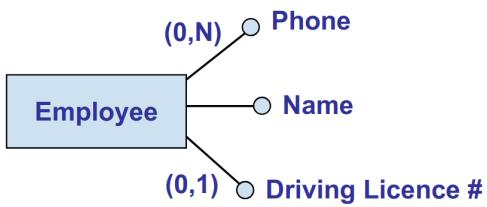


## Cardinalità degli attributi

È possibile legare la cardinalità anche agli attributi, con due scopi finali:

- indicare opzionalità (informazione parziale), può esserci o non esserci
- indicare attributi multivalore, simili ad una lista

## Rappresentazione

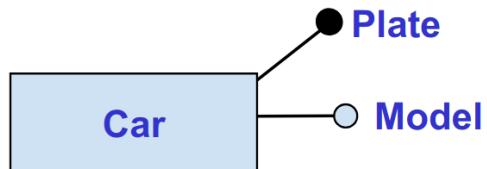


## Identificatore di entità

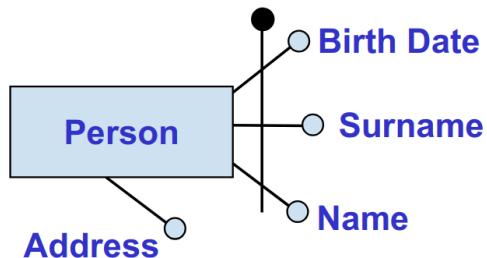
Un modo per identificare univocamente le occorrenze di un'entità. È formato da:

- attributi dell'entità → **identificatore interno**
- (attributi +) entità esterne raggiungibili tramite associazioni → **identificatore esterno** (rispetto all'entità da identificare)

## Identificatore interno



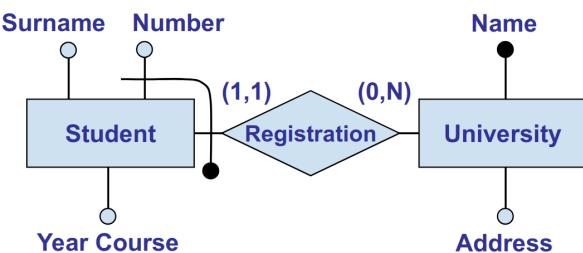
L'attributo *Plate* è l'identificatore interno di *Car* perché non ci sono due macchine con la stessa targa.



Gli attributi *Name*, *Surname* e *Birth Date* definiscono l'identificatore interno di *Person*, perché (assumiamo che) due persone con lo stesso nome, cognome e data di nascita non esistono.

## Identificatore esterno

Comprende uno o più attributi da un'altra entità con cui è in associazione per avere un'identità univoca.



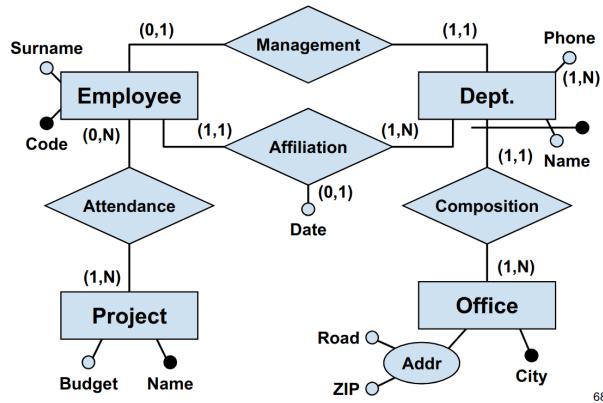
L'attributo *Number* e l'entità *University* compongono l'identificatore esterno di *Student* perché potrebbero esserci studenti con lo stesso numero (matricola) ma appartenenti ad università diverse.

## Osservazioni

Ciascuna entità deve avere almeno un identificatore, ma generalmente può averne più di uno.  
È possibile avere un identificatore esterno solo se l'entità da identificare partecipa all'associazione con una cardinalità (1,1).

Perché le associazioni non hanno identificatori? È meglio associare gli attributi alle entità piuttosto che alle associazioni.

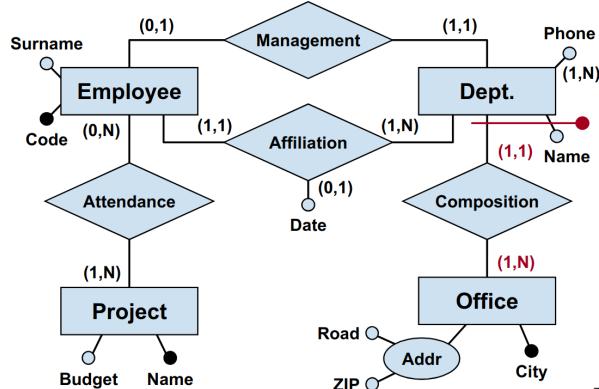
## Esempio di schema ER aggiornato



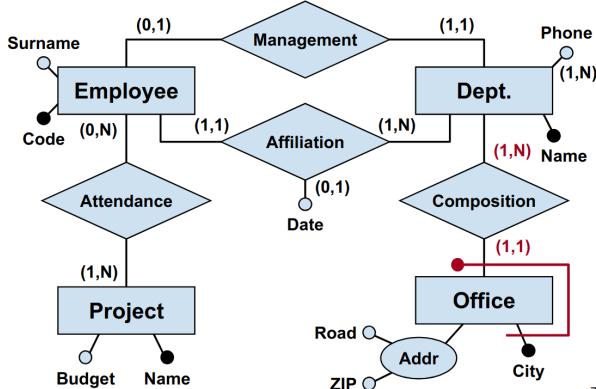
66

## Attenzione

Piccoli cambiamenti nelle cardinalità e negli identificatori potrebbero alterare il significato dell'intero diagramma.



70



71

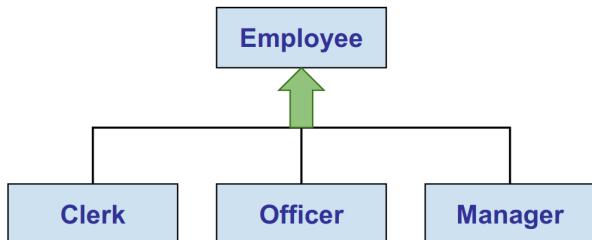
Nel primo diagramma per ogni *Dept.* c'è un solo *Office*: potremmo avere più dipartimenti con lo stesso nome, l'unico vincolo è che la città dei dipartimenti sia diversa.

Nel secondo diagramma ogni *Office* appartiene ad un solo *Dept.*: quindi nella stessa città potremmo avere molti dipartimenti diversi con nomi diversi.

# Generalizzazione

- Lega una o più entità  $E_1, \dots, E_n$  con un'altra entità  $E$ , che ha in  $E_i$  dei casi specifici.
- $E$  è una **generalizzazione** di  $E_1, \dots, E_n$
  - $E_1, \dots, E_n$  sono **specializzazioni** di  $E$

## Rappresentazione

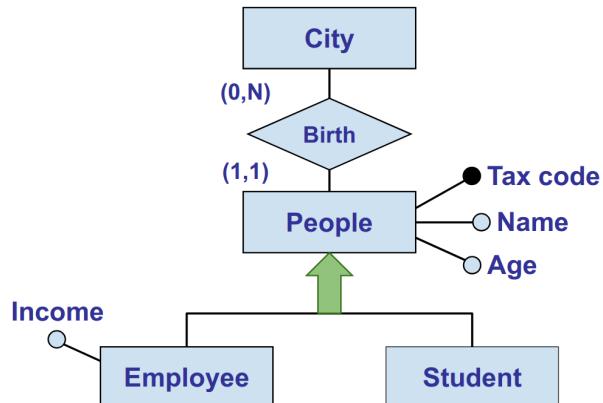


## Proprietà della generalizzazione

Se  $E$  (parent) è una generalizzazione di  $E_1, \dots, E_n$  (children), allora:

- Ogni proprietà di  $E$  è rilevante per  $E_1, \dots, E_n$
- Ogni occorrenza di  $E_1, \dots, E_n$  è anche un'occorrenza di  $E$

## Esempio



## Ereditarietà

Tutte le proprietà dell'entità padre (attributi, associazioni, generalizzazioni) sono ereditate dall'entità figlia e non devono essere rappresentate esplicitamente.

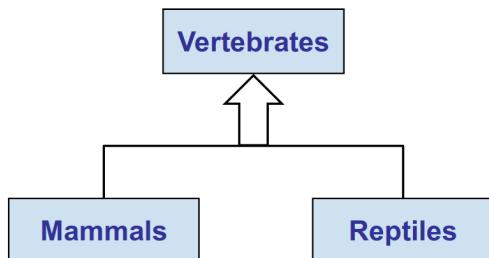
## Tipi di generalizzazione

Si dice **totale** se ogni occorrenza dell'entità padre è un'occorrenza di almeno una delle entità figlie, **parziale** se non accade.

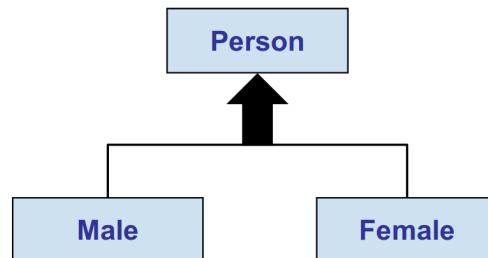
Si dice **disgiunta** se ogni occorrenza dell'entità padre è un'occorrenza di al massimo una delle entità figlie, **sovrapponente** altrimenti.

Noi considereremo soltanto generalizzazioni disgiunte e faremo distinzione tra totali e parziali.

### Generalizzazione disgiunta parziale



### Generalizzazione disgiunta totale



## Altre proprietà

Potremmo definire gerarchie multi-livello e generalizzazioni allo stesso livello.

Ciascuna entità potrebbe essere inclusa in più gerarchie differenti, sia come padre che come figlia.

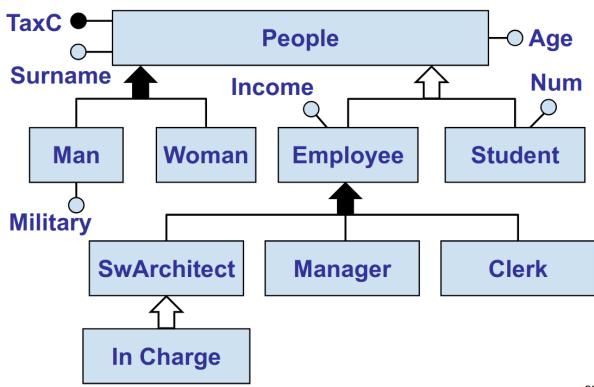
Una generalizzazione che ha solo un'entità figlia è detta sottoinsieme (subset).

Alcune configurazioni non hanno senso.

L'entità padre in una generalizzazione potrebbe non avere un identificatore, a patto che sia presente in tutte le entità figlie.

## Esercizio

People have Tax Code, Surname and Age; for each man we want to know whether he did military service; employees have an income and could be either managers, clerks or (software) architect (some of whom are in charge of a project); Students (that cannot be employees) have a (registration) Number; some people are neither Employees nor Students (and we don't need further details).



## Documentazione

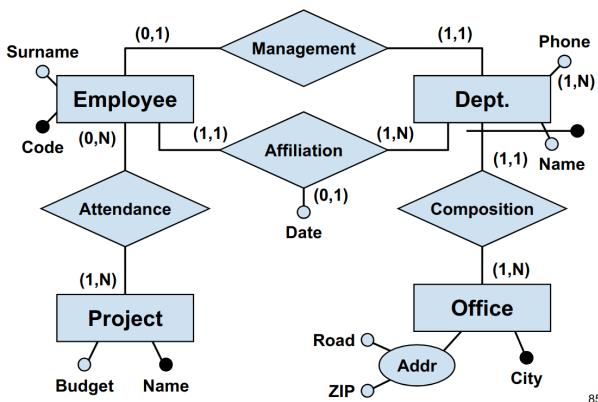
- Dizionario dei dati
  - Entità
  - Associazioni
- Vincoli non esprimibili

## Documentazione dello schema concettuale

Ci viene suggerito di produrre due documenti in fase di progettazione concettuale:

- Il primo, chiamato **dizionario dei dati** (data dictionary), ci aiuta a produrre e successivamente leggere lo schema ER.
- Il secondo, indispensabile, è un elenco di **vincoli non esprimibili**. Siccome il potere espressivo dello schema ER non è sufficientemente elevato da poter esprimere tutto ciò che riguarda la realtà di interesse, determinati vincoli sono da definire a parte.

## Schema finale di esempio



## Dizionario dei dati

Elenco delle entità e delle associazioni presenti nello schema ER, con una breve descrizione, gli attributi e l'identificatore (entità) oppure le componenti (associazione).

## Entità

Entity	Description	Attributes	Identifier
Employee	Employee in a Dept.	Code, Name, Surname	Code
Project	Projects of a Dept.	Name, Budget	Name
Department	Structure of a Dept.	Name, Phone	Name, Office
Office	Office's location	City, Address	City

## Associazioni

Relationships	Description	Components	Attributes
Management	Management of a Dept.	Employee, Dept.	
Affiliation	Affiliation to a Dept.	Employee, Dept.	Date
Attendance	Attendance for a project	Employee, Project	
Composition	Composition of Departments	Dept., Office	

## Vincoli non esprimibili

Elenco dei vincoli non esprimibili tramite lo schema ER.

Integrity Constraints on Data
1. A department director must belong to that department
2. An employee must not have an income greater than the director of the department which he is affiliated
3. A department placed in Rome must be directed by an employee with at least 10 years of service
4. An employee that isn't affiliated to any department must not attend to any project

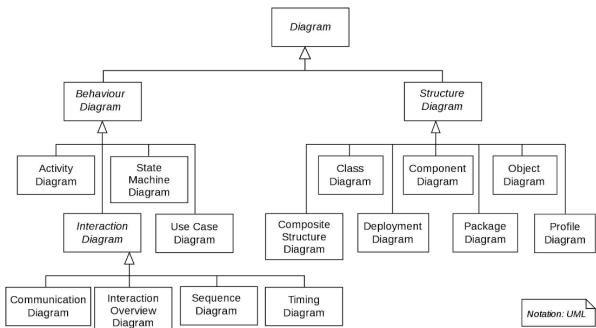
## UML

Un **linguaggio di modellazione** consente la specifica, la visualizzazione e la documentazione dello sviluppo di un sistema software.

I **modelli** sono descrizioni che gli utenti e gli sviluppatori possono usare per comunicare idee riguardo il software.

- UML 1.\* è un linguaggio di modellazione
- UML 2.\* è sempre un linguaggio di modellazione, ma è così dettagliato che può essere utilizzato anche come un linguaggio di programmazione

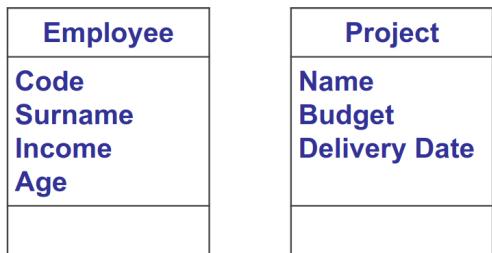
## Tipi di diagrammi in UML2



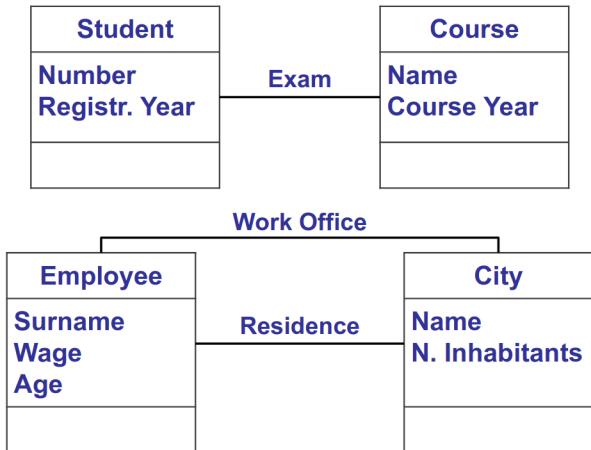
# Modellazione dei dati

UML2 può essere usato al posto di ER per la modellazione dei dati, tramite i **Class Diagram**. Cambia la rappresentazione grafica, ma l'approccio resta lo stesso. Vediamo come rappresentare i modelli concettuali con UML2.

