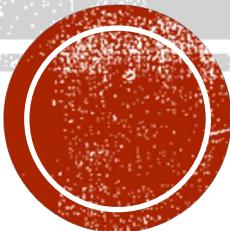


# BASI DI DATI I

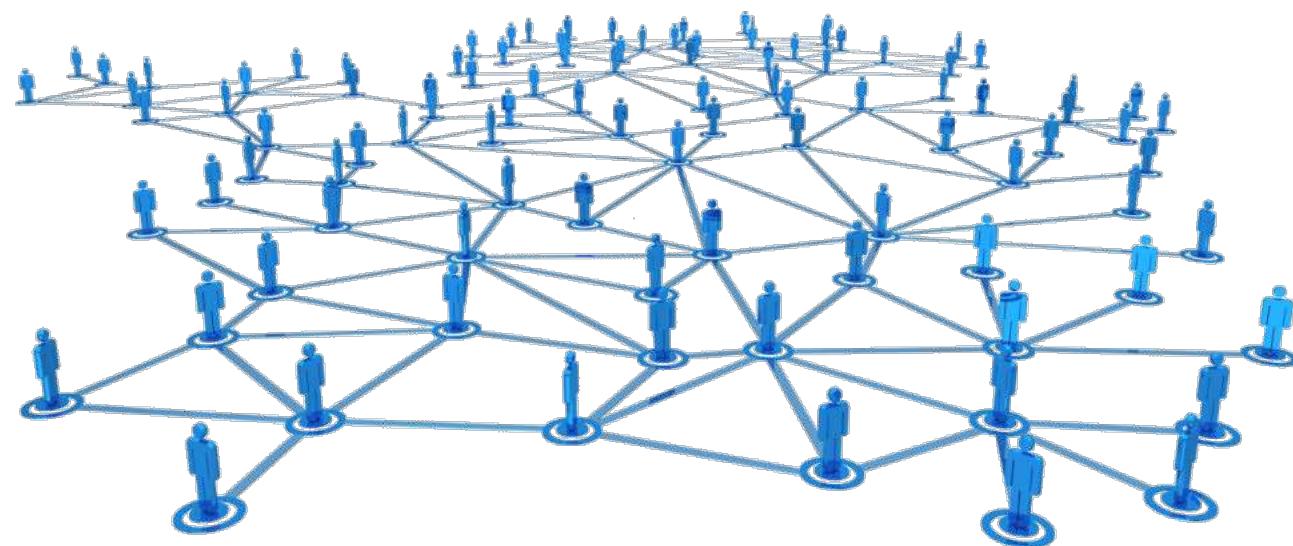
- Introduzione al corso
- Database Management Systems (DBMS)
- Basi di dati e DBMS
- Un esempio di Base di Dati
- Utilizzo di un DB
- Gli Utenti di un DB
- Vantaggi di un DBMS

# **INTRODUZIONE AL CORSO**



# SISTEMI INFORMATIVI

- Sistema informativo
- Sistema informatico
- Informazioni
- Dati



# SISTEMA INFORMATIVO

- Componente (sottosistema) di una organizzazione che gestisce (acquisisce, elabora, conserva, produce) le informazioni di interesse (cioè utilizzate per il perseguimento degli scopi dell'organizzazione).



# **SISTEMA INFORMATIVO**

- Ogni organizzazione ha un sistema informativo, eventualmente non esplicitato nella struttura.
- Quasi sempre, il sistema informativo è di supporto ad altri sottosistemi, e va quindi studiato nel contesto in cui è inserito.
- Il sistema informativo è di solito suddiviso in sottosistemi (in modo gerarchico o decentrato), più o meno fortemente integrati.



# SISTEMA ORGANIZZATIVO

- Insieme di risorse (persone, denaro, materiali, informazioni) e regole per lo svolgimento coordinato delle attività (processi) al fine del perseguimento degli scopi.



# SISTEMA ORGANIZZATIVO E SISTEMA INFORMATIVO

- Il sistema informativo è parte del sistema organizzativo.
- Il sistema informativo esegue/gestisce processi informativi (cioè i processi che coinvolgono informazioni).



# SISTEMI INFORMATIVI E AUTOMAZIONE

- Il concetto di “ sistema informativo” è indipendente da qualsiasi automatizzazione:
  - esistono organizzazioni la cui ragion d’essere è la gestione di informazioni (p. es. servizi anagrafici e banche) e che operano da secoli

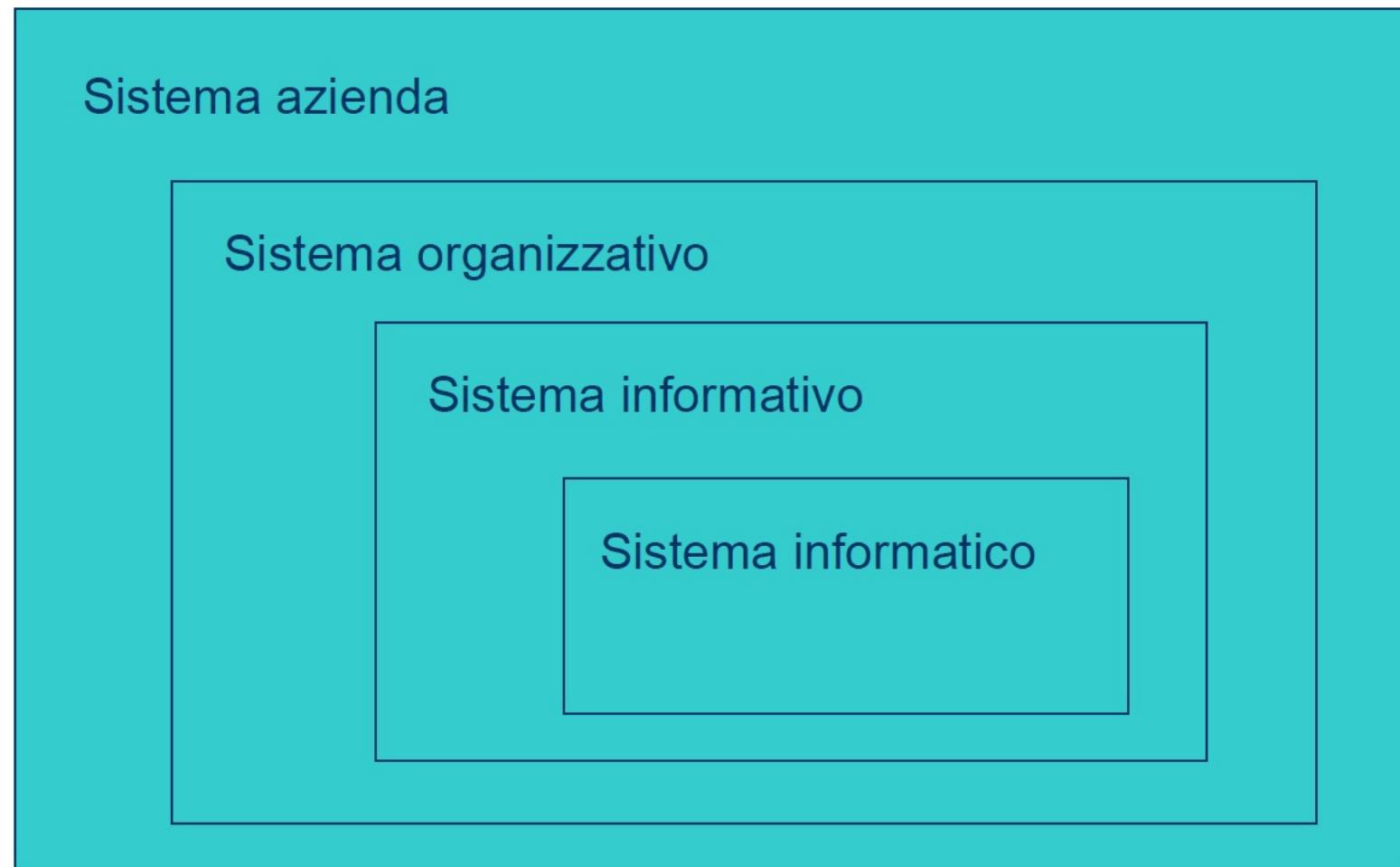


# SISTEMA INFORMATICO

- porzione automatizzata del sistema informativo:
  - la parte del sistema informativo che gestisce informazioni con tecnologia informatica.



# ORGANIZZAZIONE AZIENDALE DI UN SISTEMA INFORMATICO



# I DATABASE

I database sono ormai una componente fondamentale della vita di tutti i giorni: molte delle nostre più banali attività ci portano ad interagire con qualche tipo di database.

Qualche esempio:

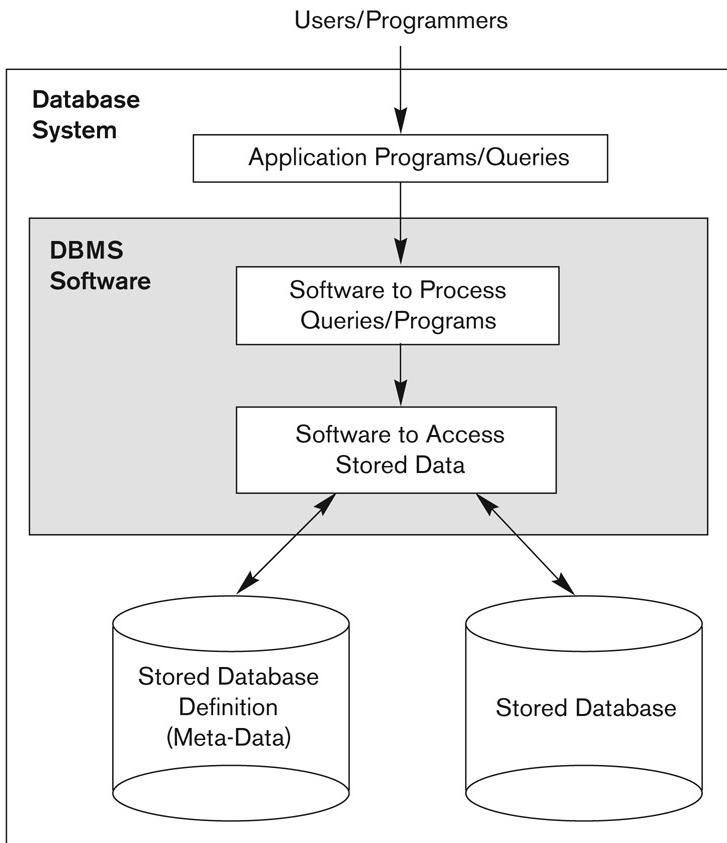
- Prenotazioni di alberghi, biglietti aerei,...
- Telepass / Viacard
- Ricerca nel catalogo elettronico di una biblioteca
- Richiesta di documenti
- Spesa al supermercato
- Operazioni bancarie
- ...



# QUALCHE DEFINIZIONE DI BASE

- Un *Database* (*DB o Base di Dati*) è una collezione di dati correlati:
  - *Esempio:* una rubrica telefonica creata usando Access, Excel, ecc.
- Per “dati” si intendono dei fatti noti, con un significato implicito, che possono essere memorizzati.
  - *Es:* nome, cognome, indirizzo e telefono di un abbonato telefonico.
- Un **mini world** è una porzione del mondo reale di cui è fatto il mirroring all'interno del database
  - Es: i voti degli studenti agli esami di profitto
- DBMS: Database management system per facilitare la creazione e la gestione di un database
  - Es: MySQL Server, Oracle, Postgres
- Database System: DBMS + Dati



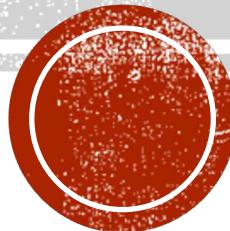


**Figure 1.1**  
A simplified database system environment.



# DATABASE ENVIRONMENT SIMPLIFICATION

# **DATABASE MANAGEMENT SYSTEM (DBMS)**



# IL DBMS

- Un *database management system (DBMS)* è una collezione di programmi che permette di creare e manutenere una base di dati.
- È un software "general-purpose" che facilita la creazione, costruzione e gestione di database per differenti applicazioni.
- Fornisce un modo per memorizzare informazioni in strutture dati efficienti, scalabili e flessibili.



# TIPICHE FUNZIONALITÀ DI UN DBMS

- **Definire** un particolare database in termini di
  - Tipi di dati
  - Strutture
  - Vincoli
- **Costruire** o **Caricare** il contenuto iniziale del database su un mezzo di memorizzazione (storaging) secondario
- **Manipolare** il database
  - Recuperare informazioni: Query e generazione di report
  - Modificare le informazioni: Inserimento, cancellazioni e aggiornamenti del contenuto del DB
  - Accedere al DB: tramite applicazioni stand-alone o web applications
- **Elaborare** i dati e **Condividere** tra gli utenti e i programmi applicativi, mantenendo i dati consistentemente

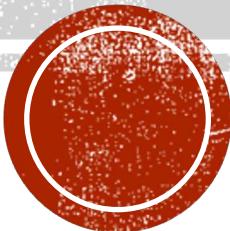


# **FUNZIONALITÀ AGGIUNTIVE DI UN DBMS**

- **Fornire misure di Protezione e/o Sicurezza** per prevenire accessi non autorizzati
- **Elaborazione Attiva** sui dati per facilitare azioni nel futuro
- **Fornire tools per la presentazione** e la visualizzazione dei dati
- **Fornire strumenti**, automatizzati o temporizzati per la manutenzione ordinaria e straordinaria delle applicazioni del database e dei programmi associati



# BASI DI DATI E DBMS



# APPLICAZIONI DEI DATABASE

Le attività appena descritte coinvolgono applicazioni di database tradizionali, avendo a che fare principalmente con testi e numeri.

I progressi tecnologici, però, stanno apreendo la strada a nuove interessantissime applicazioni di database:

- I Database Multimediali possono memorizzare immagini, videoclip, suoni, ecc..
- I Sistemi informativi geografici (GIS) possono memorizzare ed analizzare mappe ed immagini satellitari
- I sistemi Data Warehouse permettono di estrarre ed analizzare grandi quantità di dati
  - Forniscono un supporto al processo decisionale
- I motori di ricerca permettono di trovare informazioni sparse sul WWW
- Tecnologie di database real-time
  - Controllo industriale e processi di produzione



# APPLICAZIONI DEI DATABASE (2)

- Ad aumentare il ventaglio di database in uso al giorno d'oggi, l'esplosione ed il consolidamento nella nostra vita quotidiana dei Social Network
  - Facebook
  - Instagram
  - Twitter
  - LinkedIn
  - ...
- Si ha necessità di memorizzare i post, gli utenti, le foto, i video
- Nuove tecnologie stanno emergendo per venire incontro alle odierne necessità
  - Big Data Storage Systems (computer Clusters organisation)
  - NOSQL (Not Only SQL) Systems
  - Cloud Storaging



# LA NOZIONE DI DATABASE (2)

L'uso comune del termine database è più ristretto.

Un 'database' deve presentare le seguenti proprietà:

- Rappresenta alcuni aspetti del mondo reale, detto miniworld o Universo del Discorso (UOD). Cambiamenti al miniworld sono riflessi nel database.
- È una collezione di dati logicamente correlati con qualche significato inerente.
  - Un assortimento casuale di dati non può correttamente essere considerato un database.
- È progettato, costruito e riempito di dati per un utilizzo specifico. Ha una tipologia ben definita di utenti ed è realizzato per delle applicazioni a cui tali utenti sono interessati.



# DIMENSIONE DI DATABASE

Un database può avere qualsiasi dimensione e complessità **Esempi**

- Una rubrica telefonica personale può avere poche centinaia di voci.
- Il database dei contribuenti americani e delle relative dichiarazioni dei redditi ha delle dimensioni notevoli:
  - 100 milioni di contribuenti
  - mediamente 5 moduli per ciascuna dichiarazione,
  - 200 byte per ogni modulo:

$$\begin{aligned}100 \times 10^6 \times 200 \times 5 &= 100 \times 10^6 \times 100 \times 2 \times 5 = \\&= 10^2 \times 10^6 \times 10^2 \times 10 = 10^{11} \text{ bytes}\end{aligned}$$

- Tenendo traccia delle ultime quattro dichiarazioni, risulterebbe una base di dati di  $4 \times 10^{11}$  bytes = 400 Gigabytes
- Questo enorme ammontare di informazioni deve essere organizzato e gestito in modo tale che gli utenti possano interrogare, recuperare ed aggiornare i dati.



# GESTIONE DI DATABASE

- Un database può essere gestito manualmente (es. lo schedario di una biblioteca) o attraverso un elaboratore elettronico.
- Un database computerizzato può essere creato e gestito o da programmi realizzati “*ad hoc*” o da un “DBMS”
  - Come abbiamo visto in precedenza, un DBMS può fornire strumenti per la manutenzione e la gestione di una base di dati.



# DBMS VS. DATABASE

- Un DBMS è un applicativo per gestire database.
  - *Esempio: MySQL*
- Un database è un insieme di dati.
  - *Esempio: file con estensione .MDB*
- Stessa differenza esistente tra Word (**applicativo**) e file .DOC (**dati**).

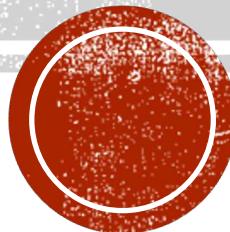


# DBMS SPECIAL-PURPOSE

- Non è necessario usare un DBMS general-purpose.  
È anche possibile scrivere un proprio insieme di programmi per gestire i dati, creando un DBMS special-purpose.
- Tale approccio può essere vantaggioso nello sviluppo di soluzioni molto piccole.



# *UN ESEMPIO: IL DATABASE UNIVERSITÀ*



# UN ESEMPIO DI DATABASE

Vogliamo realizzare il database **UNIVERSITÀ** per gestire gli studenti, i corsi (con prerequisiti) e gli esami superati.

- Organizzato in quattro file:
  - STUDENTE:
    - Contiene i dati su ciascuno studente iscritto.
  - CORSO:
    - Contiene i dati relativi a ciascun corso.
  - PREREQUISITI:
    - Contiene i prerequisiti di ciascun corso.
  - VOTAZIONE:
    - Contiene i voti riportati dagli studenti nei vari esami.
- Ogni file memorizza dei record di dati dello stesso tipo.



# *ESEMPIO: DEFINIZIONE DEL DB*

Per definire il database occorre specificare la struttura dei record di ciascun file.

Occorre cioè:

- Specificare i campi (data element) di ogni record.
- Specificare il tipo di ogni data element in ciascun record.



# ESEMPIO: DEFINIZIONE DEL DB (2)

- I data element:
  - Un record del file STUDENTE contiene dati per rappresentare il **nome** dello studente, il numero di **matricola**, e l'**anno** di iscrizione corrente.

STUDENTE	Nome	Matricola	Anno
	Neri	N86000323	2 f.c.
	Bianchi	N86000084	5
	Verdi	N86000579	3
	...	...	...

- Il tipo dei data element:
  - Gli elementi NOME, MATRICOLA ed ANNO sono tutti definiti come stringhe di caratteri.



# ESEMPIO: DEFINIZIONE DEL DB (3)

La definizione del resto del database UNIVERSITÀ:

CORSO	DENOMIN.	SEMESTRE	TITOLARE
	Basi di Dati	1	Peron
	Sistemi Operativi 2	2	Barra
	Object Orientation	1	Di Martino
	...	...	...

PREREQUISITI	DENOMIN.	PROPEDEUTICITA'
	Basi di Dati	nessuna
	Sistemi Operativi 2	Sistemi Operativi 1
	Basi di Dati 2	Basi di Dati
	...	...

VOTAZIONE	NOME	DENOMIN.	VOTO
	Bianchi	Basi di Dati	24
	Neri	Sistemi Operativi 2	28
	Verdi	Object Orientation	18
	Bianchi	Sistemi Operativi 2	23

Per ognuno dei quattro file, viene specificato il tipo di ogni data element contenuto in ciascun record del file.



# *ESEMPIO: COSTRUZIONE DEL DB*

- Per costruire il database UNIVERSITÀ memorizziamo dati per rappresentare ogni studente, corso, prerequisito e votazione nel file appropriato.
- I record nei vari file possono essere correlati:

*Esempio:*

- Il record per "Bianchi" nel file STUDENTE è in relazione con due record nel file VOTAZIONE,
- Il record per »Basi di Dati" nel file CORSO è in relazione con un record nel file PREREQUISITI e con un record nel file VOTAZIONE.



# *ESEMPIO: MANIPOLAZIONE DEL DB*

- *Manipolare il database significa interrogare e aggiornare i dati.*

*Esempio di query:*

- Quanti esami ha sostenuto “Verdi”?
- Elencare gli studenti in corso.
- Calcolare la media dei voti di uno studente.
- ...



# **ESEMPIO: MANIPOLAZIONE DEL DB (2)**

- **Manipolare** il database significa interrogare e aggiornare i dati.

*Esempio di update:*

- Inserire un nuovo studente.
- Registrare un esame.
- ...



# **FASI PER LA PROGETTAZIONE DI UN DATABASE**

- 1. Specifica ed analisti dei Requisiti**
- 2. Progettazione Concettuale**
- 3. Progettazione Logica**
- 4. Progettazione Fisica**



# **UTILIZZO DI UN DB**



# NATURA AUTODESCRITTIVA DI UN DATABASE

- Il database non contiene solo i dati ma anche la **definizione completa** (o descrizione) del database.
- Le informazioni sulla definizione, dette **metadati**, sono memorizzate nel **catalogo di sistema**.
  - Il catalogo salva la descrizione di un particolare database (strutture dati, tipi, vincoli)
  - Il DBMS accede al catalogo per recuperare le info
  - In questo modo il DBMS è in grado di lavorare con differenti DB applications

## METADATI

```
type Studente=
record
  nome : string;
  matricola : string;
  anno : string;
end;
```

## DATI

Nome	Matricola	Anno
Neri	N86000323	2 f.c.
Bianchi	N86000084	5
Verdi	N86000579	3
...	...	...



# INDIPENDENZA PROGRAMMA-DATI

File processing	<ul style="list-style-type: none"><li>◆ La struttura di un file dati è immersa nei programmi che accedono al file:<ul style="list-style-type: none"><li>▪ un cambiamento nella struttura del file richiede un cambiamento in tutti i programmi.</li></ul></li></ul>
DBMS	<ul style="list-style-type: none"><li>◆ I programmi di accesso sono scritti indipendentemente dagli specifici file.</li><li>◆ La struttura dei file dati è nel catalogo.</li></ul>



# FILE PROCESSING VS DATABASE

## ■ **File processing:**

Ogni utente definisce ed implementa i file necessari per una specifica applicazione, con notevole spreco di risorse umane/informatiche e ridondanza dei dati.

*Esempio:*

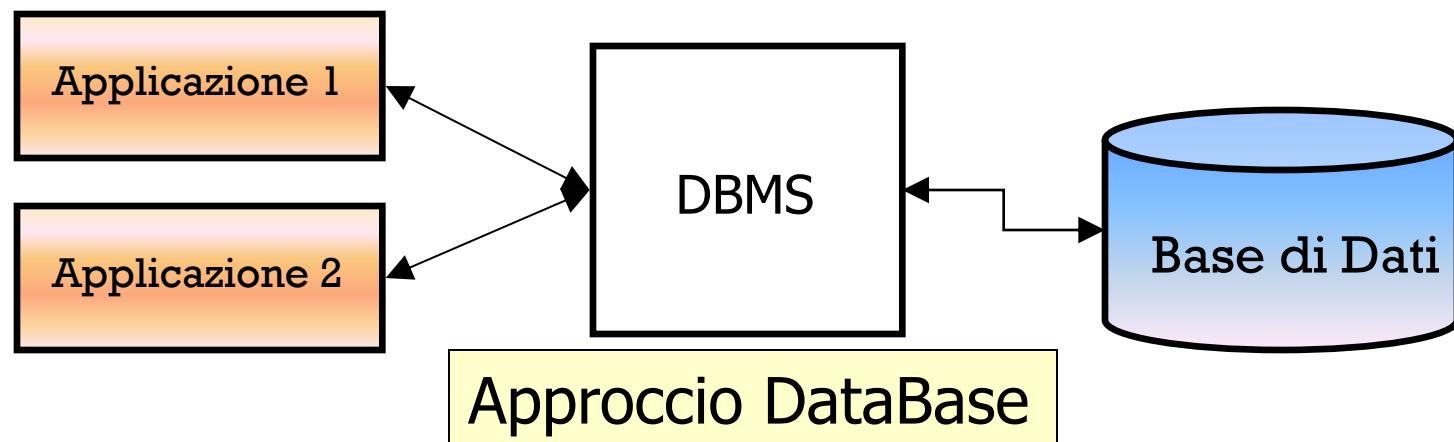
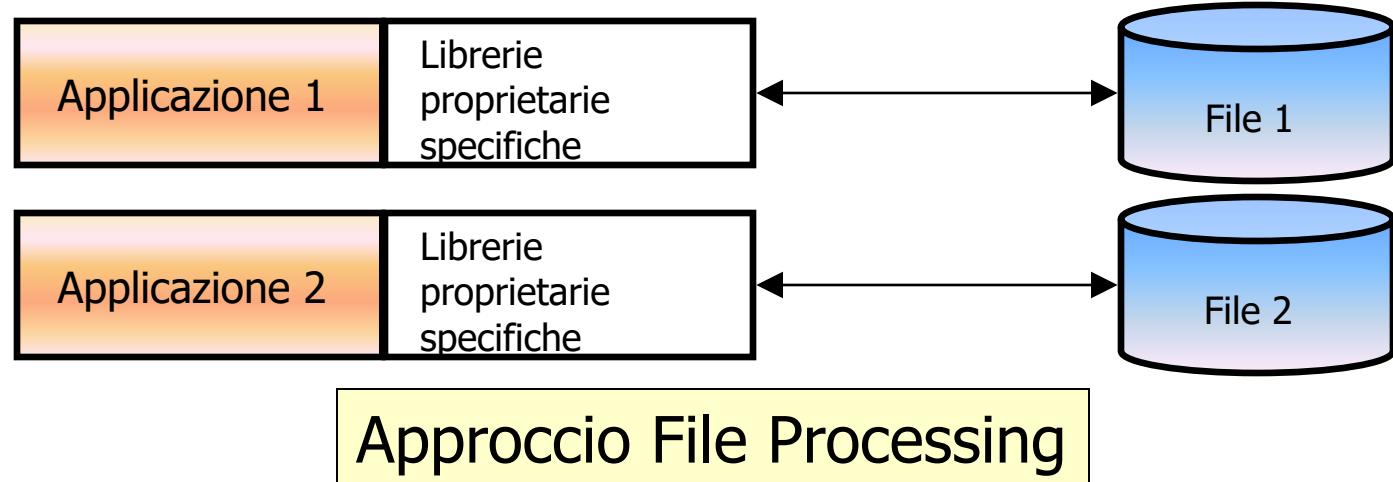
- L'ufficio esami mantiene un file che registra gli studenti e le votazioni relative
- L'ufficio iscrizioni mantiene un file che registra gli studenti e i relativi pagamenti delle tasse.

## ■ **Database:**

Si definisce una volta per tutte un singolo repository di dati, poi utilizzato dai vari utenti



# FILE PROCESSING VS DATABASE (SCHEMA)



# MODELLO DI DATI

- Un DBMS fornisce una rappresentazione concettuale dei dati che non include molti dei dettagli su come i dati sono memorizzati.
- Un modello di dati (data model) è un tipo di astrazione di dati usato per fornire la rappresentazione concettuale.
- Il modello di dati usa concetti logici quali oggetti, proprietà e loro interrelazioni, nascondendo quindi i dettagli fisici di memorizzazione.



# PROPRIETÀ DI UN DATABASE

- Abilitazione di viste multiple dei dati:
  - Un database ha molti utenti e ciascuno può averne una diversa prospettiva (o vista):
    - Una vista può essere un sottoinsieme del database,
    - o può contenere dati virtuali (derivati dal database ma non esplicitamente memorizzati).
- Un DBMS deve consentire la definizione di viste multiple.

ESAMI SUPERATI	Nome	Matricola	ESAMI DENOMIN.	VOTO
Bianchi	N86000084		Basi di Dati	24
			Sistemi Operativi 2	23
Verdi	N86000579		Object Orientation	18
			Algoritmi	27



# PROPRIETÀ DI UN DATABASE

- **Condivisione** di dati e trattamento di *transazioni* multiutente.
- Un DBMS multiutente deve consentire accesso a più utenti contemporaneamente.
  - **Online transaction processing (OLTP)**
- Deve includere software per il controllo della concorrenza che garantiscano l'aggiornamento corretto.

*Esempio:* il problema della prenotazione di posti per una compagnia aerea (applicazione di transaction processing).



# PROPRIETÀ DELLE TRANSAZIONI

- Le transazioni dovrebbero possedere alcune proprietà (*dette ACID properties, dalle loro iniziali*):
  - **Atomicità:** una transazione è un'unità atomica di elaborazione da eseguire o completamente o per niente (*responsabilità del recovery subsystem*).
  - **Consistency preserving:** una transazione deve far passare il database da uno stato consistente ad un altro (*responsabilità dei programmati*).
  - **Isolation:** Una transazione non deve rendere visibili i suoi aggiornamenti ad altre transazioni finché non è committed (*responsabilità del sistema per il controllo della concorrenza*)
  - **Durability:** Se una transazione cambia il database, e il cambiamento è committed, queste modifiche non devono essere perse a causa di fallimenti successivi (*responsabilità del sistema di gestione dell'affidabilità*)



# **GLI UTENTI DI UN DB**



# GLI UTENTI DEL DB (BEHIND THE SCENES)

- DataBase Administrator (DBA):

- Il DBA è responsabile per autorizzare l'accesso al database, coordinare e monitorare il suo uso, acquisire nuove risorse hardware e software.
- In grosse organizzazioni è assistito da uno staff.

- Database Designer (Progettista):

- È responsabile dell'individuazione dei dati da memorizzare nel DB;
- È responsabile della scelta delle strutture opportune.
- Deve capire le esigenze dell'utenza del DB e giungere a un progetto che soddisfi i requisiti:
- Sviluppa viste dei dati;
- Il DB finale deve essere in grado di supportare i requisiti di tutti i gruppi di utenti.



# GLI UTENTI DEL DB (ACTORS ON THE SCENE)

- Utenti finali

- utenti finali casuali: accedono occasionalmente al DB e lo interrogano attraverso query Languages sofisticati;
- utenti finali naive o parametrici: rappresentano una parte considerevole di utenza. Usano aggiornamenti e queries standard;
- utenti finali sofisticati: ingegneri, scienziati e analisti di affari che hanno familiarità con le facilities del DBMS per richieste complesse;
- utenti stand-alone: gestiscono databases personali usando pacchetti applicativi.

- Analisti di sistema e Programmatori di applicazioni

- gli analisti di sistema determinano i requisiti degli utenti finali e sviluppano specificazioni per le transazioni;
- i programmatori di applicazioni implementano le specifiche come programmi.

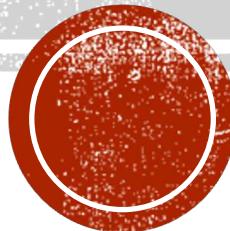


# ALTRÉ FIGURE

- **Progettisti e Implementatori di DBMS**
  - Disegnano e implementano moduli e interfacce di DBMS come un pacchetto software. Un DBMS è un complesso sistema software che consiste di molti moduli.
- **Sviluppatori di Tools**
  - Implementano pacchetti software per il progetto, il monitoraggio, l'interfaccia, la prototipazione, ecc. di databases.
- **Operatori e Personale per la Manutenzione**
  - Personale che si occupa dell'amministrazione del sistema e della manutenzione hw e sw del sistema di database.



# VANTAGGI DI UN DBMS



# VANTAGGI PRIMARI

- Controllo della ridondanza
  - Normalizzazione dei dati.
- Limitare l'accesso non autorizzato
  - Sicurezza e sottosistema di autorizzazione.
- Fornisce una memoria *persistente* per gli oggetti dei programmi
  - Oggetti complessi in Java/C++ possono essere permanentemente memorizzati.



# VANTAGGI AGGIUNTIVI

- Forniscono strutture di memorizzazione e implementano tecniche di ricerca per eseguire le query in maniera efficiente:
  - **Indici**
  - **Buffering e caching**
  - **Elaborazione ed ottimizzazione delle query**
- Forniscono strumenti di *backup* e *recovery*
- Forniscono interfacce utente multiple
  - **Graphical user interface (GUI)**



# IMPLICAZIONI ADDIZIONALI

- Permettono l'uso dei Trigger:
- Sono regole attivate dagli aggiornamenti apportati alle tabelle.
- Gestiscono le Stored procedure:
- Sono procedure di supporto usate per rispettare le regole definite sul database.
- Riduce il tempo di sviluppo dell'applicazione.
- flessibilità
- Disponibilità di aggiornamento delle informazioni
- Economie di scala
  - Vantaggio nel costo medio unitario all'aumentare della scala



# QUANDO NON USARE UN DBMS

- È desirabile usare l'approccio basato sui *file*, quando:
  - Si sviluppano semplici e ben definite applicazioni di database che non prevedono modifiche.
  - Sono richiesti requisiti rigorosi real-time che non possono essere soddisfatte a causa del sovraccarico eseguito dal DBMS.
  - Si implementano sistemi embedded con una capacità di memorizzazione limitata.
  - L'accesso ai dati non richiede utenti multipli.

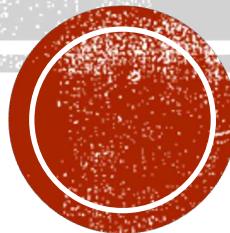




# FINE

Per eventuali domande: (in ordine di preferenza personale)

- Ora.
- Chat di Teams
- Mail: [silvio.barra@unina.it](mailto:silvio.barra@unina.it)



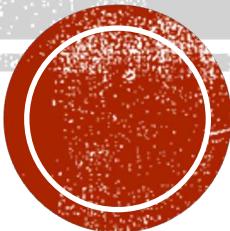
# BASI DI DATI I

- Modelli dei Dati
- Architettura dei R-DBMS
- Indipendenza logica e fisica dei dati
- Dati e Metadati
- Architettura Client Server

Prof. Adriano Peron

Prof. Silvio Barra

# MODelli DEI DATI



# MODELLI ED ASTRAZIONE DATI

- L'approccio database ha la caratteristica fondamentale di fornire una forte astrazione dei dati, nascondendo tutti i dettagli di memorizzazione non necessari agli utenti.
- L'astrazione dei dati si ottiene per mezzo di un data model.
- Un insieme di concetti che possono essere usati per descrivere la struttura di una base di dati.