## Classi

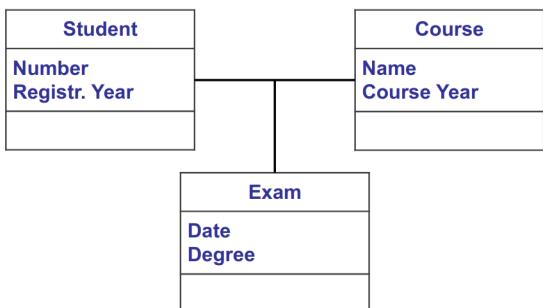


## Associazioni



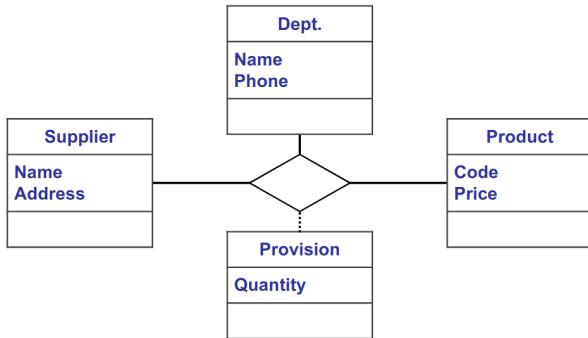
## Classi come associazioni

*Exam* è una classe-associazione utilizzata per rappresentare gli attributi *Degree* e *Date* dell'associazione tra *Student* e *Course*.



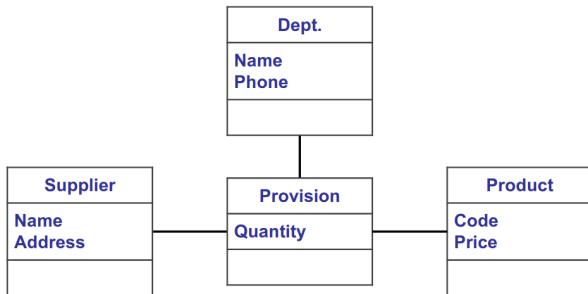
## Associazione ternaria

Un'associazione ternaria è rappresentata da un rombo con le classi coinvolte (*Supplier*, *Dept.* e *Product*), usando una classe-associazione (*Provision*) per assegnare attributi all'associazione tra queste classi.



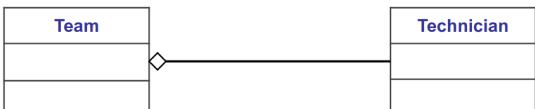
## Reificazione delle associazioni

Le associazioni N-arie non sono molto comuni: per questo motivo dovrebbero essere reificate in una singola classe (*Provision*) collegata alle classi originali tramite un'associazione binaria.

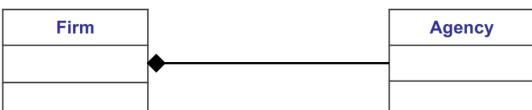


## Aggregazione e composizione

Un *Technician* appartiene ad un *Team*. Può essere rappresentato indipendentemente dal *Team* a cui appartiene → **aggregazione**.



Una *Agency* appartiene ad una *Firm*. Tale entità non può essere rappresentata indipendentemente dalla *Firm* → **composizione**.



## Cardinalità

Un *Order* ha tra 0 e 1 *Bill*. D'altra parte, un *Bill* ha solo 1 *Order*.



Nessuna molteplicità corrisponde a 1..1, quindi una *Person* può essere residente in solo una *City*.

D'altra parte, una *City* può avere da 0 a N *Person* residenti.



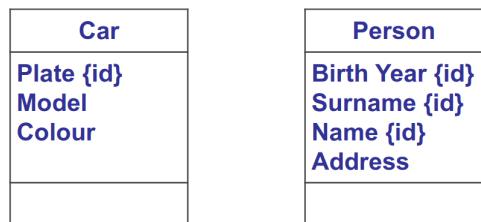
Un *Tourist* può prenotare da 1 a N *Hotel*. D'altra parte, un *Hotel* può essere prenotato da 0 a N *Tourist*.



## Identifieri

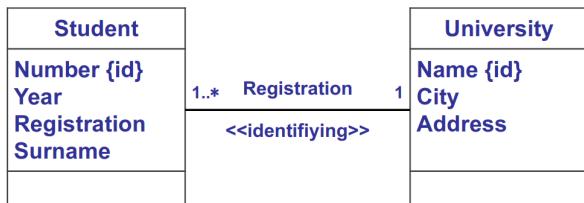
*Plate* è l'identificatore per la classe *Car*.

Gli attributi *Birth Year*, *Surname* e *Name* sono l'identificatore per la classe *Person*.



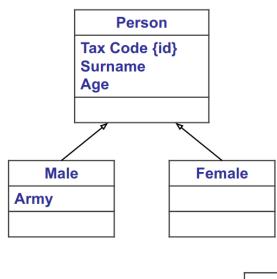
## Identifieri esterni

Lo stereotipo <> specifica che l'associazione tra *Student* e *University* insieme con il *Number id* identifica un determinato studente.

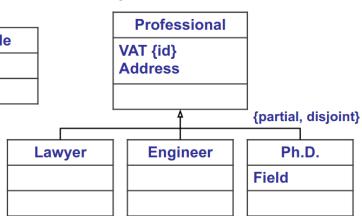


# Generalizzazione

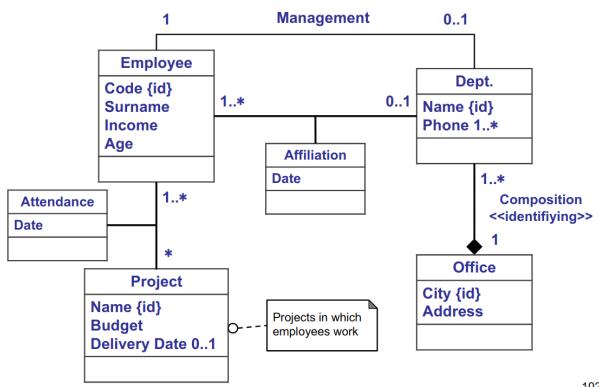
Disjoint and Total



Disjoint and Partial

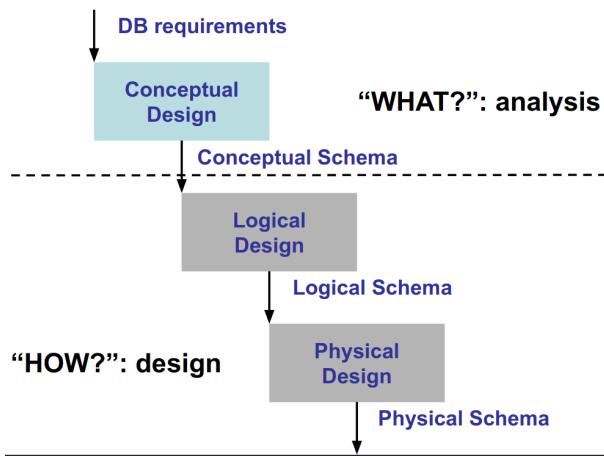


## Schema finale precedente in UML



# Progettazione concettuale

## Analisi dei requisiti



## Modellazione concettuale

Copre i seguenti compiti (interconnessi):

- Raccolta dei requisiti
- Analisi dei requisiti
- Costruzione di uno schema concettuale
- Costruzione di un glossario dei termini

## Possibili sorgenti

- Utenti e clienti:
  - interviste (colloqui)
  - documentazione specifica, ad-hoc
- Documentazione (già esistente):
  - regolamentazioni (leggi, regolamenti d'industria)
  - regolamenti interni, processi aziendali
  - soluzioni pre-esistenti (sistema informatico già presente)
- Questionari

# **Acquisizione dei requisiti**

L'acquisizione dei requisiti è un'attività difficoltosa e non standardizzata. Comincia con la raccolta dei primi requisiti. Tali requisiti spesso necessitano di essere rifiniti (tramite altre acquisizioni).

## **Acquisizione tramite interviste**

Differenti tipi di utenti forniscono informazioni diverse. Gli utenti che appartengono ai livelli più alti nell'organizzazione spesso hanno una visione più ampia ma meno dettagliata. Le interviste spesso portano ad ulteriori interviste per raffinare i requisiti.

## **Interagire con gli utenti**

Qualche suggerimento per interagire con le persone da intervistare:

- Accertarsi frequentemente della comprensione e consistenza
- Esempi di casi d'uso sono molto utili (casi generici e casi limite)
- Chiedere definizioni e classificazioni
- Domandare quali aspetti sono essenziali e quali di contorno

## **Documentazione descrittiva**

I requisiti raccolti vanno scritti seguendo alcune indicazioni:

- Scegliere il giusto livello di astrazione
- Mantenere la struttura delle frasi il più standard possibile
- Dividere le frasi più verbose
- Distinguere le frasi riguardanti i dati da quelle che riguardano le funzioni

## **Termini e concetti**

Una buona pratica è seguire alcune regole per le descrizioni:

- Costruire un glossario dei termini
- Unificare omonimi e sinonimi sotto un unico termine
- Chiarificare esplicitamente le relazioni tra i termini
- Ordinare le frasi per concetto

## Esempio: Bibliography Database

We want to automate the data for bibliographical references.

We want to handle the data of interest concerning the bibliographical references. For each reference we want an ID code formed by 7 characters, formed by the authors' initial letters, the year of publication, and a disambiguation character.

A reference could be either a monograph (we are interested on editor, date and place of editing) or journal papers (paper name, #volume, number, pages and year of publication); for both papers we are interested on the authors' names.

## Esempio: Training company

A "training company" requires a database providing some courses, for which we want to handle both lectures and teachers.

There are ~5000 students, which have an ID, their tax code (Tax), a surname, age, sex, place of birth, their employers' name, where they have worked previously and when, address and mobile phone, the attended courses (~200) and the final grade.

We want to store the workshops attended by the students and, for each day of the year, where and when the lectures are hold.

The courses have a code, a title and could have different editions, have a beginning and an end date and have a number of participants.

If a student is freelancer, we store the area of interest and, if they have it, an honorific.

Regarding the teachers (~300), we store their surname, their age, the place of birth, the name of the taught course, the set of the courses that they taught in the past and the set of the courses that they could teach in the future. We want to keep all their phone records. Teachers could be either house employees or independent contractors.

## Glossario

Term	Description	Synonym	Related To
<b>Participant</b>	<i>Who takes part to the courses</i>	Student	Course, Society
<b>Lecturer</b>	<i>The courses' lecturer. He could be a house employee</i>	Teacher	Course
<b>Course</b>	<i>Internal course. It could have several editions.</i>	Workshop	Lecturer
<b>Company</b>	<i>Current (or past) place of work</i>	Place	Participant

# Strutturare i requisiti in gruppi di frasi omogenei

## General sentences

A “training company” requires a database providing some courses, for which we want to handle both lectures and teachers data.

## Participants' sentences

There are ~5000 **participants**, which have an ID, their tax code (Tax), a surname, age, sex, place of birth, their employers' name, where they have worked previously and when (**starting and end date**), address and mobile phone, the attended courses (**current and past courses**) with their final grade.

## Specific participants' sentences

If a student is freelancer, we store the area of interest and, if they have it, an honorific.

If a student belongs to the same organization, we want to know their level within the management structure and their working position.

## Employer's statement

We store the participants' employers, both current and from the past. In particular we represent their name, address and phone number.

## Courses' statement

The courses (~200) have a code, a title and could have different editions, have a beginning and an end date. For each edition we keep the number of participants, the day of the week, the rooms and when the courses are held.

## Lecturers' statement

Regarding the **lecturers** (~300), we store their surname, their age, the place of birth, the name of the taught course, the set of the courses that they taught in the past and the set of the courses that they could teach in the future. We want to keep all their phone records. **Lecturers** could be either house employees or independent contractors.

# Schemi concettuali

Quale costruttore ER dovremmo usare per ogni specifico termine all'interno dei requisiti?

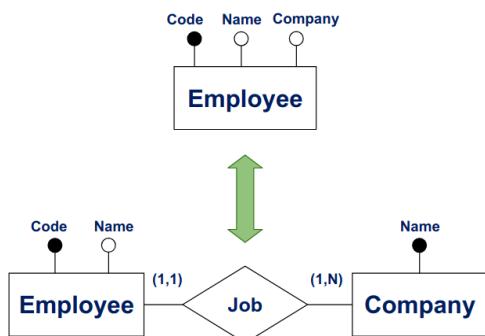
Riguardiamo le definizioni dei costruttori ER:

- **Entità** se il termine ha proprietà rilevanti e descrive oggetti che hanno senso anche da soli
- **Attributo** se è un semplice termine senza ulteriori specifiche
- **Associazione** quando un termine si lega ad altri termini
- **Generalizzazione** se un termine è un caso più generale di un altro termine

## Design pattern

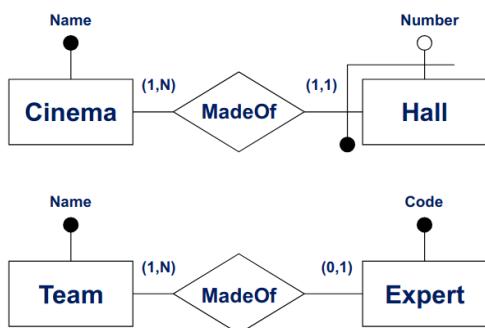
Buone pratiche comuni nella progettazione software per problemi ricorrenti. Gli ingegneri del software li usano giornalmente. Diamo un'occhiata ad alcuni design pattern ER.

### Reificazione dell'attributo in un'entità



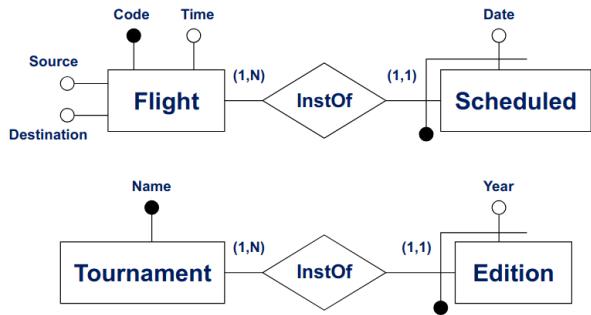
*Company* è solitamente un concetto diverso da *Employee* (es. quando le compagnie sono coinvolte in altre associazioni).

### Parte di



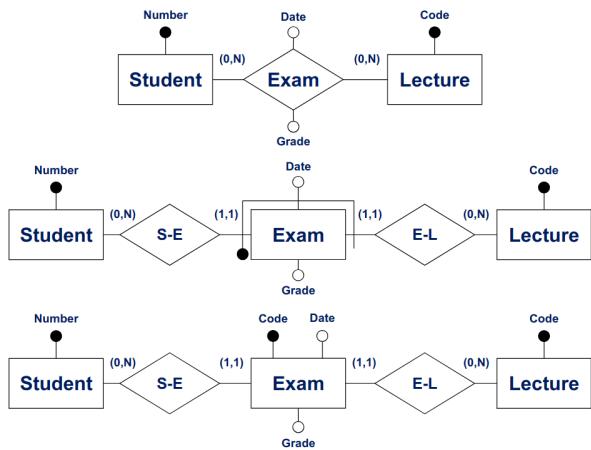
Le associazioni (1,N) potrebbero rappresentare relazioni di "parte di". Potrebbero rappresentare **composizioni** (un *Cinema* è composto da molte *Hall*, ogni *Hall* appartiene ad un *Cinema*) oppure **aggregazioni** (un *Team* è formato da molti *Expert*).