# MODELLI DEI DATI

- Un data model è un insieme di concetti per descrivere
  - La **struttura** di un database
    - *Es:* costrutti per definire gli elementi ed il loro tipo, entità, record e relazioni
  - Le **operazioni** per manipolare le strutture
    - *Es:* Inserzione, cancellazione, modifica, ritrovamento di un oggetto.
    - *Es:* Un'operazione COMPUTE\_GPA che calcola la media dei voti e che può essere applicata a un oggetto studente.
  - Alcuni **vincoli** cui il database deve sottostare
    - I vincoli specificano restrizioni sui dati per renderli validi



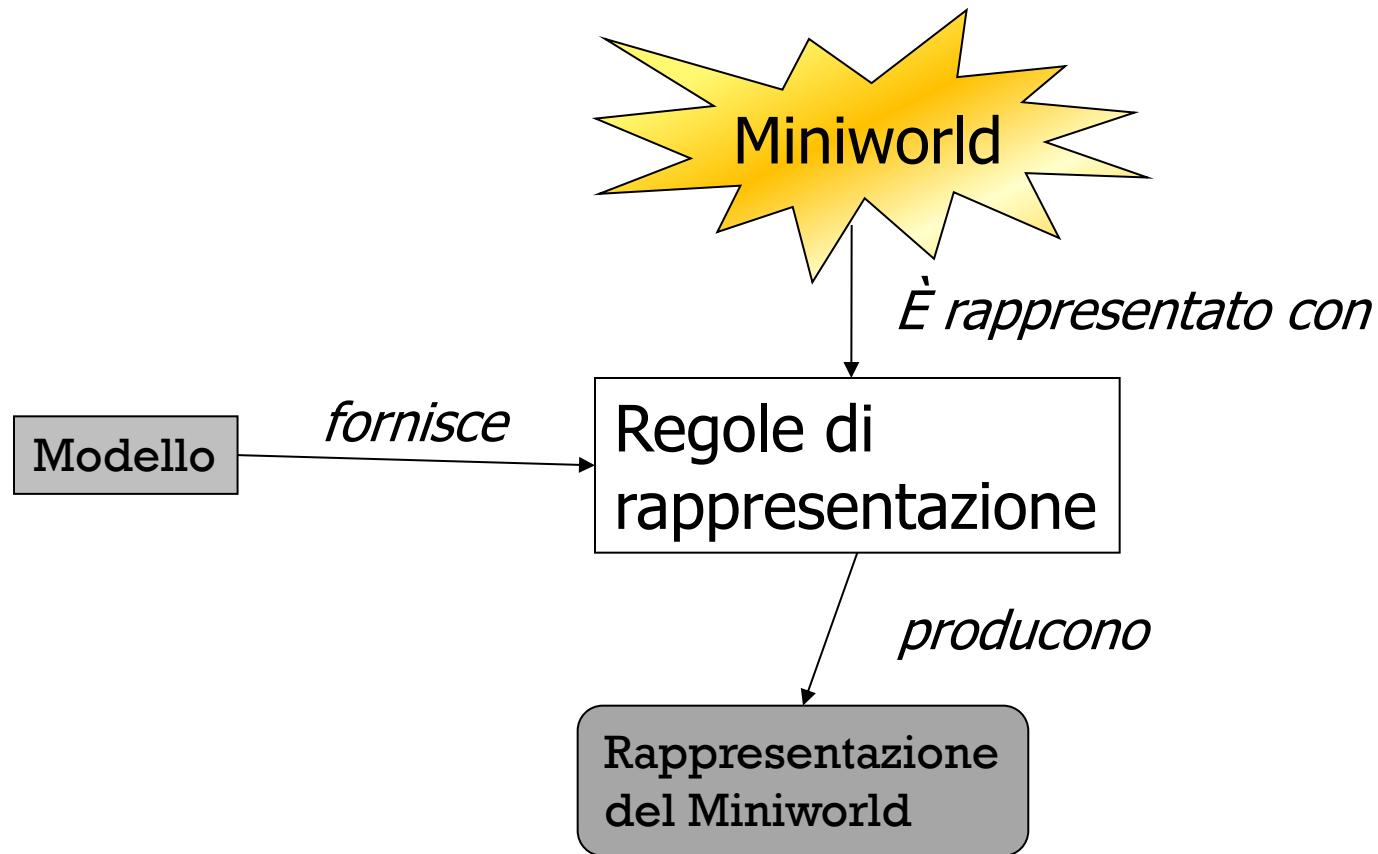
# COSA DEVE FARE UN MODELLO

*Un modello deve:*

- **Rappresentare una certa realtà:**
  - **Es:** Una mappa rappresenta una porzione di territorio.  
È quindi un modello del territorio, che rappresenta alcune caratteristiche, nascondendone altre.
- **Fornire un insieme di strutture simboliche per descrivere la rappresentazione della realtà:**
  - **Es:** Una mappa ha una serie di simboli grafici per rappresentare delle entità reali.

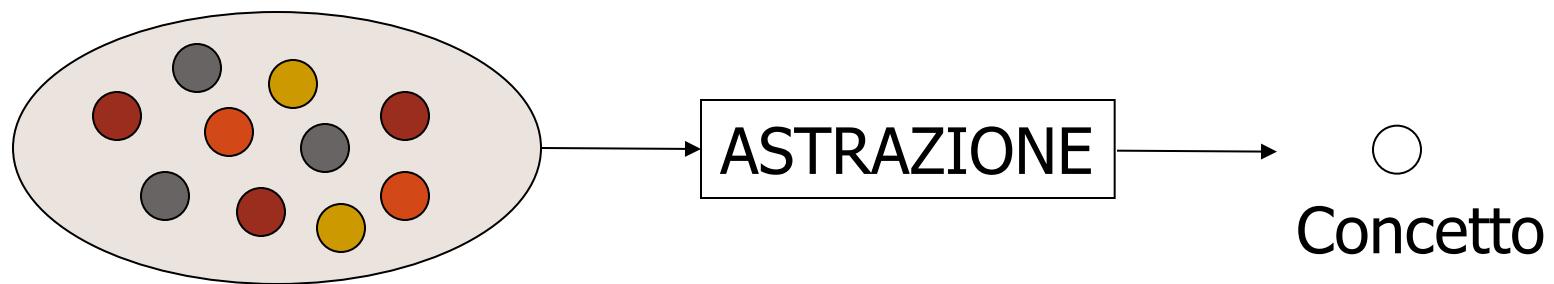


# COS'È UN MODELLO (SCHEMA)



# ASTRAZIONE

- L'astrazione è un procedimento mentale che sostituisce con un concetto un insieme di oggetti in base ad alcune loro proprietà:



## *Esempio:*

l'astrazione consente di definire il concetto di “automobile”. Grazie ad esso è possibile descrivere e riconoscere tutte le automobili della realtà.



# ASTRAZIONE E MODELLI

- L'astrazione è importante perché permette di comunicare ed elaborare una rappresentazione del miniworld.
- Per comunicare ed elaborare tale rappresentazione si utilizzano dei modelli concettuali e logici.
- Un **modello concettuale** fornisce i simbolismi per rappresentare concetti astratti in modo indipendente da qualsiasi elaboratore.
- Un **modello logico** traduce le strutture concettuali in strutture logiche processabili da un DBMS.



# CATEGORIE DI DATA MODEL

*È possibile categorizzare i vari data model proposti secondo i concetti utilizzati per descrivere la struttura del database.*

- I data model di alto livello o concettuali forniscono concetti che sono vicini al modo di percepire i dati degli utenti.
  - Anche noti come **entity-based** o **object-based** data models
- I data model di basso livello o fisici forniscono concetti che descrivono dettagli su come i dati sono memorizzati.
- I data model rappresentazionali o di implementazione forniscono concetti comprensibili agli utenti finali, ma che non sono troppo lontani dal modo in cui i dati sono fisicamente organizzati.
  - Essi nascondono alcuni dettagli della memorizzazione dei dati ma possono essere implementati in maniera diretta.
- I data model auto-descrittivi combinano la descrizione dei dati con i valori dei dati stessi
  - Ad esempio, come si fa con l'XML



# DATA MODEL DI ALTO LIVELLO

- I data model di alto livello usano concetti quali entità, attributi e relazioni:
  - Un'entità rappresenta un oggetto o concetto del mondo reale.
    - *Es:* Un impiegato o un progetto.
  - Un attributo rappresenta qualche proprietà importante che descrive ulteriormente un'entità.
    - *Es:* Il nome o lo stipendio di un impiegato.
  - Una relazione tra due o più entità rappresenta un'interazione tra le entità.
    - *Es:* Una relazione “lavora su” tra un impiegato e un progetto.
- Il più comune data model di alto livello è il modello entità-relazione.

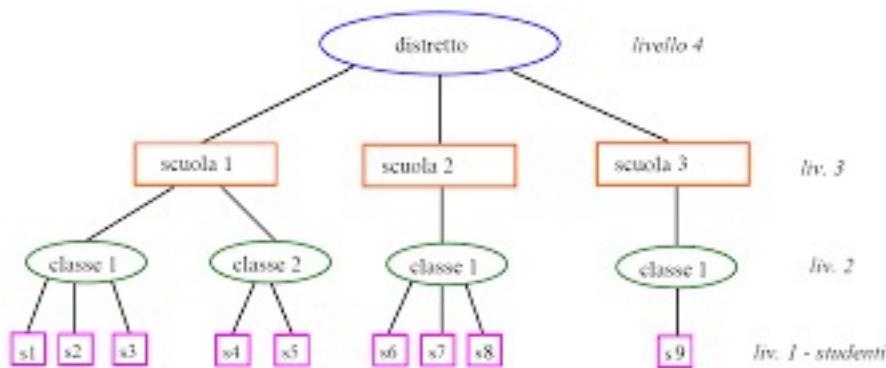


# DATA MODEL RAPPRESENTAZIONALI

- I data model rappresentazionali, per la facilità con cui possono essere implementati, comprendono i tre data model più usati dai DBMS commerciali:
  - Il modello **relazionale** (i dati sono incapsulati in record a struttura fissa. La relazione è rappresentata da una tabella)
  - Il modello **reticolare** (si basa sull'utilizzo di grafi e si basano sui concetti di record e set. Utilizzato per limitare la ridondanza)
  - Il modello **gerarchico** (utilizza strutture ad albero con i record come nodi e le associazioni come archi. Ha problemi per quanto riguarda la ridondanza dei dati)
- Tali modelli rappresentano i dati usando strutture di record. Per tale motivo sono chiamati anche modelli record-based.
- I modelli **object-oriented**, grazie alla forte astrazione dei dati, costituiscono una nuova famiglia di data model, posti a metà strada tra il rappresentazionale e il concettuale.



# ESEMPI DI DATA MODEL RAPPRESENTAZIONALI



gerarchico

reticolare

CLIENTI			
Bianchi	Mazzini	Bergamo	24127
Neri	Garibaldi	Napoli	80120
Rossi	Rosmini	Milano	20125
Gialli	Cavour	Bari	70122
Verdi	Dante	Roma	00128
Rosa	Crispi	Torino	10137
Moro	Colombo	Milano	20143

CONTI		
9000	120345	
7000	500	
4500	3500	
5100	58500	
8000	6320	
4310	37	
...		
21/08/2010	Prel	350,00
21/08/2010	Prel	536,00

MOVIMENTI		
23/12/2009	Vers	2.500,00
20/04/2010	Prel	1.250,00
27/05/2010	Vers	550,00

relazionale

Activity Code	Activity Name
23	Patching
24	Overlay
25	Crack Sealing

Activity Code	Date	Route No.
24	01/12/01	I-95
24	02/08/01	I-66

Date	Activity Code	Route No.
01/12/01	24	I-95
01/15/01	23	I-495
02/08/01	24	I-66



# DATA MODEL FISICI

- I data model fisici descrivono come sono memorizzati i dati nel calcolatore rappresentando informazioni quali formati di record, ordinamenti di record, percorsi di accesso.
  - Un percorso di accesso è una struttura che rende efficiente la ricerca di particolari record di database.



# SCHEMI E ISTANZE DI DATABASE

- In qualsiasi data model è necessario distinguere tra la descrizione del database ed il database stesso.
- La differenza è simile a quella tra tipi e variabili nei linguaggi di programmazione.
- Lo schema è la struttura logica del database.
- Un'istanza è il contenuto del database in un particolare istante di tempo.



# SCHEMI DI DATABASE

- La descrizione del database è detta schema (o metadati).
- Lo schema del database è specificato in fase di progetto e non ci si aspetta che cambi frequentemente.
  - Per rappresentare uno schema si crea un diagramma di schema, secondo opportune convenzioni definite dal data model.
  - Un diagramma di schema visualizza la struttura dei record ma non le reali istanze dei record.
- Il DBMS memorizza lo schema nel catalogo per poterne fare riferimento ogni qualvolta gli occorre.



# **SCHEMI DI DATABASE:**

## ***ESEMPIO DB UNIVERSITÀ***

STUDENTE		
NOME	MATRICOLA	ANNO
CORSO		
DENOMIN.	SEMESTRE	TITOLARE
PREREQUISITI		
DENOMIN.	PROPEDEUTICITA'	
VOTAZIONE		
NOME	DENOMIN.	VOTO

- Ogni oggetto nello schema è detto costrutto di schema.
  - *Es:* STUDENTE o CORSO sono ciascuno un costrutto di schema.



# ISTANZE DI DATABASE

- I dati reali in un db possono cambiare frequentemente.
  - *Es:* Il db UNIVERSITA' cambia ogni volta che si inserisce un nuovo studente o si registra un nuovo esame per uno studente.
- Uno stato del database (detto anche istanza o insieme di occorrenze del db) è l'insieme dei dati presenti nel database in un particolare istante di tempo.



# ISTANZE DI DATABASE (2)

- Dato un stato del database, ogni costrutto di schema ha un proprio insieme corrente di istanze
  - *Es:* Il costrutto STUDENTE conterrà l'insieme di entità (o record) studenti individuali come sue istanze.
- Per un dato schema di database possono essere costruiti diversi stati di database.
- Ogni volta che si inserisce o cancella un record o si modifica il valore di un data item, si cambia lo stato del database.



# L'IMPORTANZA DELLO SCHEMA DI DATABASE

- Quando si crea un nuovo database, si specifica al DBMS lo schema del database:
  - in tale fase il db è nello “*stato vuoto*”, senza dati. Si passa nello “*stato iniziale*”, quando si inseriscono i dati per la prima volta.
- Il DBMS è parzialmente responsabile nel garantire che ogni stato del DB sia valido, cioè che soddisfi la struttura ed i vincoli specificati nello schema.
- Quindi:
  - specificare uno schema corretto è estremamente importante; lo schema deve essere progettato con la massima cura.



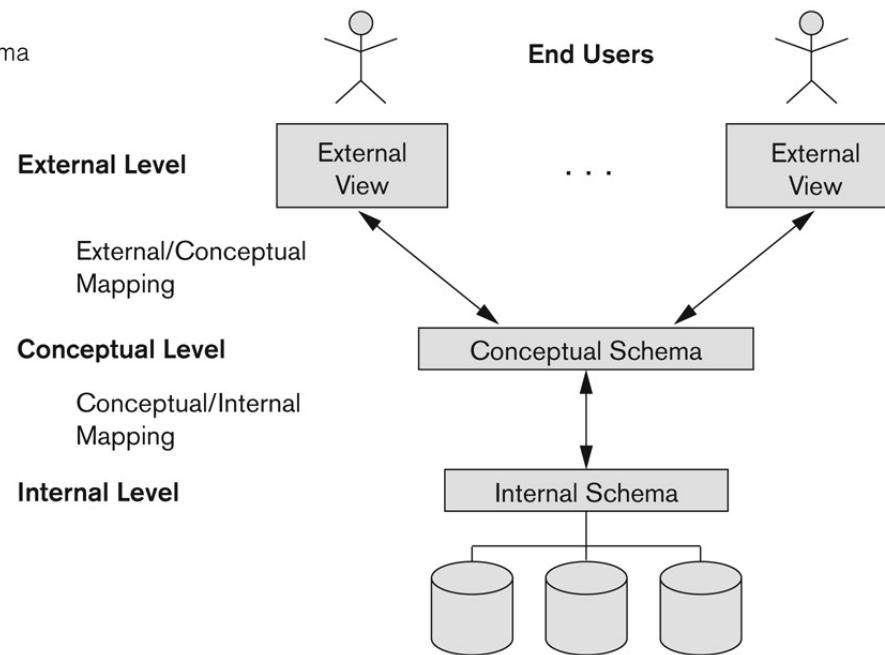
# **ARCHITETTURA DEI R-DBMS**



# L'ARCHITETTURA THREE-SCHEMA (2)

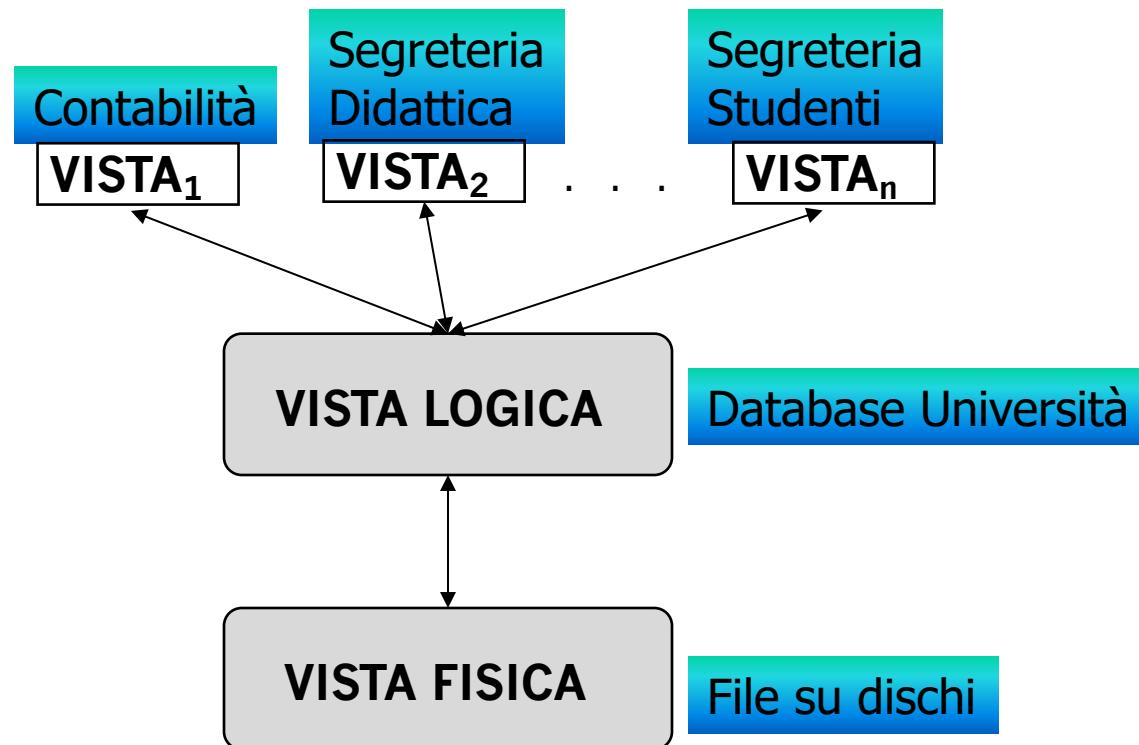
**Figure 2.2**

The three-schema architecture.



- Ricordiamo le caratteristiche principali dell'approccio di database:
  - supporto di viste multiple per gli utenti;
  - uso di un catalogo per memorizzare la descrizione del DB (lo schema);
  - condivisione dei dati ed elaborazione delle transazioni in modalità multiutente.
- L'architettura per database system chiamata **three-schema** aiuta ad ottenere tali caratteristiche, definendo il database in tre livelli e separando quindi le applicazioni utente dal database fisico.

# L'ARCHITETTURA THREE-SCHEMA: UN'ESEMPIO



Gli schemi possono essere specificati a tre livelli:

- **Interno (Internal Schema)**
  - Per descrivere le strutture di storing fisiche (*physical data model*)
- **Concettuale (Conceptual Schema)**
  - Descrive la struttura e i vincoli del database
- **Esterno (External Schema)**
  - Descrive le varie viste per gli utenti



# LIVELLO INTERNO

*Livello interno come schema interno:*

- Descrive la struttura di memorizzazione fisica del DB.
- Lo schema interno usa un data model fisico e descrive i dettagli completi del data storage e gli access paths del DB.
- **Esempio:**  
Dividi i record del file studenti in tre partizioni sui dischi 5, 6 e 7.



# LIVELLO CONCETTUALE

*Livello concettuale come schema concettuale:*

- Descrive la struttura del DB per una comunità di utenti.
- Lo schema concettuale nasconde i dettagli delle strutture di storage fisico e si concentra sulla descrizione delle entità, tipi di dati, relazioni, operazioni utente e vincoli.
- Può usare un data model ad alto livello o uno di implementazione.
- **Esempio:**  
**type** Studente= **record**  
    **nome** : **string**;  
    **matricola** : **string**;  
    **anno** : **string**;  
**end**;



# LIVELLO ESTERNO

*Livello esterno come schema esterno:*

- Definisce un sottoinsieme del DB per una particolare applicazione.
  - Include più schemi esterni o viste utente.
  - Usa un data model di alto livello o di implementazione.
  - Ogni schema esterno descrive la parte del DB cui è interessato un particolare utente e nasconde il resto del DB a quel gruppo.
- 
- **Esempio:**  
La segreteria didattica ha necessità di vedere tutte le informazioni sugli esami, ma non deve vedere informazioni sugli stipendi.

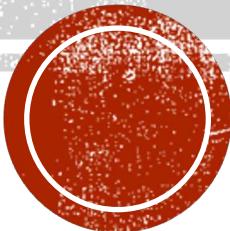


# I MAPPING TRA I LIVELLI DELL'ARCHITETTURA

- I tre schemi sono solo delle descrizioni di dati: gli unici dati che realmente esistono sono a livello fisico.
- In un DBMS basato sull'architettura three-schema, ogni gruppo di utenti utilizza una propria vista esterna.
- Un mapping è un processo di trasformazione delle richieste e dei risultati. Il DBMS trasforma:
  - una richiesta specificata su uno schema esterno...
  - ...in una richiesta secondo schema concettuale e poi...
  - ...in una richiesta sullo schema interno per procedere sul database memorizzato.



# **INDIPENDENZA LOGICA E FISICA DEI DATI**



# INDIPENDENZA DEI DATI

L'architettura three-schema è utile per evidenziare il concetto di indipendenza dei dati, ovvero la capacità di cambiare lo schema a un livello del database senza dover cambiare lo schema al livello superiore.

Si definiscono due tipi di indipendenza dei dati:

- ***Indipendenza logica dei dati:***

- lo schema concettuale può essere cambiato senza dover cambiare gli schemi esterni o i programmi applicativi.

- ***Indipendenza fisica dei dati:***

- lo schema interno può essere cambiato senza dover cambiare gli schemi concettuali (o esterni).



# INDIPENDENZA LOGICA

- Indica la capacità di cambiare lo schema concettuale senza dover cambiare lo schema esterno e gli applicativi correlati
- Lo schema concettuale può essere cambiato per espandere oppure per ridurre il database (aggiungendo o rimuovendo, rispettivamente, un tipo di record o data item).
- Se si elimina un tipo di record, gli schemi esterni che si riferiscono solo ai dati restanti non devono essere alterati.
  - Se uno schema ad un livello più basso viene modificato, soltanto il **mapping tra questo schema e quelli di livello più alto necessitano di essere cambiati**;
  - i programmi applicativi che fanno riferimento agli schemi esterni sono indifferenti alle modifiche, siccome si riferiscono solo agli schemi esterni.



# INDIPENDENZA LOGICA: ESEMPIO

Lo schema esterno

ESAMI SUPERATI	Nome	Matricola	ESAMI	
			DENOMIN.	VOTO
	Rossi	N86000484	Basi di Dati	24
			Object Orientation	28
	Verdi	N86000323	Algoritmi	30
			Sistemi Biometrici	27

non dovrebbe essere alterato cambiando il file votazione da

VOTAZIONE		
NOME	DENOMIN.	VOTO
Rossi	Basi di Dati	24
Rossi	Object Orientation	28
Verdi	Algoritmi	30
...	...	...

a

VOTAZIONE			
NOME	MATRICOLA	DENOMIN.	VOTO
Rossi	N86000484	Basi di Dati	24
Rossi	N86000484	Object Orientation	28
Verdi	N86000323	Algoritmi	30
...	...	...	...



# INDIPENDENZA FISICA

- Indica la capacità di cambiare lo schema interno senza dover cambiare lo schema concettuale
- Un cambiamento dello schema interno può essere dovuto alla riorganizzazione di qualche file fisico (**es:** creando ulteriori strutture di accesso, o modificando degli indici), per migliorare l'esecuzione del ritrovamento o dell'aggiornamento.
  - Se i dati non subiscono alterazioni, non è necessario cambiare lo schema concettuale.
- L'indipendenza fisica si ottiene più facilmente di quella logica.



# INDIPENDENZA FISICA: *ESEMPIO*

- Fornire un percorso di accesso per migliorare il ritrovamento dei record del file **CORSO** con **Denominaz.** e **Semestre** non dovrebbe richiedere variazioni a una query come

*“ritrova tutti i corsi tenuti nel secondo semestre”*

sebbene la query possa essere eseguita in maniera più efficiente.



# L'ARCHITETTURA THREE-SCHEMA: VANTAGGI E SVANTAGGI

## VANTAGGI

- L'architettura three-schema consente facilmente di ottenere una reale indipendenza dei dati.
- Il database risulta più flessibile e scalabile.

## SVANTAGGI

- Il catalogo del DBMS multilivello deve avere dimensioni maggiori, per includere informazioni su come trasformare le richieste e di dati tra i vari livelli.
- I due livelli di mapping creano un overhead durante la compilazione o esecuzione di una query di un programma, causando inefficienze nel DBMS.



# **DBMS LANGUAGES**



# LINGUAGGI DBMS

- *Un DBMS deve fornire ad ogni categoria di utenti delle interfacce e dei linguaggi adeguati.*
- Alla fase di progetto del DB segue quella di specifica degli schemi concettuale ed interno e di qualsiasi mapping tra i due.
- Nei DBMS dove non c'è una netta separazione tra livelli, progettisti e DBA usano un **data definition language (DDL)** per definire entrambi gli schemi.
- Il compilatore del DDL (contenuto nel DBMS) ha la funzione di:
  - Trattare gli statement DDL per identificare le descrizioni dei costrutti dello schema.
  - Memorizzare la descrizione dello schema nel catalogo del DBMS.



# LINGUAGGI DBMS (2)

- Dove c'è una chiara distinzione tra livello concettuale ed interno si usa:
  - uno **storage definition language (SDL)** per specificare lo schema interno,
  - il **DDL** per specificare soltanto lo schema concettuale.
- I mapping tra i due schemi possono essere specificati in uno qualsiasi dei due linguaggi.
- Nelle architetture three-schema viene usato un **view definition language (VDL)** per specificare le viste utente e i loro mapping sullo schema concettuale.



# LINGUAGGI DBMS (3)

- Una volta che gli schemi sono compilati ed il DB è riempito di dati:
  - il DBMS fornisce un **data manipulation language (DML)** per consentire agli utenti di manipolare (cioè recuperare, inserire, cancellare e aggiornare) dati.
- Nei DBMS moderni si tende ad utilizzare un unico linguaggio integrato che comprende costrutti:
  - per la definizione di schemi concettuali;
  - per la definizione di viste;
  - per la manipolazione dei DB e
  - per la definizione della memorizzazione.
- ***Esempio:***
  - Il linguaggio di database relazionali **structured query language (SQL)** rappresenta una combinazione di DDL, VDL, DML e SDL.



# I DML DI ALTO LIVELLO

- Consentono da soli di specificare operazioni di database complesse in maniera concisa.
- In molti DBMS gli statement di DML di alto livello
  - Possono essere specificati interattivamente da terminale.
  - Possono essere inglobati (*embedded*) in un linguaggio di programmazione general-purpose (in questo caso gli statement DML sono identificati all'interno del programma in modo che il DBMS possa trattarli (*precompilazione*)).
  - Sono detti **set-at-a-time** o **set-oriented** perché possono specificare e ritrovare molti record in singolo statement DML (es: l'SQL).
  - Sono detti dichiarativi perché una query di un DML high-level spesso specifica **quale** dato deve essere ritrovato piuttosto che **come** ritrovarlo.



# I DML DI BASSO LIVELLO

- Deve essere inglobato in un linguaggio general-purpose;
- È detto **record-at-a-time** perché tipicamente ritrova record individuali nei DB e tratta ciascun record separatamente;
  - ha quindi bisogno di usare costrutti di programmazione come l'iterazione per ritrovare e trattare ogni record da un insieme di record.



# SQL, UN LINGUAGGIO INTERATTIVO

- “*Trovare i corsi tenuti in aule a piano terra*”

**Corsi**

Corso	Docente	Aula
Basi di dati	Rossi	DS3
<b>Sistemi</b>	Neri	<b>N3</b>
<b>Reti</b>	Bruni	<b>N3</b>
Controlli	Bruni	G

**Aule**

Nome	Edificio	Piano
<b>DS1</b>	OMI	<b>Terra</b>
<b>N3</b>	OMI	<b>Terra</b>
G	Pincherle	Primo



# SQL, UN LINGUAGGIO INTERATTIVO (2)

**SELECT** Corso, Aula, Piano

**FROM** Aule, Corsi

**WHERE** Nome = Aula **AND** Piano = “*Terra*”

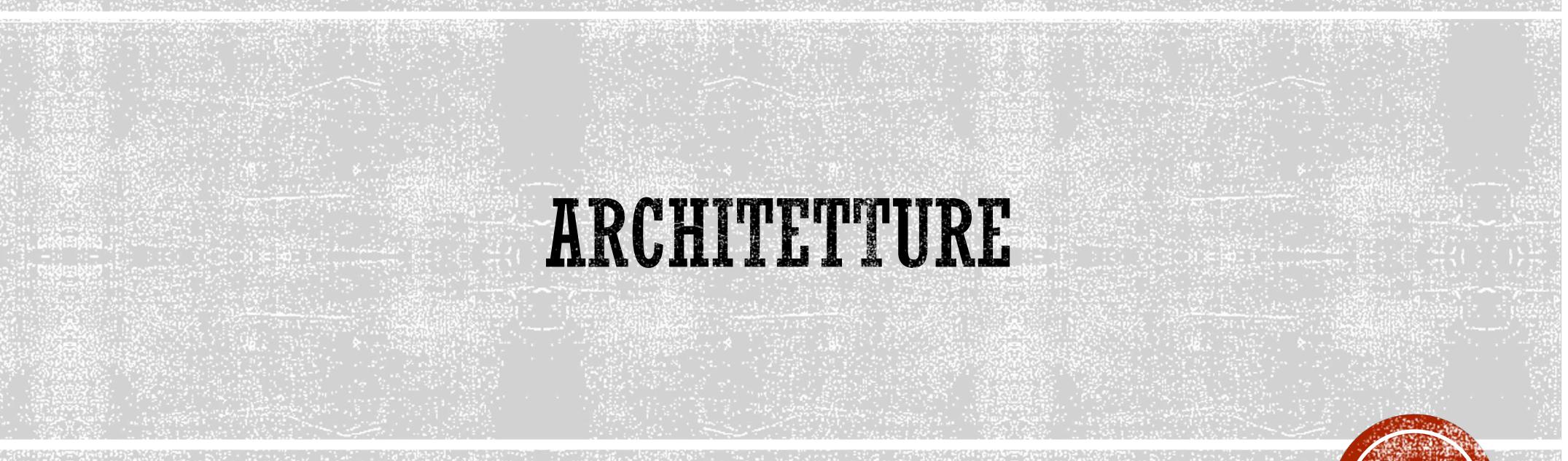
Corso	Aula	Piano
Sistemi	N3	Terra
Reti	N3	Terra



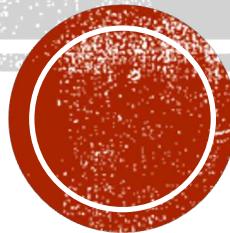
# SQL IMMERSO (EMBEDDED) IN LINGUAGGIO OSPITE

```
write('nome della citta?'); readln(citta);
EXEC SQL DECLARE P CURSOR FOR
    SELECT NOME, REDDITO
    FROM PERSONE
    WHERE CITTA = :citta ;
EXEC SQL OPEN P ;
EXEC SQL FETCH P INTO :nome, :reddito ;
while SQLCODE = 0 do
    begin
        write('nome della persona:', nome, 'aumento?'); readln(aumento);
        EXEC SQL UPDATE PERSONE
            SET REDDITO = REDDITO + :aumento
            WHERE CURRENT OF P ;
        EXEC SQL FETCH P INTO :nome, :reddito ;
    end;
EXEC SQL CLOSE CURSOR P ;
```



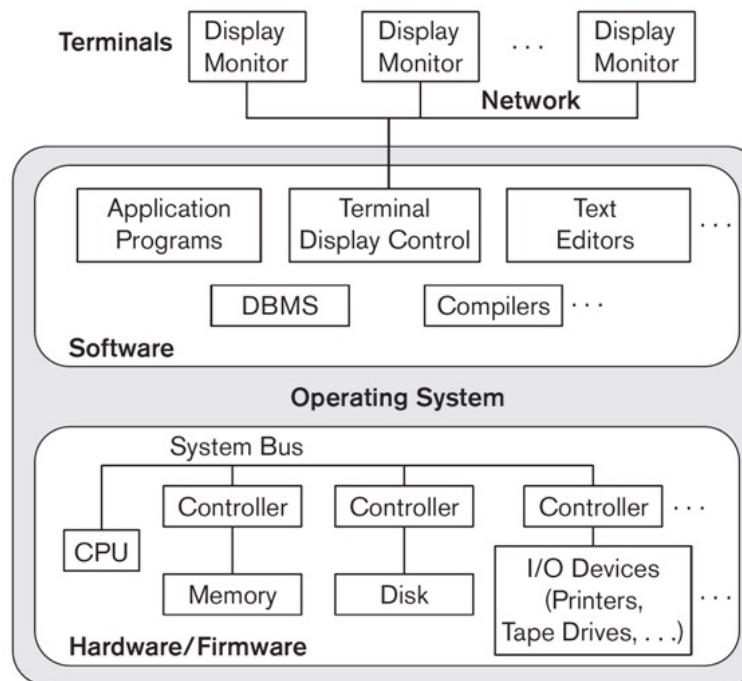


# ARCHITETTURE



# ARCHITETTURA DEL DBMS CENTRALIZZATO

- Tutte le funzionalità del DBMS, l'esecuzione del programma applicativo, e l'elaborazione dell'interfaccia utente sono eseguite su una sola macchina.



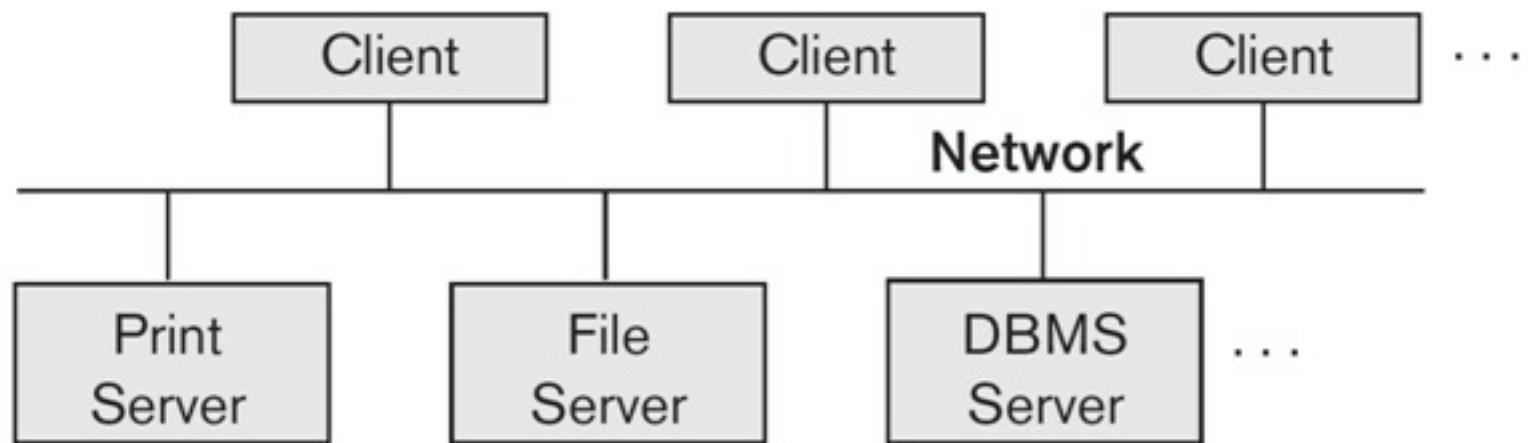
# ARCHITETTURA CLIENT/SERVER BASE

- Sono **Server** con specifiche funzionalità:
  - **File server**
    - Mantiene e gestisce i file delle macchine client.
  - **Printer server**
    - È connesso a diverse stampanti,
    - Tutte le richieste di stampa dalle macchine client sono trasmesse a questa macchina.
  - **Web server o e-mail server**
- **Le macchine Client:**
  - Forniscono l'utente di un interfaccia appropriata per utilizzare i server.
  - Hanno capacità operazionali locali per eseguire applicazioni in locale.



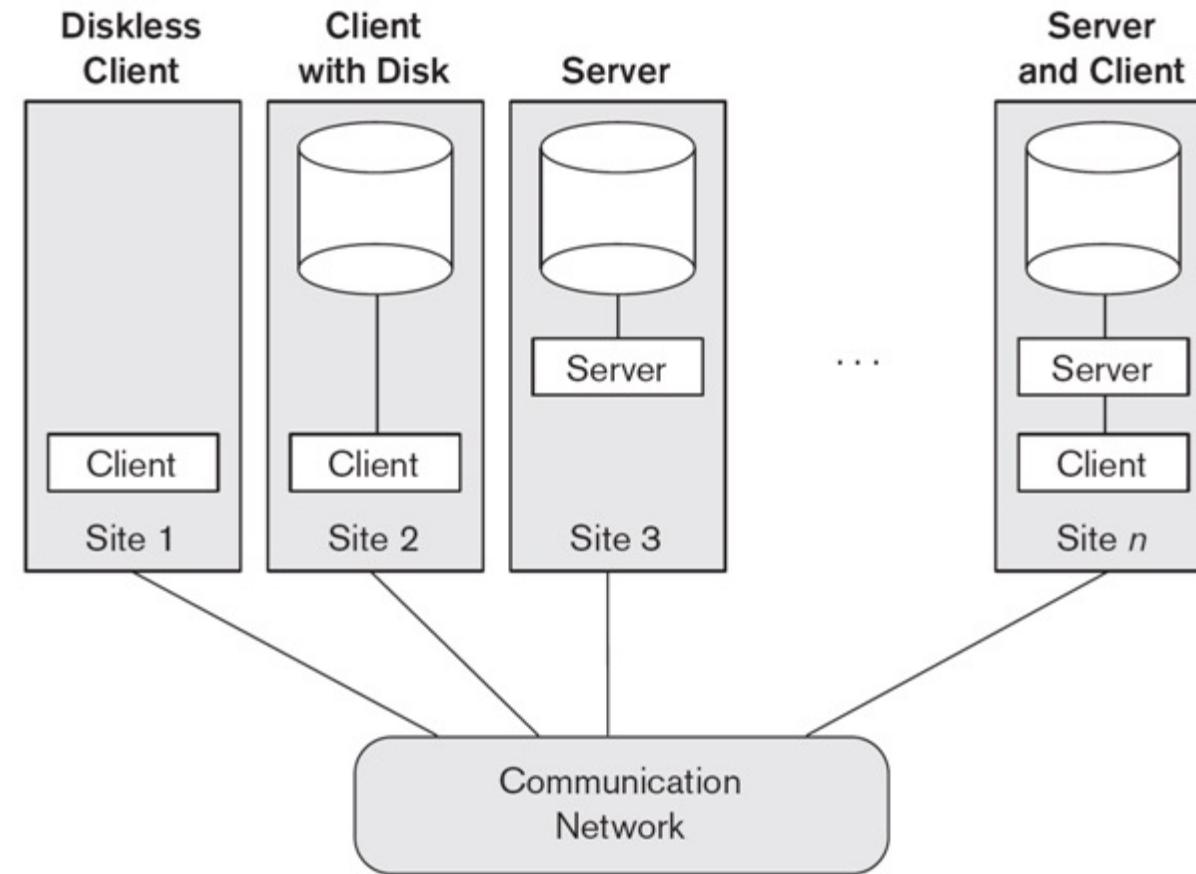
# ARCHITETTURA CLIENT/SERVER TWO-TIER

- Architettura logica:



# ARCHITETTURA CLIENT/SERVER TWO-TIER (2)

- Architettura fisica:



# ARCHITETTURA CLIENT/SERVER TWO-TIER (3)

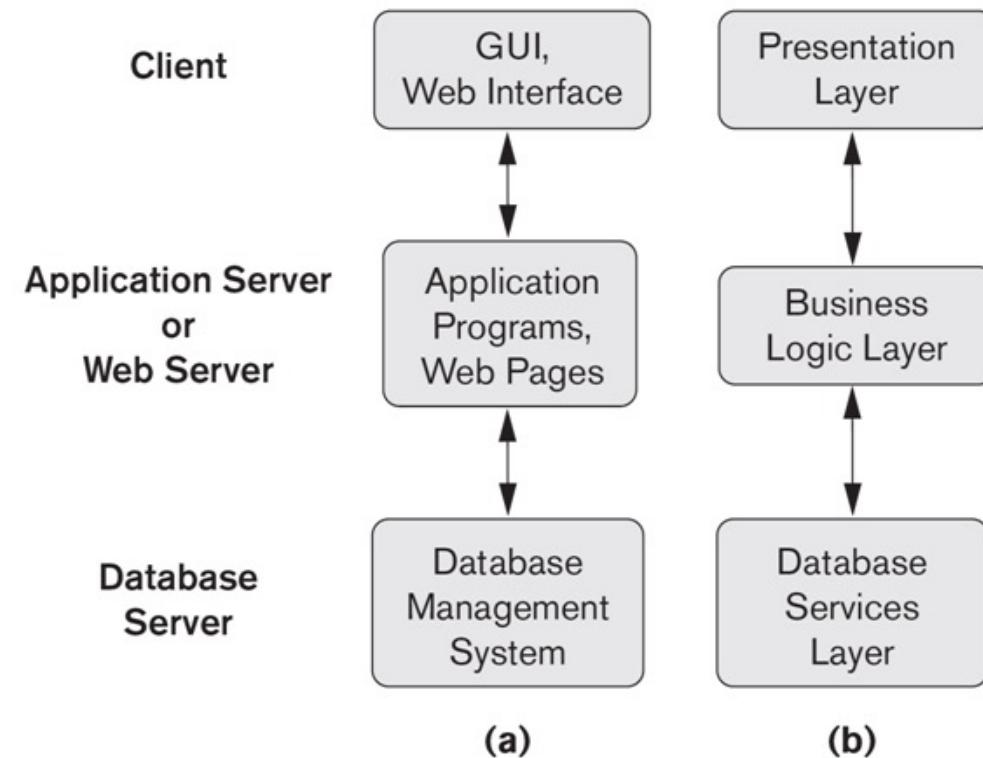
- Open Database Connectivity (ODBC)
  - Fornisce le Application Programming Interface (API).
  - Permette ad un programma in esecuzione sulla macchina client di comunicare con il DBMS
    - Entrambi le macchine client e server devono avere installato il software necessario.
- Java DataBase Connectivity (JDBC)
  - Permette ai programmi client scritti in **Java** di accedere ad uno o più DBMS attraverso un interfaccia standard.



# ARCHITETTURA CLIENT/SERVER THREE-TIER

- **Application server o Web server**

- Aggiunge uno strato intermedio tra il client ed il database server.
- Esegue i programmi applicativi e memorizza le regole di business.



(a)

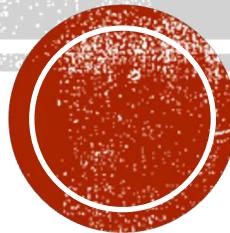
(b)



# FINE

Per eventuali domande: (in ordine di preferenza personale)

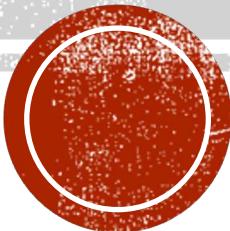
- Ora.
- Chat di Teams
- Mail: [silvio.barra@unina.it](mailto:silvio.barra@unina.it)



# BASI DI DATI I

- Progettazione di una Base di Dati
- Modellazione del DB: ER vs UML
- Entità e Attributi (UML vs ER)
- Esempio: Il database *Company*
- Le Relazioni (ER) e le Associazioni (UML)
- Esempi

# **PROGETTAZIONE DI UNA BASE DI DATI**



# PROGETTAZIONE DI BASI DI DATI

- È una delle attività del processo di sviluppo dei sistemi informativi.
- Va quindi inquadrata in un contesto più generale:
  - il ciclo di vita dei sistemi informativi.



# SISTEMA INFORMATIVO

- **Componente (sottosistema)**

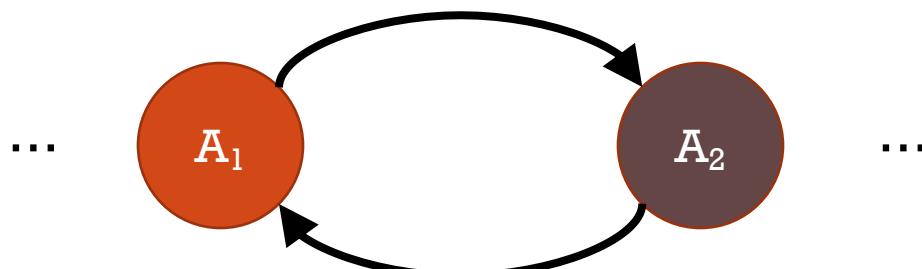
**di una organizzazione che gestisce** (acquisisce, elabora, conserva, produce)

**le informazioni di interesse** (cioè utilizzate per il perseguitamento degli scopi dell'organizzazione).



# IL CICLO DI VITA DEI SISTEMI INFORMATIVI

- Insieme e sequenzializzazione delle attività svolte da analisti, progettisti, utenti, nello sviluppo e nell'uso dei sistemi informativi.
- È un attività iterativa, quindi è rappresentata attraverso un **ciclo**.



# IL CICLO DI VITA DEI SISTEMI INFORMATIVI



- Il ciclo di vita è una attività iterativa, rappresentata tramite un ciclo



# FASI (TECNICHE) DEL CICLO DI VITA

## ■ Studio di fattibilità

- Si analizzano le potenziali aree di applicazione, si effettuano degli studi di *costi/benefici*, si determina la complessità di dati e processi, e si impostano le priorità tra le applicazioni.

## ■ Raccolta e analisi dei requisiti

- Comprende una raccolta dettagliata dei requisiti con interviste ai potenziali utenti, per definire le funzionalità del sistema.

## ■ Progettazione di dati e funzioni

## ■ Realizzazione

- Si implementa il sistema informativo, si carica il DB e si implementano e si testano le transazioni.

## ■ Validazione e collaudo

- Si verifica che il sistema soddisfi i requisiti e le performance richieste.

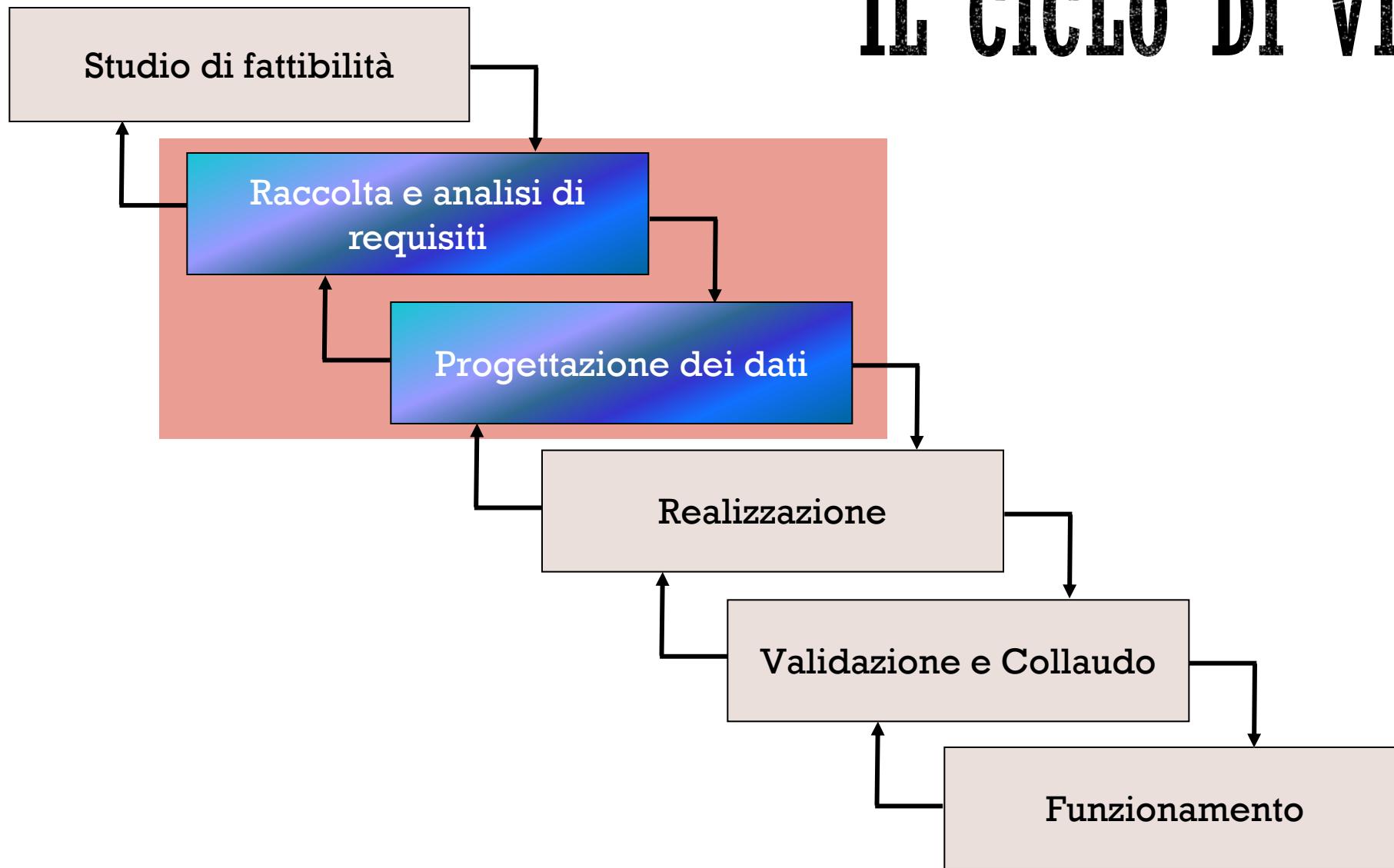
## ■ Funzionamento

- La fase operativa del nuovo sistema parte quando tutte le funzionalità sono state validate.

Il rilascio può essere preceduto da una fase di addestramento del personale al nuovo sistema.

Se emergono nuovi funzionalità da implementare, si ripetono i passi precedenti, per includerle nel sistema (*manutenzione*).

# IL CICLO DI VITA



# LA METODOLOGIA DI PROGETTO

- Per garantire prodotti di buona qualità è opportuno seguire una
  - metodologia di progetto, con:
    - articolazione delle attività in fasi indipendenti tra loro;
    - strategie da seguire nei vari passi e criteri di scelta (alternative),
    - modelli di rappresentazione per descrivere i dati in ingresso e uscita delle varie fasi.
  - proprietà:
    - generalità,
    - qualità del prodotto in termini di correttezza, completezza ed efficienza rispetto alle risorse impiegate,
    - facilità d'uso delle strategie e dei modelli.



# **MODELLO DEI DATI**

- I prodotti della varie fasi sono schemi di alcuni modelli di dati:
  - Schema concettuale
  - Schema logico
  - Schema fisico



# MODELLO DEI DATI

- È un insieme di costrutti utilizzati per organizzare i dati di interesse e descriverne la dinamica.
- Componente fondamentale: meccanismi di strutturazione (o costruttori di tipo).
  - Come nei linguaggi di programmazione esistono meccanismi che permettono di definire nuovi tipi, così ogni modello dei dati prevede alcuni costruttori .
- **Esempio:** il modello relazionale prevede il costruttore relazione, che permette di definire insiemi di record omogenei.



# SCHEMI E ISTANZE

- In ogni base di dati esistono:
  - lo **schema**, sostanzialmente invariante nel tempo, che ne descrive la struttura (*aspetto intensionale*):
    - **Es:** nel modello relazionale, le intestazioni delle tabelle.
  - l'**istanza**, i valori attuali, che possono cambiare anche molto rapidamente (*aspetto estensionale*):
    - **Es:** nel modello relazionale, il “corpo” di ciascuna tabella.



# DUE TIPI (PRINCIPALI) DI MODELLI

- **Modelli concettuali:** permettono di rappresentare i dati in modo indipendente da ogni sistema:
  - cercano di descrivere i concetti del mondo reale,
  - sono utilizzati nelle fasi preliminari di progettazione.
- **Modelli logici:** utilizzati nei DBMS esistenti per l'organizzazione dei dati:
  - utilizzati dai programmi,
  - indipendenti dalle strutture fisiche.

**Esempi:** **relazionale**, reticolare, gerarchico, a oggetti.



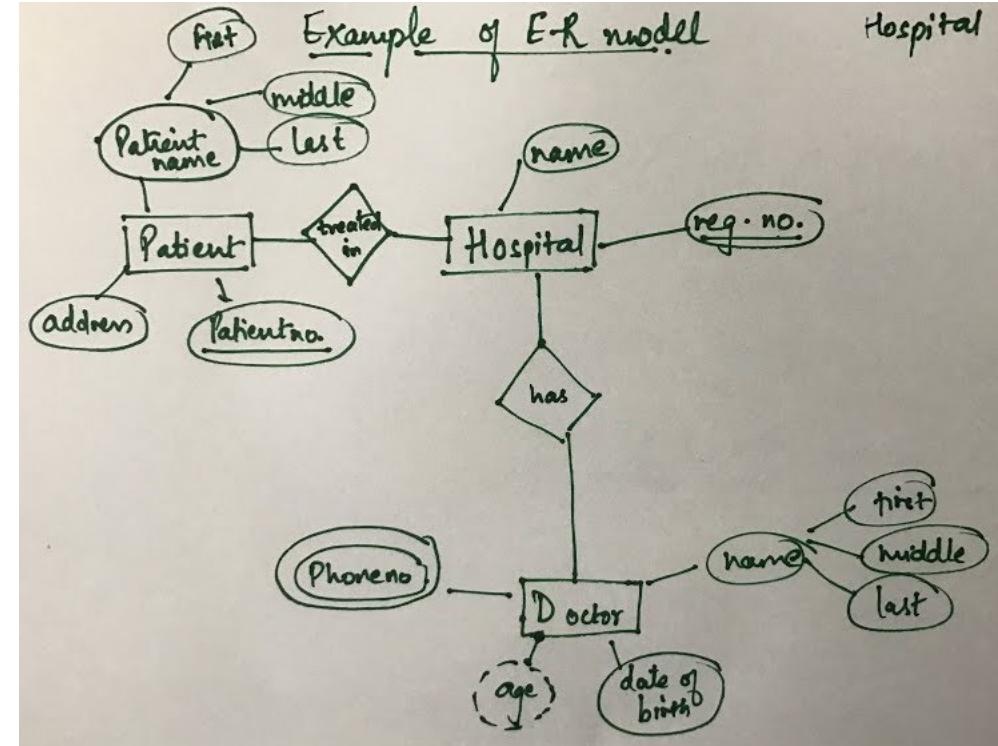
# **MODELLI CONCETTUALI, *PERCHÉ?***

- Proviamo a modellare una applicazione definendo direttamente lo schema logico della base di dati:
  - da dove cominciamo?
  - rischiamo di perderci subito nei dettagli;
  - dobbiamo pensare subito a come correlare le varie tabelle (chiavi, relazioni, etc.);
  - i modelli logici sono rigidi.



# MODELLI CONCETTUALI, PERCHÉ? (2)

- Servono per ragionare sulla realtà di interesse, indipendentemente dagli aspetti realizzativi.
- Permettono di rappresentare le classi di dati di interesse e le loro correlazioni.



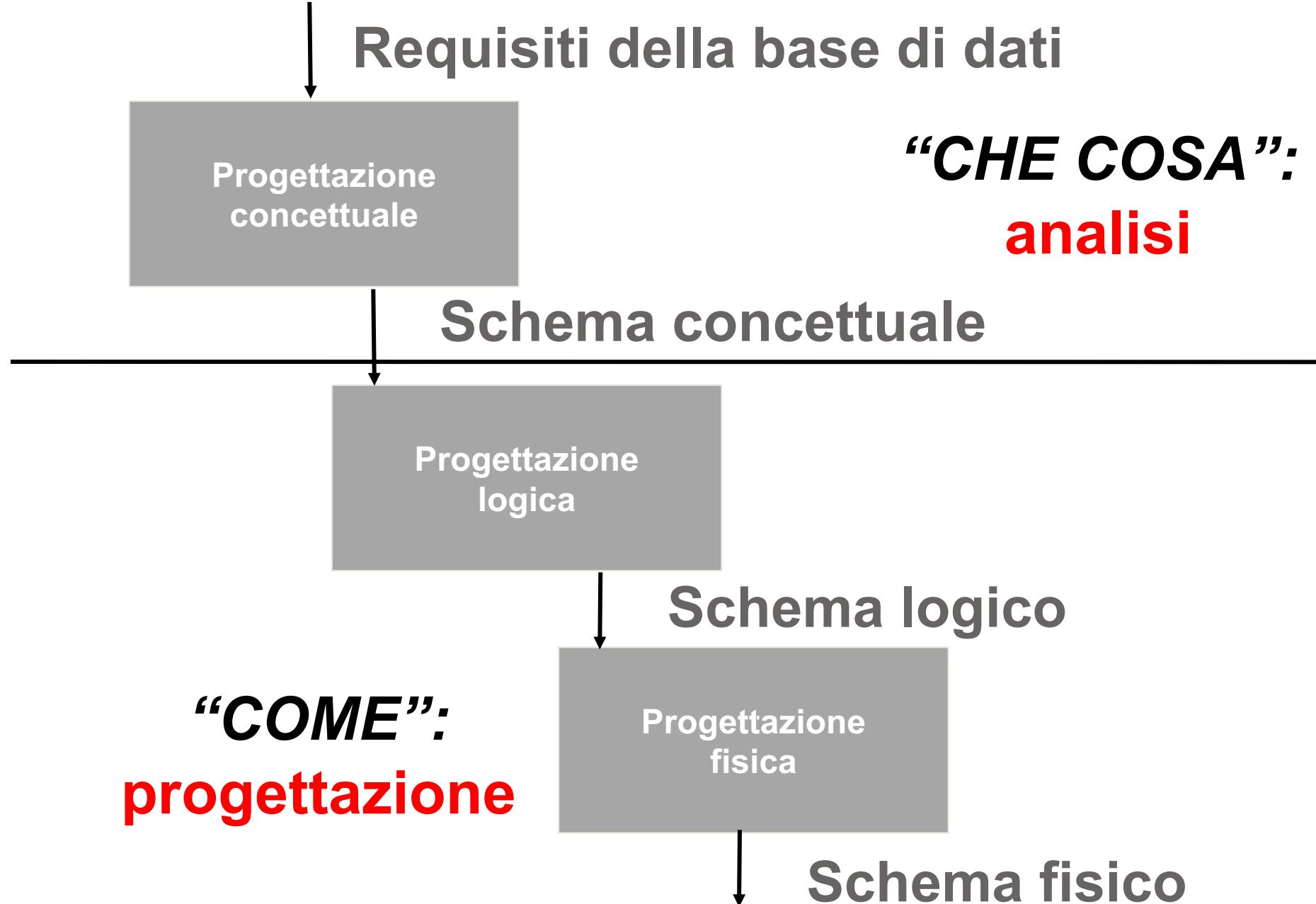
- Prevedono efficaci rappresentazioni grafiche (utili anche per documentazione e comunicazione).

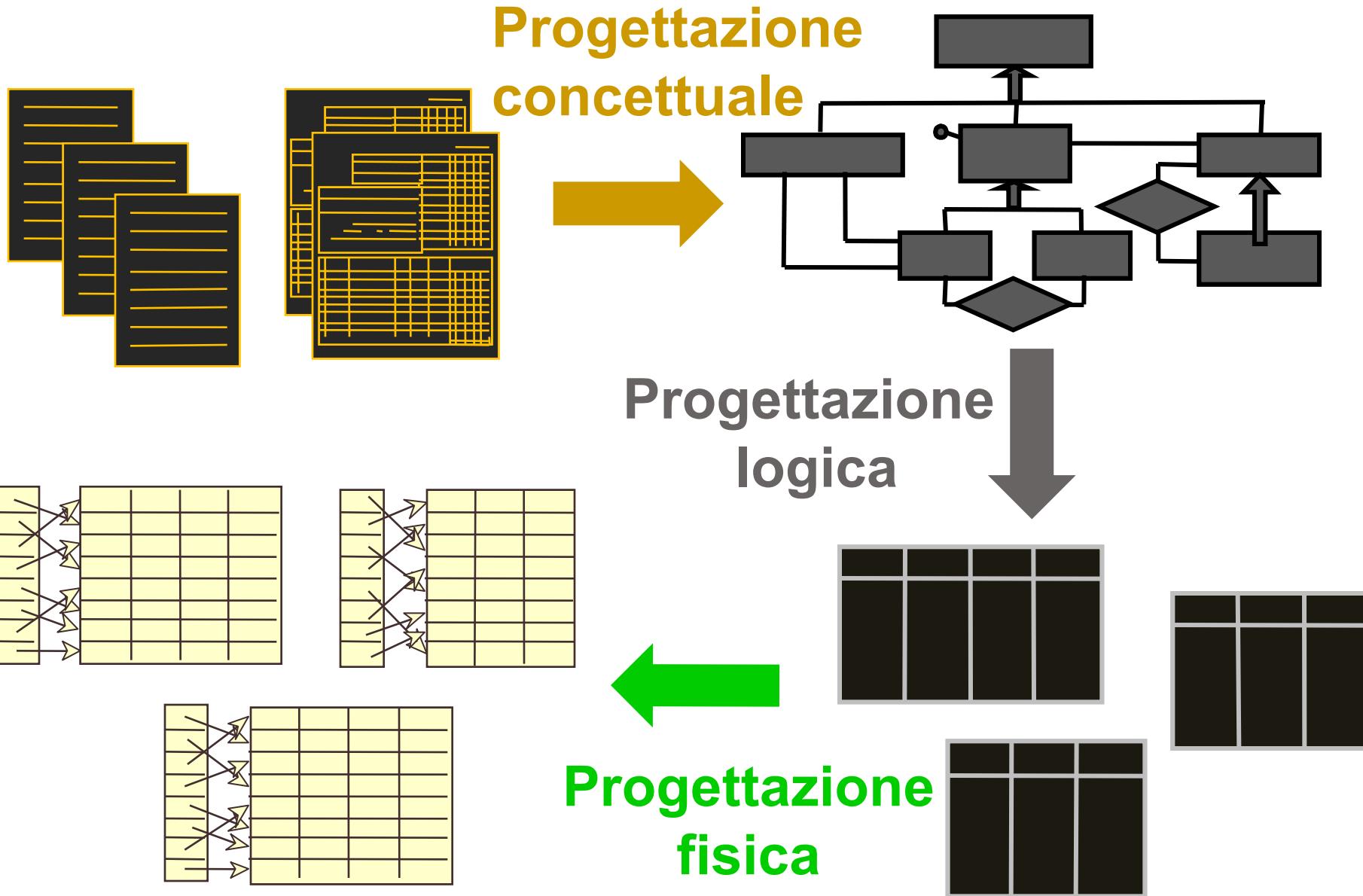
# **IL PROCESSO DI PROGETTAZIONE DEL DATABASE**

**Il processo consta di quattro fasi:**

- 1. Raccolta e analisi dei requisiti**
- 2. Disegno del database concettuale (Progettazione Concettuale)**
- 3. Disegno del database logico (Progettazione Logica)**
- 4. Disegno del database fisico (Progettazione Fisica)**







# PROGETTAZIONE CONCETTUALE

- Individuare l'informazione rilevante da memorizzare per soddisfare le richieste informative e funzionali del caso applicativo
- Interesse specifico per la parte statica del problema: i dati
- Individuazione nel problema degli aspetti informativi rilevanti:
  - Gli elementi informativi di base (Entità)
  - Il modo in cui sono interrelati gli elementi informativi di base (Associazioni)
  - I vincoli che i dati memorizzati nel database dovrebbero soddisfare per garantire la qualità e la coerenza dell'informazione
  - Le più importanti forme di interazione che gli utenti possono richiedere al database (ad esempio le interrogazioni della base di dati)
  - La frequenza con cui le interazioni avvengono
  - Il numero di utenti che simultaneamente possono interagire.



# CARATTERISTICHE DELLA PROGETTAZIONE CONCETTUALE

- Dovrebbe essere indipendente da uno specifico modello dei dati e da uno specifico DBMS (scelta da adottare successivamente)
- Deve escludere dettagli implementativi legati a alla scelta del modello dei dati o alla scelta di uno specifico DBMS



# **DOCUMENTAZIONE ALLA PROGETTAZIONE CONCETTUALE**

- Per la descrizione statica delle entità e delle loro associazioni si utilizzeranno i Class Diagram di UML;
- Tradizionalmente viene proposta un linguaggio di descrizione grafico chiamato Schemi E-R (Entità-Relazione);
  - E' uno schema facilmente esprimibile mediante Class Diagram UML
  - I Class Diagram di UML sono lo standard per la rappresentazione delle strutture dati nella programmazione ad oggetti
  - I Class Diagram sono indipendenti dal modello dei dati relazionale e agevolano le descrizioni per il modello relazionale
- Dizionari delle entità e delle associazioni
  - Descrivono e commentano integrando la presentazione del Class Diagram
- Dizionario dei vincoli
- Dizionario delle interrogazioni e indicazione della loro frequenza

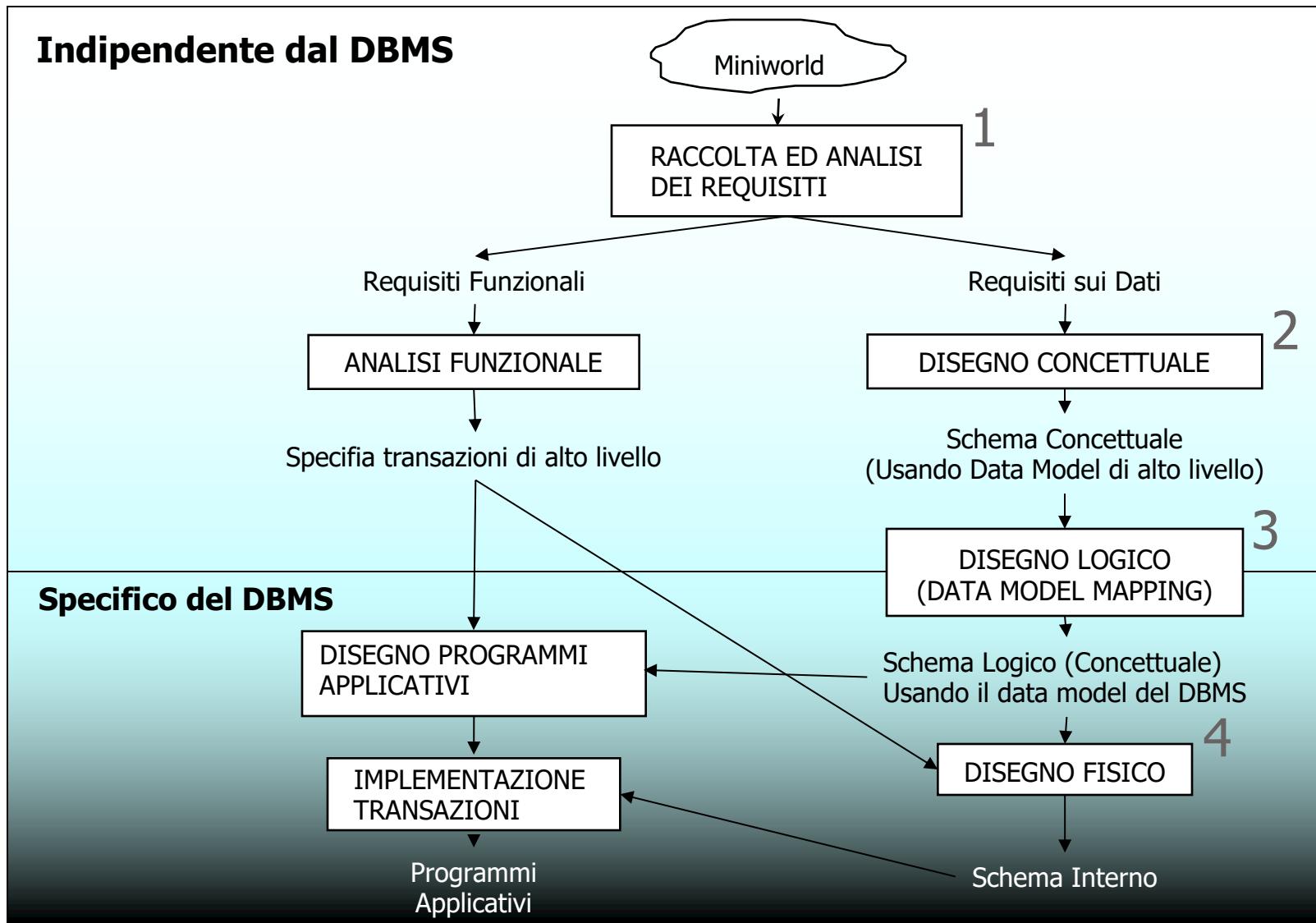


# PASSI SUCCESSIVI ALLA PROGETTAZIONE CONCETTUALE

- Progettazione logica
  - Scelta del modello dei dati
  - Codifica delle entità ed associazioni nel modello dei dati
- Progettazione fisica
  - Scelta del DBMS
  - Definizione dei metadati utilizzando il DDL del DBMS
  - Implementazione dei vincoli di consistenza
  - Definizione dei dettagli implementativi utilizzando le funzionalità specifiche del DBMS scelto



# LE FASI DI PROGETTAZIONE DI UN DATABASE



# 1. RACCOLTA E ANALISI DEI REQUISITI

- Il progettista del DB intervista i potenziali utenti dei DB per capire e documentare i requisiti utente.