# Istanza di

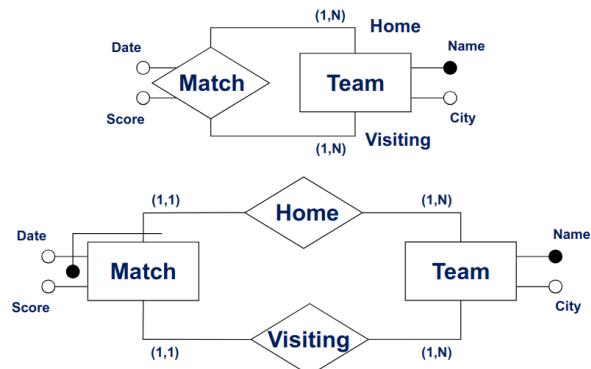


A volte sono necessarie due entità distinte, una definisce una rappresentazione astratta e l'altra memorizza le informazioni richieste dai nostri requisiti.

## Reificazione di associazioni binarie

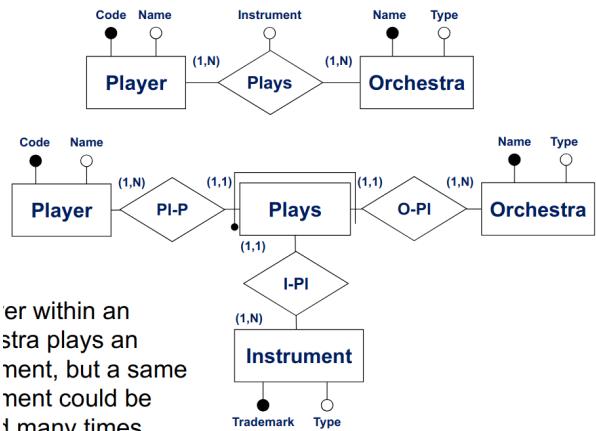


## Reificazione di associazioni ricorsive



Un *Match* può essere espresso come un'associazione binaria tra due *Team*. Dato che qualunque *Team* può giocare più volte un *Match*, potremmo reificare *Match* come nel secondo schema ER (quindi fornendo la rappresentazione più vicina).

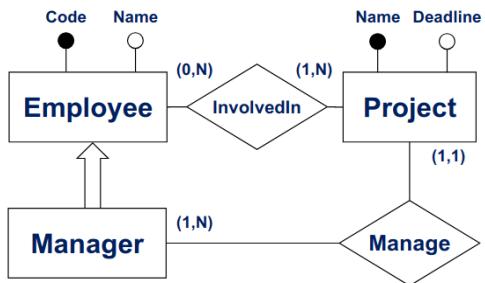
# Reificazione di attributi in associazioni



er within an  
stra plays an  
ment, but a same  
ment could be  
ì many times

Un *Player* all'interno di un'*Orchestra* suona uno strumento, ma lo stesso strumento potrebbe essere suonato più volte.

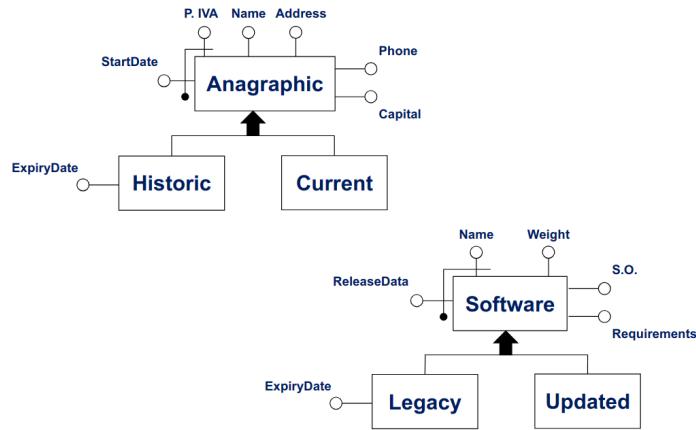
## Casi specifici



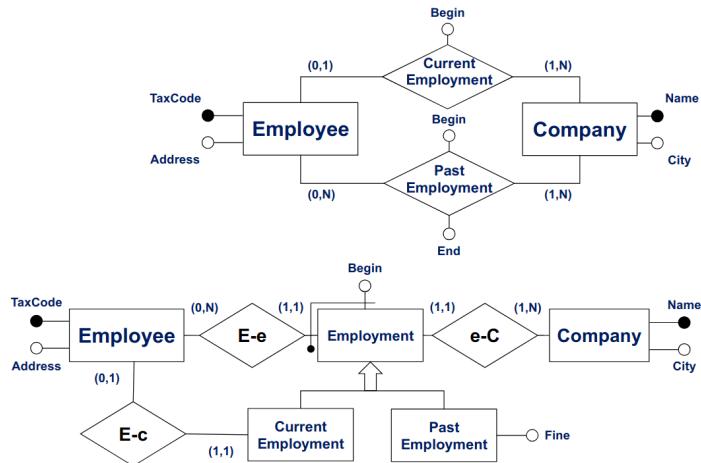
Le generalizzazioni sono usate per definire un caso specifico (*Manager*) di una determinata entità (*Employee*). In questo caso non tutti gli impiegati gestiscono un progetto.

# Storicizzazione di un concetto

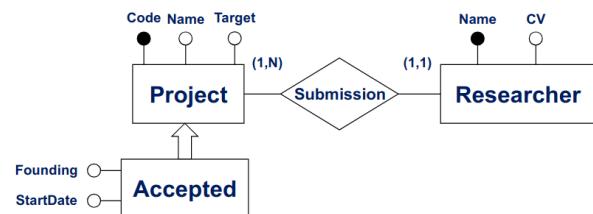
Mantenere determinati dati di rilevanza.



Un ulteriore esempio:

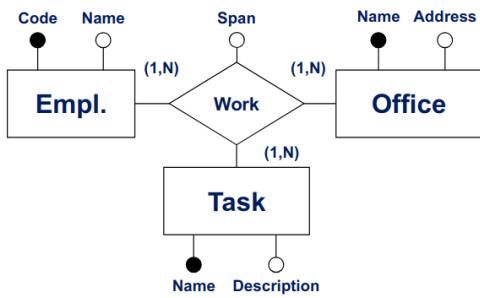


# Estendere un concetto



Le generalizzazioni possono essere usate per estendere le implementazioni attuali quando il *Project* è già in funzione. In questo caso il *Project Accepted* richiede più informazioni degli altri *Project* (sia *idle* che *rejected*).

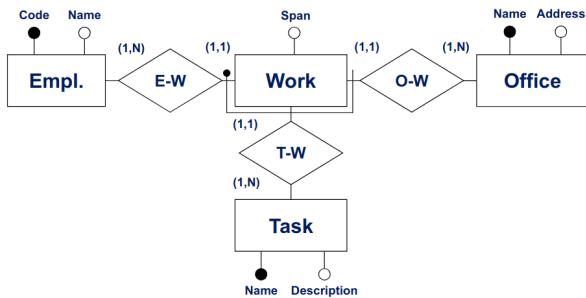
# Reificazione di associazioni ternarie



Gli *Employee* possono lavorare su numerosi *Task* in differenti *Office*.

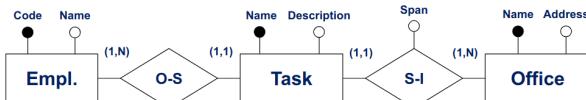
Gli *Office* possono ospitare differenti *Employee* che lavorano a numerosi *Task*.

I *Task* possono essere effettuati da differenti *Employee* in differenti *Office*.



Ogni *Work* è definito da un *Employee* (E-W) che lavora in un certo *Office* (O-W) per uno specifico *Task* (T-W).

Se un *Task* potesse essere effettuato da un *Employee* solo in un *Office*, la reificazione potrebbe essere semplificata nel seguente schema ER:



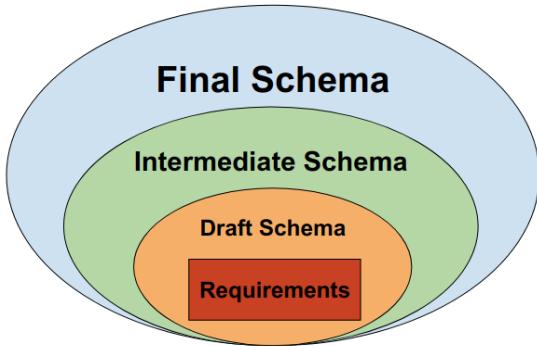
# Strategie

Come possiamo mappare i nostri requisiti in istanze di schemi ER? Ci sono diverse strategie possibili:

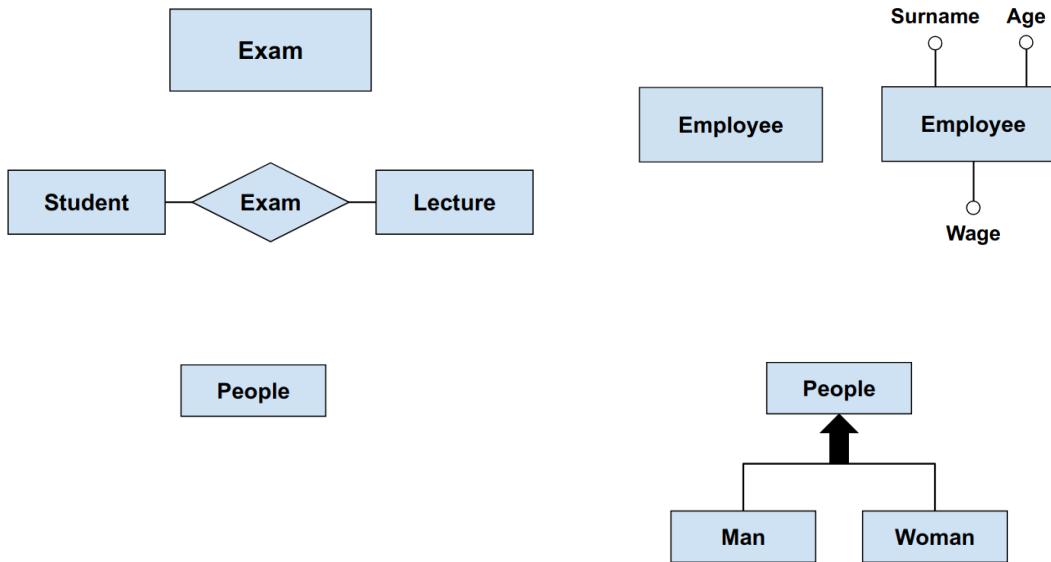
- **Top-Down**
- **Bottom-Up**
- **Inside-Out**

## Top-Down

L'idea alla base della strategia **Top-Down** è partire dai requisiti per produrre una prima bozza di schema ER. Questo schema verrà poi esteso arrivando ad uno schema intermedio. Infine si completa l'opera generando lo schema finale che coprirà tutti e soli i requisiti del sistema.

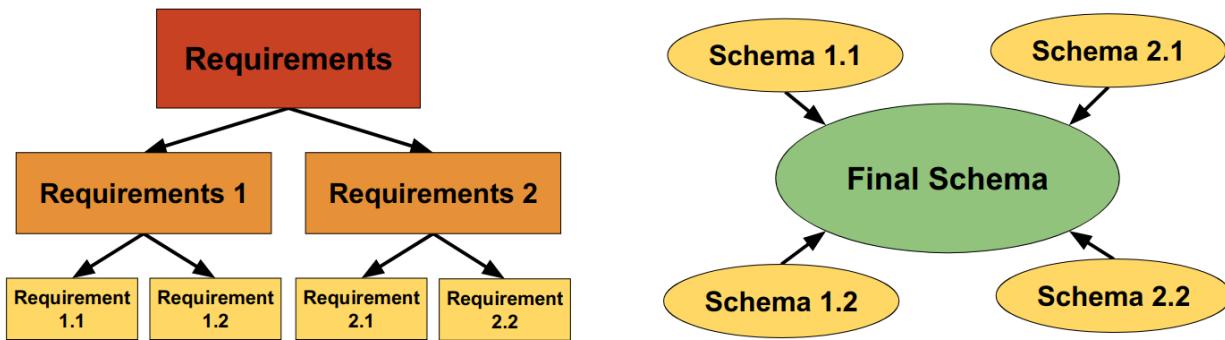


## Rifiniture



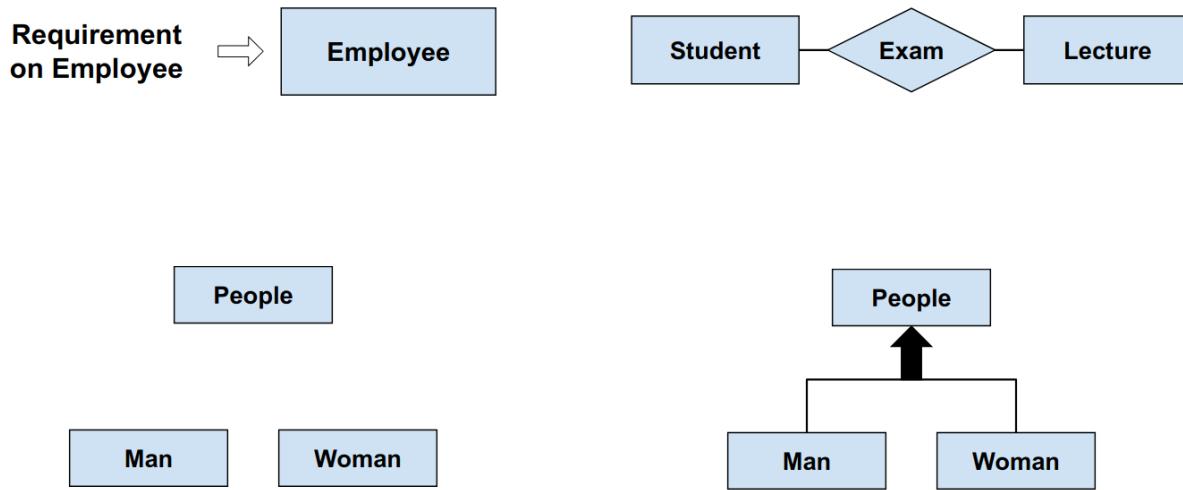
# Bottom-Up

Nella strategia **Bottom-Up** si costruisce gradualmente lo schema cominciando dalle unità base. Si parte dai requisiti e si cerca di capire quali sono le entità di base, andandole poi a raffinare man mano che si analizzano le informazioni. Si producono così pezzi di schema che coprono parte dei requisiti, che successivamente verranno integrati per ottenere lo schema finale.



## Rifiniture

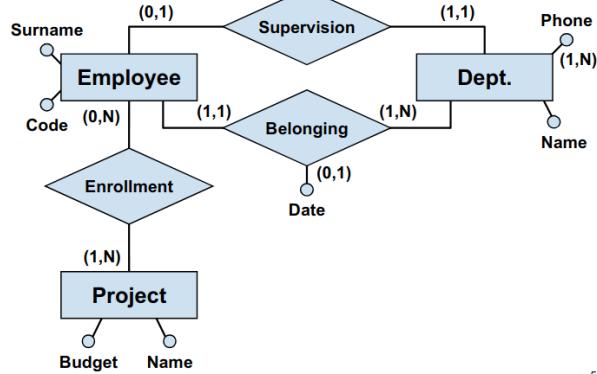
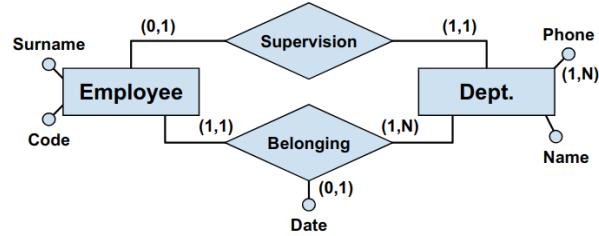
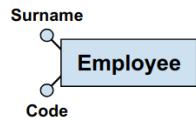
Si individua un'entità e piano piano si aggiungono gli attributi. Oppure, si individua un'entità e successivamente si raffina scoprendo che è una generalizzazione di altre entità.



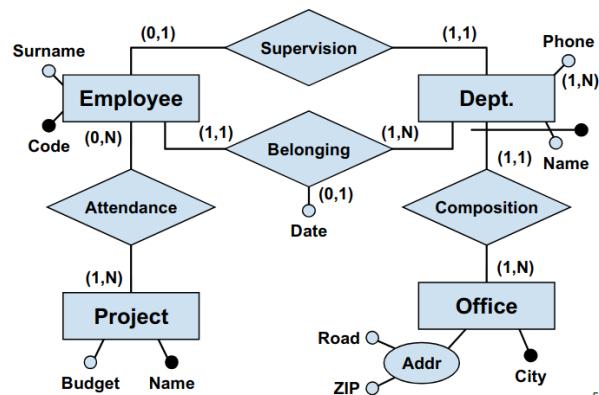
# Inside-Out

Con la strategia **Inside-Out** si parte da un punto e si costruisce lo schema gradualmente leggendo i requisiti, senza partire dalle entità base o finali. Per certi versi è il modo più semplice quando si inizia.

# Evoluzione



54



55

56

57

## Regole empiriche

Utilizzare sempre uno stile **misto**:

1. Per prima cosa si crea uno schema bozza usando le entità più rilevanti
2. Successivamente si decompone lo schema
3. Infine si rifiniscono le entità tramite Top-Down, si integrano tramite Bottom-Up e si espandono tramite Inside-Out

## Abbozzare lo schema ER

Cominciare dalle entità più rilevanti, che solitamente sono le più citate oppure sono esplicitamente dichiarate tali, e produrre un primo schema ER base.

# Best practices

- **Analisi dei requisiti**
  - risolvere le ambiguità
  - creare un glossario
  - raccogliere i requisiti in frasi simili
- **Caso base**
  - definire un primo schema abbozzato con i concetti più rilevanti
- **Caso iterativo** (continuare finché non soddisfa)
  - rifinire i concetti base usando i requisiti
  - aggiungere concetti per descrivere i requisiti non rappresentati
- **Controllo qualità** (ripeterlo durante lo sviluppo)
  - controllare la qualità dello schema e modificare di conseguenza
  - alcune misure: correttezza, completezza, chiarezza, minimalismo

## Esempio di flusso 1

1. Analisi dei requisiti
2. Caso base
3. Decomposizione
  - i. Decomporre i requisiti complessi basandosi sullo schema abbozzato
4. Caso iterativo per ogni sotto-schema
5. Integrazione
  - i. Integrare i numerosi sotto-schemi in uno schema totale, usando lo schema abbozzato come riferimento
6. Controllo qualità

## Esempio di flusso 2

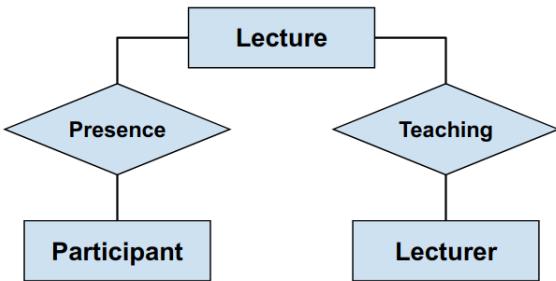
1. Analisi dei requisiti
2. Decomposizione
  - i. Identificare le aree di interesse e partizionare i requisiti
3. Per ogni area:
  - i. Caso base
  - ii. Caso iterativo
4. Integrazione
5. Controllo qualità

# Esempio finale comprensivo: *The Training Company*

## General statement

A “training company” requires a database providing some courses, for which we want to handle both lectures and teachers.

## Sketched schema



## Refining: Participants

There are ~5000 participants, which have an ID, their tax code (Tax), a surname, age, sex, place of birth, their employers' name, where they have worked previously and when (starting and end date), address and mobile phone, the attended courses (current and past courses) with their final grade.

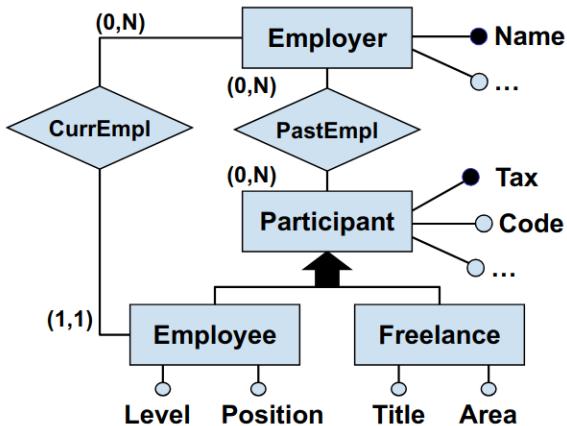
If a student is freelancer, we store the area of interest and, if they have it, an honorific.

If a student belongs to the same organization, we want to know their level within the management structure and their working position.

## Refining: Employer

We store the participants' employers, both current and from the past. In particular we represent their name, address and phone number.

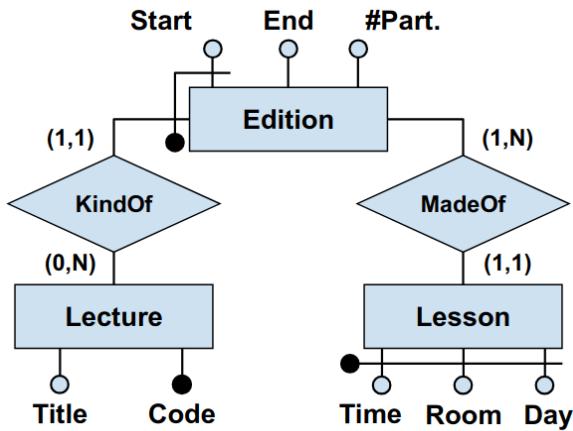
## Partial ER Schema (1)



## Refining: Course

The courses (~200) have a code, a title and could have different editions, have a beginning and an end date. For each edition we keep the number of participants, the day of the week, the rooms and when the courses are held.

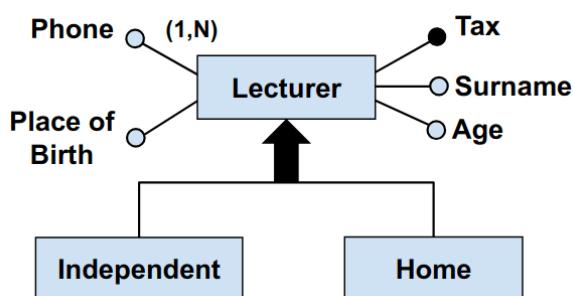
## Partial ER Schema (2)



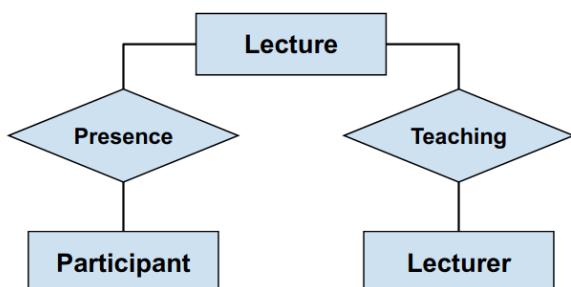
## Refining: Lecturers

Regarding the lecturers (~300), we store their surname, their age, the place of birth, the name of the taught course, the set of the courses that they taught in the past and the set of the courses that they could teach in the future. We want to keep all their phone records. Lecturers could be either house employees or independent contractors.

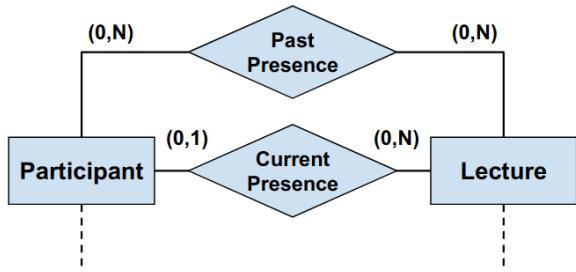
## Partial ER Schema (3)



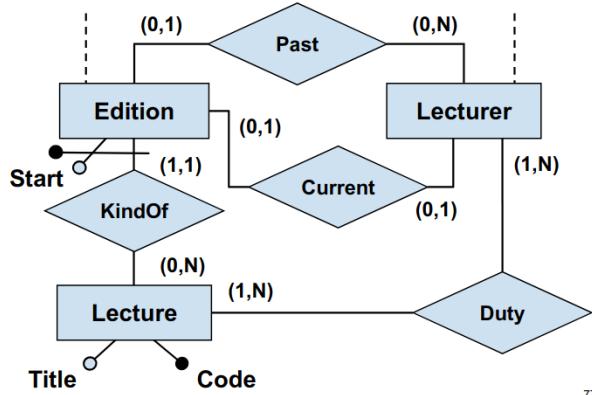
## Sketched Schema: Integration



## Intermediate Schema (1)

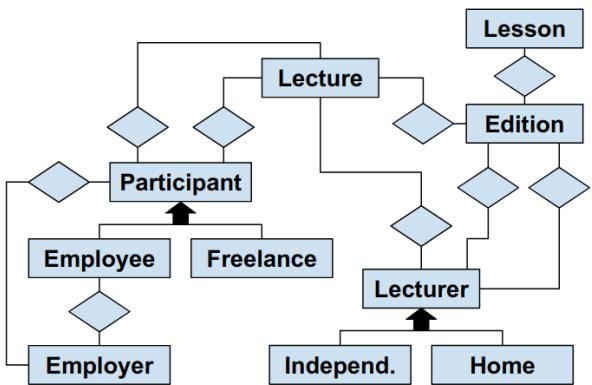


## Intermediate Schema (2)



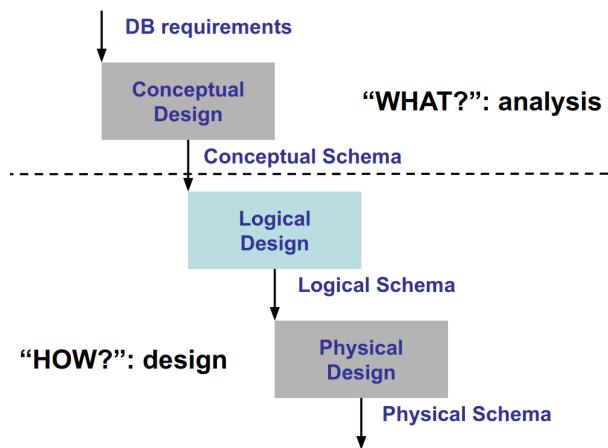
77

## Final Schema: Just Entities and Relations



# Progettazione logica

## Introduzione



L'obiettivo della progettazione logica è "tradurre" lo schema concettuale in uno schema logico rappresentante gli stessi dati, in modo sia corretto che efficiente. Qui è la prima volta che viene affrontato il concetto di efficienza.

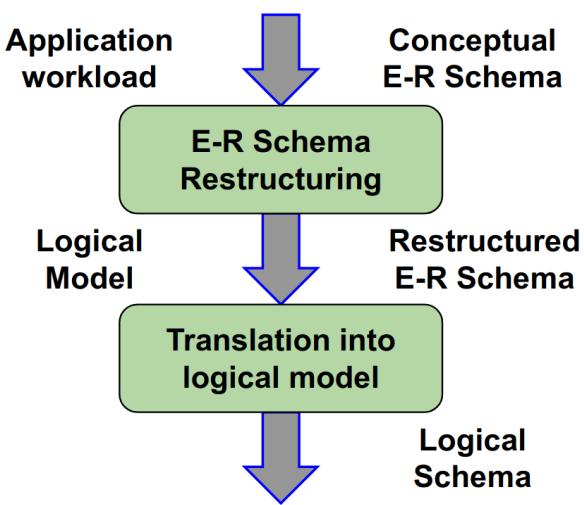
In input viene preso uno schema concettuale, un modello logico ed alcune informazioni sul carico di lavoro. In output viene restituito uno **schema logico** (che sia relazionale, OO, a grafo...) e la documentazione associata ad esso.

## Sotto-fasi della trasformazione

Quella che stiamo cercando di fare non è una semplice traduzione, perché alcuni concetti del modello logico non sono direttamente rappresentabili nel modello concettuale. In questa fase dobbiamo inoltre considerare la performance (efficienza).

La progettazione logica è composta da due fasi:

- **Ristrutturazione dello schema ER**, il quale arriva in input insieme al carico di lavoro applicativo che dovrà supportare il db. Questa fase produce lo schema ER ristrutturato.
- **Traduzione in modello logico**, che con lo schema ristrutturato ed il modello logico di riferimento produce in output lo schema logico con la documentazione.



## Ristrutturazione dello schema ER

Ristrutturare lo schema consente di semplificare la traduzione ed ottimizzare le prestazioni.

Nota che uno schema ER ristrutturato non è più uno "schema concettuale" nel senso proprio del termine.

## Prestazioni

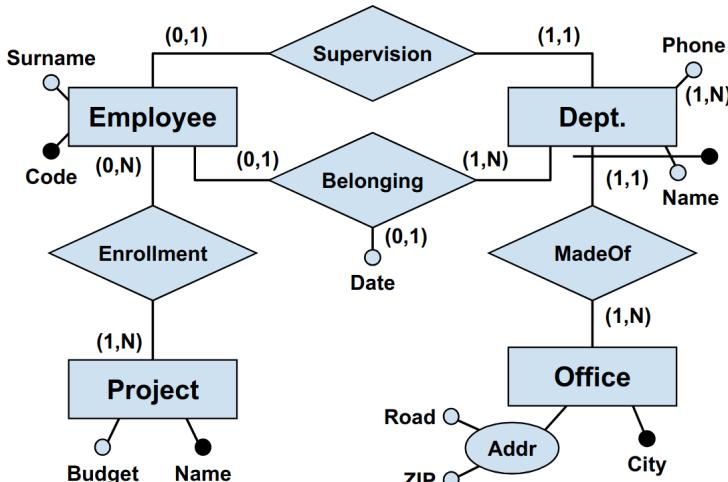
Per ottimizzare i risultati bisogna analizzare le prestazioni. Quando si parla di **prestazioni** si intendono genericamente **indicatori di performance**, in quanto i valori assoluti dipendono dal sistema operativo, dall'hardware...

## Indicatori

Consideriamo alcuni **indicatori di performance**, parametri da cui dipendono le prestazioni reali.

- **Spazio**: quante istanze ci aspettiamo di dover memorizzare.
- **Tempo**: numero di istanze (entità ed associazioni) da visitare per eseguire una certa operazione.