*Output:*

- Requisiti sui dati
- Requisiti funzionali
  - Operazioni definite dall'utente (transazioni) che saranno applicate al DB.
    - *Es:* aggiornamenti, ricerche, ...
  - Specifica dei requisiti funzionali attraverso Data Flow Diagrams.



## 2. DISEGNO DEL DATABASE CONCETTUALE

- Creazione dello schema concettuale, usando un data model concettuale ad alto livello.

### *Output:*

- Descrizione concisa dei requisiti utente dei dati inclusiva di una descrizione dettagliata di tipi di dati, relazioni e vincoli fatta usando i concetti del data model ad alto livello.
  - Poiché non vi sono dettagli implementativi, tale descrizione può essere usata per comunicare con utenti non tecnici.
  - Tale documentazione può essere usata anche come riferimento per assicurarsi di aver considerato tutti i requisiti dell'utente.
- Questo approccio abilita il progettista a concentrarsi sui dati ignorando i dettagli di memorizzazione.
- Dopo aver disegnato lo schema concettuale, le operazioni di base del data model possono essere usate per specificare le operazioni di alto livello individuate durante l'analisi funzionale (*passo 1*).



# **3. DISEGNO DEL DATABASE LOGICO**

- Implementazione del database usando un DBMS commerciale.
  - Detto anche data model mapping.
  - Molti DBMS usano un data model di implementazione, in modo da trasformare facilmente lo schema concettuale da data model ad alto livello in data model di implementazione.

*Output:*

- Schema del database nel data model di implementazione specifico del DBMS scelto.



# **4. DISEGNO DEL DATABASE FISICO**

- Specifica delle strutture di memorizzazione interne, degli access path e dell'organizzazione dei file.
- Progettazione dei programmi applicativi e implementazione delle transazioni.



# IL PROCESSO DI PROGETTAZIONE DEL DATABASE

1. Raccolta e analisi dei requisiti
  2. Disegno del database concettuale
  3. Disegno del database logico
  4. Disegno del database fisico
- 
- La prima fase mal si presta all'utilizzo di rigorosi formalismi, essendo essenzialmente un grosso documento di testo.
  - La seconda fase, invece, permette la definizione di modelli formali per supportare il lavoro del progettista.



# MODELLAZIONE DEL DB: ER VS UML



# MODELLAZIONE DEL DATABASE

- *La seconda fase, invece, permette la definizione di modelli formali per supportare il lavoro del progettista.*
- Abbiamo due tipologie di modelli che possiamo utilizzare
  - ER (Entity Relationship):
    - Il diagramma ER è definito proprio per la modellazione di basi di dati relazionali
    - I costrutti e le forme utilizzate sono adattate alla modellazione di un database
  - UML (Unified Modelling Language)
    - I diagrammi UML sono utilizzati per molte attività di modellazione nel ciclo di vita di un progetto software
      - Sequence Diagram, Activity Diagram, StateChart Diagram, Class Diagram...
      - Ma è anche esso un linguaggio di modellazione (UNIFICATO) e può essere facilmente essere adattato anche per la descrizione di una base di dati
    - All'interno del corso, utilizzeremo questo.
    - In particolare, utilizzeremo la notazione dei Class Diagram per la modellazione di una base di dati.



# MODELLO ER

- Il modello Entità-Relazione (ER) è un diffusissimo data model di alto livello, estesamente utilizzato per definire lo schema concettuale di un database.
- È stato concepito per essere più vicino ai concetti “*umani*”, e quindi facilmente comprensibile anche ad utenti non tecnici.
- Il modello ER ha avuto una grandissima diffusione principalmente per i formalismi grafici semplici e chiari che incorpora.
- Il modello ER descrive i dati con tre concetti fondamentali:
  - Entità
  - Attributi
  - Relazioni



# MODELLAZIONE DEI DATI IN UML

- UML (**Unified Modeling Language**) è un linguaggio grafico per la modellazione di applicazioni software basate sulla programmazione orientata agli oggetti.
  - Permette di rappresentare (attraverso una serie di diagrammi) tutti gli aspetti di un'applicazione software: dati, operazioni, processi e architetture.
- UML può essere utilizzato in alternativa al modello ER:
  - I *diagrammi delle classi* sono usati per descrivere le classi di oggetti di interesse e le relazioni che intercorrono tra di esse.
- *Cambia la rappresentazione diagrammatica ma non l'approccio alla progettazione.*



# UML VS. ER

- UML
  - modellazione di un'applicazione software
    - aspetti strutturali e comportamentali (dati, operazioni, processi e architetture)
  - formalismo ricco
    - diagramma delle classi, degli attori, di sequenza, di comunicazione, degli stati, ...
- ER/EER
  - modellazione di una base di dati
  - aspetti strutturali di un'applicazione
  - costrutti funzionali alla modellazione di basi di dati



# UML VS. ER

- Principali differenze di UML rispetto ad ER
  - assenza di notazione standard per definire gli identificatori
  - possibilità di aggiungere note per commentare i diagrammi
  - possibilità di indicare il verso di navigazione di una associazione (non rilevante nella progettazione di una base di dati)
- Formalismi diversi
- Il diagramma delle classi di un'applicazione è diverso dallo schema ER della base di dati
- Il diagramma delle classi, anche se progettato per uso diverso, può essere adattato per la descrizione del progetto concettuale di una base di dati



# UML: LE ORIGINI

- Proposto a metà degli anni '90 quale **formalismo unificante** per la modellazione o.o. di applicazioni software, è stato standardizzato sotto l'egida dell'Object Management Group (OMG).
- UML offre una molteplicità di diagrammi, corredati da una descrizione testuale della loro semantica: **molteplicità di viste** della medesima applicazione.



# UML: IL MODELLO DELL'APPLICAZIONE

- L'insieme dei diagrammi definisce il **modello dell'applicazione** (controparte dello schema ER):
  - UML è un metamodello per la descrizione di modelli di applicazioni software.
- Nella sua versione corrente UML prevede un certo numero di **diagrammi fondamentali**:
  - diagramma delle classi, degli oggetti, dei casi d'uso, di sequenza, di comunicazione, delle attività, degli stati, dei componenti e di distribuzione dei componenti.



# UML: I DIAGRAMMI PRINCIPALI (1)

- Diagramma delle classi:
  - descrive le caratteristiche statiche e dinamiche delle componenti (**classi**) e delle loro relazioni (**associazioni**).
- Diagramma degli oggetti:
  - rappresentazione delle possibili istanze delle classi (**oggetti**) e dei loro collegamenti.
- Diagramma dei casi d'uso:
  - modalità di utilizzo del sistema da parte di persone/altri sistemi (**attori**) e interazioni tra sistema e attori.



# UML: I DIAGRAMMI PRINCIPALI (2)

- Diagramma di sequenza:
  - ordinamento temporale di messaggi (**invocazione di metodi**) scambiati tra i diversi oggetti dell'applicazione.
- Diagramma delle attività:
  - comportamento dinamico di un processo dell'applicazione in termini dei flussi di **attività** da svolgere.
- Diagramma degli stati:
  - descrive il ciclo di vita di un oggetto dell'applicazione attraverso gli **stati** che può assumere.

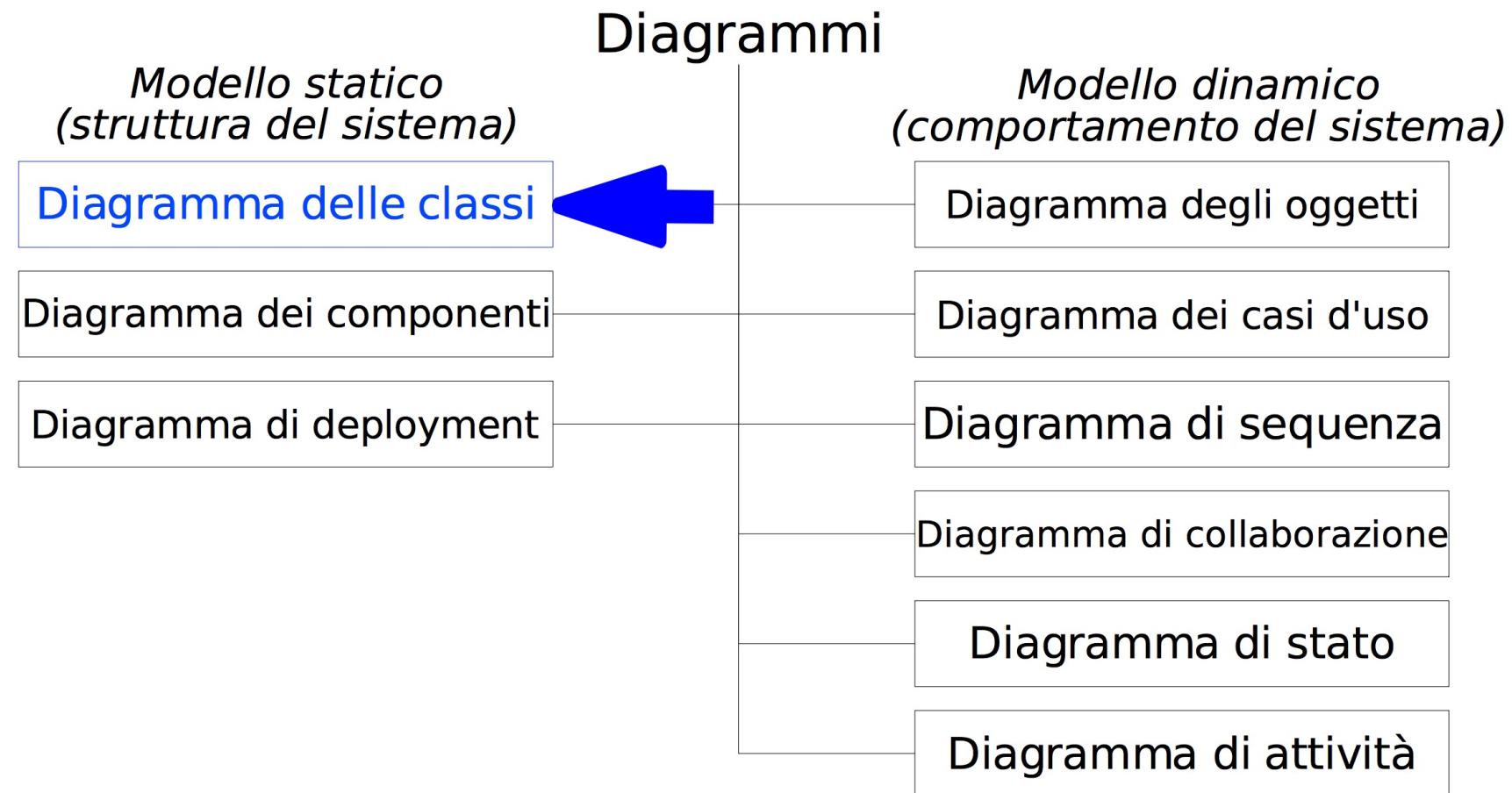


# UML: I DIAGRAMMI PRINCIPALI (3)

- Diagramma di collaborazione (comunicazione):
  - descrive lo scambio di **messaggi** tra gli oggetti, ma utilizzando notazione e prospettiva diverse.
- Diagramma dei componenti:
  - descrive l'organizzazione delle **componenti** fisiche del sistema (file, moduli, ..) e le loro dipendenze.
- Diagramma di distribuzione dei componenti (deployment):
  - descrivere la **dislocazione** dei nodi hardware del sistema e le loro associazioni.



# UML: I DIAGRAMMI PRINCIPALI (4)



# MODELLAZIONE STRUTTURALE

- Modellazione strutturale:
  - Rappresenta una vista di un sistema software che pone l'accento sulla struttura degli oggetti (classi di appartenenza, relazioni, attributi, operazioni)
- Il **diagramma delle classi** descrive il tipo degli oggetti che compongono il sistema e le relazioni statiche esistenti tra loro



# **ENTITÀ E ATTRIBUTI (UML E ER)**

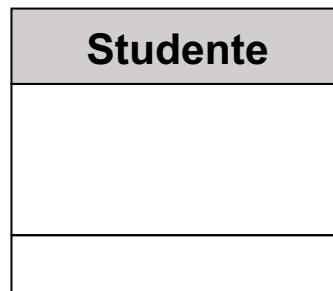


# ENTITÀ

- Le entità corrispondono a classi di oggetti del mondo reale (fatti, persone, ...) che hanno proprietà omogenee, comuni ed esistenza “autonoma” ai fini dell'applicazione di interesse.
- Un'entità può essere un oggetto fisico (casa, impiegato, ...) o un oggetto concettuale (un lavoro, una società, ...).



Rappresentazione ER  
dell'entità "*Studente*"



Rappresentazione UML  
Della classe "*Studente*"



# RAPPRESENTAZIONE DI DATI CON I DIAGRAMMI DELLE CLASSI

- **Classi:** Sono le componenti principali dei diagrammi delle classi.
  - Corrispondono alle entità del modello ER.
  - È possibile aggiungere i metodi (i.e., le operazioni ammissibili sugli oggetti della classe).
  - I dati vengono descritti *insieme* alle operazioni da svolgere su di essi.

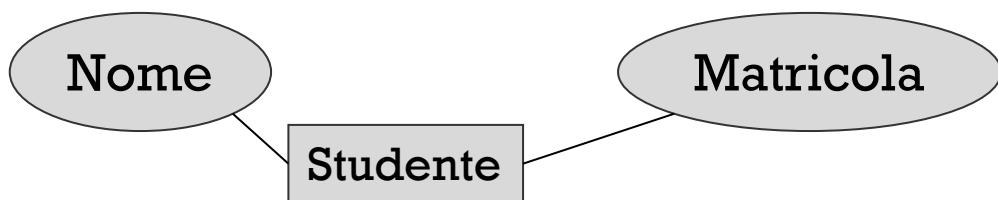
Impiegato
Codice
Cognome
Stipendio
Età
<i>SetStipendio()</i>

Progetto
Nome
Budget
Data consegna

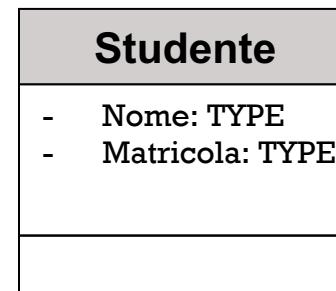


# ATTRIBUTI

- Ogni entità ha delle proprietà dette attributi.
  - *Es:* l'entità *Impiegato* ha attributi **nome**, **età**, **indirizzo**, **salario**, **telefono**, ...
- Ogni entità è caratterizzata da un valore per i suoi attributi.



Rappresentazione ER  
dell'entità "*Studente*" con gli  
attributi "*Nome*" e "*Matricola*"

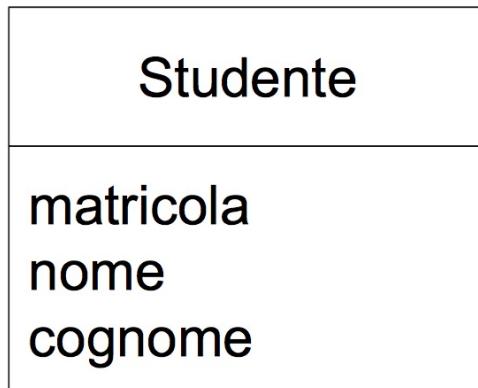


Rappresentazior **UML**  
della classe "*Studente*" con gli  
attributi "*Nome*" e "*Matricola*"



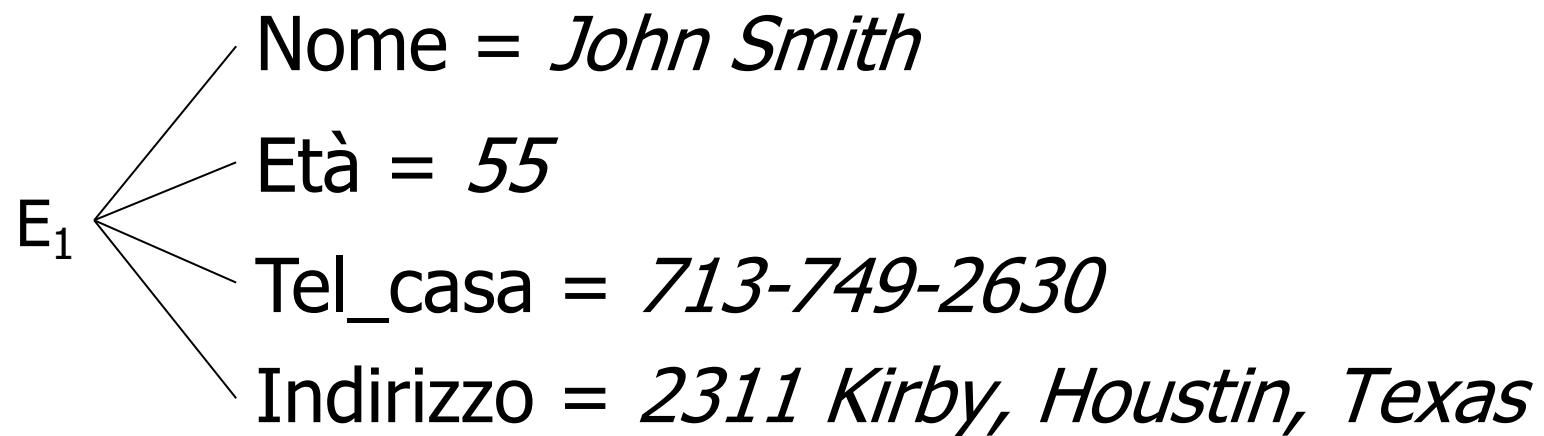
# ATTRIBUTI IN ER E UML

- ER:
  - un attributo descrive una proprietà elementare di un'entità o di una relazione.
- UML:
  - un attributo rappresenta una proprietà elementare degli oggetti di una classe.



# ENTITÀ E ATTRIBUTI: *ESEMPIO*

- ***Esempio:*** entità *Impiegato*



# TIPO DI ENTITÀ

- Entità con gli stessi attributi di base sono raggruppati in un tipo di entità.
- Esempio:** Tutte le persone che lavorano per un dipartimento possono essere definite con l'entità *Impiegato*.

Nome del Tipo di Entità:  
(Schema o Intenzione)

**Impiegato**

Nome, Età, Stipendio

$e_1$	■	
(John Smith, 55, 80k)		
$e_2$	■	
(Fred Brown, 40, 30k)		
$e_3$	■	
(Judy Clark, 25, 20k)		
	:	

Insieme di Entità  
(Estensione)

Un tipo di entità descrive lo schema (o intenzione) per un insieme di entità

Le entità individuali di un particolare tipo di entità sono raggruppate in una collezione o insieme detta estensione del tipo di entità.



# TIPI DI ATTRIBUTI

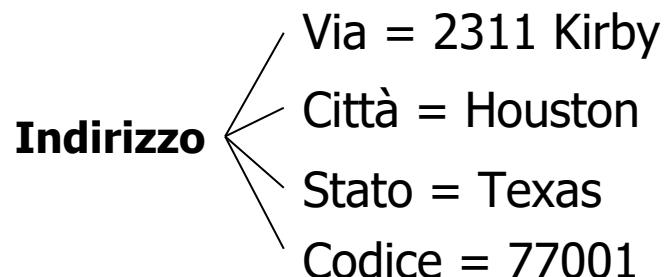
- Gli attributi possono essere di diversi tipi

Divisibile?	Più valori?	Calcolabile?
Semplice	Single-valued	Memorizzato
Composto	Multivalued	Derivato



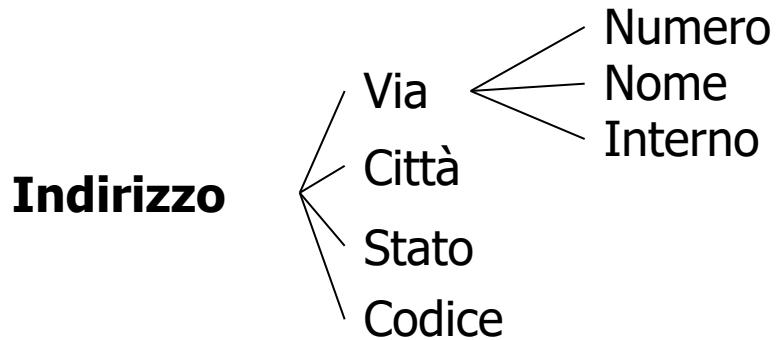
# ATTRIBUTI SEMPLICI E COMPOSTI

- *Attributi semplici*: ogni entità ha un valore singolo (atomico) per tale attributo.
- *Attributi composti*: possono essere divisi in sottoparti, che rappresentano informazioni di base con loro significati indipendenti.
  - **Esempio:** l'attributo **Indirizzo**:



# ATTRIBUTI SEMPLICI E COMPOSTI (2)

- Gli attributi composti possono formare una gerarchia:



- L'utilizzo di un attributo semplice o di uno composto dipende dalla necessità o meno di trattare separatamente le sottoparti.



# SINGLE-VALUED E MULTIVALUED

## ■ *Attributi single-valued:*

- hanno un solo valore per ciascuna entità.
- **Esempio:** l'età di un impiegato.

## ■ *Attributi multivalued:*

- può avere un insieme di valori per la stessa entità.
- **Esempio:** l'attributo **Colore** per un'entità auto. Un'auto può avere più colori.
- Può essere determinato un limite inferiore e un limite superiore al numero di valori per un'entità.



# MEMORIZZATI E DERIVATI

- *Attributi derivati:* alcuni attributi possono essere in relazione tra loro.
  - **Esempio:** l'età e la data di nascita:

■ Età	<b>attributo derivato</b>
■ Data_nascita	<b>attributo memorizzato</b>
  - Alcuni valori di attributi possono essere derivati da entità correlate.  
**Esempio:**  
Numero\_di\_Impiegati di una entità “*dipartimento*” può essere derivato contando il numero di impiegati relati al (che lavorano per) dipartimento.
- Alcuni attributi possono avere valore **null** col significato di “*non noto*” o “*mancante*” o “*non applicabile*”.



# PARTICOLARITÀ DEGLI ATTRIBUTI IN UML

- Non è possibile definire attributi composti.
- È possibile associare agli attributi i rispettivi domini
  - Interi, reali, stringhe, etc.

Impiegato
<b>String</b> Codice
<b>String</b> Cognome
<b>Float</b> Stipendio
<b>int</b> Età
<i>SetStipendio()</i>



# ATTRIBUTI CHIAVE DI UN TIPO DI ENTITÀ

- Un importante vincolo sulle entità di un tipo di entità è la chiave o vincolo di unicità.
  - L'attributo chiave di un tipo di entità è un attributo che deve avere un valore univoco per cui ogni entità.
  - **Esempio:** Il codice fiscale di una persona.
- Talvolta più attributi insieme formano una chiave:
  - In tal caso tali attributi possono essere raggruppati in un attributo composto che diventa chiave.
- Alcuni tipi di entità possono avere più di un attributo chiave.
- In notazione ER l'attributo chiave è rappresentato **sottolineato** nell'ovale.



# **ESEMPIO: ENTITÀ AUTO**

AUTO

Targa(Numero,Provincia), Telaio, Marca, Modello, Anno\_imm, {Colore}

Car 1 .

((ASC 123, TEXAS), tk629, Ford Mustang, convertible, 1989, {red, black})

Car 2 .

((ABO 123, NEW YORK), WPS872, N~san sontra 2~cor, 1992, {blue})

Car3 .

((VYS 720, TEXAS), TD729, Chrysler LeBaron, 4dcor, 1993, {white, blue})

Gli attributi multivalued sono mostrati tra parentesi {}.  
Le componenti di un attributo composto sono mostrate tra parentesi ().



# DOMINIO DI UN ATTRIBUTO

- Ciascun attributo semplice è associato a un dominio o insieme di valori che rappresenta l'insieme dei valori che l'attributo può assumere.
- **Esempio:** l'età dell'impiegato può variare nel dominio (16, 70).
- *Matematicamente:*
  - Un attributo semplice  $A$ , del tipo di entità  $E$ , avente dominio  $V$ , è una funzione:  
 $A: E \rightarrow P(V)$        $P(V)$  insieme potenza dei sottoinsiemi di  $V$ :  
Un valore **null** è rappresentato con  $\emptyset$ .
  - $A(e)$  il valore dell'attributo  $A$  per l'entità  $e$ ,
  - $A(e) = \text{singleton}$  per attributi single-valued.



# DOMINIO DI UN ATTRIBUTO (2)

- Per un attributo composto A, del tipo di entità E, il dominio V è il prodotto cartesiano

$$V = P(V_1) \times P(V_2) \times \dots \times P(V_n)$$

dove  $V_i$  è il dominio dell'attributo semplice  $i$ -mo di A,

- **Esempio:** l'indirizzo telefonico di una persona con più residenze, dove ogni residenza può avere più telefoni, è specificato come segue:

{ Indirizzo ({Telefono (Prefisso, Numero)}, Indirizzo (Via (Numero, Nome, Interno), Città, Stato, Codice) }



# ESEMPIO: IL DATABASE COMPANY



# REQUISITI PER IL DATABASE COMPANY

- La compagnia è organizzata in DIPARTIMENTI. Ogni Dipartimento ha un nome, un numero ed un impiegato che lo gestisce. Bisogna tener traccia della data di insediamento del manager. Un dipartimento può avere più locazioni.
- Ogni dipartimento controlla una serie di PROGETTI. Ogni progetto ha un nome, un numero ed una singola locazione.
- Per IMPIEGATO si tiene traccia di: nome, SSN, indirizzo, salario, sesso e data di nascita. Ogni impiegato lavora per un dipartimento e può lavorare su più progetti. Teniamo traccia del numero di ore settimanali che un impiegato spende su un progetto e del supervisore di ogni impiegato.
- Ogni impiegato ha una serie di PERSONE A CARICO. Per ogni persona a carico, registriamo: nome, sesso, data di nascita e parentela con l'impiegato.



# DISEGNO CONCETTUALE DEL DATABASE COMPANY

- Descriviamo i tipi di entità per il database COMPANY.
- In accordo ai requirement possiamo identificare quattro tipi di entità:

## 1.DIPARTIMENTO

Nome, Numero, {Sedi}, Manager, Datains\_manager

Nome e Numero sono entrambi attributi chiave.

## 2.PROGETTO

Nome, Numero, Luogo, Dip\_controllo

Nome e Numero sono entrambi attributi chiave.



# DISEGNO CONCETTUALE DEL DATABASE COMPANY (2)

## 3. IMPIEGATO

Nome, SSN, Sesso, Indirizzo, Stipendio, DataNascita, Dipartimento, Supervisore

SSN è un attributo chiave.

Nome e Indirizzo sono attributi composti (occorrerebbe verificare con l'utente se ha bisogno di riferire ai componenti individuali).

## 4. PERS\_A\_CARICO

Sesso, DataNascita, Impiegato, Nome\_pers\_carico, Parentela



# DISEGNO CONCETTUALE DEL DATABASE COMPANY (3)

- Dobbiamo però rappresentare:
  - Il fatto che un impiegato può lavorare su più progetti.
  - Il numero di ore settimanali di un impiegato su ciascun progetto.
- Si può aggiungere un attributo a IMPIEGATO “*Lavora\_su*” composto di due componenti semplici (**Progetto, Ore**):

## IMPIEGATO

Nome (FName, Minit, LName), SSN, Sesso, Indirizzo, Stipendio, DataNascita, Dipartimento, Supervisore, {Lavora\_su(Progetto, Ore)}

- In alternativa, le stesse informazioni si potrebbero mantenere nel tipo di entità PROGETTO con un attributo composto:
  - **Addetti (Impiegato, Ore)**

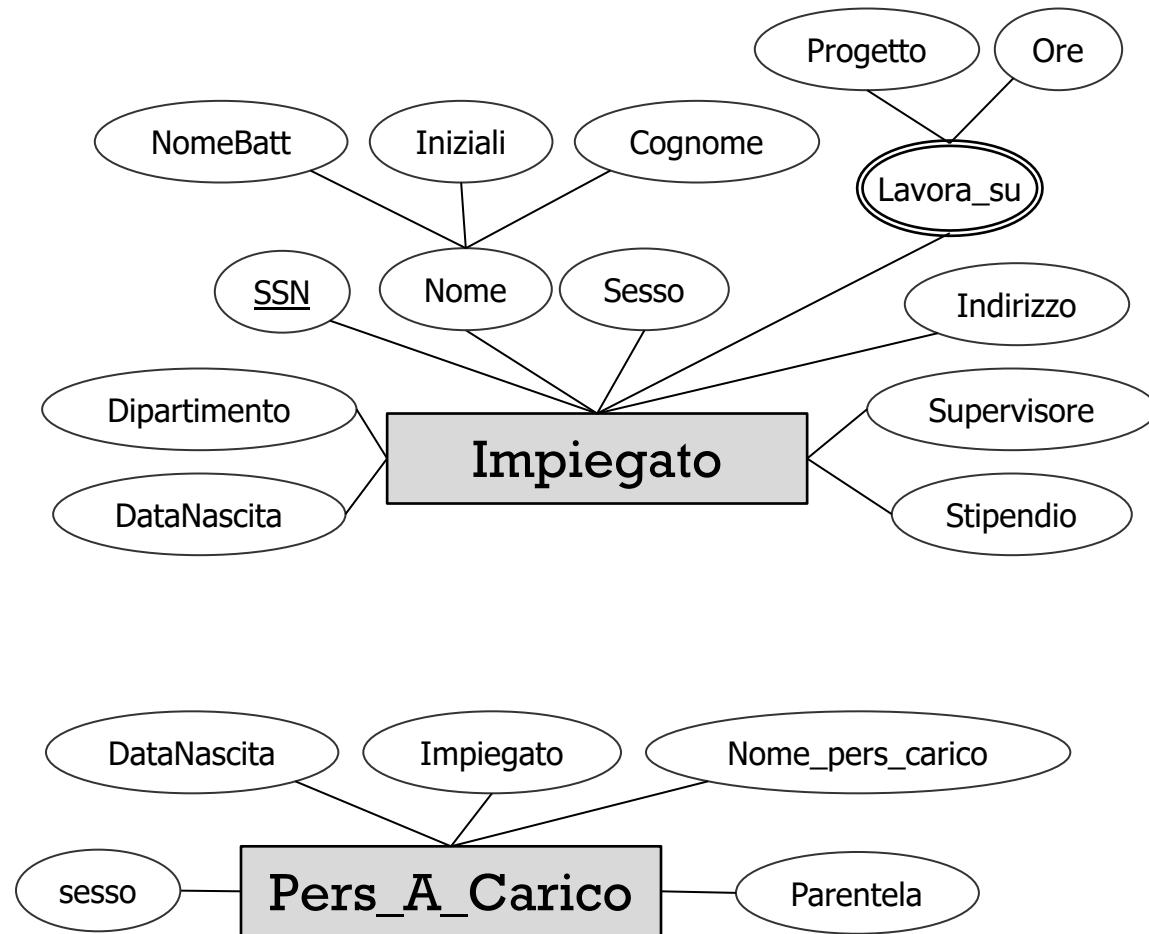
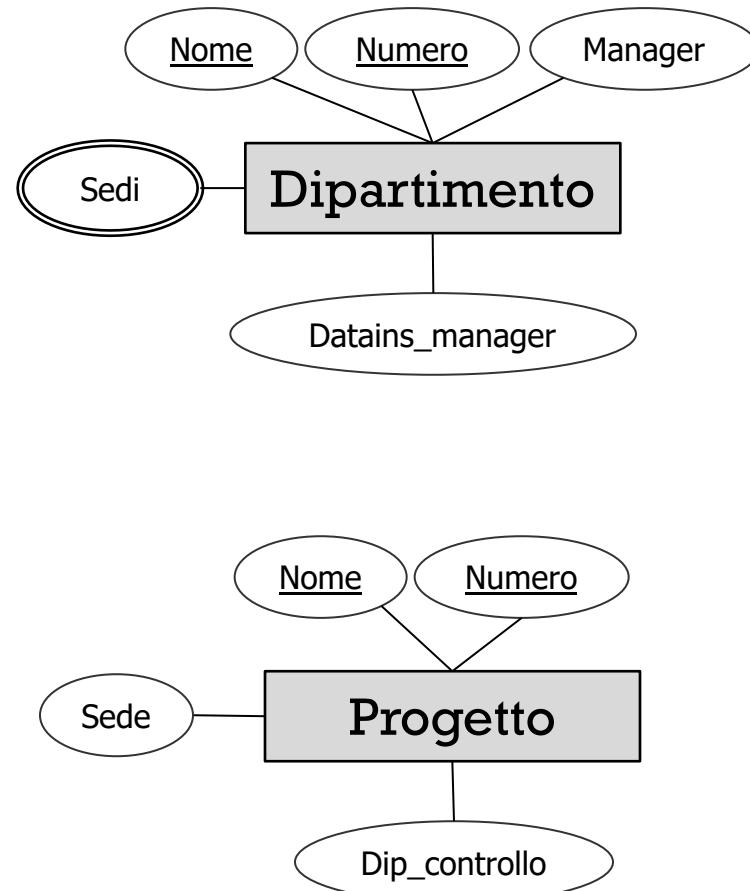


# DISEGNO CONCETTUALE DEL DATABASE COMPANY (4)

- **Esistono varie relazioni implicite:**
  - L'attributo Manager di DIPARTIMENTO si riferisce a un impiegato che gestisce il dipartimento;
  - L'attributo Dip\_controllo di PROGETTO si riferisce al dipartimento che controlla il progetto;
  - L'attributo Dipartimento di IMPIEGATO si riferisce al dipartimento per cui lavora l'impiegato;
  - ...
- Nel disegno iniziale queste associazioni tra entità sono rappresentabili come attributi, ma durante il processo di raffinamento nel modello ER questi riferimenti dovrebbero essere rappresentati come relazioni.



# PROGETTAZIONE ER PRELIMINARE PER IL DB COMPANY



# PROGETTAZIONE UML (CLASS DIAGRAM) PRELIMINARE PER IL DB COMPANY

Dipartimento
Nome
Numero
Manager
Datains_Manager
Sedi

Progetto
Nome
Numero
Sede
Dip_controllo

Impiegato
SSN
Dipartimento
DataNascita
{Nome}
Sesso
Indirizzo
Supervisore
Stipendio
{Lavora_Su}

Pers_a_carico
Nome
Parentela
Impiegato
Sesso
DataNascita





# LE RELAZIONI (ER)



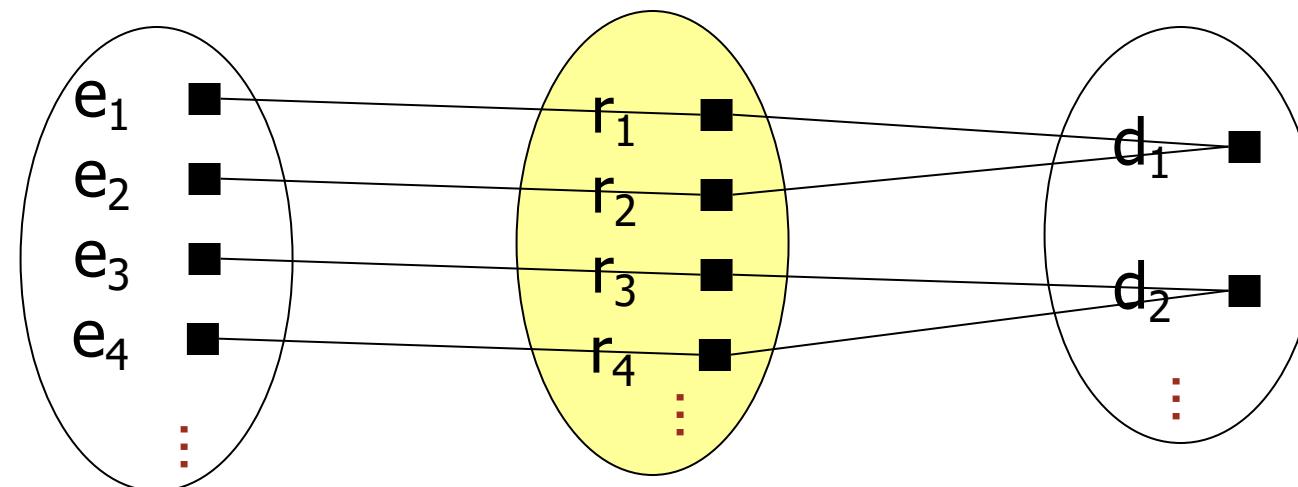
# TIPI E Istanze di Relazioni

- Le relazioni corrispondono a legami logici tra entità, significativi ai fini dell'applicazione di interesse.
- Un tipo di relazione è un'associazione tra n tipi di entità  $E_1, E_2, \dots, E_n$ .
- Le occorrenze o istanze di relazione associano n entità dei tipi di relazione richiesti.
- Ogni tipo di entità è detto partecipare al tipo di relazione.
- Il grado di un tipo di relazione è il numero di entità che vi partecipano. Se il grado è 2, la relazione è detta binaria.



# TIPI E Istanze DI RELAZIONI (2)

- **Esempio:** vogliamo rappresentare il fatto che ogni impiegato  $e_i$  lavora per un dipartimento  $d_j$ .
- Definiamo il tipo di relazione LAVORA\_PER tra i due tipi di entità IMPIEGATO e DIPARTIMENTO: ogni relazione  $r_i$  associa una entità IMPIEGATO  $e_i$  con una entità DIPARTIMENTO  $d_j$ .



Impiegato:  
Tipo di Entità

Lavora\_Per:  
Relazione

Dipartimento:  
Tipo di Entità



# TIPI E ISTANZE DI RELAZIONI (3)

*Matematicamente:*

R è un insieme di istanze di relazione  $r_i$  dove ogni  $r_i$  associa n entità ( $e_1, e_2, \dots, e_n$ ), e ciascuna entità  $e_j$  in  $r_i$  è un membro del tipo di entità  $E_j$ , con  $1 \leq j \leq n$ .

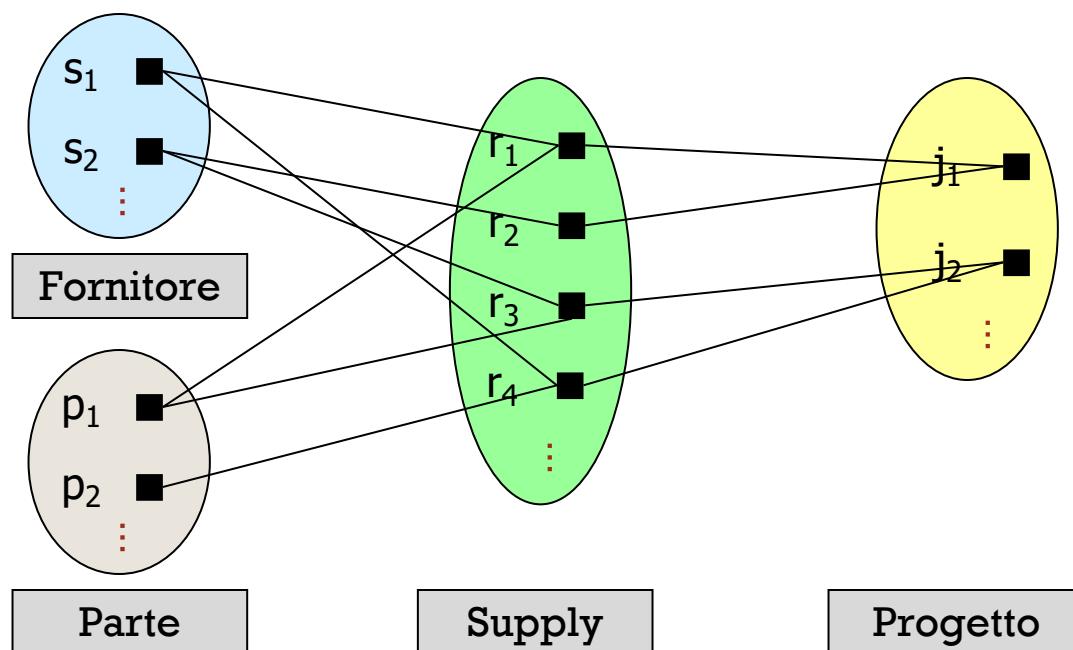
Quindi un tipo di relazione è una funzione matematica su  $E_1, E_2, \dots, E_n$  o alternativamente può essere definita come un sottoinsieme del prodotto cartesiano  $E_1 \times E_2 \times \dots \times E_n$ .

Ciascun  $E_j$  è detto partecipare al tipo di relazione R e analogamente ogni entità individuale  $e_j$  è detta partecipare all'istanza di relazione  $r_i = (e_1, e_2, \dots, e_n)$ .



# GRADO DI UN TIPO DI RELAZIONE

- Il grado di un tipo di relazione è il numero di tipi di entità partecipanti.
- Esempio:**
  - LAVORA\_PER è di grado 2 (binaria).
  - La relazione SUPPLY è un tipo di relazione ternaria, dove ogni istanza di relazione  $r_i$  associa tre entità, un fornitore  $s$ , una parte  $p$  e un progetto  $j$  ogni volta che  $s$  fornisce la parte  $p$  al progetto  $j$ .



Le relazioni possono essere di qualsiasi grado ma le più ricorrenti sono quelle binarie.



# RAPPORTO DI CARDINALITÀ

- Deve essere indicato per ciascun tipo di entità che partecipa ad una relazione, e permette di specificare il numero minimo e massimo di istanze di relazione a cui le occorrenze delle entità coinvolte possono partecipare.



- Rappresentazione ER della relazione “*Impiegato lavora per Dipartimento*”, con rapporto di cardinalità N:1 – N impiegati lavorano per un dipartimento:
  - MAX:** Ogni dipartimento può avere numerosi impiegati, e ciascun impiegato lavora per un solo dipartimento.
  - MIN:** Un dipartimento potrebbe non avere impiegati, mentre un impiegato deve sempre essere assegnato ad un dipartimento.



# **ESEMPIO: RELAZIONE LAVORA\_PER**

**Esempio:** Impiegato dipartimento

(rossi, ricerca)

(bianchi, ricerca)

(neri, amministrazione)

(verdi, ricerca)

- Rossi è presente una volta nella relazione.
- Ricerca è presente 3 volte.



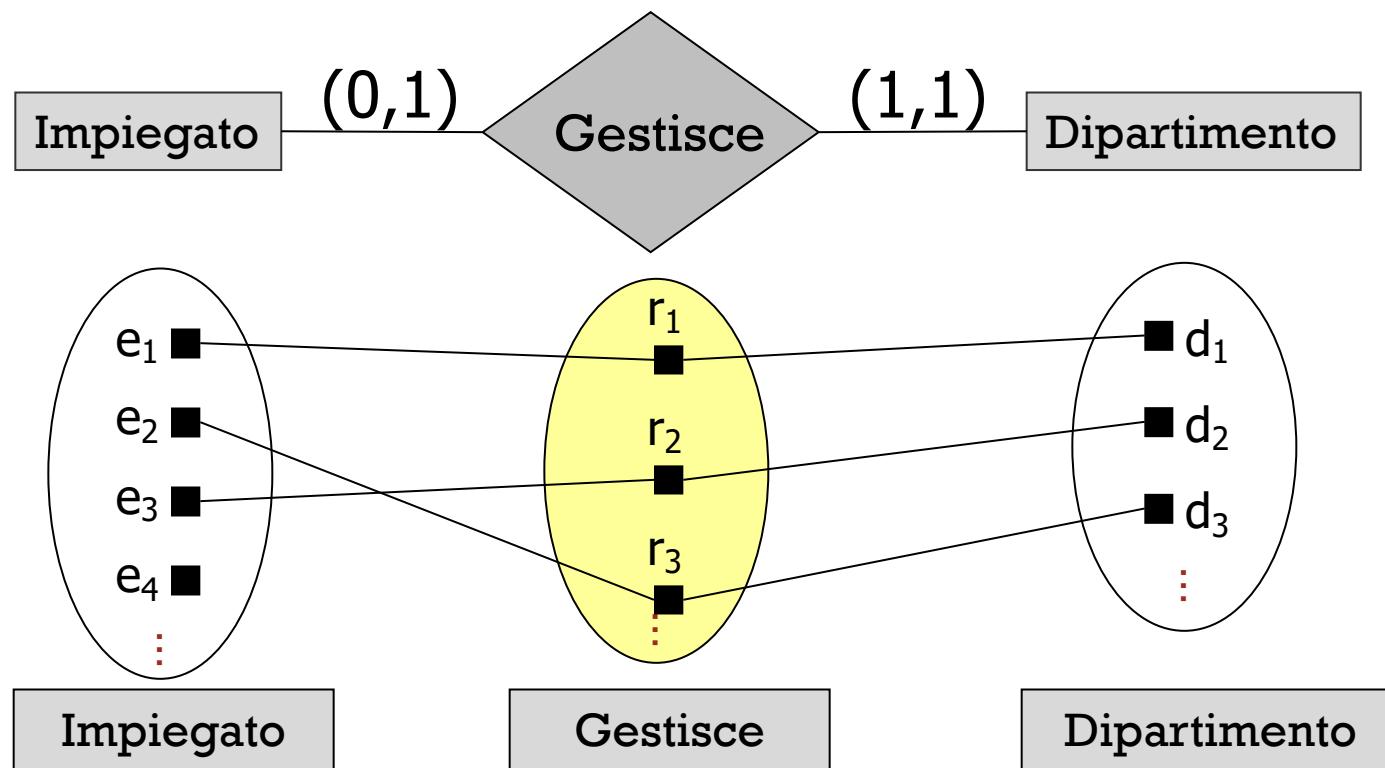
# RAPPORTO DI CARDINALITÀ (2)

- È possibile assegnare un qualunque intero non negativo a un rapporto di cardinalità, con l'ovvio vincolo che la cardinalità minima deve essere minore o uguale alla cardinalità massima.
- Nella maggior parte dei casi si utilizzano solo tre valori: 0, 1 e N:
  - Il valore 0 per la cardinalità minima indica una partecipazione opzionale del tipo di entità alla relazione.
  - Il valore 1 per la cardinalità minima indica una partecipazione obbligatoria del tipo di entità alla relazione.
- La cardinalità minima può eventualmente essere omessa, quella massima deve essere sempre specificata.



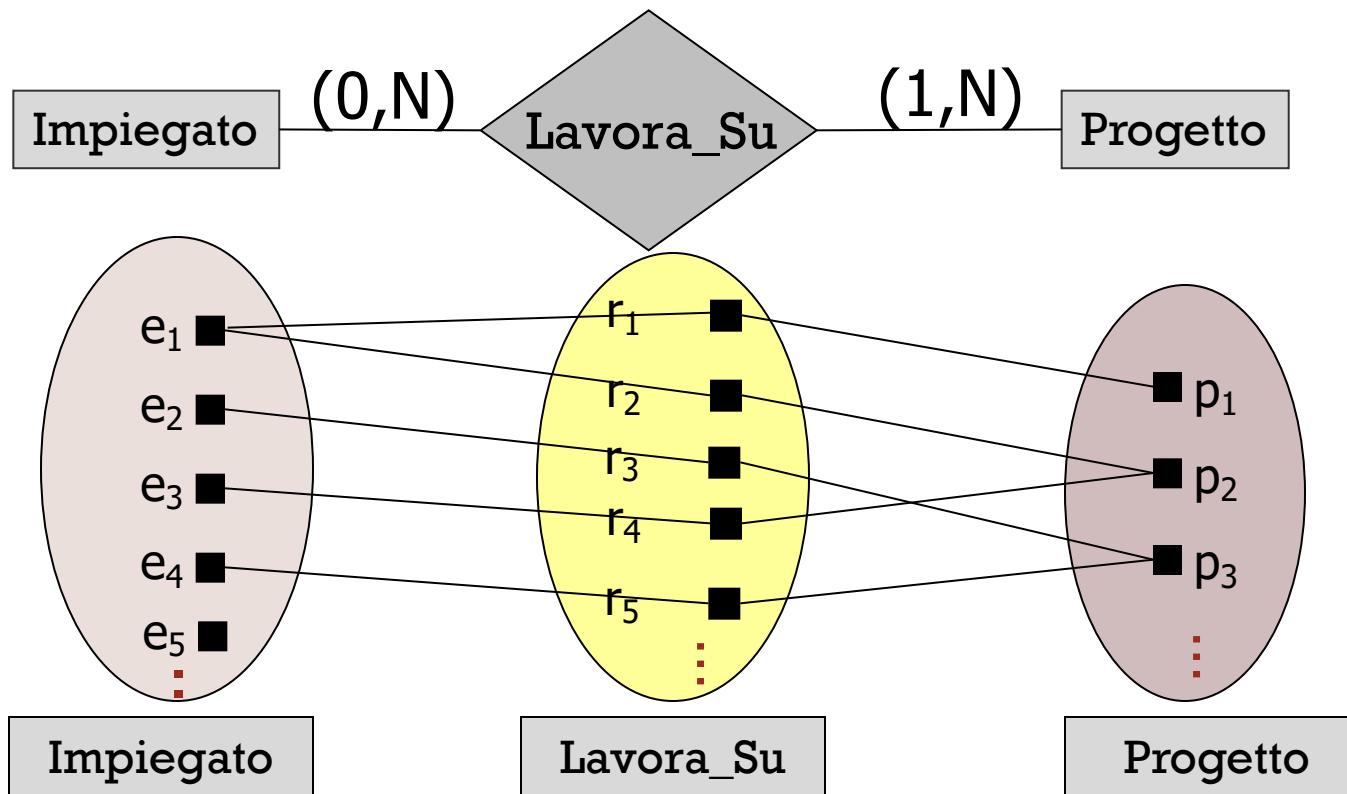
# RAPPORTO DI CARDINALITÀ: ESEMPIO

- La relazione binaria GESTISCE tra IMPIEGATO e DIPARTIMENTO è di rapporto di cardinalità 1:1 (un impiegato può gestire al più un dipartimento, ed un dipartimento deve sempre avere un solo manager).

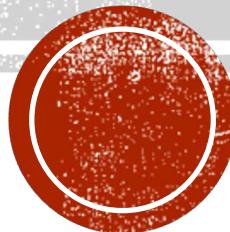


# RAPPORTO DI CARDINALITÀ: ESEMPIO

- La relazione binaria LAVORA\_SU tra IMPIEGATO e PROGETTO è di rapporto di cardinalità M:N, (poiché un impiegato può lavorare su più progetti, e più impiegati possono lavorare sullo stesso progetto).

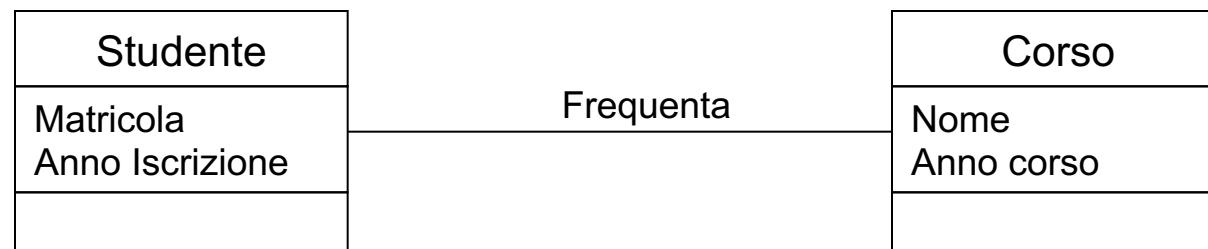
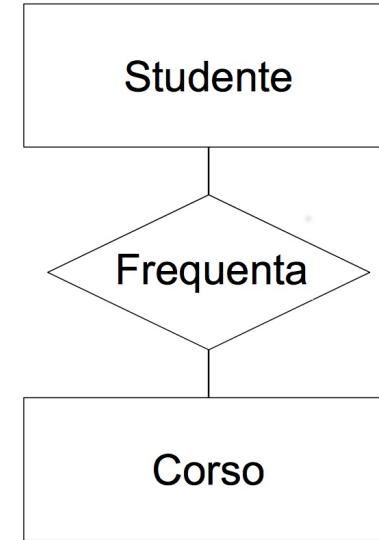


# LE ASSOCIAZIONI (UML)



# LE ASSOCIAZIONI

- **Associazioni:** Corrispondono alle relazioni del modello ER.
  - Le associazioni binarie si rappresentano con semplici linee che congiungono le classi coinvolte.
  - Il nome della relazione viene posto sulla linea.
    - *Non è obbligatorio:* infatti, in UML possono esistere relazioni senza nome.



# LE ASSOCIAZIONI (2)

- Si possono definire più associazioni tra le medesime classi.

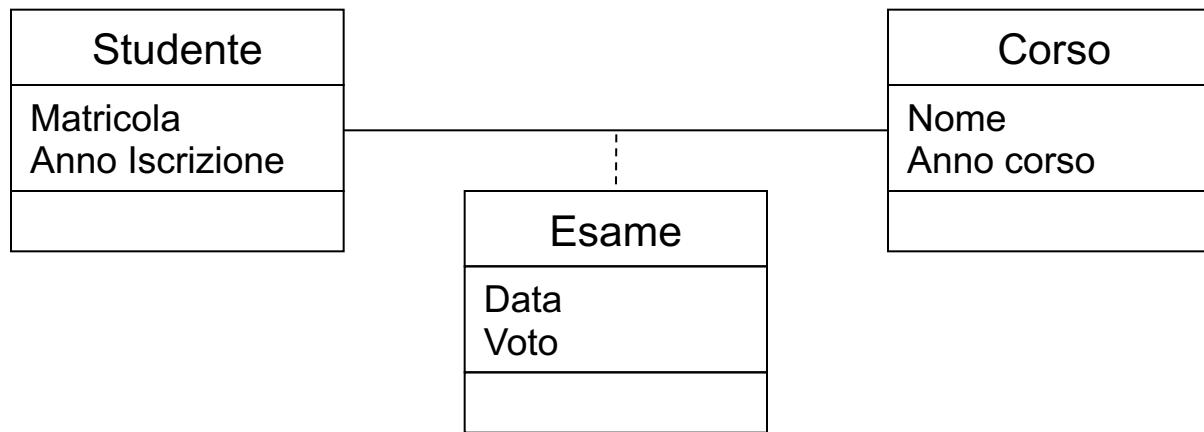


- È possibile associare *ruoli* alle classi coinvolte nelle associazioni.
- Non è possibile assegnare attributi alle associazioni.



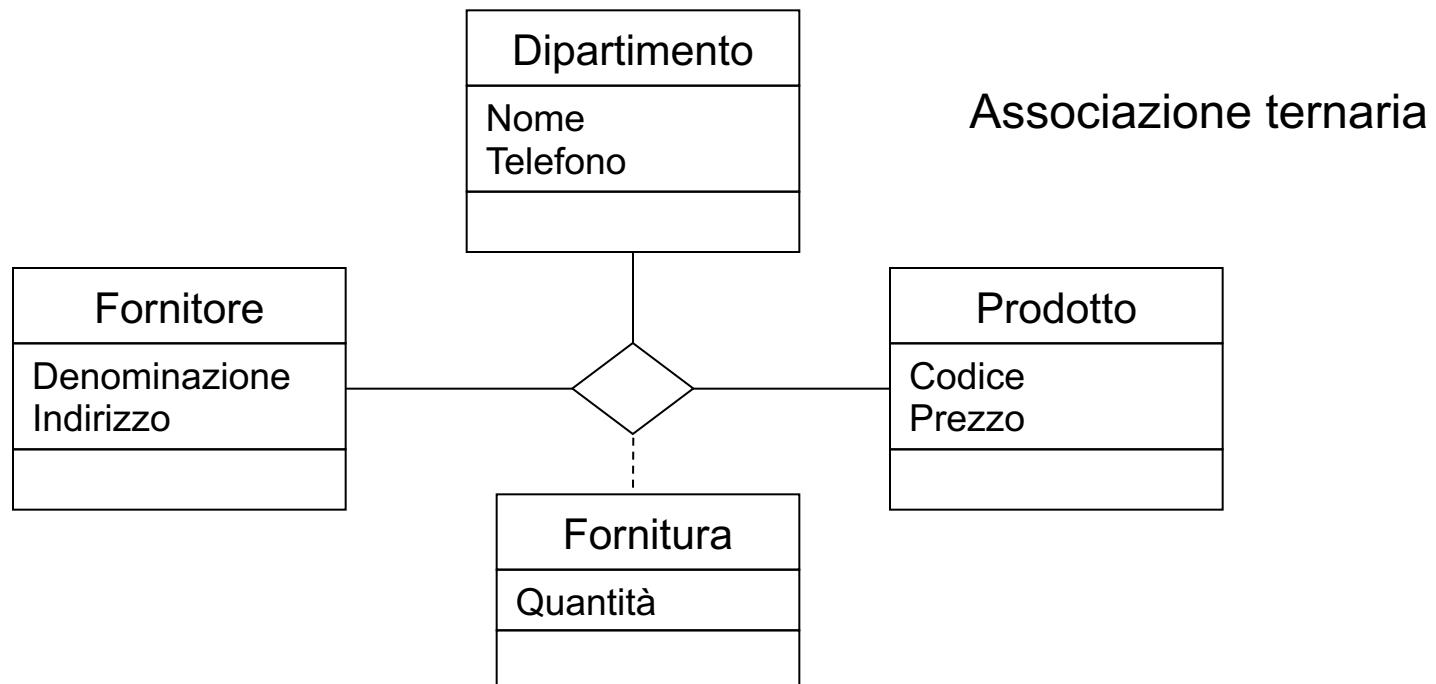
# LE ASSOCIAZIONI (3)

- Per assegnare attributi ad una associazione si fa uso delle *classi di associazione* per descrivere le proprietà di una associazione.
  - Queste classi vengono collegate all'associazione da descrivere mediante una linea tratteggiata.



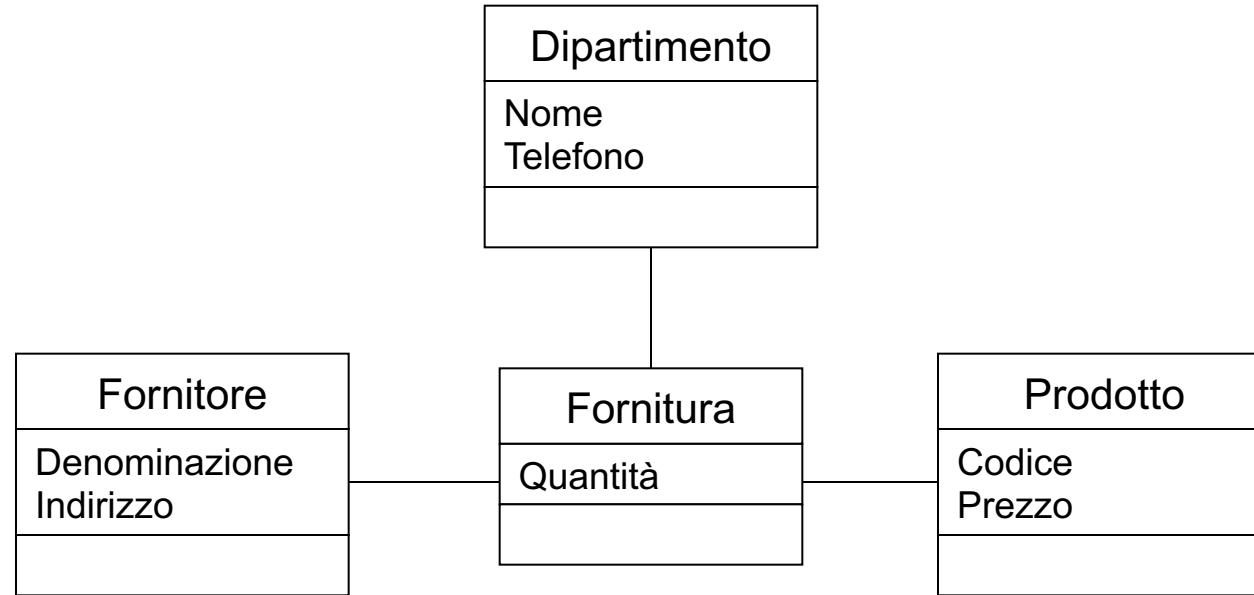
# ASSOCIAZIONI N-ARIE

- L'associazione viene rappresentata da un *rombo* collegato con le classi che partecipano all'associazione.
  - Si può usare una *classe di associazione* per assegnare attributi all'associazione n-aria.



# ASSOCIAZIONI N-ARIE (2)

- L'uso di associazioni n-arie è raro nel diagramma delle classi.
  - Si può applicare un processo di “*reificazione*”: ovvero trasformare l'associazione in una classe legata alle altre tramite associazioni binarie.



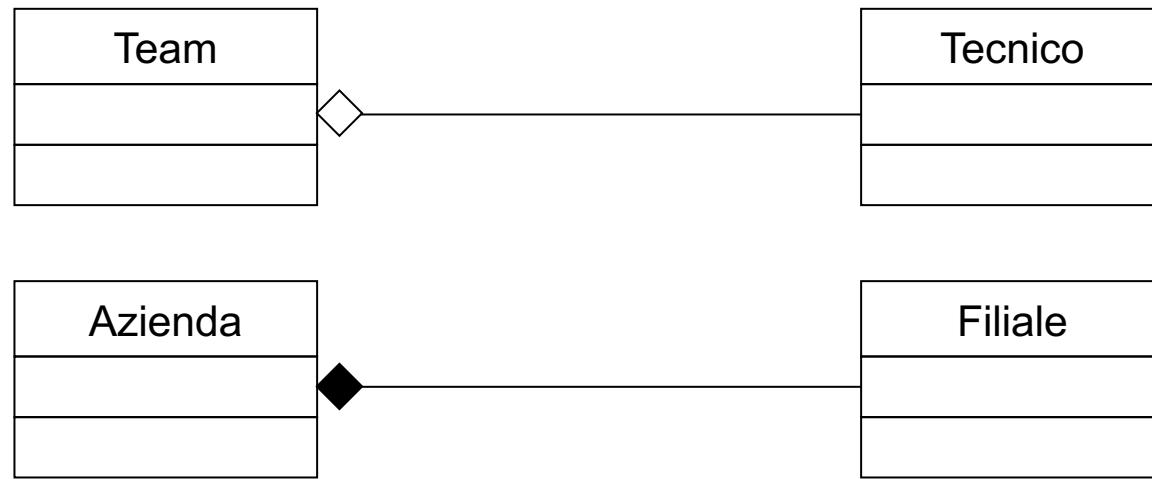
# PROPRIETÀ DELLE ASSOCIAZIONI

- Con una freccia è possibile indicare un verso privilegiato di *navigabilità* di un'associazione.
- Si possono specificare *aggregazioni* di concetti: relazioni tra un oggetto composito e uno o più concetti che ne costituiscono una sua parte.
  - Sono indicate da linee avente un rombo dal lato del concetto “aggregante”.



# PROPRIETÀ DELLE ASSOCIAZIONI (2)

- Il rombo bianco indica che un oggetto della classe “parte” può esistere senza dover appartenere a un oggetto della classe “aggregante”. **(Aggregazione)**
- Il rombo nero indica bianco indica che un oggetto della classe “parte” **non** può esistere senza appartenere a un oggetto della classe “aggregante”. **(Associazione)**

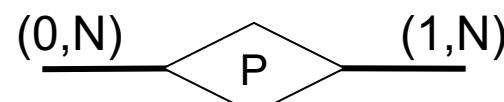
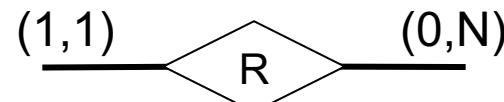


# ASSOCIAZIONI CON MOLTEPLICITÀ

- La molteplicità di default è 1 (che denota la coppia di cardinalità 1..1).
  - Le cardinalità di default possono essere omesse.
- La cardinalità “*molti*” viene rappresentata dal simbolo \* (dove si intende 0..\*, ovvero (0, N) dello schema ER).
- La cardinalità di default per l’aggregazione è \* (i.e., 0..\*).



*Corrispondenza (invertita)  
con le molteplicità dell'ER:*



# IDENTIFICATORI (INTERNI)

- In UML non esiste una notazione per esprimere identificatori di classi.
- Si possono definire vincoli di integrità su associazioni e attributi specificandoli tra parentesi graffe (*vincolo utente*).
  - **Es.**  $\{id\}$  indica che l'attributo è un identificatore.
  - **Es.** assegnare  $\{id\}$  a più attributi indica un identificatore composto.

Automobile
Targa $\{id\}$
Modello
Colore

Persona
Data Nascita $\{id\}$
Cognome $\{id\}$
Nome $\{id\}$
Indirizzo



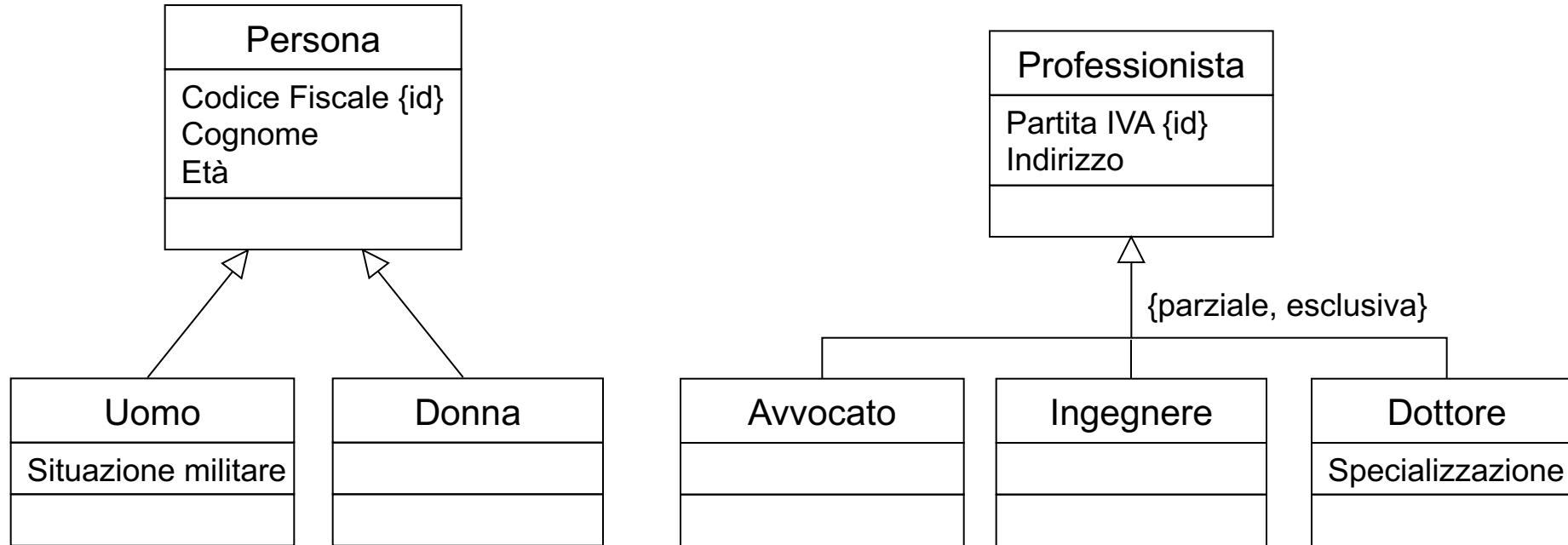
# IDENTIFICATORE ESTERNO

- Per gli identificatori esterni si usa uno *stereotipo*.
  - Si usano per estendere i costrutti base dell'UML:
    - quando si vuole modellare un concetto che non può essere modellato con i costrutti di base.
    - Vengono indicati da un nome racchiuso tra i simboli << e >>.
    - **Es.** Lo stereotipo <<identificante>> insieme alla *Matricola* identifica univocamente uno *Studente* rispetto una particolare *Università*.