## Schema ER in input e tabella delle dimensioni



Name	Type	Size
Office	E	10
Dept.	E	80
Employee	E	2'000
Project	E	500
MadeOf	R	80
Belonging	R	1'900
Supervision	R	80
Enrollment	R	6'000

## Valutazione dei costi

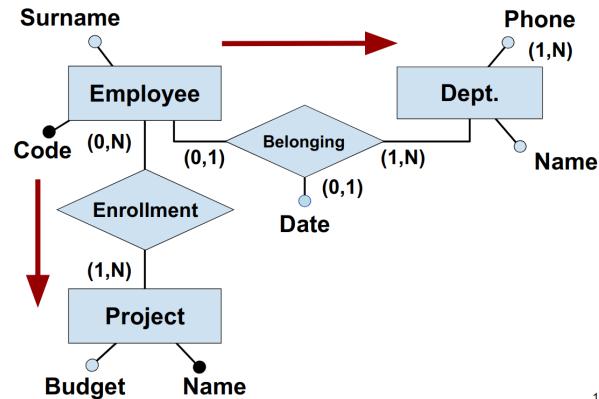
Bisogna conoscere la cardinalità iniziale ed il tasso di crescita previsto nel corso del tempo. Se durante la progettazione il database viene ottimizzato su un certo parametro, e poi questo varia sensibilmente, le prestazioni potrebbero deteriorarsi.

Utilizzando insieme le informazioni di cardinalità (nello schema) e tabella delle dimensioni si possono ricavare i volumi delle associazioni.

Ci interesserà andare a determinare il costo delle operazioni. Per fare ciò è necessario costruire la **tabella degli accessi**: a partire dallo schema, serve capire come verrà "navigato" per effettuare le operazioni.

## Esempio di valutazione

Ritornare tutti i dati di un determinato *Employee*, i dati del *Dept.* a cui appartiene e i dati dei *Project* a cui lavora.



Name	Type	Accesses Number	Accesses Type	Accesses Order
Dept.	E	1	R	3
Employee	E	1	R	1
Project	E	3	R	5
Belonging	R	1	R	2
Enrollment	R	3	R	4

La tabella degli accessi andrebbe costruita per tutte le operazioni eseguite sul db. Di solito si individuano le 10 operazioni più frequenti, e si costruisce la tabella degli accessi soltanto per quelle.

# Attività di ristrutturazione

## Analisi delle ridondanze

Una **ridondanza** in uno schema ER è un'informazione rilevante che può essere derivata da altre informazioni.

In questa fase dobbiamo decidere se tenere, rimuovere o creare ridondanze.

### PRO:

- Semplificano le query

### CONTRO:

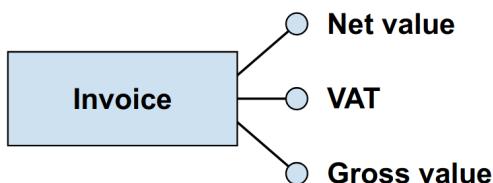
- Gli aggiornamenti richiedono più tempo
- La dimensione in memoria aumenta

## Tipi di ridondanze

- Attributi derivabili:
  - da altri attributi all'interno della stessa entità (o associazione)
  - da attributi di altre entità (o associazioni)
- Associazioni derivabili dalla composizione di associazioni differenti, ovvero i cicli di associazioni

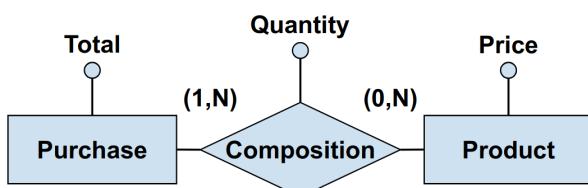
### Attributi derivabili all'interno dell'entità

Uno dei tre attributi è derivabile dagli altri due.



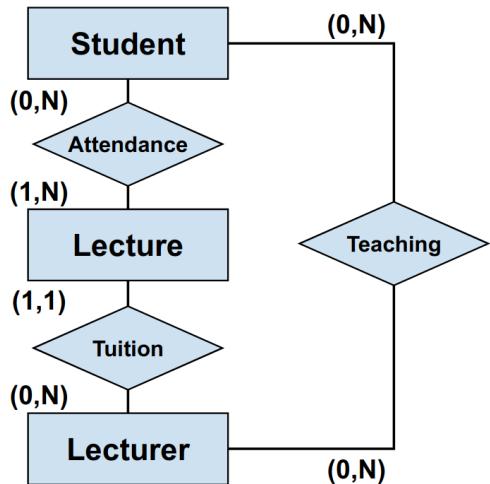
### Attributi derivabili da altre entità

Anche qui uno dei tre attributi è derivabile, in genere il totale.



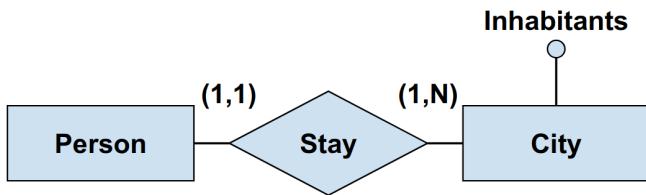
## Ridondanza da un ciclo

L'associazione *Teaching* non è necessaria. Tutte le volte che abbiamo un ciclo c'è spazio per la rimozione di un'associazione.



## Esempio di analisi delle ridondanze

Un attributo derivabile dal conteggio delle occorrenze.



Name	Type	Size
City	E	200
Person	E	1'000'000
Stay	R	1'000'000

**Operazione #1:** salvare una nuova persona con la sua città di residenza (500 volte al giorno).

**Operazione #2:** stampare i dati di tutte le città, incluso il numero di abitanti (2 volte al giorno).

## Dettagli delle operazioni

Con ridondanza:

Operation #1			
Name	Type	Accesses Number	Accesses Type
Person	E	1	W
Stay	R	1	W
City	E	1	R
City	E	1	W

Operation #2			
Name	Type	Accesses Number	Accesses Type
City	E	1	R

Senza ridondanza:

Operation #1			
Name	Type	Accesses Number	Accesses Type
Person	E	1	W
Stay	R	1	W

Operation #2			
Name	Type	Accesses Number	Accesses Type
City	E	1	R
Stay	R	5'000	R

## Costi

Contiamo gli accessi in scrittura come doppi: l'idea è che è un'operazione più brigosa.

*Con ridondanza:*

Operazione #1: 1500 scritture + 500 letture al giorno

Operazione #2: ignorabile (2 operazioni)

Un totale di 3500 operazioni al giorno.

*Senza ridondanza:*

Operazione #1: 1000 scritture al giorno

Operazione #2: 10000 letture al giorno

Un totale di 12000 operazioni al giorno.

Conclusione: in questo caso specifico conviene tenere la ridondanza.

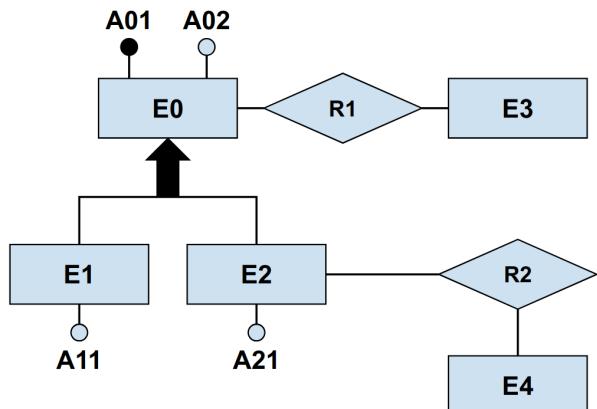
## Eliminazione delle generalizzazioni

Il modello dati relazionale non supporta direttamente le generalizzazioni, quindi non possono essere immediatamente rappresentate, mentre le entità e le associazioni sono direttamente rappresentabili. Rimuoviamo perciò le gerarchie sostituendole con entità e associazioni.

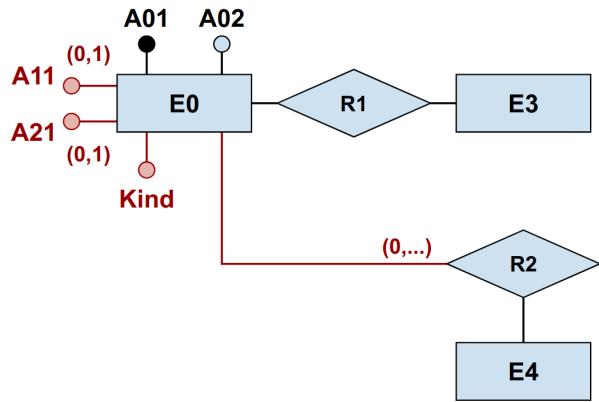
Tre possibili soluzioni:

- 1) **Parent embedding:** inserire le entità figlie della generalizzazione nel padre.
- 2) **Children embedding:** inserire l'entità padre della generalizzazione nel figlio.
- 3) **Using relationship:** sostituire la generalizzazione con un'associazione.

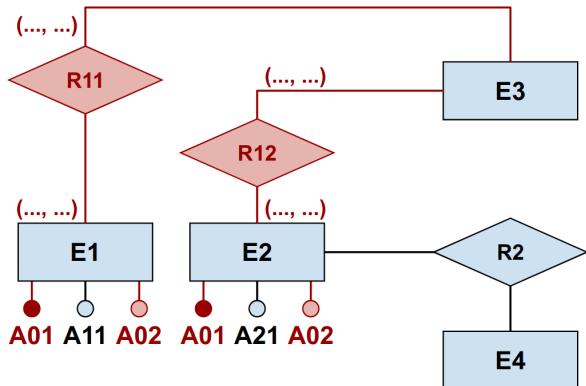
## Esempio di eliminazione delle gerarchie



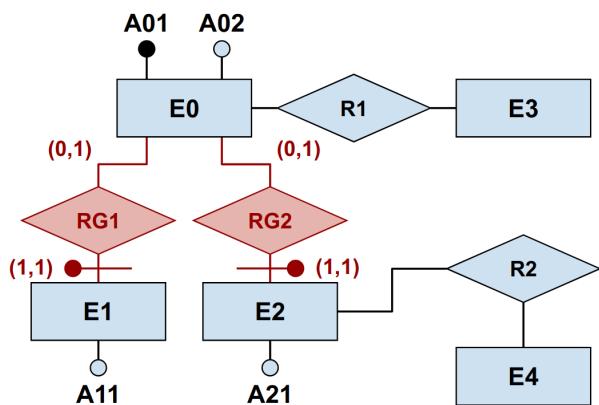
1) Inserire le entità figlie della generalizzazione nel padre.



2) Inserire l'entità padre della generalizzazione nel figlio.



3) Sostituire la generalizzazione con un'associazione.



## Commenti

Possiamo decidere fra queste tre alternative in base alle dimensioni e alla tabella degli accessi (considerare solo il numero di accessi non è abbastanza).

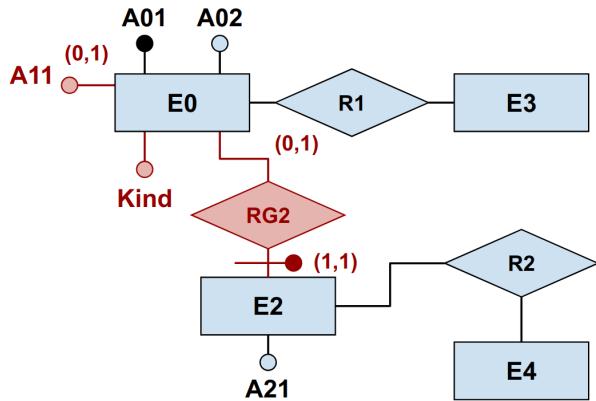
Le tre soluzioni dovrebbero essere usate quando...

- 1) **Parent embedding** quando l'accesso alle entità figlie e padre avviene allo stesso momento.
- 2) **Children embedding** quando le entità figlie sono accedute indipendentemente l'una dall'altra.
- 3) **Using relationship** quando le entità figlie sono accedute indipendentemente dall'entità padre.

## Soluzione ibrida

È possibile applicare anche delle soluzioni ibride, specialmente quando si presentano gerarchie su più livelli.

*Soluzione ibrida (1) + (3).*



## Raggruppamento/partizionamento di entità e associazioni

La ristrutturazione può fornire operazioni più efficienti tramite una semplice riduzione del numero di accessi:

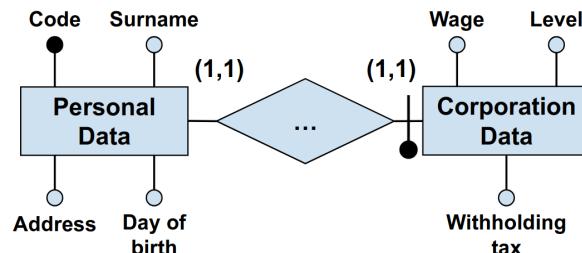
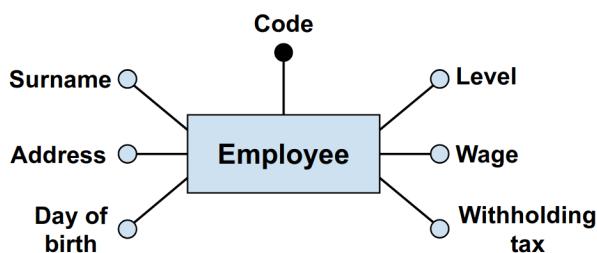
- Attributi acceduti separatamente possono essere divisi.
- Attributi acceduti allo stesso momento possono essere raggruppati, anche nel caso appartengano a differenti entità o associazioni.

## Casi principali

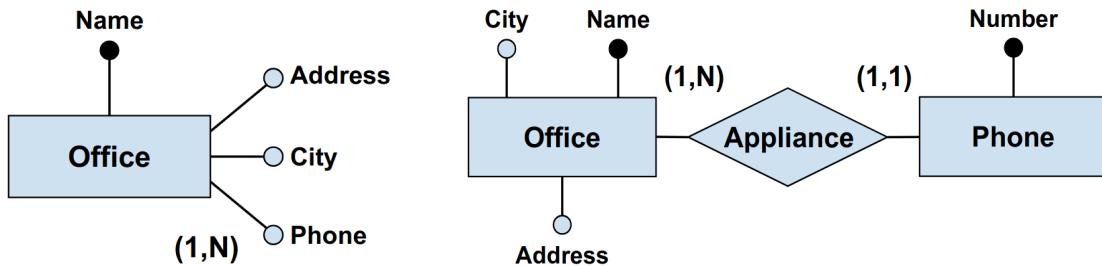
- 1) Partizionamento verticale di un'entità
- 2) Ristrutturazione di attributi multi-valore
- 3) Raggruppamento di entità o associazioni
- 4) Partizionamento orizzontale di un'associazione

### Esempi

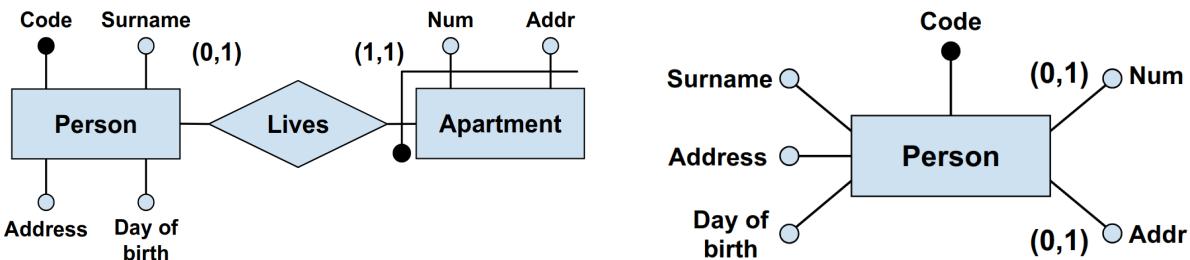
#### 1) Partizionamento verticale di un'entità



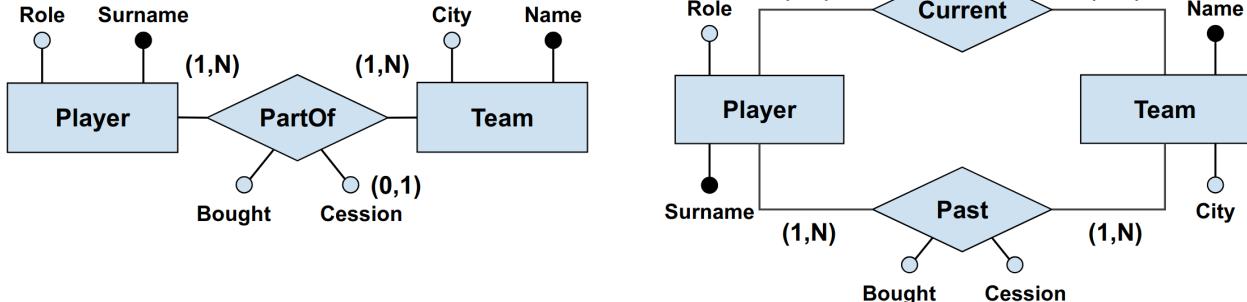
## 2) Ristrutturazione di attributi multi-valore



## 3) Raggruppamento di entità o associazioni



## 4) Partizionamento orizzontale di un'associazione



## Identificare le chiavi primarie

Un'operazione fondamentale per la traduzione in un modello relazionale.

Tra i criteri per la scelta della chiave primaria troviamo che l'informazione deve essere obbligatoria, semplice ed usata nelle operazioni più frequenti o rilevanti.

## Nuovi attributi

Cosa succede se nessuna delle condizioni sopracitate è soddisfatta?

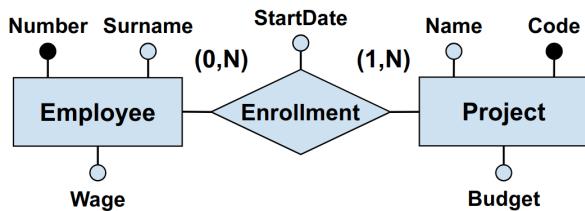
Bisogna introdurre dei nuovi attributi usando codici generati appositamente.

# Traduzione in modello relazionale

Due regole semplici ma efficaci:

- Le **entità** diventano delle tabelle, il cui schema corrisponde agli attributi dell'entità.
- Le **associazioni** diventano anch'esse delle tabelle, il cui schema corrisponde agli attributi dell'associazione uniti alle chiavi esterne delle entità coinvolte.

## Associazioni molti a molti



EMPLOYEE(Number, Surname, Wage)

PROJECT(Code, Name, Budget)

ENROLLMENT(Number, Code, StartDate)

Vincoli di integrità referenziale tra:

- *Number* in ENROLLMENT e la chiave di EMPLOYEE.
- *Code* in ENROLLMENT e la chiave di PROJECT.

Per le chiavi esterne (Foreign Key) usare nomi espressivi.

EMPLOYEE(Number, Surname, Wage)

PROJECT(Code, Name, Budget)

ENROLLMENT(Number, Code, StartDate)

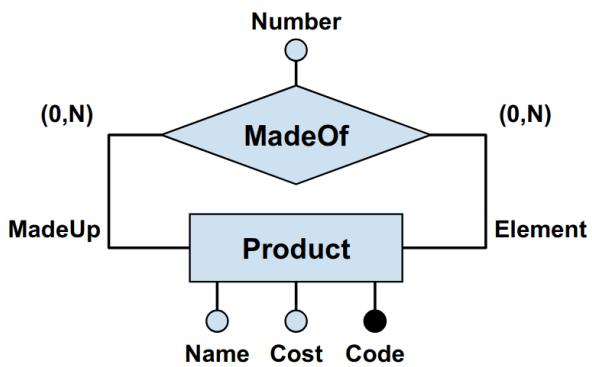


ENROLLMENT(Employee, Project, StartDate)

Tale traduzione non tiene conto della cardinalità minima delle associazioni molti a molti.

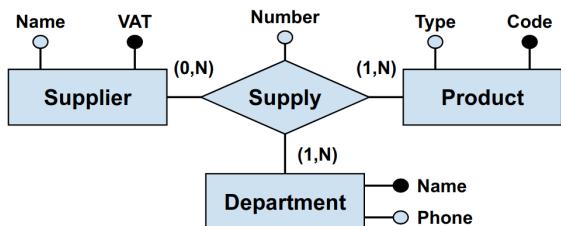
Sarebbe così anche se usassimo vincoli CHECK molto rari e complessi.

## Associazioni ricorsive



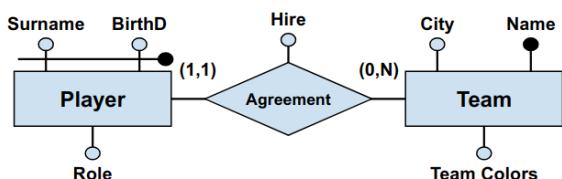
`PRODUCT(Code, Name, Cost)`  
`MADEOF(MadeUp, Element, Number)`

## Associazioni N-arie



`SUPPLIER(VAT, Name)`  
`PRODUCT(Code, Type)`  
`DEPARTMENT(Name, Phone)`  
`SUPPLY(Supplier, Product, Department, Number)`

## Associazioni uno a molti



`PLAYER(Surname, BirthD, Role)`  
`AGREEMENT(SurnameP, BirthDP, Team, Hire)`  
`TEAM(Name, City, TeamColors)`

È corretto così? C'è una soluzione meno ridondante:

`PLAYER(Surname, BirthD, Role)`  
`AGREEMENT(SurnameP, BirthDP, Team, Hire)`  
`TEAM(Name, City, TeamColors)`



`PLAYER(Surname, BirthD, Team, Role, Hire)`  
`TEAM(Name, City, TeamColors)`

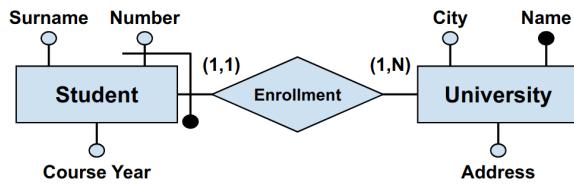
Vincolo di integrità referenziale tra *Team* in PLAYER e la chiave di TEAM.

Se la cardinalità minima dell'associazione è 0, *Team* in PLAYER deve ammettere anche valori NULL.

Tale traduzione potrebbe rappresentare il caso in cui la cardinalità minima è 0 e la massima è 1:

- 0: valori NULL ammessi
- 1: valori NULL non ammessi

## Entità con un identificatore esterno

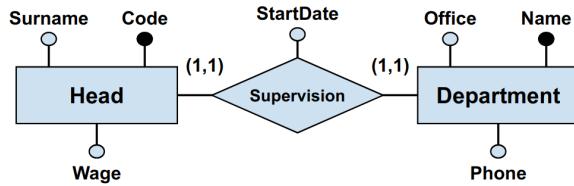


STUDENT(Number, University, Surname, CourseYear)

UNIVERSITY(Name, City, Addr)

**Vincolo:** ogni studente è iscritto ad una sola università.

## Associazioni uno a uno



Abbiamo opzioni differenti:

- Merge di un'entità con l'associazione (una o l'altra)
- Merge tutto insieme (fattibile?)

HEAD(Code, Surname, Wage, Department, StartDate)

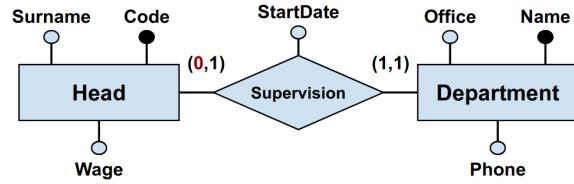
DEPARTMENT(Name, Office, Phone)

**OR**

HEAD(Code, Surname, Wage)

DEPARTMENT(Name, Office, Phone, Head, StartDate)

## Caso speciale

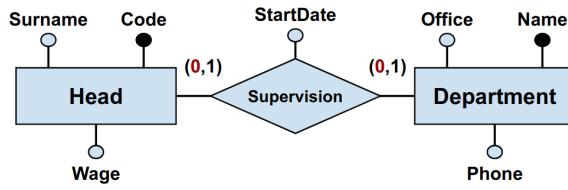


HEAD(Code, Surname, Wage)

DEPARTMENT(Name, Office, Phone, Head, StartDate)

Un vincolo di integrità referenziale, NULL non ammesso.

## Altro caso speciale



HEAD(Code, Surname, Wage)

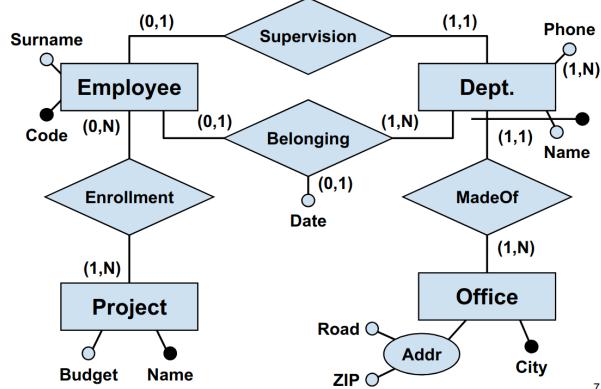
DEPARTMENT(Name, Office, Phone)

SUPERVISION(Head, Department, StartDate)

Due vincoli di integrità referenziale, NULL non ammesso.

## Schema finale

### Concettuale



### Logico

EMPLOYEE(Code, Surname, Dept, Office, Date\*)

DEPT(Name, City, Phone, Head)

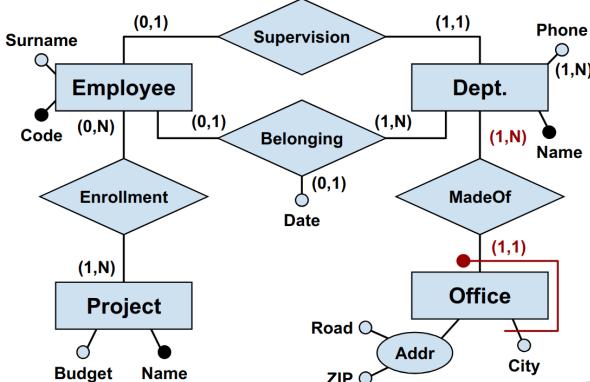
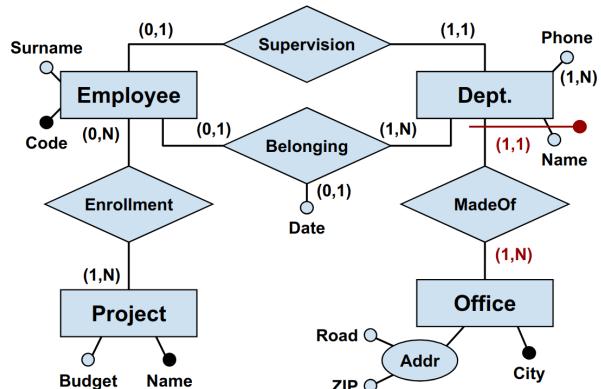
OFFICE(City, Road, Zip)

PROJECT(Name, Budget)

ENROLLMENT(Employee, Project)

## Attenzione

Differenze apparentemente piccole nella cardinalità e nella scelta degli identificatori potrebbero portare a schemi molto differenti.



**EMPLOYEE**(Code, Surname, Dept, Office, Date\*)

**DEPT**(Name, Phone, Head)

**OFFICE**(City, Dept, Road, Zip)

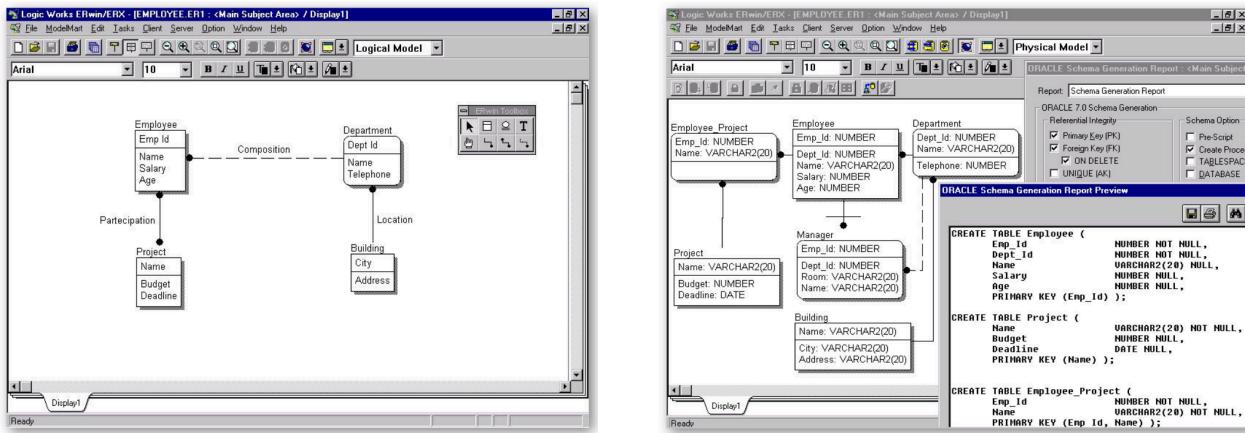
**PROJECT**(Name, Budget)

**ENROLLMENT**(Employee, Project)

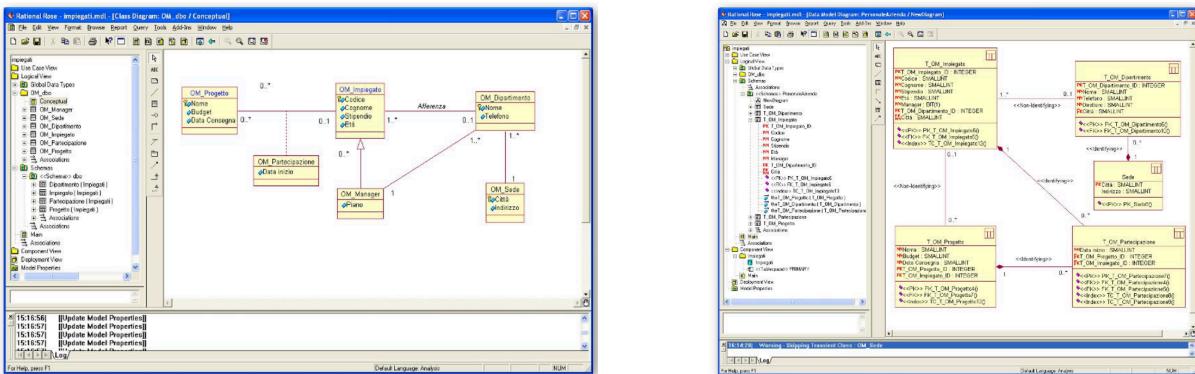
## Strumenti

Esistono alcuni **Computer-Aided Software Engineering (CASE)**, software che forniscono supporto attraverso tutte le fasi della progettazione di database.

### ERwin / ERX



### IBM Rational Rose



# Normalizzazione

## Introduzione

Siamo ora interessati a capire se lo schema prodotto è di "buona qualità".

## Forme normali

La **forma normale** è una proprietà dei database relazionali, che ne garantisce la qualità, ovvero l'assenza di determinati difetti.

Quando una relazione non è in forma normale presenta delle ridondanze e può avere comportamenti indesiderati durante le operazioni di aggiornamento.

Le forme normali sono solitamente definite sul modello relazionale, ma hanno significato anche in altri contesti, ad esempio nel modello ER.

## Normalizzazione

Questa attività permette la trasformazione di uno schema non normalizzato in uno schema che soddisfa una certa forma normale.

La normalizzazione deve essere usata come **tecnica di verifica** per controllare i risultati della progettazione del database, non è una metodologia per progettare il db.

## Una relazione con anomalie

Con anomalia solitamente si intende una ridondanza.