# GENERALIZZAZIONI

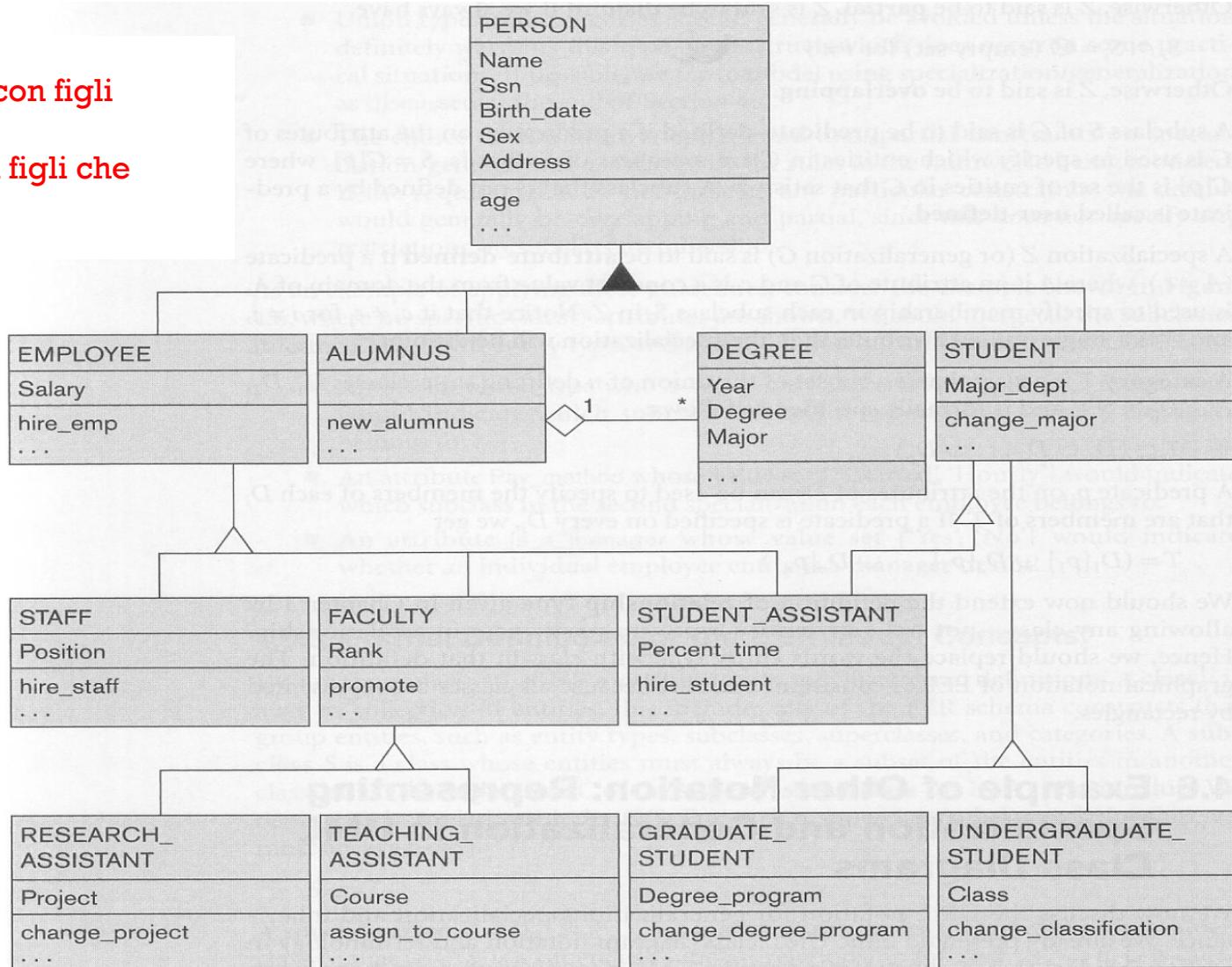
- Le generalizzazioni sono molto simili a quelle del modello ER.
- Eventuali proprietà possono essere rappresentate con vincoli racchiusi tra le parentesi { e }.



# UNA GERARCHIA DI GENERALIZZAZIONI

Triangolo **bianco** = generalizzazione con figli disgiunti

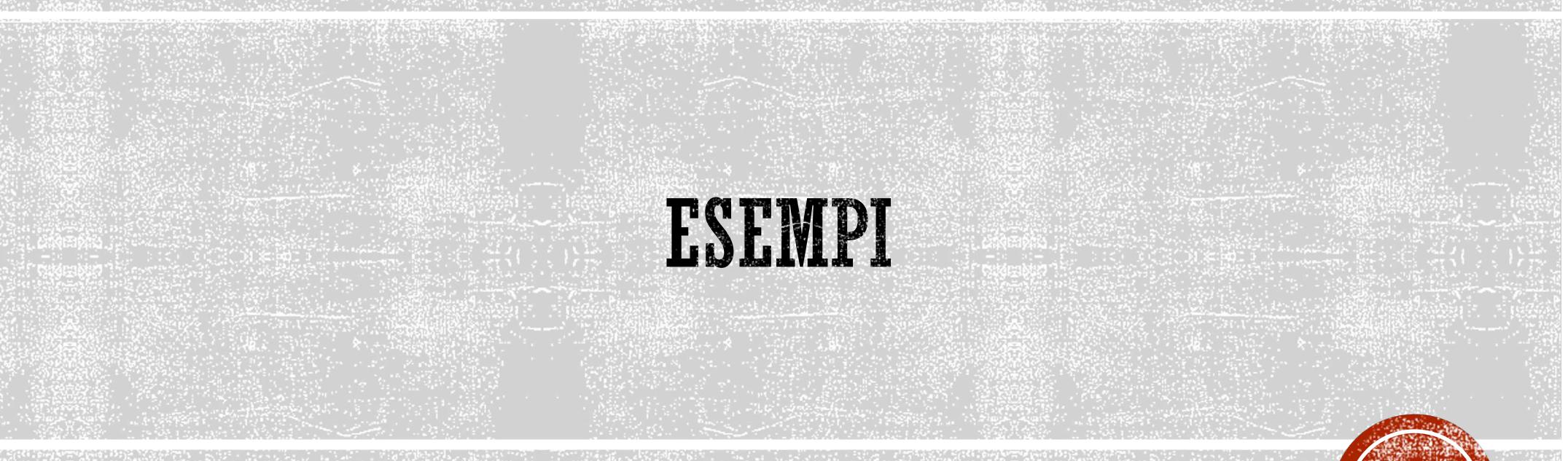
Triangolo **nero** = generalizzazioni con figli che possono sovrapporsi



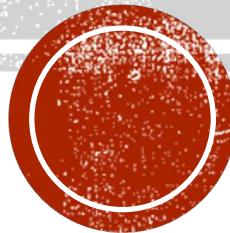
# LE NOTE

- Un diagramma delle classi può essere documentato con delle *note*.
  - Consistono in semplici commenti testuali.
  - Sono riportate sul diagramma in un rettangolo con l'angolo superiore destro ripiegato.
  - Una nota può essere associata ad un particolare elemento del diagramma attraverso una linea tratteggiata.





**ESEMPI**

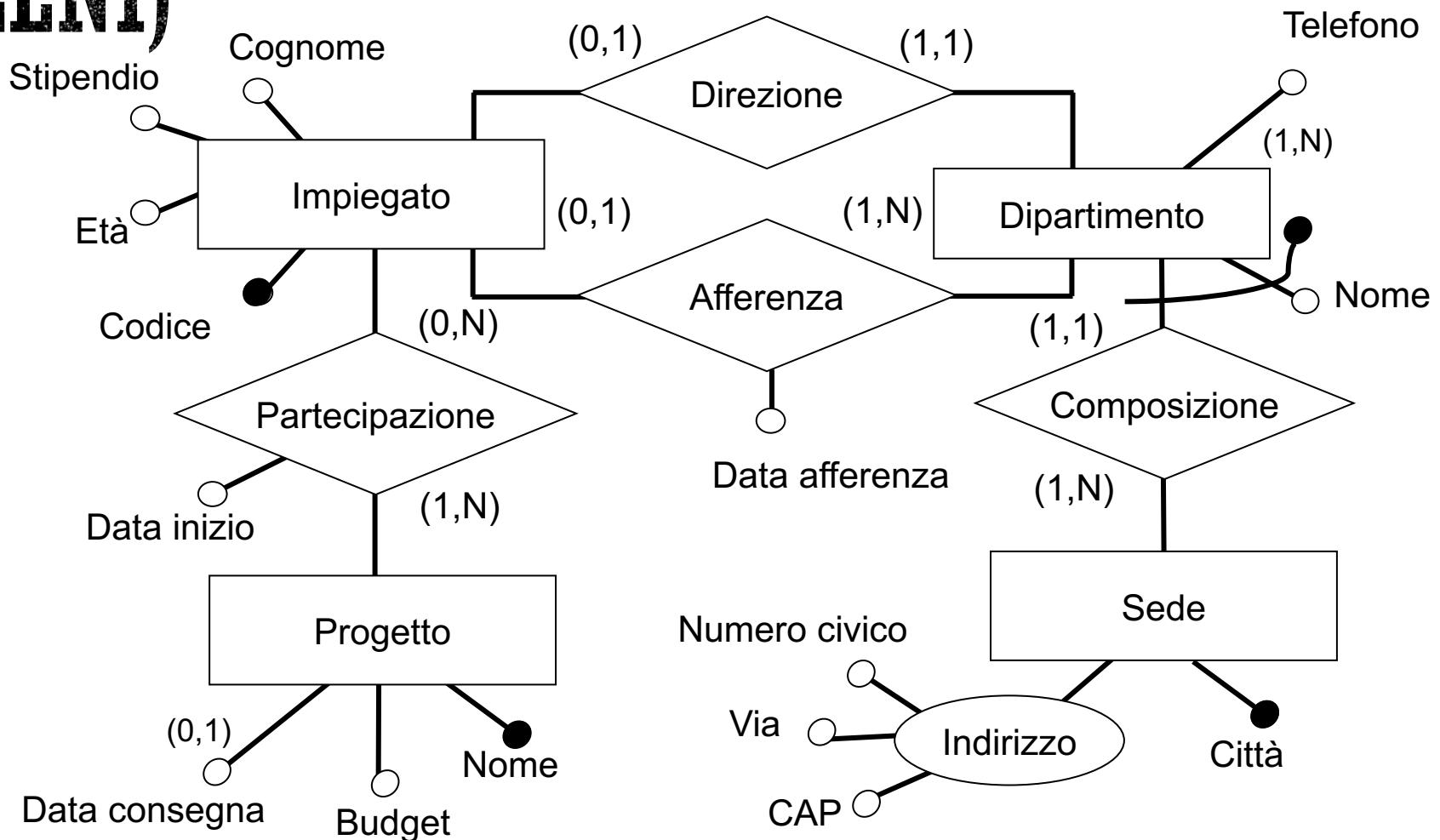


# ESEMPIO

- *Modellare uno schema riguardanti le informazioni di carattere organizzativo relative ad una azienda con diverse sedi.*
  - Una **sede** dell'azienda è identificata dalla città ed è composta da una serie di dipartimenti che non possono essere definiti al di fuori della sede. Una sede ha anche un indirizzo.
  - Un **dipartimento** è identificato dal nome e dalla sede di appartenenza e possiede diversi numeri di telefono.
  - Ai dipartimenti afferiscono (a partire da una certa data) uno o più impiegati; e un impiegato li dirige.
  - Gli **impiegati** vengono rappresentati dal cognome, stipendio, età e un codice che serve per identificarli.
  - Gli impiegati lavorano su zero o più progetti a partire da una certa data.
  - Ogni **progetto** ha un nome, un budget e una data di consegna che può non essere specificata.
  - Una **nota** associata alla classe progetto ne descrive il significato.



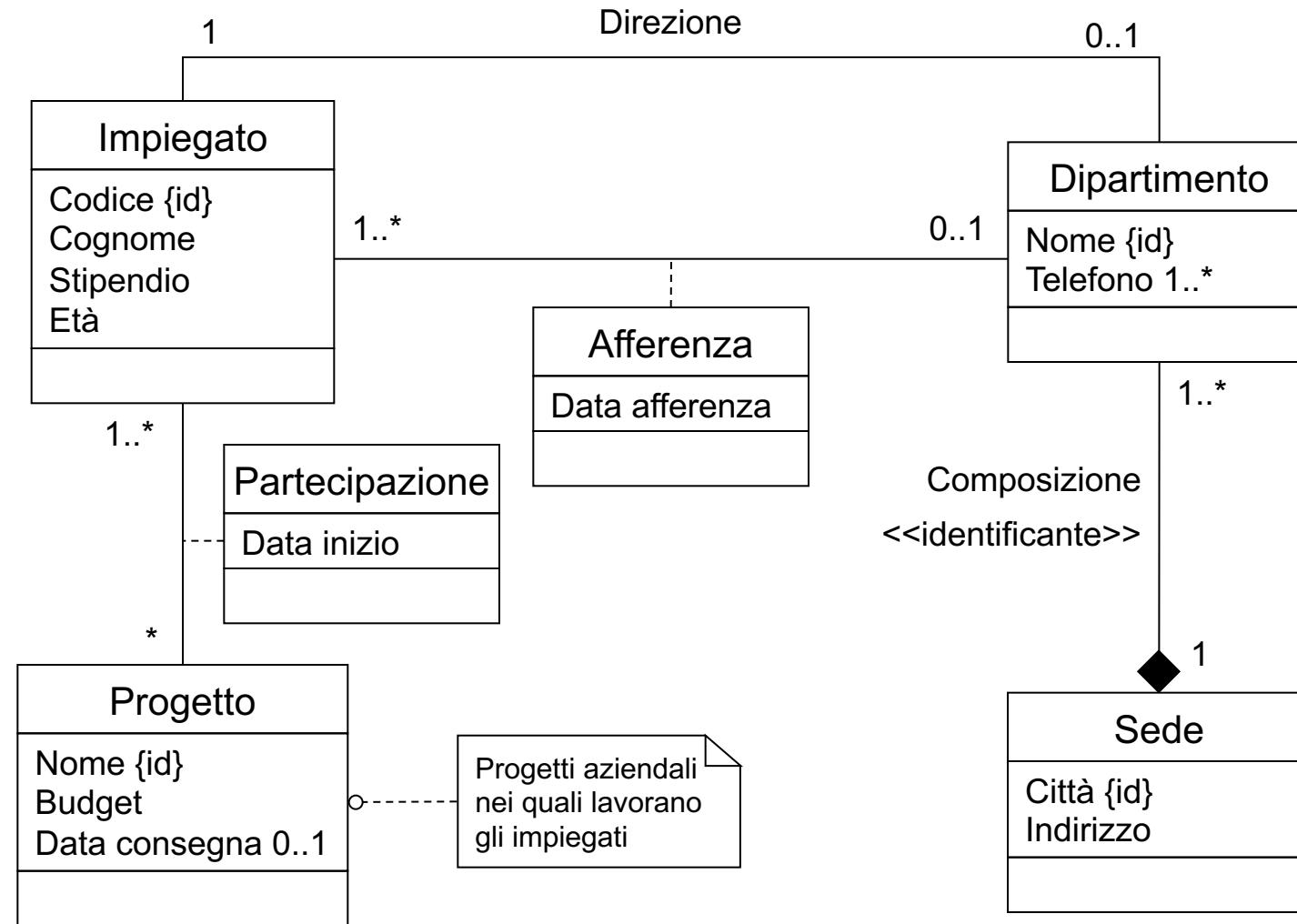
# LO SCHEMA CONCETTUALE (NOTAZIONE ATZENI)



*Indirizzo è stato modellato come attributo composto: Via, Numero civico, CAP*



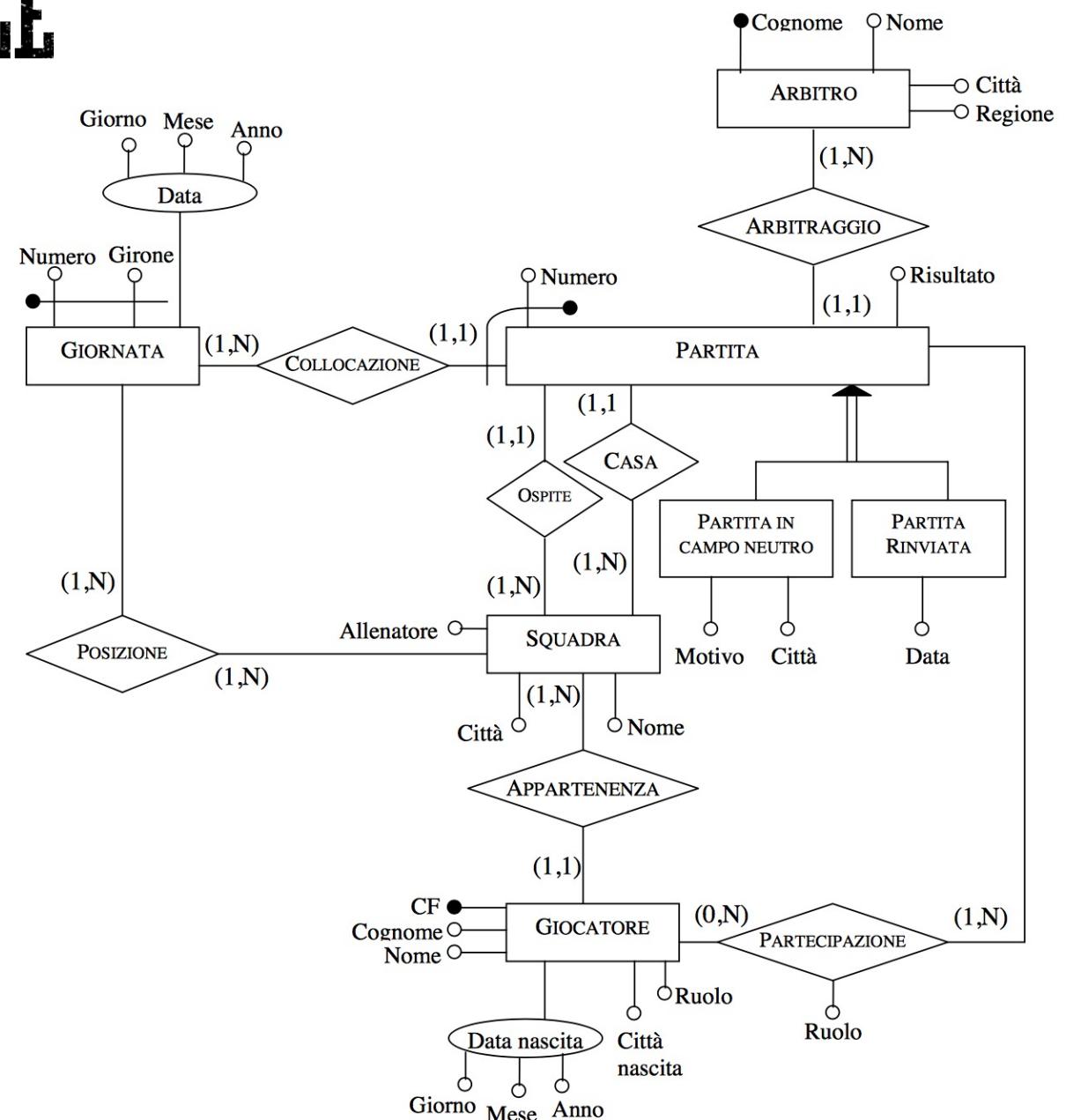
# LO SCHEMA CONCETTUALE IN UML



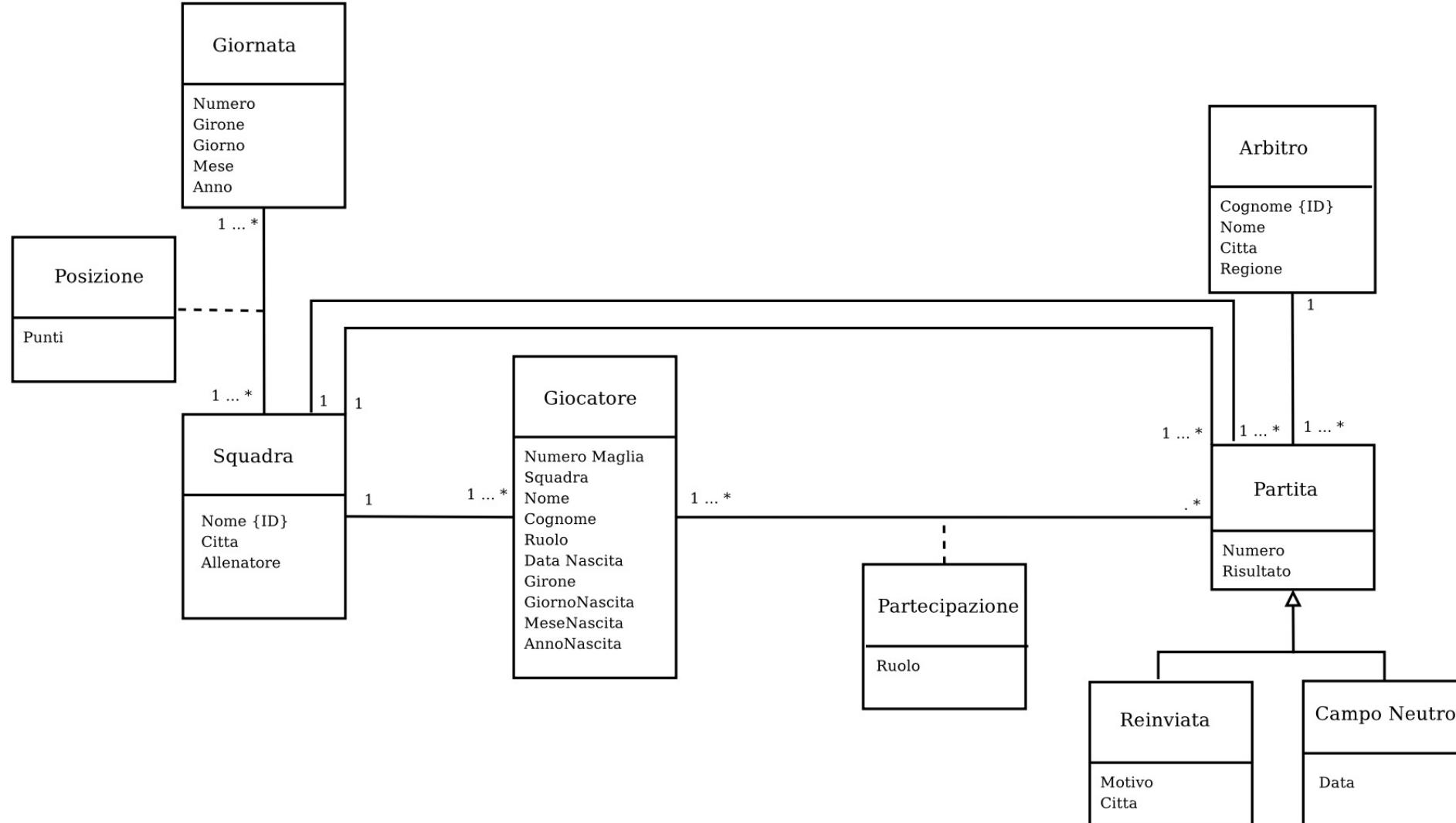
# ESEMPIO 2

- *Modellare uno schema riguardanti le informazioni di un campionato di calcio:*
  - L'entità **SQUADRA** rappresenta tutte le squadre del campionato, indicando per ognuna di esse il nome, la città e il nome dell'allenatore.
  - L'entità **GIOCATORE** rappresenta i giocatori delle squadre: ogni giocatore ha un contratto con una sola squadra e ogni squadra ha più giocatori. I giocatori sono identificati dal loro Codice Fiscale e per ognuno di essi è indicato il nome, il cognome, il ruolo nella squadra, la città di nascita e la data di nascita.
  - Lo schema contiene anche informazioni sulle partite. Una **PARTITA** è identificata con un numero (che deve essere differente per tutte le partite dello stesso giorno) e con un riferimento al giorno (attraverso la relazione COLLOCAZIONE e l'entità giornata).
  - Le relazioni CASA e OSPITE rappresentano le due squadre che giocano la partita: per ogni partita è indicato il risultato e l'**ARBITRO**, con la relazione ARBITRAGGIO tra partita e arbitro; questa entità rappresenta tutti gli arbitri del campionato e per ognuno di essi è indicato il Nome, il Cognome, la Città e la Regione. Un arbitro è rappresentato solo se ha arbitrato almeno una partita.
  - Una partita può essere giocata su campo neutrale o può essere rinviata ad un'altra data (ma questi due eventi non sono ammessi contemporaneamente nello schema).
  - La relazione Partecipazione rappresenta il fatto che un giocatore abbia giocato in una partita, la sua posizione (che può essere diversa dalla sua solita). Lo schema non esprime la condizione che i giocatori che giocano una partita devono avere un contratto con una delle due squadre.
  - L'entità **GIORNATA** rappresenta la giornata del campionato. Sono identificate con Numero e Girone. La relazione Posizione dà il punteggio di ogni squadra in ogni giornata.

# LO SCHEMA CONCETTUALE (NOTAZIONE ATZENI)



# LO SCHEMA CONCETTUALE IN UML





# FINE

Per eventuali domande: (in ordine di preferenza personale)

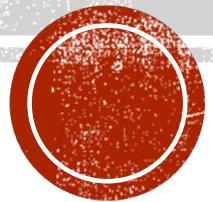
- Ora.
- Chat di Teams
- Mail: [silvio.barra@unina.it](mailto:silvio.barra@unina.it)



# BASI DI DATI I

- Rapido Recap
- Il Modello Relazionale
- I Concetti del Modello Relazionale
- Vincoli nel Modello Relazionale

# RAPIDO RECAP



# RECAP #1

- Cos'è un sistema informativo e dove il sistema di basi di dati agisce all'interno del sistema organizzativo.
- Cos'è un miniworld.
- Cos'è un DBMS e le funzionalità di un DBMS.
- Fasi per la progettazione di un DB
  - Specifica
  - Progettazione Concettuale
  - Progettazione Logica
  - Progettazione Fisica
- Indipendenza tra il programma e i dati
- File Processing VS Database



# RECAP #2

- Modelli dei dati e Astrazione dei dati
- Cosa deve fare un modello
- Le categorie e i vari tipi di data model (alto-livello, rappresentazionali, fisici)
- Schemi e Istanze di DB
- Architetture dei R-DBMS
  - Architettura 3 schema
    - Vista interna
    - Vista logica
    - Vista Esterna
- Mapping tra i vari livelli
- Linguaggi del DBMS
  - DDL, SDL, VDL, DML, SQL

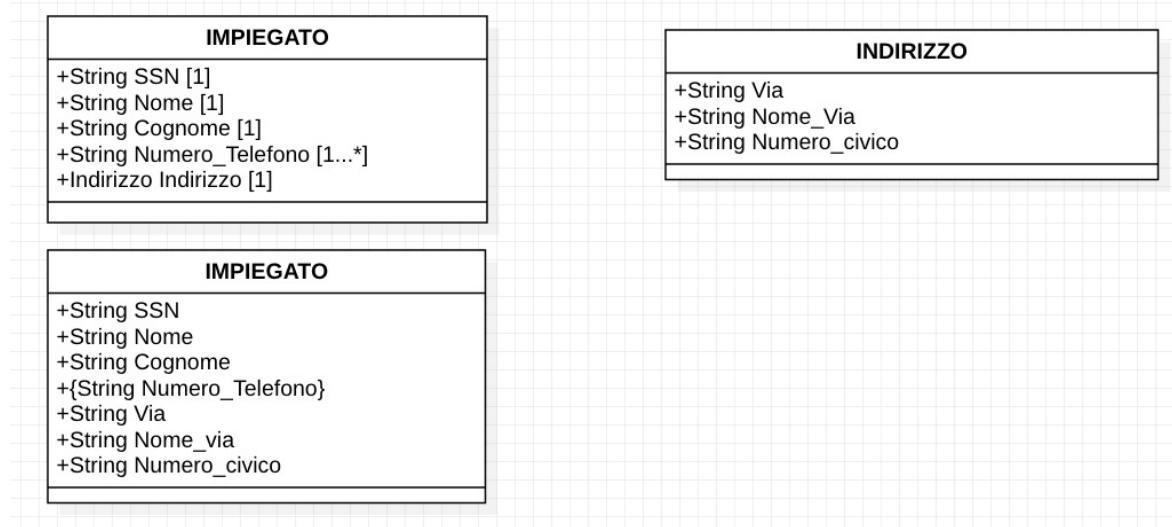


# RECAP #3 – PROGETTAZIONE CONCETTUALE

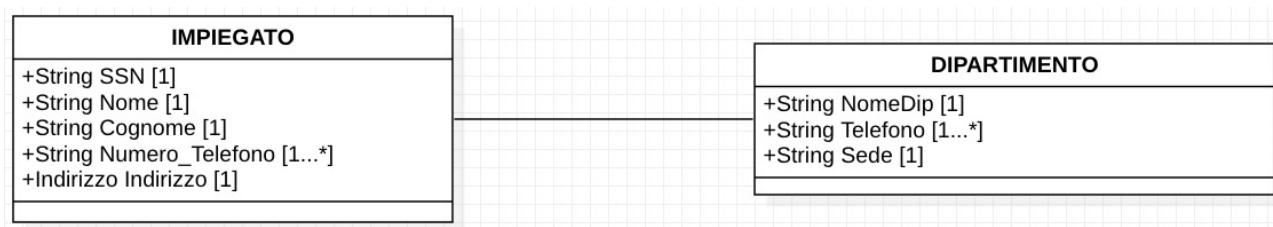
- UML e ER e le differenze tra l'uno e l'altro
  - ER è pensato per la modellazione dei dati: contiene tutti i formalismi necessari per la modellazione dei dati
  - UML è più orientato alla modellazione ad oggetti: è comunque un linguaggio di modellazione e adattabile al contesto dei dati. Non esistono formalismi per tutti i concetti, ma ad ogni modo possiamo stereotipare alcuni concetti per adattarlo alla modellazione dei dati.
  - ER utilizza i concetti di entità e relazioni
  - UML utilizza i concetti di classi e associazioni



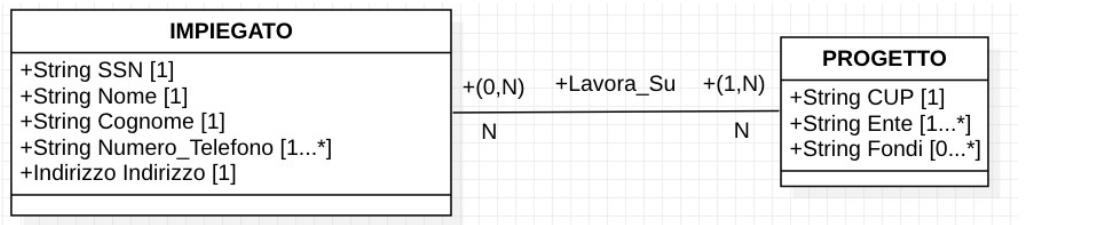
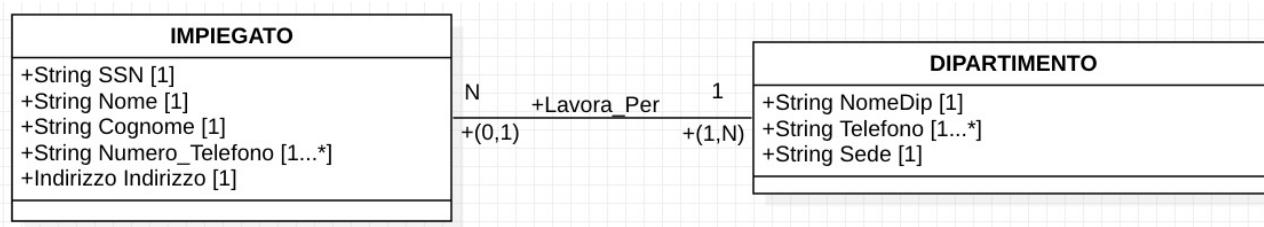
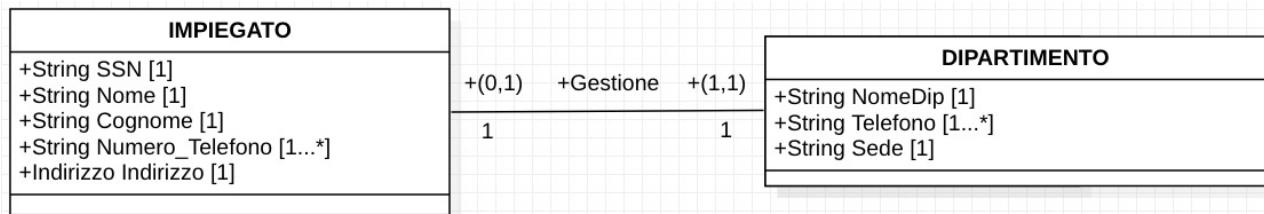
## CLASSE



## ASSOCIAZIONI



## CARDINALITA' DELLE ASSOCIAZIONI



## CARDINALITA' DEGLI ATTRIBUTI (min, max)

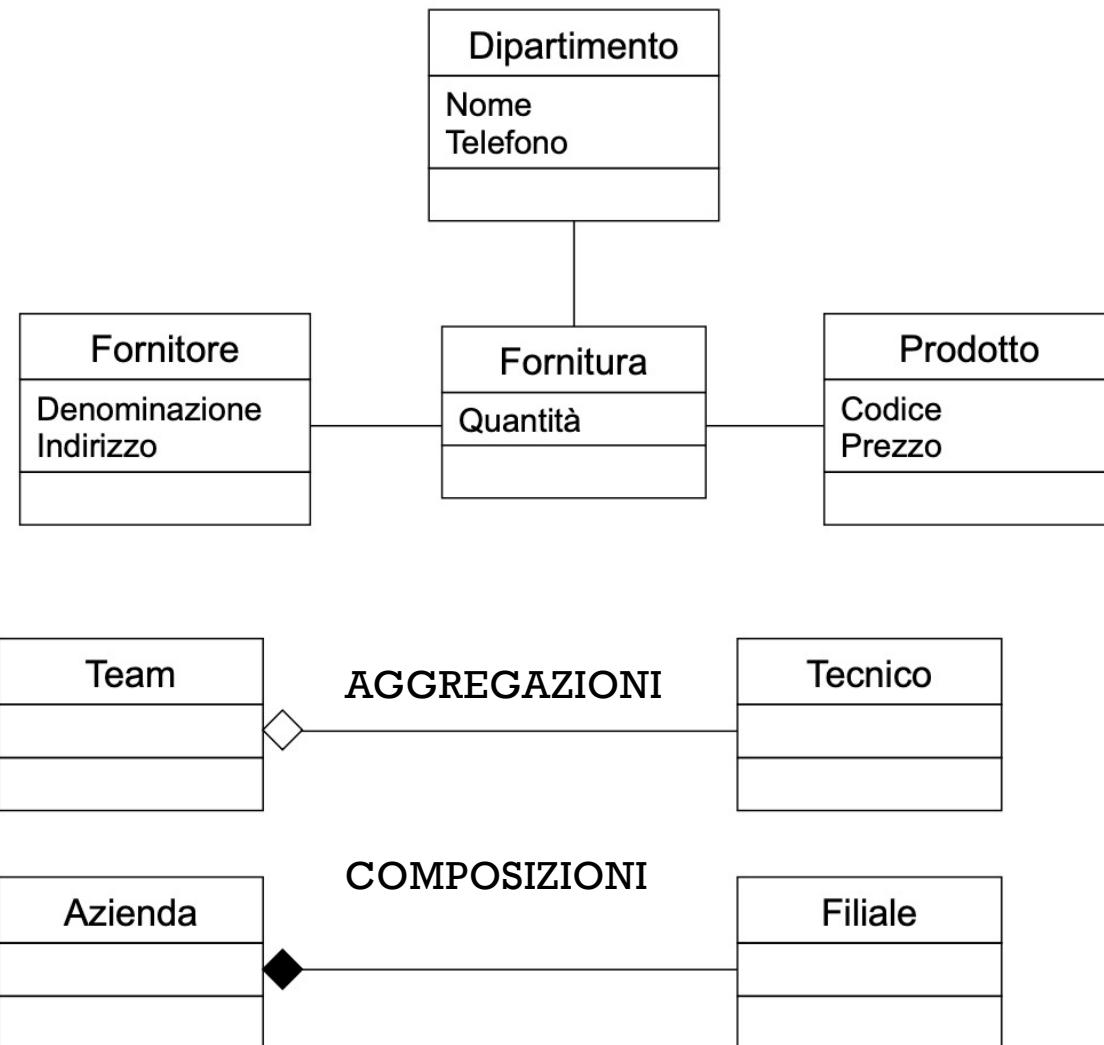
- (0,1): ATTRIBUTO A VALORE SINGOLO PARZIALE (NON è OBBLIGATORIO)
- (1,1): ATTRIBUTO A VALORE SINGOLO TOTALE (OBBLIGATORIO)
- (0,N): ATTRIBUTO A VALORE MULTIPLO PARZIALE (con  $N \geq 1$ )
- (1,N): ATTRIBUTO A VALORE MULTIPLO TOTALE (con  $N \geq 1$ )
- (0,\*): ATTRIBUTO A VALORE MULTIPLO PARZIALE ILLIMITATO
- (1, \*): ATTRIBUTO A VALORE MULTIPLO TOTALE ILLIMITATO