Employee	Wage	Project	Budget	Role
Jones	20	Mars	2	Technician
Smith	35	Jupiter	15	Designer
Smith	35	Venus	15	Designer
Williams	55	Venus	15	Chief
Williams	55	Jupiter	15	Consultant
Williams	55	Mars	2	Consultant
Brown	48	Mars	2	Chief
Brown	48	Venus	15	Designer
White	48	Venus	15	Designer
White	48	Jupiter	15	Director

## Anomalie presenti

- **Ridondanza:** lo stipendio di ciascun impiegato è ripetuto più volte nella relazione.
- **Anomalia di aggiornamento:** quando lo stipendio di un impiegato cambia, dobbiamo modificarne tutte le occorrenze.
- **Anomalia di eliminazione:** quando un impiegato finisce di lavorare a tutti i suoi progetti, viene completamente rimosso dal database.
- **Anomalia di inserimento:** non possiamo creare un impiegato senza nessun progetto associato.

## Perché questa situazione è indesiderata?

Perché parti diverse dell'informazione sono rappresentate nella stessa relazione:

- Gli impiegati ed il loro stipendio
- I progetti ed il loro budget
- Il ruolo di ciascun impiegato all'interno di un progetto a cui sta lavorando

## Dipendenze funzionali

Per studiare in maniera sistematica il concetto che abbiamo appena introdotto dobbiamo usare uno strumento matematico chiamato **dipendenza funzionale**.

Una dipendenza funzionale è uno specifico vincolo di integrità per il modello relazionale che descrive i limiti funzionali tra gli attributi di una relazione.

### L'idea di base

Ogni impiegato ha sempre lo stesso stipendio (anche se lavora a svariati progetti).

Ogni progetto ha sempre lo stesso budget.

Ogni impiegato ha sempre lo stesso ruolo in ogni progetto (anche se possono avere funzioni differenti in progetti diversi).

## Definizione

Data una relazione **r** avente uno schema **R(X)**.

Dati **Y** e **Z**, due sotto-insiemi non vuoti degli attributi **X**.

Una **dipendenza funzionale (FD)** esiste tra **Y** e **Z** se e solo se per ogni coppia di tuple  $t_1$  e  $t_2$  di **r** aventi gli stessi valori per gli attributi **Y**, risulta che  $t_1$  e  $t_2$  hanno gli stessi valori anche per gli attributi **Z**.

# Notazione

$$Y \rightarrow Z$$

Esempi:

- Employee → Wage
- Project → Budget
- Employee Project → Role

## Esempio

Employee	Wage	Project	Budget	Role
Jones	20	Mars	2	Technician
Smith	35	Jupiter	15	Designer
Smith	35	Venus	15	Designer
Williams	55	Venus	15	Chief
Williams	55	Jupiter	15	Consultant
Williams	55	Mars	2	Consultant
Brown	48	Mars	2	Chief
Brown	48	Venus	15	Designer
White	48	Venus	15	Designer
White	48	Jupiter	15	Director

- Employee → Wage
- Project → Budget
- Employee Project → Role

## Proprietà delle dipendenze funzionali

Employee Project → Project

Tale FD è **triviale**, ossia è sempre vera.

Una FD  $Y \rightarrow A$  è **non triviale** se una delle seguenti condizioni è soddisfatta:

- A è un attributo e non appartiene a Y
- A è un insieme di attributi e nessuno di essi appartiene a Y

## Anomalie e FD

Soltanente le anomalie dipendono da una qualche FD, ad esempio:

Gli impiegati devono avere un solo stipendio: Employee → Wage

I progetti devono avere un solo budget: Project → Budget

## Non tutte le FD provocano anomalie

In ogni progetto, ciascun impiegato ha un solo ruolo:

Employee Project → Role

Il vincolo è banalmente soddisfatto poiché Employee Project è una chiave.

## Conclusioni

Le prime due FD non sono chiavi e causano anomalie.

La terza FD è una chiave (Employee Project) e non causa anomalie.

La relazione contiene alcune informazioni legate alla chiave ed altre informazioni legate ad attributi che non compongono una chiave.

Quindi, le anomalie sono causate da informazioni eterogenee:

- Proprietà degli impiegati (lo stipendio)
- Proprietà dei progetti (il budget)
- Proprietà per la chiave Employee Project (il ruolo)

## Boyce-Codd Normal Form (BCNF)

Le forme normali sono proprietà (utili) soddisfatte solo in assenza di anomalie, definendo vincoli sulle dipendenze funzionali.

La forma normale più importante è quella chiamata di Boyce e Codd, o **BCNF**.

## Definizione

Una relazione **r** è in BCNF se e solo se, per ogni dipendenza funzionale (non triviale)  $X \rightarrow Y$  definita su **r**, **X** ha una chiave **K** di **r**, cioè **X** è superchiave per **r**.

## Decomposizione

Può capitare di avere una relazione che non soddisfa la BCNF. Nella maggior parte dei casi, possiamo sostituire tale relazione con due o più relazioni normalizzate che soddisfano la BCNF. Questo processo è chiamato **normalizzazione**.

Il processo è basato su un semplice criterio: se una relazione rappresenta più di un concetto dipendente, allora deve essere decomposta in relazioni più piccole, una per ciascun concetto.

## Esempio di decomposizione

Employee	Wage	Project	Budget	Role
Jones	20	Mars	2	Technician
Smith	35	Jupiter	15	Designer
Smith	35	Venus	15	Designer
Williams	55	Venus	15	Chief
Williams	55	Jupiter	15	Consultant
Williams	55	Mars	2	Consultant
Brown	48	Mars	2	Chief
Brown	48	Venus	15	Designer
White	48	Venus	15	Designer
White	48	Jupiter	15	Director

Employee	Wage
Jones	20
Smith	35
Williams	55
Brown	48
White	48

Project	Budget
Mars	2
Jupiter	15
Venus	15

Employee	Project	Role
Jones	Mars	Technician
Smith	Jupiter	Designer
Smith	Venus	Designer
Williams	Venus	Chief
Williams	Jupiter	Consultant
Williams	Mars	Consultant
Brown	Mars	Chief
Brown	Venus	Designer
White	Venus	Designer
White	Jupiter	Director

## Decomposizione con perdita

È possibile effettuare una decomposizione con perdita di contenuto informativo. Le decomposizioni che ci interessano sono solo quelle senza perdita.

Employee	Project	Office
Jones	Mars	Rome
Smith	Jupiter	Milan
Smith	Venus	Milan
White	Saturn	Milan
White	Venus	Milan

**Employee → Office**

Employee	Office
Jones	Rome
Smith	Milan
White	Milan

**Project → Office**

Project	Office
Mars	Rome
Jupiter	Milan
Venus	Milan
Saturn	Milan

Proviamo a ricomporre la tabella originale

Employee	Office
Jones	Rome
Smith	Milan
White	Milan



Office	Project
Rome	Mars
Milan	Jupiter
Milan	Venus
Milan	Saturn



Employee	Office	Project
Jones	Rome	Mars
Smith	Milan	Jupiter
Smith	Milan	Venus
Smith	Milan	Saturn
White	Milan	Jupiter
White	Milan	Saturn
White	Milan	Venus



Employee	Office	Project
Jones	Rome	Mars
Smith	Milan	Jupiter
Smith	Milan	Venus
White	Milan	Saturn
White	Milan	Venus

DIFFERENT FROM THE ORIGINAL RELATION!

## Ricordi dall'algebra

Il risultato del join tra  $R_1$  e  $R_2$  ha un numero di tuple compreso tra zero e  $|R_1| \times |R_2|$

$$0 \leq |R_1 \bowtie R_2| \leq |R_1| \times |R_2|$$

Se il join coinvolge una chiave da  $R_2$ , allora il numero di tuple risultanti è tra 0 e  $|R_1|$

$$0 \leq |R_1 \bowtie R_2| \leq |R_1|$$

Se il join coinvolge una chiave da  $R_2$  ed un vincolo di integrità referenziale, il numero di tuple è  $|R_1|$

$$|R_1 \bowtie R_2| = |R_1|$$

## Proprietà del join senza perdita

Definizione:

Una relazione  $r$  può essere decomposta senza perdite in due relazioni  $q(X)$  e  $s(Y)$  se e solo se il join tra le proiezioni di  $r$  su  $X$  e  $Y$  è uguale ad  $r$  stessa, cioè

$$\pi_X(r) \bowtie \pi_Y(r) = r$$

Questa proprietà è verificata se gli attributi comuni contengono una chiave per almeno una delle relazioni decomposte.

## Decomposizione senza perdita

Dati una relazione  $r(X)$  ed  $X_1$  e  $X_2$  sottoinsiemi di  $X$ , tali che  $X_1 \cup X_2 = X$ .

Dato  $X_0 = X_1 \cap X_2$ .

$r(X)$  può essere **decomposta senza perdita** in due relazioni  $q(X_1)$  e  $s(X_2)$  se e solo se:  
 $X_0 \rightarrow X_1$  è soddisfatta oppure  $X_0 \rightarrow X_2$  è soddisfatta.

## Esempio di decomposizione senza perdita

Employee	Project	Office
Jones	Mars	Rome
Smith	Jupiter	Milan
Smith	Venus	Milan
White	Saturn	Milan
White	Venus	Milan

Employee	Office
Jones	Rome
Smith	Milan
White	Milan

Employee	Project
Jones	Mars
Smith	Jupiter
Smith	Venus
White	Saturn
White	Venus

## Inserire una tupla

Supponiamo che venga inserita una nuova tupla (*White, Mars, Milan*).

Employee	Project	Office
Jones	Mars	Rome
Smith	Jupiter	Milan
Smith	Venus	Milan
White	Saturn	Milan
White	Venus	Milan
<b>White</b>	<b>Mars</b>	<b>Milan</b>

Employee	Project	Office
Jones	Mars	Rome
Smith	Jupiter	Milan
Smith	Venus	Milan
Smith	Mars	Milan
White	Saturn	Milan
White	Venus	Milan
White	Mars	Milan

Employee	Office
Jones	Rome
Smith	Milan
White	Milan

Project	Office
Mars	Rome
Jupiter	Milan
Venus	Milan
Saturn	Milan
Mars	Milan

Employee	Project	Office
Jones	Mars	Rome
Smith	Jupiter	Milan
Smith	Venus	Milan
White	Saturn	Milan
White	Venus	Milan
White	Mars	Milan

**Employee → Office**  
**Project → Office**

28

## Decomposizioni che preservano le dipendenze

Diciamo che una decomposizione **preserva le dipendenze** se ogni dipendenza funzionale dello schema originale coinvolge attributi che appaiono tutti insieme in uno degli schemi composti. Nel nostro caso Project → Office non viene preservata.

## Qualità delle decomposizioni

Il processo di normalizzazione attraverso decomposizione deve inoltre assicurare l'esistenza di proprietà aggiuntive che gli schemi relazionali, presi assieme, dovrebbero avere:

- Proprietà del **join senza perdita**, che assicura la ricostruzione dell'informazione originale
- **Preservazione delle dipendenze**, che assicura il mantenimento dei vincoli di integrità originali

## Problemi di decomposizione

Una relazione non in forma normale:

Chief	Project	Office
Smith	Mars	Rome
Johnson	Jupiter	Milan
Johnson	Mars	Milan
White	Saturn	Milan
White	Venus	Milan

**Project Office → Chief**

**Chief → Office**

La dipendenza **Project Office → Chief** coinvolge tutti gli attributi, quindi nessuna decomposizione può preservarla.

In alcuni casi la BCNF non può essere raggiunta.

# Terza forma normale (3NF)

La BCNF è più rigorosa della 3NF, in quanto la 3NF ammette relazioni con alcune anomalie.

La 3NF può sempre essere raggiunta (c'è un teorema).

Se una relazione ha soltanto una chiave, è in BCNF se e solo se è in 3NF.

## Definizione

Una relazione  $r$  è in **terza forma normale** se e solo se, per ogni FD non triviale  $X \rightarrow Y$  su  $r$ , almeno una delle seguenti condizioni è soddisfatta:

- $K \subset X$ ,  $X$  è superchiave in  $r$
- Ogni attributo  $Y$  è in almeno una chiave di  $r$

## Relazione non in forma normale

Chief	Project	Office
Smith	Mars	Rome
Johnson	Jupiter	Milan
Johnson	Mars	Milan
White	Saturn	Milan
White	Venus	Milan

$$\begin{array}{ll} \text{Project Office} \rightarrow \text{Chief} & \Leftarrow X \text{ is key} \\ \text{Chief} \rightarrow \text{Office} & \Leftarrow Y \subset \text{key} \end{array}$$

## Decomposizione in 3NF

Creare una relazione per ogni insieme di attributi coinvolto in una dipendenza funzionale.

Controllare che al termine almeno una relazione abbia una chiave della relazione originale.

Il processo dipende dalle FD trovate.

## Una possibile soluzione

Se la relazione non è normalizzata, decomporla in 3NF. Poi, controllare se lo schema risultante è in BCNF. Nella maggior parte dei casi, decomporre una relazione per arrivare alla 3NF permette di arrivare anche alla BCNF.

## Una relazione che non può essere messa in BCNF

Chief	Project	Office
Smith	Mars	Rome
Johnson	Jupiter	Milan
Johnson	Mars	Milan
White	Saturn	Milan
White	Venus	Milan

**Project Office → Chief**

**Chief → Office**

## Possibile riorganizzazione

Chief	Project	Office	Dept.
Smith	Mars	Rome	1
Johnson	Jupiter	Milan	1
Johnson	Mars	Milan	1
White	Saturn	Milan	2
White	Venus	Milan	2

**Dept. Office → Chief**

**Chief → Office Dept.**

**Project Office → Dept.**

## Decomposizione in BCNF

Chief	Office	Dept.
Smith	Rome	1
Johnson	Milan	1
White	Milan	2

Project	Office	Dept.
Mars	Rome	1
Jupiter	Milan	1
Mars	Milan	1
Saturn	Milan	2
Venus	Milan	2

## Teoria delle dipendenze

Il concetto appena introdotto potrebbe essere automatizzato con un processo algoritmico.

Data una **relazione** ed un insieme di **dipendenze funzionali** su di essa, generare una decomposizione di tale relazione contenente solo relazioni in **forma normale** che soddisfino le sopracitate proprietà di decomposizione, ovvero decomposizione senza perdita e preservazione delle dipendenze.

## Implicazioni

Se dalle dipendenze funzionali valide possiamo determinare altre dipendenze, diciamo che la prima implica le seconde.

Un insieme di dipendenze funzionali  $\mathbf{F}$  implica  $\mathbf{f}$  se e solo se ogni relazione che soddisfi tutte le dipendenze in  $\mathbf{F}$  soddisfa anche  $\mathbf{f}$ .

## Esempio

Employee	Type	Wage
Smith	3	30.000
Johnson	3	30.000
White	4	50.000
Williams	4	50.000
Brown	5	72.000

$\text{Employee} \rightarrow \text{Type}$  and  $\text{Type} \rightarrow \text{Wage}$

**IMPLY**

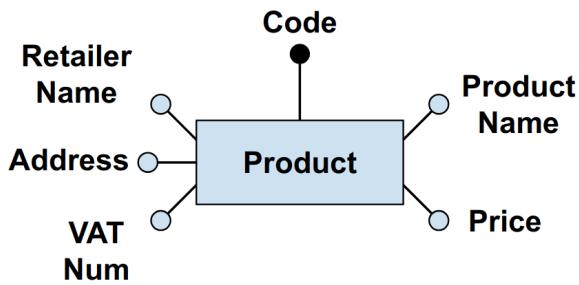
$\text{Employee} \rightarrow \text{Wage}$

Ogni relazione che soddisfi le prime due dipendenze soddisfa anche la terza.