## ATTRIBUTI COMPOSTI

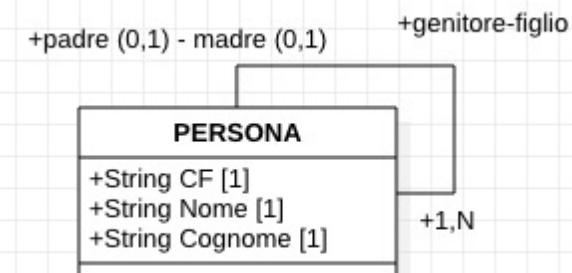
- IN UML NON SONO AMMESSI.
- NON ESISTE UN FORMALISMO PER RAPPRESENTARLI (IN ER IL FORMALISMO LO ABBIAMO)  
AD OGNI MODO, ESSENDO IL LINGUAGGIO UML ALTAMENTE STEREOTIPATO,  
POSSIAMO TROVARE IL MODO PER RAPPRESENTARLI
  - POSSIAMO INSERIRE DELLE NOTE CHE DESCRIVONO LA STRUTTURA DELL'ATTRIBUTO COMPOSTO
  - POSSIAMO DEFINIRE UNA CLASSE CHE RAPPRESENTA COME è COMPOSTO L'ATTRIBUTO COMPOSTO



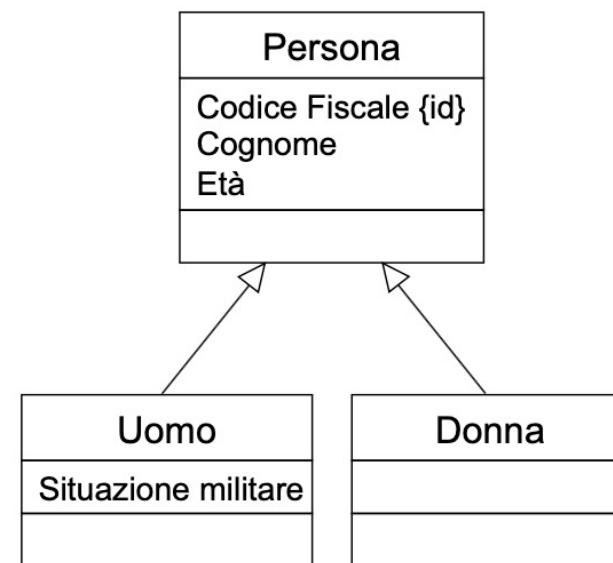
## ASSOCIAZIONI N-ARIE



## ASSOCIAZIONI RICORSIVE

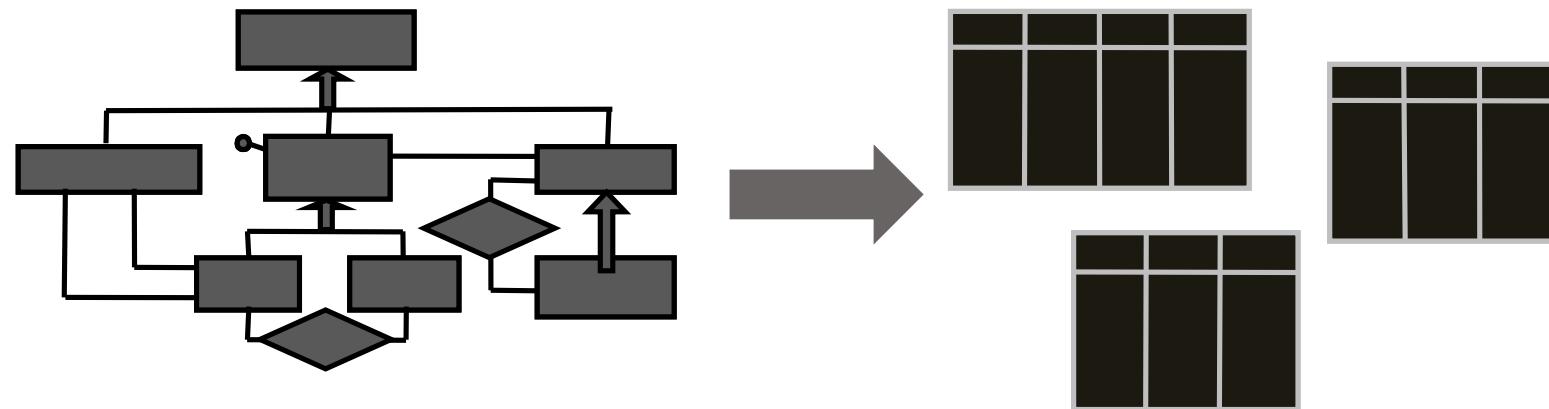


## GENERALIZZAZIONI (E SPECIALIZZAZIONI)

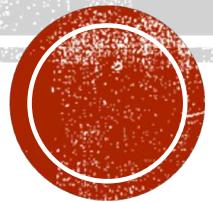


# DA SCHEMA CONCETTUALE A LOGICO

Terminata la fase di analisi concettuale del database e creato un modello di alto livello (UML) che descrive il *miniworld*, passiamo alla definizione di uno schema logico, più vicino al DBMS ma meno comprensibile ai “*non addetti ai lavori*”.



# **IL MODELLO RELAZIONALE**



# **IL DATA MODEL RELAZIONALE**

- Fu proposto da Codd nel 1970 per favorire l'indipendenza dei dati e reso disponibile come modello logico in DBMS reali nel 1981.
- È il modello più diffuso, sia a livello teorico che commerciale.
- La forza del modello relazionale è nella sua semplicità e nei solidi formalismi matematici su cui si poggia.



# IL DATA MODEL RELAZIONALE (2)

- Si basa sul concetto matematico di **Relazione**.
- Le relazioni hanno una rappresentazione naturale per mezzo di **tabelle**:
  - Ciascuna riga rappresenta una collezione di valori di dati relati.
- Il database è rappresentato come una collezione di relazioni.



# IL DATA MODEL RELAZIONALE - ESEMPIO

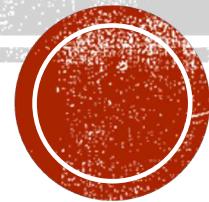
EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
John	B	Smith	123456789	09-JAN-55	731 Fondren, Houston, TX	M	30000	333445555	5	
Franklin	T	Wong	333445555	08-DEC-45	638 Voss, Houston, TX	M	40000	888665555	5	
Alicia	J	Zelaya	999887777	19-JUL-58	3321 Castle, Spring, TX	F	25000	987654321	4	
Jennifer	S	Wallace	987654321	20-JUN-31	291 Berry, Bellaire, TX	F	43000	888665555	4	
Ramesh	K	Narayn	666884444	15-SEP-52	975 Fire Oak, Humble, TX	M	38000	333445555	5	
Joyce	A	English	453453453	31-JUL-62	5631 Rice, Houston, TX	F	25000	333445555	5	
Ahmad	V	Jabbar	987987987	29-MAR-59	980 Dallas, Houston, TX	M	25000	987654321	4	
James	E	Borg	888665555	10-NOV-27	450 Stone, Houston, TX	M	55000	null	1	

DEPARTMENT	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE	DEPT_LOCATIONS	DNUMBER	DLOCATION
Research		5	333445555	22-MAY-78		1	Houston
Administration		4	987654321	01-JAN-85		4	Stafford
Headquarters		1	888665555	19-JUN-71		5	Bellaire
						5	Sugarland
						5	Houston

**Esempio** di una parte del database “Company” nel data model Relazionale.



# **CONCETTI DEL MODELLO RELAZIONALE**



# RELAZIONE DAL PUNTO DI VISTA MATEMATICO

- Siano  $D_1, D_2, \dots, D_n$   $n$  insiemi.
- Il prodotto cartesiano  $D_1 \times D_2 \times \dots \times D_n$ , è l'insieme di tutte le  $n$ -uple ordinate  $(d_1, d_2, \dots, d_n)$  tali che  
 $d_1 \in D_1, d_2 \in D_2, \dots, d_n \in D_n$ .
- Una **relazione matematica** su  $D_1, D_2, \dots, D_n$  è un **sottoinsieme** del prodotto cartesiano  $D_1 \times D_2 \times \dots \times D_n$ .
- $D_1, D_2, \dots, D_n$  sono i **domini** della relazione. Una relazione su  $n$  domini ha grado  $n$ .
- Il numero di  $n$ -uple è la **cardinalità** della relazione. Nelle applicazioni reali, la cardinalità è sempre finita.



# RELAZIONE MATEMATICA - ESEMPIO

$$D_1 = \{a, b\}; D_2 = \{x, y, z\}$$

Prodotto cartesiano  $D_1 \times D_2$

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

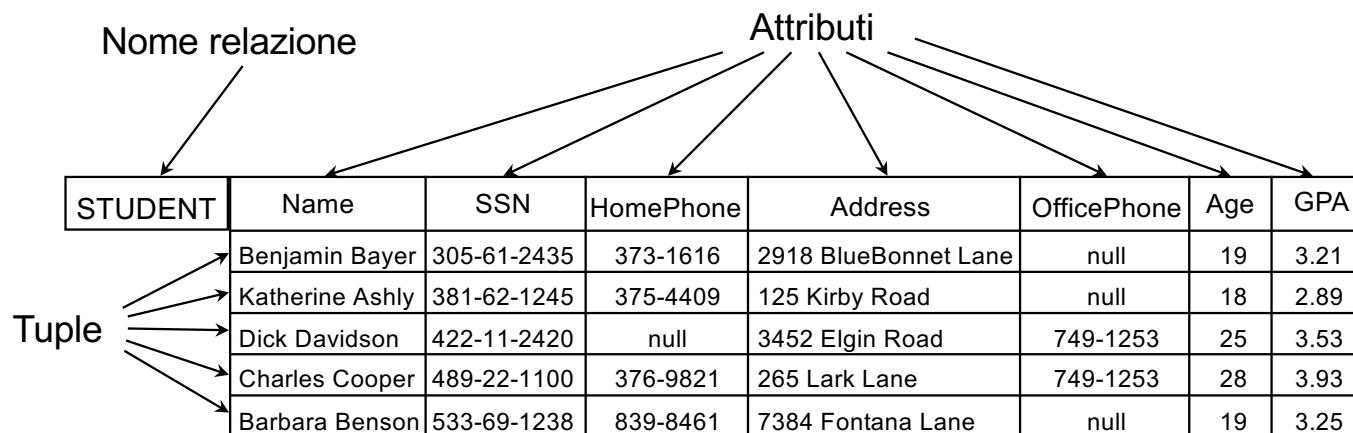
Una relazione  $r \subseteq D_1 \times D_2$

a	x
a	y
b	y
b	z



# RELAZIONI NEL MODELLO RELAZIONALE

- A ogni dominio (attributo) è associato un nome, unico nella relazione, che “*descrive*” il ruolo del dominio.
- L’ordinamento fra gli attributi è irrilevante:
  - la struttura è **non posizionale**.
- Nella terminologia del modello relazionale, una riga è detta **tupla**:



**Esempio** di Relazione



# **DOMINI, ATTRIBUTI, TUPLE E RELAZIONI**

- Nel modello relazionale, un dominio D è un insieme di valori atomici, cioè indivisibili.
- Un metodo per specificare un dominio è specificare un tipo di dato da cui sono presi i dati che formano il dominio.



# ESEMPI DI DOMINI

- **Usa\_Phone\_Numbers**: insieme di numeri a 10 cifre che rappresentano numeri telefonici validi negli stati uniti.
- **Social\_Security\_Number**: insieme di SSN validi, di 9 cifre.
- **Names**: insieme di nomi di persone.
- **Employee\_Ages**: possibile età dei dipendenti, da 16 a 80 anni.
- **Academic\_Department**: insieme di dipartimenti universitari (matematica e informatica, economia, ecc...).



# DOMINI

- Per ogni dominio viene specificato un tipo di dato (o formato).
  - *Esempio:* `USA_PHONE_NUMBER` può essere dichiarato come una stringa **(ddd)ddd-dddd**.
- Potrebbe essere necessario specificare l'unità di misura per interpretare i valori di un dominio.
  - *Esempio:* `peso_persona` è espresso con l'unità di peso **kg**.



# SCHEMI DI RELAZIONE

- Uno **schema di relazione**, denotato da  $R(A_1, A_2, \dots, A_n)$ , descrive una relazione.
- Uno schema di relazione è formato da:
  - Un nome di relazione  $R$ ;
  - Una lista di attributi  $(A_1, A_2, \dots, A_n)$ .
- Ciascun  $A_i$  è il nome di un ruolo giocato da qualche dominio  $D$  nello schema  $R$ .
- $D$  è detto dominio di  $A_i$ :  $D = \text{Dom}(A_i)$ .
- Il **grado** di una relazione è il numero di attributi,  $n$ , del suo schema di relazione.



# SCHEMI DI RELAZIONE - *ESEMPIO*

Nome della relazione: Student

STUDENT						
Name	SSN	HomePhone	Address	OfficePhone	Age	GPA

- Grado 7
- Dom(Name)=Names
- Dom(SSN)=Social\_Security\_Numbers



# ISTANZE DI RELAZIONE

- Una relazione (o istanza di relazione)  $r$  dello schema  $R(A_1, A_2, \dots, A_n)$ , denotata  $r(R)$   
è un insieme di tuple  $r = \{t_1, t_2, \dots, t_n\}$ .
- Ogni  $t_i$  è una lista ordinata di  $n$  valori  $t = \langle v_1, v_2, \dots, v_n \rangle$  dove ciascun  $v_i \in \text{dom}(A_i) \cup \{\text{null}\}$ 
  - Intensione della relazione  $\rightarrow R$  (schema)
  - Estensione della relazione  $\rightarrow r(R)$  istanza di relazione



# **SCHEMI E ISTANZE DI RELAZIONE - ESEMPIO**

**STUDENT**

Name	SSN	HomePhone	Address	OfficePhone	Age	GPA
------	-----	-----------	---------	-------------	-----	-----

Schema di relazione

Benjamin Bayer	305-61-2435	373-1616	2918 BlueBonnet Lane	null	19	3.21
Katherine Ashly	381-62-1245	375-4409	125 Kirby Road	null	18	2.89
Dick Davidson	422-11-2420	null	3452 Elgin Road	749-1253	25	3.53
Charles Cooper	489-22-1100	376-9821	265 Lark Lane	749-1253	28	3.93

Istanza di relazione



# CARATTERISTICHE DI UNA RELAZIONE

- L'ordinamento delle tuple di una relazione non è parte della definizione.

La definizione non specifica alcun ordine.

- Una definizione alternativa di relazione considera non significativo anche l'ordine degli attributi;

In accordo a tale definizione una tupla può essere considerata come un insieme di coppie (**<attributo>**,**<valore>**).



# CARATTERISTICHE DI UNA RELAZIONE (2)

- Relazioni equivalenti, con diversi ordinamenti di righe e colonne:

**STUDENT**

Name	SSN	Home-Phone	Address	Office-Phone	Age	GPA
Benjamin Bayer	305-61-2435	373-1616	2918 BlueBonnet Lane	null	19	3.21
Katherine Ashly	381-62-1245	375-4409	125 Kirby Road	null	18	2.89
Dick Davidson	422-11-2420	null	3452 Elgin Road	749-1253	25	3.53
Charles Cooper	489-22-1100	376-9821	265 Lark Lane	749-1253	28	3.93

**STUDENT**

Name	Home-Phone	SSN	Age	Address	Office-Phone	GPA
Charles Cooper	376-9821	489-22-1100	28	265 Lark Lane	749-1253	3.93
Katherine Ashly	375-4409	381-62-1245	18	125 Kirby Road	null	2.89
Dick Davidson	null	422-11-2420	25	3452 Elgin Road	749-1253	3.53
Benjamin Bayer	373-1616	305-61-2435	19	2918 BlueBonnet Lane	null	3.21



# SCHEMA DI DATABASE RELAZIONALE

- Uno **schema** di database relazionale è un insieme di schemi di relazione:

$$S_0\{R_1, R_2, \dots, R_n\}$$

- Una **istanza** di database relazionale DB di S è un insieme di istanze di relazione

$DB = \{r_1, r_2, \dots, r_m\}$  tale che  $r_i$  è una istanza di  $R_i$ .



# SCHEMA DI DATABASE RELAZIONALE (2)

**EMPLOYEE**

FNAME	MINIT	LNAME	<u>SSN</u>	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
-------	-------	-------	------------	-------	---------	-----	--------	----------	-----

**DEPARTMENT**

DNAME	<u>DNUMBER</u>	MGRSSN	MGRSTARTDATE
-------	----------------	--------	--------------

**DEPT\_LOCATION**

<u>DNUMBER</u>	DLOCATION
----------------	-----------

**PROJECT**

PNAME	<u>PNUMBER</u>	PLOCATION	DNUM
-------	----------------	-----------	------

**WORKS ON**

<u>ESSN</u>	PNO	HOURS
-------------	-----	-------

**DEPENDENT**

<u>ESSN</u>	DEPARTMENT_NAME	SEX	BDATE	RELATIONSHIP
-------------	-----------------	-----	-------	--------------

Lo schema del database 'Company'



# ISTANZA DI DATABASE RELAZIONALE

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
John	Smith		123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5	
Franklin	Wong		333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5	
Alicia	Zelaya		999887777	1968-01-19	3321 Castle Spring, TX	F	25000	987654321	4	
Jennifer	Wallace		987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4	
Ramesh	Narayan		666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5	
Joyce	English		453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5	
Ahmad	Jaber		987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4	
James	Borg		888665555	1937-11-10	450 Stone, Houston, TX	M	56000	null	1	

DEPARTMENT	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE	DEPT_LOCATIONS	DNUMBER	DLOCATION
Research		5	333445555	1988-05-22			Houston
Administration		4	987654321	1995-01-01			Stafford
Headquarters		1	888665555	1981-06-19			Bellaire
							Sugarland

WORKS_ON	ESSN	PNO	HOURS
	123456789	1	32.5
	123456789	2	7.5
	666884444	3	40.0
	453453453	1	20.0
	453453453	2	20.0
	333445555	2	10.0
	333445555	3	10.0
	333445555	10	10.0
	333445555	20	10.0
	999887777	30	30.0
	999887777	10	10.0
	987987987	10	35.0
	987987987	30	5.0
	987654321	30	20.0
	987654321	20	15.0
	888665555	20	null

PROJECT	PNAME	PNUMBER	PLOCATION	DNUM
ProductX		1	Bellaire	5
ProductY		2	Sugarland	5
ProductZ		3	Houston	5
Computerization		10	Stafford	4
Reorganization		20	Houston	1
Newbenefits		30	Stafford	4

DEPENDENT	ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP
	333445555	Alice	F	1986-04-05	DAUGHTER
	333445555	Theodore	M	1983-10-25	SON
	333445555	Joy	F	1958-05-03	SPOUSE
	987654321	Abner	M	1942-02-28	SPOUSE
	123456789	Michael	M	1988-01-04	SON
	123456789	Alice	F	1988-12-30	DAUGHTER
	123456789	Elizabeth	F	1967-05-05	SPOUSE

Un'istanza del database "Company"



# NOTAZIONI DEL MODELLO RELAZIONALE

- Uno schema di relazione  $R$ , di grado  $n$ , è denotato da

$R(A_1, A_2, \dots, A_n)$ .

- Una tupla  $t$  in una relazione  $r(R)$  è denotata da

$t = <v_1, v_2, \dots, v_n>$  dove  $v_i$  è il valore per l'attributo  $A_i$

- $t[A_i]$  si riferisce al valore  $v_i$  per l'attributo  $A_i$ ;
- $t[A_u, A_w, \dots, A_z]$  dove  $A_u, A_w, \dots, A_z$  è una lista di attributi di  $R$  e riferisce alle sottotuple di valori  $v_u, v_w, \dots, v_z$ .

- Le lettere **Q**, **R**, **S** denotano nomi di relazioni.
- Le lettere **q**, **r**, **s** denotano stati di relazioni.
- Le lettere **t**, **u**, **v** denotano tuple.



# VINCOLI NEL MODELLO RELAZIONALE



# VINCOLI DEL MODELLO RELAZIONALE

- Nel modello relazionale, i valori presenti in un'istanza di relazione devono soddisfare una serie di vincoli:
  1. Vincoli di dominio
  2. Vincoli di chiave
  3. Vincoli di integrità di entità
  4. Vincoli di integrità referenziale



# VINCOLI DI DOMINIO

- Il valore di ciascun attributo di  $A$  deve essere un valore atomico  
*{carattere, stringa a lunghezza fissa e variabile, data, ora, valuta, ecc...}*  
appartenente a  $\text{dom}(A)$ .



## VINCOLI DI CHIAVE – DEFINIZIONE DI SUPERCHIAVE

- Una relazione è definita come un **insieme di tuple**. Per definizione tutti gli elementi di un insieme sono distinti, quindi tutte le tuple devono essere **distinte**.
- Devono allora esistere dei sottoinsiemi di attributi con la proprietà di non avere la stessa combinazione di valori in più tuple.  
Sia  $sk$  un tale sottoinsieme di attributi di  $R$ :

$$t_1[sk] \neq t_2[sk]$$

- L'insieme di attributi  $sk$  è detto **superchiave** di  $R$ .



# VINCOLI DI CHIAVE – DEFINIZIONE DI CHIAVE

- Formalmente, una **chiave k** di uno schema di relazione R è una superchiave tale che, rimovendo uno dei suoi attributi, non è più una superchiave.
  - **k** è detta anche **superchiave minimale**.
- Informalmente, una chiave **k** è un insieme di attributi minimale che permette di identificare univocamente una tupla.



## VINCOLI DI CHIAVE – DEFINIZIONE DI CHIAVE (2)

- In una relazione possono esistere più chiavi, dette **chiavi candidate**:
  - in tal caso se ne sceglie una, detta **chiave primaria**.
- Una chiave deve godere anche delle proprietà di **invarianza nel tempo**.
- **Esempio** di chiave:

Name	SSN	HomePhone	Address	OfficePhone	Age	GPA
Benjamin Bayer	305-61-2435	373-1616	2918 BlueBonnet Lane	null	19	3.21
Katherine Ashly	381-62-1245	375-4409	125 Kirby Road	null	18	2.89
Dick Davidson	422-11-2420	null	3452 Elgin Road	749-1253	25	3.53
Charles Cooper	489-22-1100	376-9821	265 Lark Lane	749-1253	28	3.93
Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	null	19	3.25

- $\{SSN\}$  è una chiave. Ogni insieme di attributi che include SSN è una superchiave.



# VINCOLI DI CHIAVE

- In una relazione R, non possono esistere valori duplicati per attributi chiave **k**.



# VINCOLI DI INTEGRITÀ DI ENTITÀ

- Nessun valore di chiave primaria può essere “**null**”.
- Questo perché:
  - Se ciò fosse permesso, non si avrebbe modo di identificare l'entità descritta nella tupla.
  - Non si vogliono memorizzare informazioni su entità non identificabili.



# VINCOLI DI INTEGRITÀ REFERENZIALE

- Specificati tra due relazioni, sono usati per **mantenere consistenza** tra tuple delle due relazioni.
- *Informalmente:* una tupla di una relazione, che riferisce ad una tupla di un'altra relazione, deve riferire ad una tupla esistente.
  - *È il concetto portante del modello relazionale!*



## VINCOLI DI INTEGRITÀ REFERENZIALE - ESEMPIO

- L'attributo *DNO* di Employee deve riferire ad un *DNUMBER* esistente nella relazione Department.
- La relazione Employee è detta essere **relata** a Department tramite l'attributo *DNO*.

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
John	B	Smith	123456789	09-JAN-55	731 Fondren, Houston, TX	M	30000	333445555		5
Franklin	T	Wong	333445555	08-DEC-45	638 Voss, Houston, TX	M	40000	888665555		5
Alicia	J	Zelaya	999887777	19-JUL-58	3321 Castle, Spring, TX	F	25000	987654321		4
Jennifer	S	Wallace	987654321	20-JUN-31	291 Berry, Bellaire, TX	F	43000	888665555		4
Ramesh	K	Narayn	666884444	15-SEP-52	975 Fire Oak, Humble, TX	M	38000	333445555		5
Joyce	A	English	453453453	31-JUL-62	5631 Rice, Houston, TX	F	25000	333445555		5
Ahmad	V	Jabbar	987987987	29-MAR-59	980 Dallas, Houston, TX	M	25000	987654321		4
James	E	Borg	888665555	10-NOV-27	450 Stone, Houston, TX	M	55000	null		1

DEPARTMENT	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
Research		5	333445555	22-MAY-78
Administration		4	987654321	01-JAN-85
Headquarters		1	888665555	19-JUN-71



## VINCOLI DI INTEGRITÀ REFERENZIALE – DEFINIZIONE DI CHIAVE ESTERNA

- *Formalmente:* un insieme di attributi  $FK$  in uno schema di relazione  $R_i$  è una **chiave esterna** se vale:
  1. gli attributi in  $FK$  hanno lo stesso dominio degli attributi della chiave primaria  $PK$  di un altro schema di relazione  $R_j$  (*gli attributi in  $FK$  riferiscono alla relazione  $R_j$* )
  2. Un valore di  $FK$  in una tupla  $t_i$  di  $R_i$  o occorre come un valore di  $PK$  per qualche tupla  $t_j$  di  $R_j$  o è **null**.  
 $t_i[FK] = t_j[PK]$  oppure  $t_i[FK] = null$



## VINCOLI DI INTEGRITÀ REFERENZIALE (2)

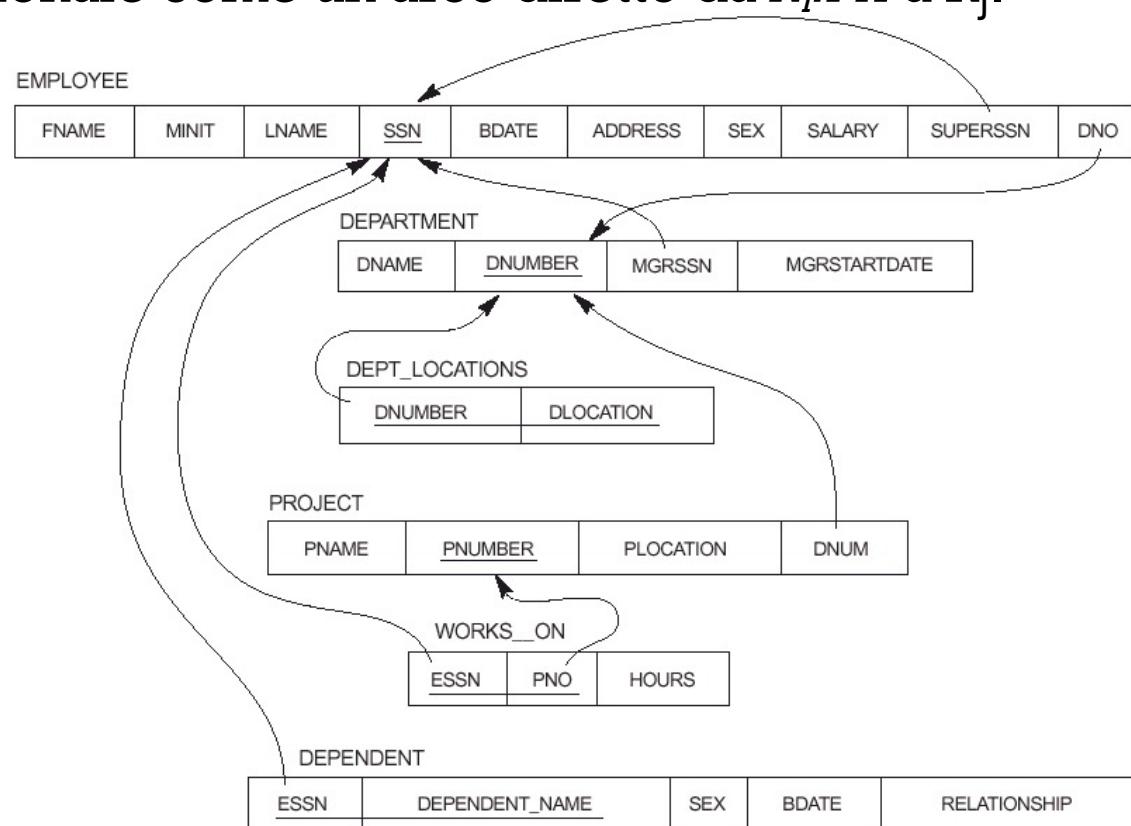
- Le chiavi esterne sono simili al concetto dei “*puntatori*” in C, che o riferiscono ad una variabile allocata o sono *null*.
- L’attributo *DNO* di *Employee* è una chiave esterna, poiché rispetta le condizioni appena elencate.
- Una tupla  $t_i$  di una relazione  $R_i$  è detta **referenziare** una tupla  $t_j$  di una relazione  $R_j$  se vale

$$t_i[FK] = t_j[PK].$$



## VINCOLI DI INTEGRITÀ REFERENZIALE - ESEMPIO

- Un vincolo di integrità referenziale può essere mostrato in uno schema relazionale come un arco diretto da  $R_i.FK$  a  $R_j$ .



# VINCOLI E OPERAZIONI DI AGGIORNAMENTO SU RELAZIONI (INSERT)

- Insert può violare tutti e quattro i tipi di vincoli:
  - Dominio
    - Un valore di un attributo può non apparire nel corrispondente dominio.
  - Chiave
    - Il valore della chiave nella nuova tupla già esistente nella relazione  $r(R)$ .
  - Integrità di entità
    - La chiave primaria è inserita a *null*.
  - Integrità referenziale
    - Il valore di una chiave esterna riferisce ad una tupla che non esiste nella relazione referenziata.



## **VINCOLI E OPERAZIONI DI AGGIORNAMENTO SU RELAZIONI (INSERT) (2)**

- **Come gestire la violazione?**
  - Forzare l'inserimento completo (della relazione riferita);
  - Rifiutare l'inserimento.
- Nel primo caso la violazione può riguardare in cascata l'inserimento su altre relazioni.



# VINCOLI E OPERAZIONI DI AGGIORNAMENTO SU RELAZIONI (DELETE)

- La delete può violare solo l'integrità referenziale.
- Gestione violazione:
  - Rigettare la cancellazione.
  - Tentare di propagare la cancellazione.
  - Modificare i valori dell'attributo referenziante (posto a *null*).
  - Combinazioni delle tre (ed il DBMS dovrebbe permettere all'utente di specificare la gestione).



## **VINCOLI E OPERAZIONI DI AGGIORNAMENTO SU RELAZIONI (MODIFY)**

- Nessun problema per attributi che non sono né chiave primaria né chiave esterna.
- Modifica chiave primaria:
  - Analogico a cancellare una tupla e inserirne un'altra.
- Modifica chiave esterna:
  - Il dbms deve verificare che riferisca ad una tupla esistente.