# Progettazione e normalizzazione

La teoria della normalizzazione può essere usata durante la progettazione logica per controllare lo schema della relazione finale. Potrebbe anche essere usata durante la fase di progettazione concettuale per verificare la qualità dello schema concettuale.

## Normalizzazione sulle entità



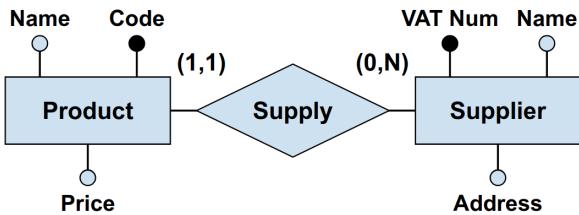
**VAT Num → RetailerName Address**

### Verifica

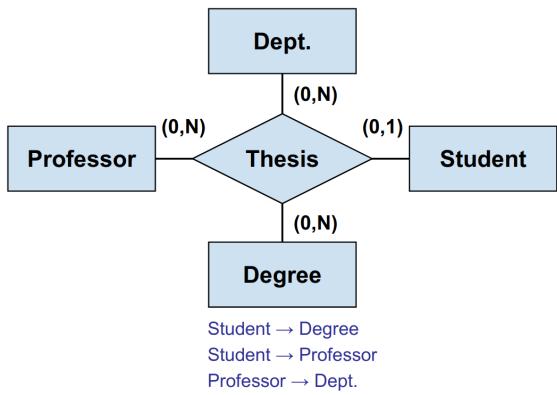
L'entità viola la forma normale a causa della dipendenza: **VAT Num → RetailerName Address**  
Possiamo decomporre l'entità usando tale dipendenza.

Porsi sempre il dubbio quando sullo schema sono presenti entità con tanti attributi che mettono insieme ambiti diversi.

## Decomposizione dell'entità



# Normalizzazione sulle associazioni

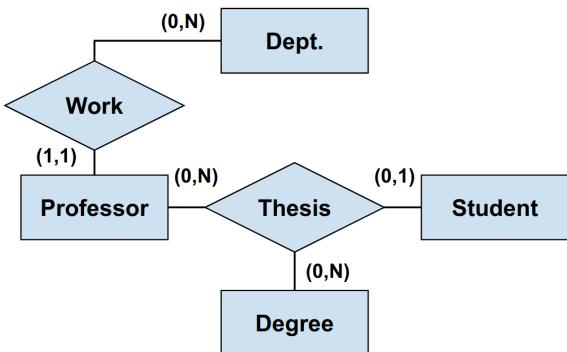


## Verifica

Non è in 3NF a causa della dipendenza: **Professor → Dept.**

Possiamo decomporre l'associazione usando questa dipendenza.

## Decomposizione dell'associazione



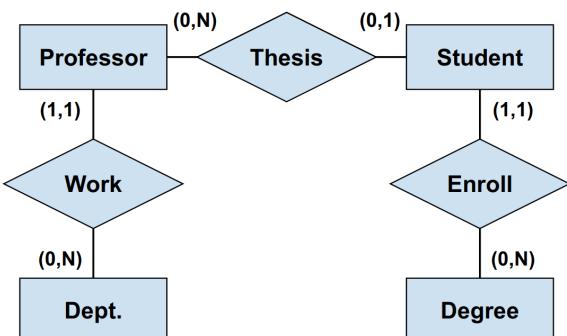
## Altra verifica

*Thesis* è in BCNF basandosi sulle dipendenze:

- **Student → Degree**
- **Student → Professor**

Le due proprietà sono indipendenti, quindi possiamo effettuare un'ulteriore decomposizione.

## Altra decomposizione



Durante la progettazione favorire le associazioni binarie in quanto più semplici da decomporre se necessario.

# Database attivi

## Introduzione

### Database passivi

Le strategie di reazione nei vincoli di integrità sono il primo esempio della necessità di introdurre un comportamento **reattivo** nei database.

```
ON < DELETE | UPDATE > < CASCADE | SET NULL | SET DEFAULT | NO ACTION >
```

L'idea è di introdurre **costrutti linguistici** specifici per questo obiettivo. Questi costrutti sono chiamati **regole** (attive) per gestire una parte del comportamento procedurale di un'applicazione. Essendo a livello di database, questo comportamento è condiviso tra molte applicazioni, ottenendo l'indipendenza dai dati.

## Database attivi

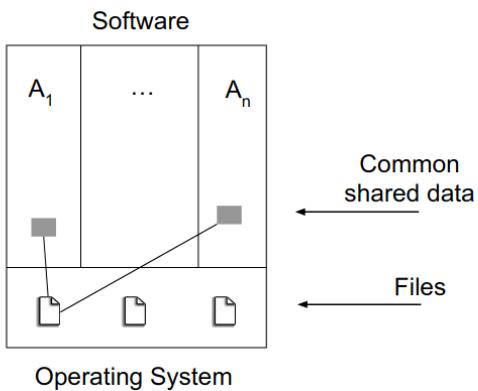
Un database attivo ha un componente per gestire le regole attive del tipo **Evento-Condizione-Azione**:

- gli eventi sono modifiche nel database
- una condizione è verificata, in base a un valore true/false
- vengono eseguite una o più azioni

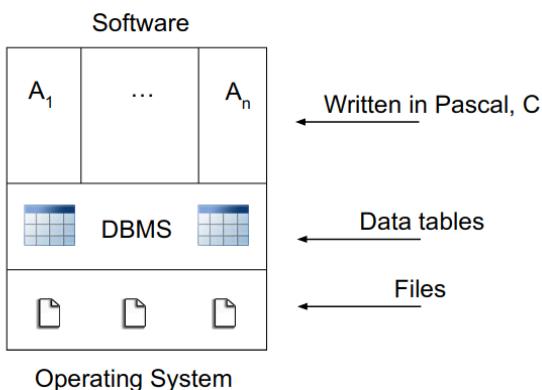
Questi database hanno un comportamento **reattivo** (opposto a **passivo**): non eseguono solamente le transazioni utente, ma anche le regole.

I DBMS commerciali (SQL3) utilizzano i **trigger** per specificare le regole, come quelle nella definizione dello schema.

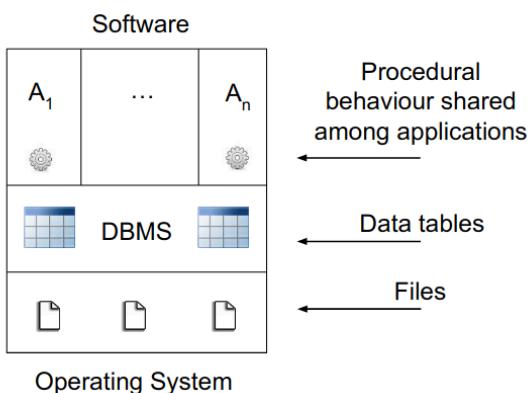
## Anni '70: nessun DBMS



## Anni '80: primi DBMS



## Anni '90: comportamento procedurale



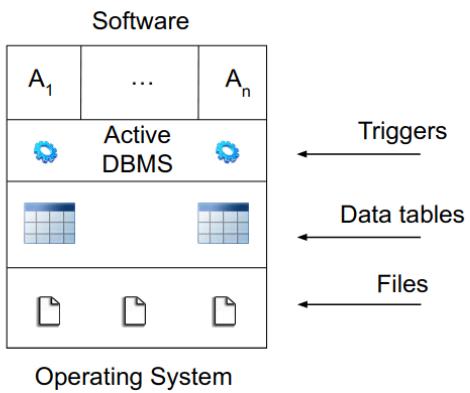
## Avanzamenti nei DBMS

Le stored procedure sono state introdotte per condividere i comportamenti procedurali comuni tra software diversi.

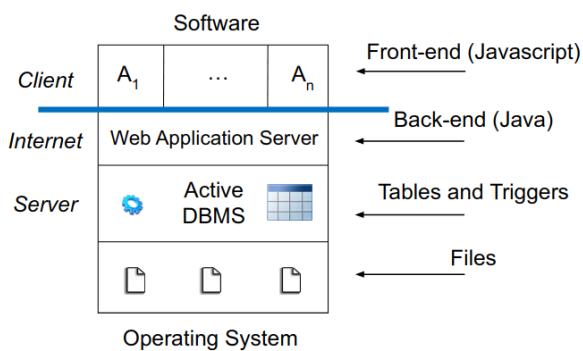
Le stored procedure non sono standardizzate e sono affette dal problema di "impedance mismatch" con il linguaggio utilizzato per esprimere tali procedure.

Di conseguenza sono state introdotte regole specifiche (*trigger*) per modellare il comportamento procedurale condiviso tra software diversi che sono gestite dal DBMS stesso.

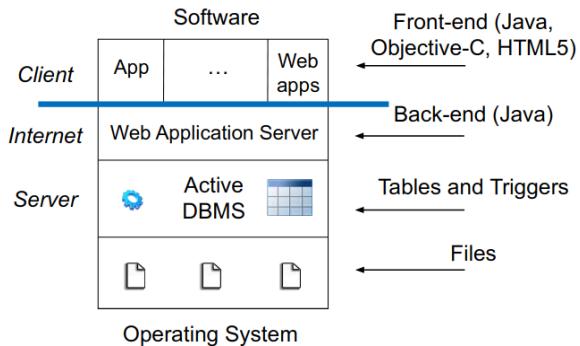
## Anni '90: DBMS attivi



## Anni 2000: web



## Anni 2010: applicazioni mobili



## Database attivo

Un database che contiene regole attive, chiamate **trigger**:

- Definizione di trigger in SQL:1999
- Definizione di trigger in Oracle e DB2
- Problemi di progettazione per applicazioni basate sull'uso di trigger

# Trigger

Viene definito con istruzioni DDL (**create trigger**).

Si basa sul paradigma Evento-Condizione-Azione (ECA):

- **Event**: cambiamento nei dati, specificato con `insert`, `delete` o `update`

- **Condition**: predicato SQL (opzionale)

- **Action**: sequenza di istruzioni SQL (oppure estensioni, come PL/SQL in Oracle)

Intuitivamente, quando si verifica un evento (attivazione), se la condizione è soddisfatta (verifica), allora esegui l'azione (esecuzione).

Ciascun trigger fa riferimento ad una tabella target, ovvero risponde ad eventi relativi a tale tabella.

## Granularità e modalità

Granularità:

- **di tupla** (a livello di riga): attivazione per ogni tupla coinvolta nell'operazione

- **di operazione** (a livello di istruzione): una sola attivazione per un'istruzione SQL, con riferimento a tutte le tuple coinvolte

Modalità:

- **immediata**: subito dopo (o subito prima) dell'evento

- **differita**: al momento del commit

## Modello computazionale

Sia  $T^U = U_1; \dots; U_n$  la transazione dell'utente.

Se le regole di  $P$  hanno la forma  $E, C \rightarrow A$ , con  $E$  evento,  $C$  condizione e  $A$  azione, allora:

- Semantica **immediata** genera  $T^I = U_1; U_1^P; \dots; U_n; U_n^P$

- Semantica **differita** genera  $T^D = U_1; \dots; U_n; U_1^P; \dots; U_n^P$

dove  $U_i^P$  rappresenta la sequenza di azioni indotte da  $U_i$  su  $P$ .

Alcuni problemi sono terminazione, confluenza ed equivalenza.

## Sintassi in SQL:1999

```
create trigger triggerName
  { before | after }
  { insert | delete | update [of column] }
    on targetTable
  [referencing
    {[old table [as] oldTableVar] [new table [as] newTableVar] } |
    {[old [row] [as] oldTupleVar] [new [row] [as] newTupleVar] }
  ]
  [for each { row | statement }]
  [when condition]
SQLProceduralStatement
```

### Tipi di evento

- **Before**
  - il trigger viene considerato ed eventualmente eseguito prima dell'evento
  - di solito viene utilizzato quando si desidera verificare una modifica prima che avvenga e "modificare la modifica"
- **After**
  - il trigger viene considerato ed eseguito dopo l'evento
  - questa è la modalità più comune, adatta alla maggior parte delle applicazioni

### Clausola referencing

Dipende dalla granularità:

- a livello di istruzione: due tavole di transizione (**old table** e **new table**) contengono i valori precedenti e successivi delle tuple modificate dall'istruzione
- a livello di riga: due variabili di transizione (**old** e **new**) rappresentano il valore prima o dopo che una tupla è stata modificata

**old** e **old table** non sono presenti con l'evento `insert`.

**new** e **new table** non sono presenti con l'evento `delete`.

## Sintassi in Oracle

```
create trigger triggerName
  { before | after } event [, event [, event]]
  [[referencing
    [old [row] [as] oldTupleVar]
    [new [row] [as] newTupleVar]
  ]
  for each { row | statement } [when condition]]
PL/SQLblock
```

- `{ before | after }` : tipo di evento
- `event` : insert, update o delete
- `for each` : specifica la granularità
- `referencing` : consente di definire nomi delle variabili (solo per la granularità di tupla)

## Semantica

Modalità **immediata**, sia con `before` che con `after`.

Piano di esecuzione:

- trigger `before statement`
- per ogni tupla coinvolta:
  - trigger `before row`
  - esecuzione e controllo dei vincoli
  - trigger `after row`
- trigger `after statement`

In caso di errore, viene scartato interamente. Priorità fra i trigger basata sui timestamp. Al massimo 32 trigger attivati in cascata.

## Esempio

```
create trigger Reorder
  after update of QtyAvbl on Warehouse
  when (new.QtyAvbl < new.QtyLimit)
  for each row
    declare X number;
begin
  select count(*) into X
  from PendingOrders
  where Part = new.Part;
  if X = 0
  then
    insert into PendingOrders
    values (new.Part, new.QtyReord, sysdate);
  end if;
end;
```

Warehouse	Part	QtyAvbl	QtyLimit	QtyReord
	1	200	150	100
	2	780	500	200
	3	450	400	120

```
T1: update Warehouse
  set QtyAvbl = QtyAvbl - 70
  where Part = 1

T2: update Warehouse
  set QtyAvbl = QtyAvbl - 60
  where Part <= 3
```

## Sintassi in DB2

```
create trigger triggerName
  { before | after } event on targetTable
  [referencing reference]
  for each level
  [when (SQLPredicate)]
  SQLProceduralStatement
```

- `{ before | after }`: tipo di evento
- `event` : insert, update o delete
- `for each level` : specifica la granularità
- `referencing` : consente di definire nomi delle variabili (dipende dalla granularità)

## Semantica

Modalità **immediata**, sia con `before` che con `after`.

I trigger `before` non possono modificare il database, a parte varianti sulle modifiche causate dall'evento (quindi non possono in generale attivare altri trigger)

In caso di errore, tutto viene scartato. Nessuna priorità fra i trigger (l'ordine è definito dal sistema, ad esempio con i timestamp), interazione con azioni compensative sui vincoli di integrità referenziale. Al massimo 16 trigger attivati in cascata.

## Esempio

```
create trigger checkWage
  after update of Wage on Employee
  for each row
  when (new.Wage < old.Wage * 0.97)
  begin
    update Employee
    set Wage = old.Wage * 0.97
    where EmpCode = new.EmpCode;
  end;
```

## Estensioni (non sempre disponibili)

- Eventi temporali (anche periodici) o "definiti dall'utente"
- Combinazioni booleane di eventi
- Clausola `instead of`
- Esecuzione "scollegata", ovvero viene attivata una transazione autonoma
- Definizione delle priorità
- Regole di gruppo, attivabili e disattivabili
- Regole associate anche alle query, non solo agli aggiornamenti

## Applicazioni

Funzionalità interne ed applicative:

- Gestione dei vincoli di integrità
- Replicazione
- Gestione delle viste
  - materialized: propagazione
  - virtual: modifica delle query
- Descrizione del comportamento del database