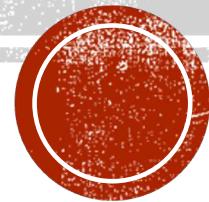


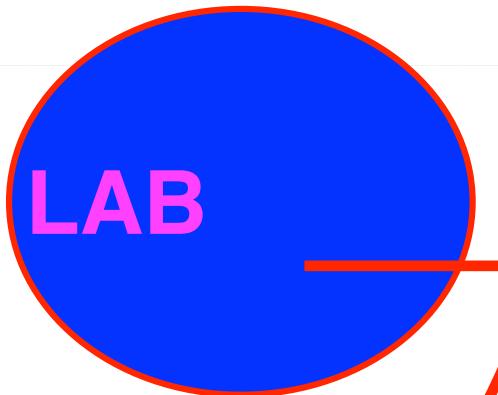


# FINE

Per eventuali domande: (in ordine di preferenza personale)

- Ora.
- Chat di Teams
- Mail: [silvio.barra@unina.it](mailto:silvio.barra@unina.it)





Testo



Lezione 06

# BASI DI DATI I

A.A. 2021/2022

Prof. Adriano Peron

Prof. Silvio Barra

# TRADUZIONE VERSO UN MODELLO LOGICO

- **Modello relazionale:**

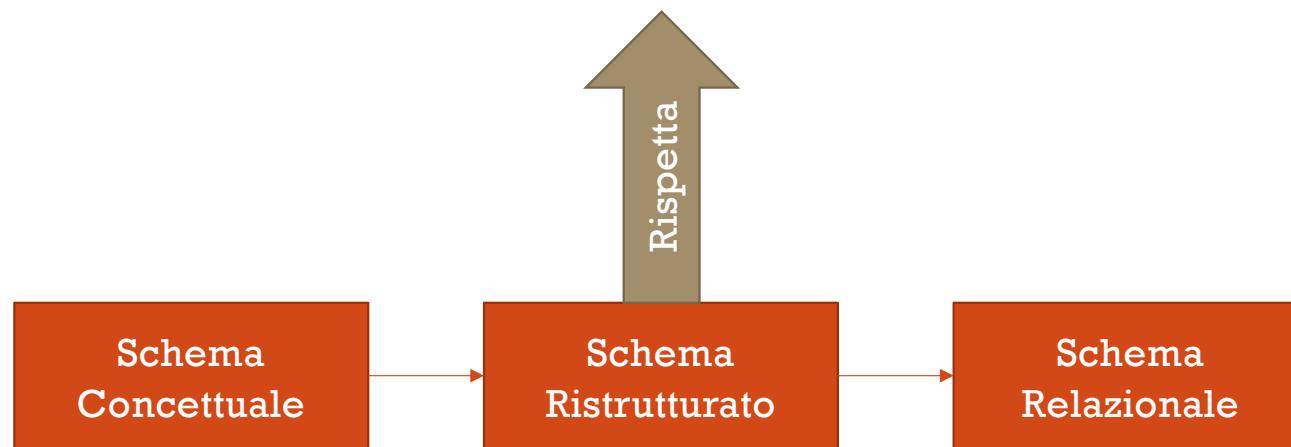
- Per ogni entità, uno schema di relazione con lo stesso nome avente i medesimi attributi dell'entità e per chiave il suo identificatore.
- Per ogni relazione nel modello UML, uno schema di relazione con lo stesso nome avente per attributi gli attributi della relazione e per chiave gli identificatori delle entità coinvolte:

- **Distinguere i diversi i casi in base ai vincoli di partecipazione delle entità coinvolte.**



# RESTRIZIONI

1. Non ci sono gerarchie (Generalizzazioni, Specializzazioni)
2. Non ci sono attributi multipli all'interno delle entità
3. Non ci sono attributi strutturati all'interno delle entità



# PASSAGGI PER IL MAPPING

1. Codificare le entità
2. Individuare la chiave primaria
3. Codificare le associazioni



# MAPPING DELLE ENTITÀ (CASO GENERALE)

A
+A1
+A2
+...
+...
+An

$A (\underline{A_1, A_2, \dots, A_k}, \dots, A_n)$   
 $PK$



# ESEMPIO

STUDENTE
+String MATRICOLA
+String CODICE_FISCALE
+String NOME
+String COGNOME
+Date DATA_N
+String ANNO

Ho 2 chiavi candidate

- Matricola
- Codice\_Fiscale

STUDENTE(Matricola, Codice\_Fiscale, Nome, Cognome, Data\_N, Anno)

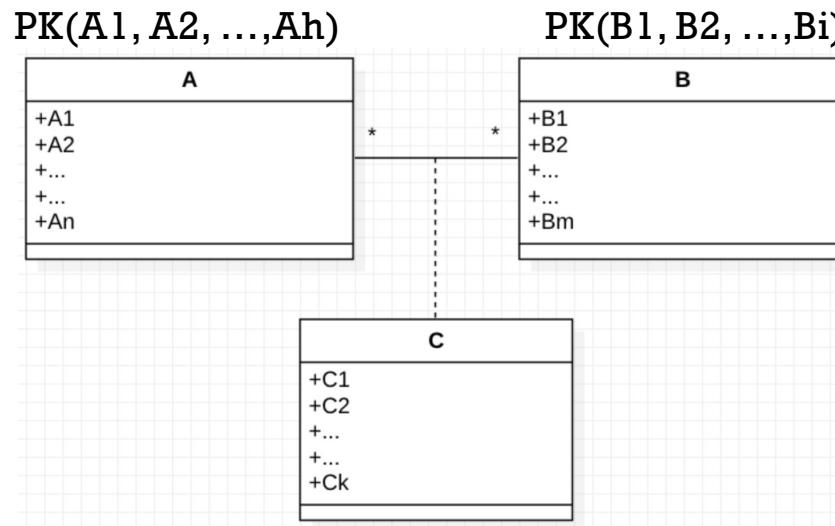


# MAPPING ASSOCIAZIONI

- Anche le associazioni vanno codificate come relazioni
- Per la codifica delle associazioni devo fare attenzione a determinati aspetti
  - La cardinalità delle associazioni
  - I vincoli di partecipazione delle entità alle associazioni
  - Il tipo di entità



# ASSOCIAZIONI M:N (CASO GENERALE)



A(A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>h</sub>, ..., A<sub>n</sub>)

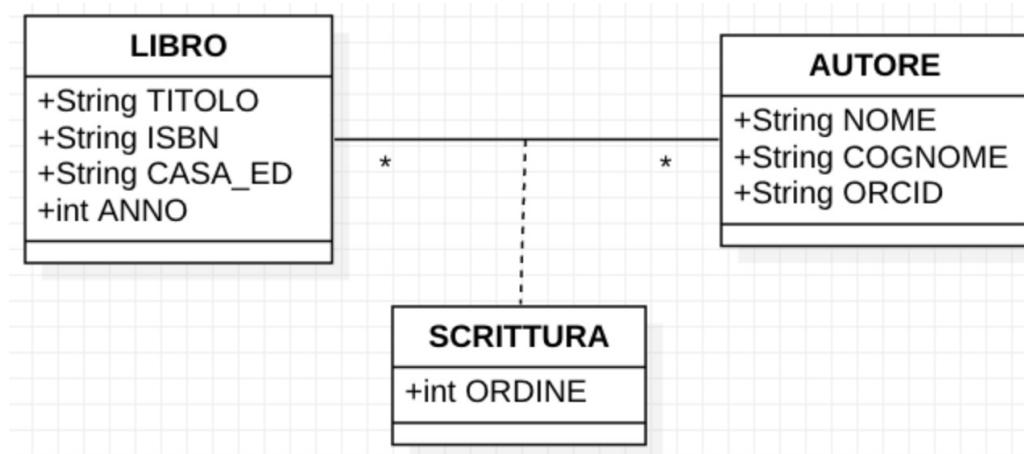
B(B<sub>1</sub>, B<sub>2</sub>, ..., B<sub>i</sub>, ..., B<sub>m</sub>)

C(A<sub>1</sub>, A<sub>2</sub>, ...A<sub>h</sub>, B<sub>1</sub>, B<sub>2</sub>, ..., B<sub>i</sub>, C<sub>1</sub>, C<sub>2</sub>, ..., C<sub>k</sub>)

PK



# ESEMPIO



LIBRO(Titolo, ISBN, Casa\_Ed, Anno)

AUTORE(Nome, Cognome, ORCID)

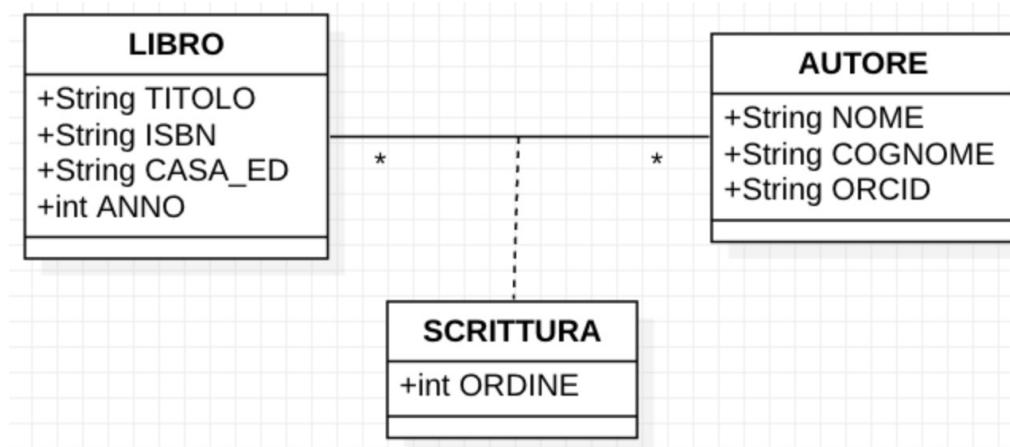
SCRITTURA(ISBN, ORCID, Ordine)

SCRITTURA.ISBN → LIBRO.ISBN

SCRITTURA.ORCID → AUTORE.ORCID



# RINOMINARE



LIBRO(Titolo, ISBN, Casa\_Ed, Anno)

AUTORE(Nome, Cognome, ORCID)

SCRITTURA(ISBN , ORCID, Ordine)

LIBROT

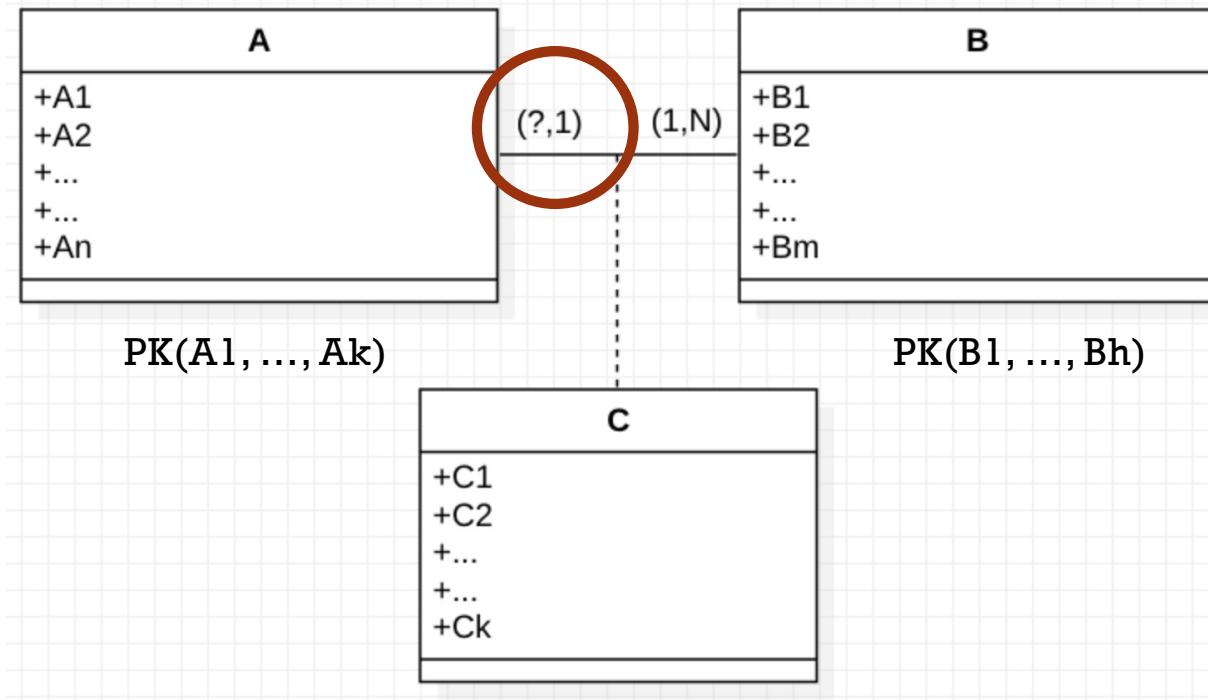
dom(SCRITTURA.ISBN) = dom(LIBRO.ISBN)

dom(SCRITTURA.ORCID) = dom(AUTORE.ORCID)

sono gli stessi domini delle relative chiavi primarie



# ASSOCIAZIONI 1:N (CASO GENERALE)



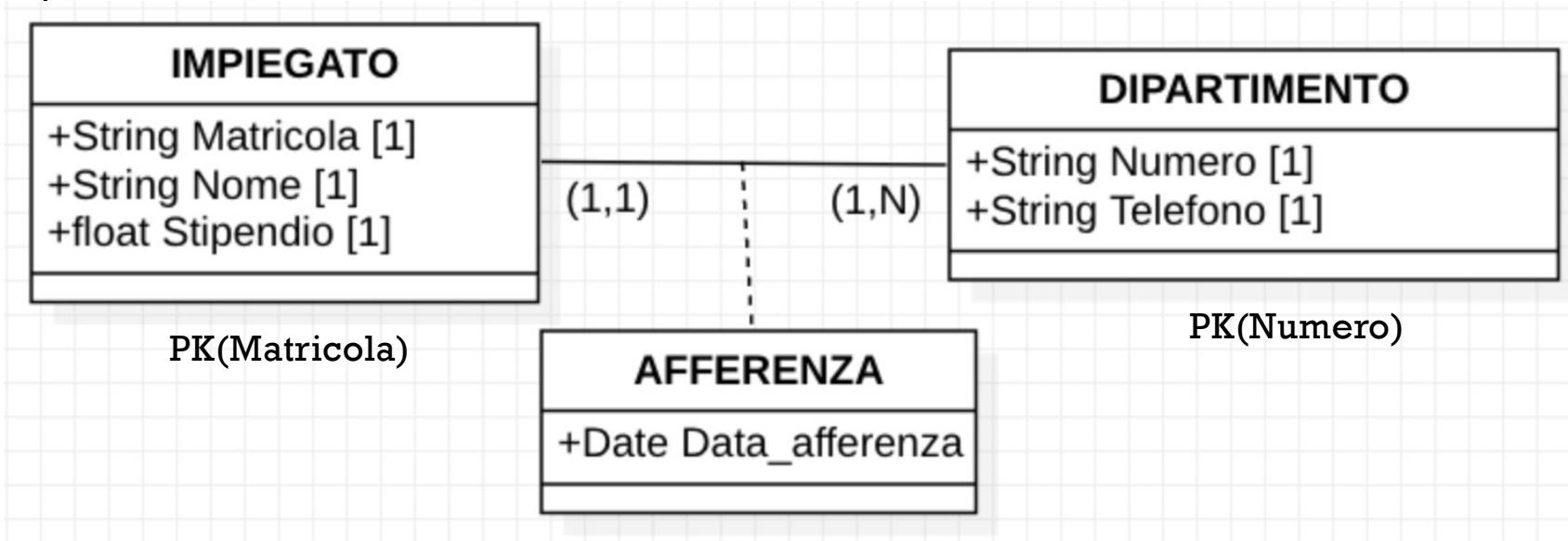
È necessario distinguere 3 situazioni

- A ha una partecipazione totale nell'associazione
- A ha una partecipazione parziale nell'associazione
- A è un'entità debole



# ASSOCIAZIONI 1:N

1) A HA UNA PARTECIPAZIONE TOTALE NELL'ASSOCIAZIONE



DIPARTIMENTO(Numero, Telefono)

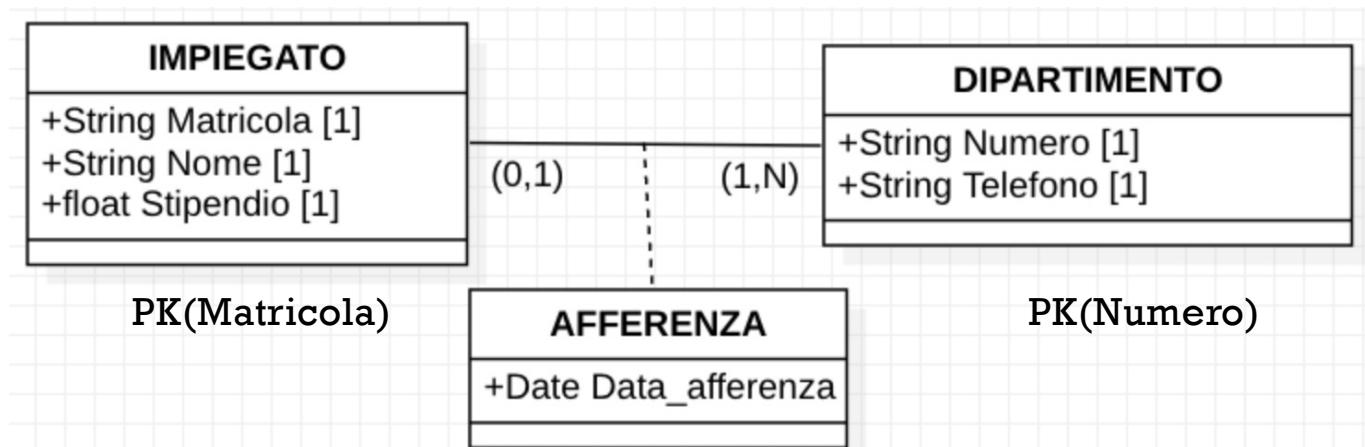
IMPIEGATO(Matricola, Nome, Stipendio, Numero\_Dipartimento, Data\_Afferenza)

IMPIEGATO.Numero\_Dipartimento → DIPARTIMENTO.Numero



# ASSOCIAZIONI 1:N

## 2) A HA UNA PARTECIPAZIONE PARZIALE NELL'ASSOCIAZIONE



DIPARTIMENTO(Numero, Telefono)

IMPIEGATO(Matricola, Nome, Stipendio)

AFFERENZA (Matricola\_I, Numero\_D, Data\_afferenza)

Matricola\_I → IMPIEGATO.Matricola

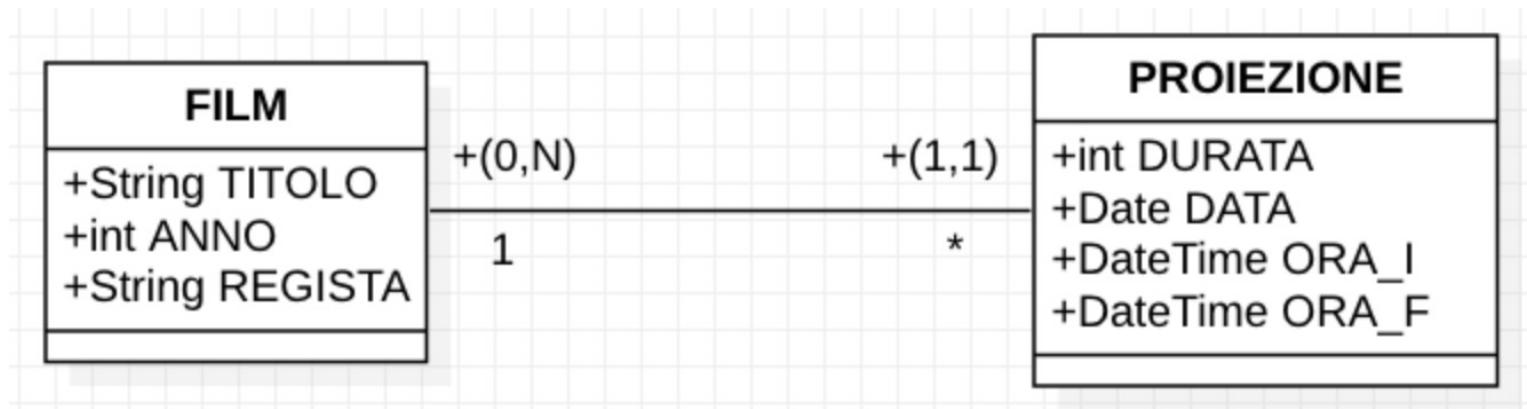
Numero\_D → Dipartimento.Numero

**N.B.** Questa soluzione conviene nel momento in cui ho pochi Impiegati che afferiscono ad un dipartimento



# ASSOCIAZIONI 1:N

## 3) A È UN'ENTITÀ DEBOLE



PK(TITOLO,ANNO)

ENTITA' FORTE

La descrizione di una proiezione dipende da FILM

ENTITA' DEBOLE

FILM(TITOLO, ANNO, REGISTA)

CHIAVE DELL'ENTITA' POSSESSORE

PROIEZIONE(DURATA, DATA, ORA\_I, ORA\_F, TITOLO, ANNO)

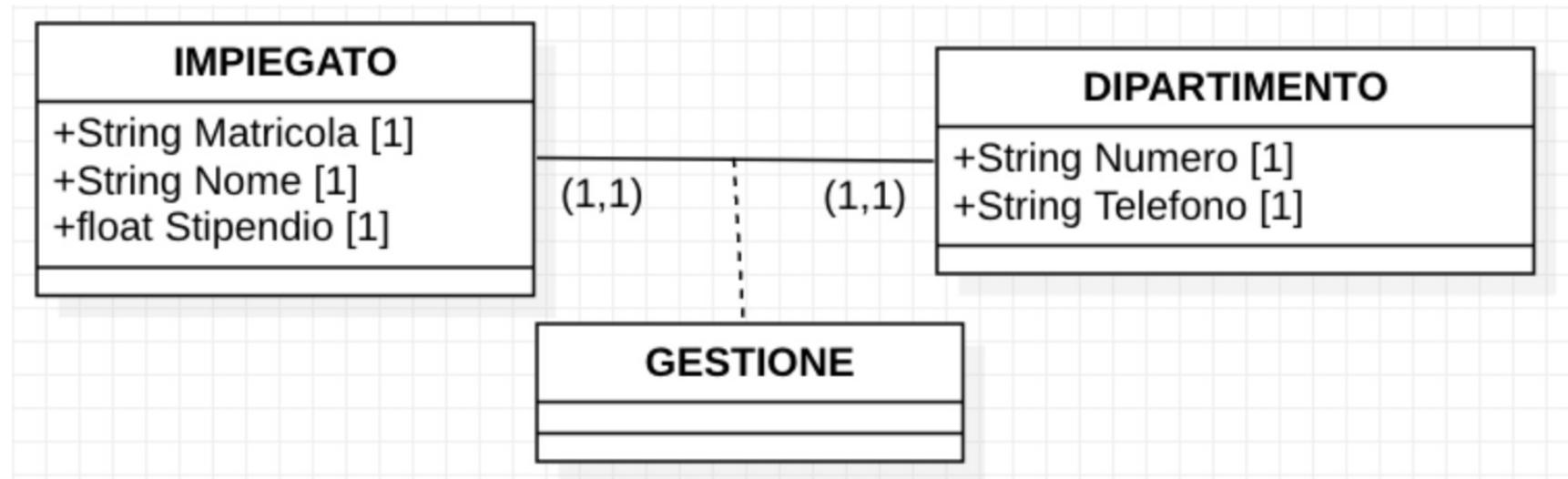
PK (Titolo, Anno, Data, Ora\_I)

Chiave Parziale

Foreign Key



# ASSOCIAZIONI 1:1 (PARTECIPAZIONE TOTALE)

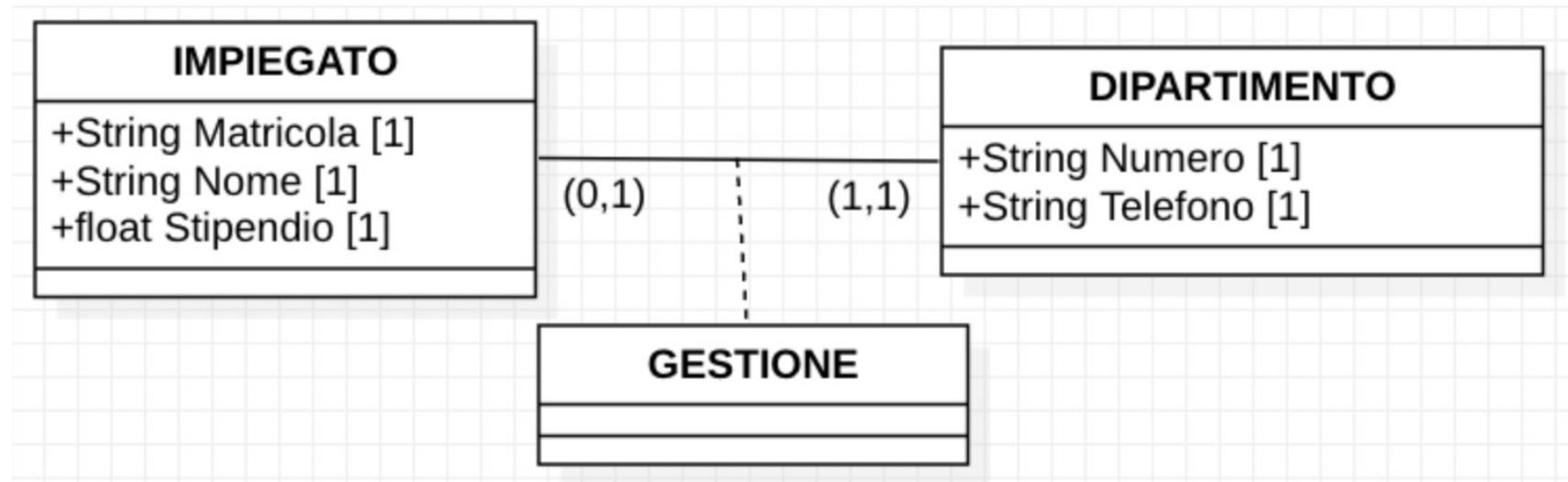


DIPARTIMENTO(Numero, Telefono, Matricola\_I)  
Matricola\_I → IMPIEGATO.Matricola  
IMPIEGATO(Matricola, Nome, Stipendio)

O DIPARTIMENTO(Numero, Telefono)  
IMPIEGATO(Matricola, Nome, Stipendio, Numero\_D)  
Numero\_D → DIPARTIMENTO.Numero



# ASSOCIAZIONI 1:1 (PARTECIPAZIONE PARZIALE)

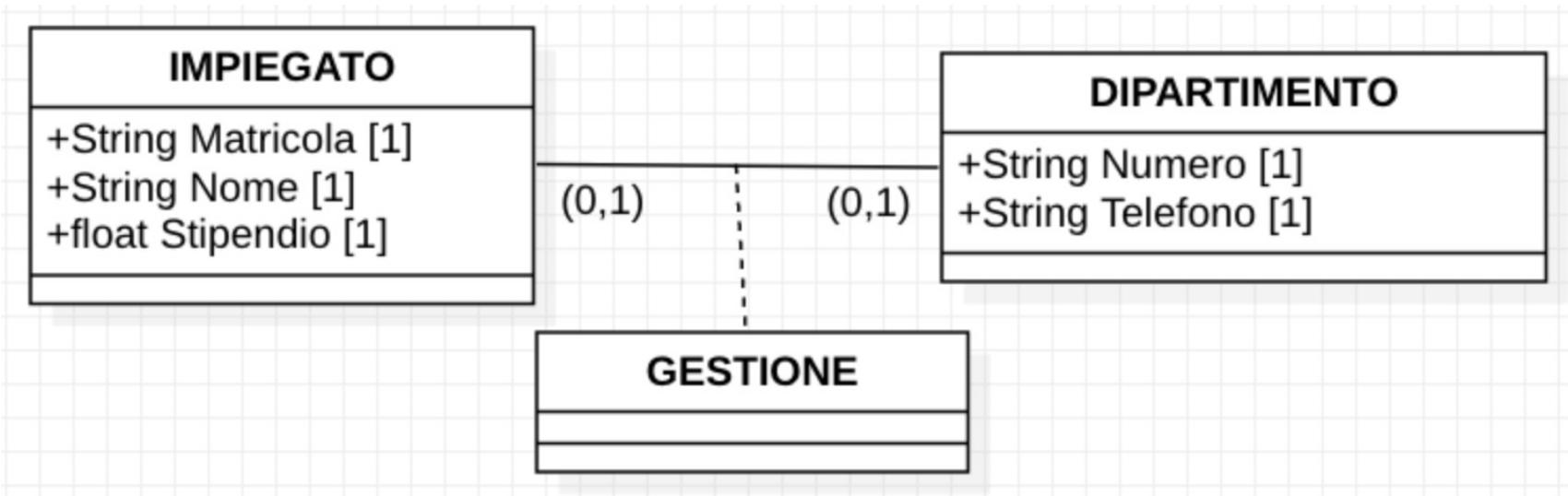


DIPARTIMENTO(Numero, Telefono, Matricola\_I)  
Matricola\_I → IMPIEGATO.Matricola  
IMPIEGATO(Matricola, Nome, Stipendio)

Rispetto il vincolo di integrità referenziale e non ho valori nulli



# ASSOCIAZIONI 1:1 ALTRO CASO)



DIPARTIMENTO(Numero, Telefono)

IMPIEGATO(Matricola, Nome, Stipendio)

GESTIONE (Matricola\_I, Dipartimento\_N)

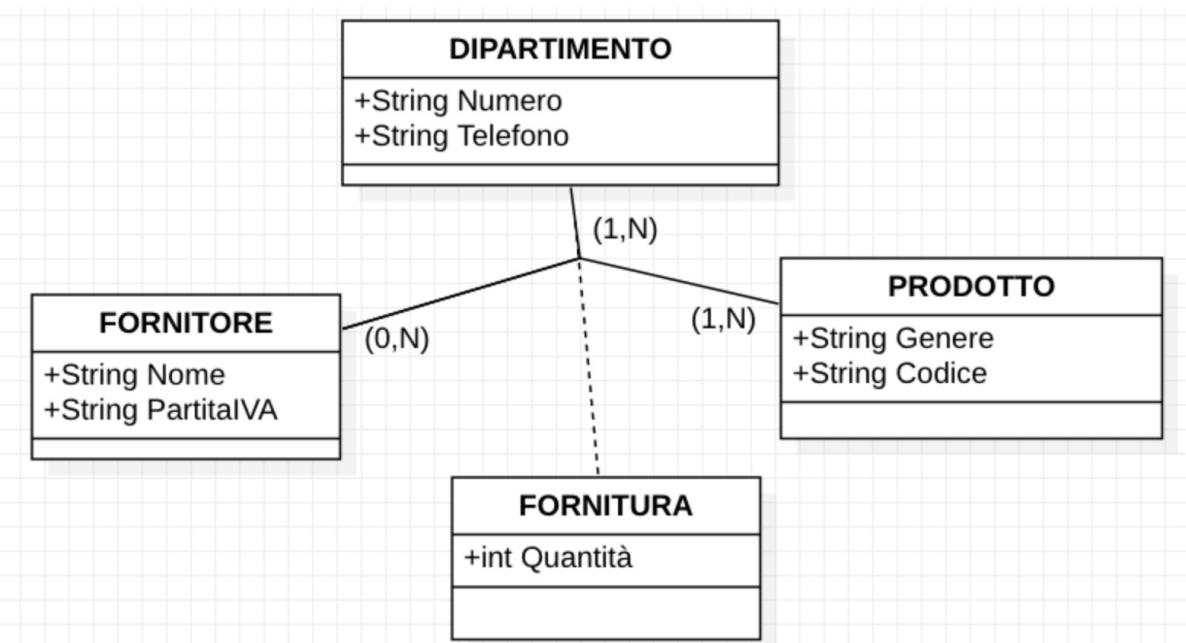
Matricola\_I → IMPIEGATO.Matricola

Dipartimento\_N → DIPARTIMENTO.Numero

Rispetto il vincolo di integrità referenziale e non ho valori nulli



# ASSOCIAZIONI N-ARIE



- Tutte le associazioni sono M:N

Fornitore(Nome, PartitaIVA)

Dipartimento(Numero, Telefono)

Prodotto(Genere, Codice)

Fornitura(PartitaIVAF, NumeroD, CodiceP)

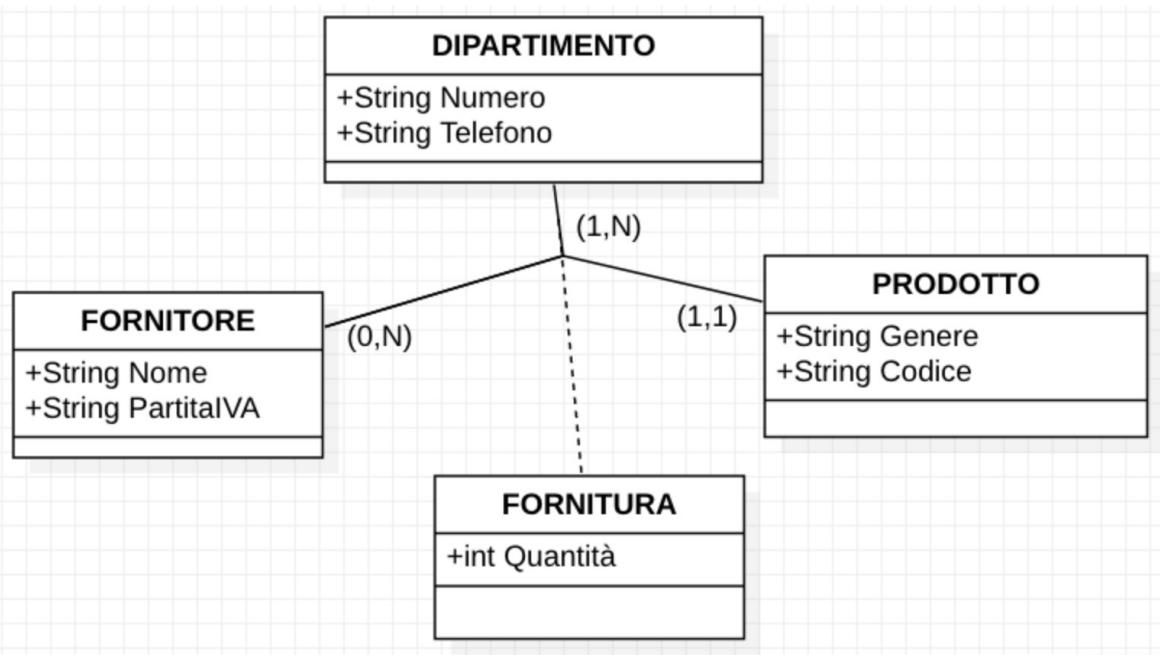
PartitaIVAF → Fornitore.PartitaIVA

NumeroD → Dipartimento.Numero

CodiceP → Prodotto.Codice



# ASSOCIAZIONI N-ARIE



- Abbiamo un'associazione 1:N

Fornitore(Nome, PartitaIVA)

Dipartimento(Numero, Telefono)

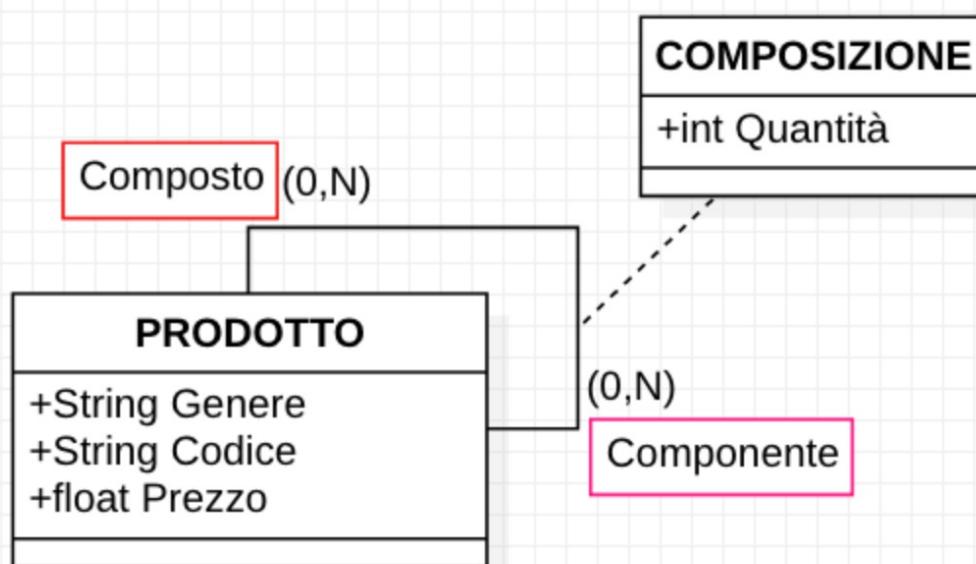
Prodotto(Genere, Codice, PartitaIVAF,  
NumeroD, Quantità)

PartitaIVAF → Fornitore.PartitaIVA

NumeroD → Dipartimento.Numero



# ASSOCIAZIONI RICORSIVE



**Rinominare, in questi casi, è necessario**

**PRODOTTO**(Genere, Codice, Prezzo)  
**COMPOSIZIONE**(Componente, Composto, Quantità)



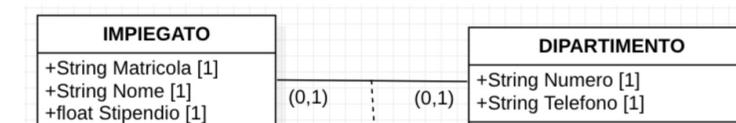
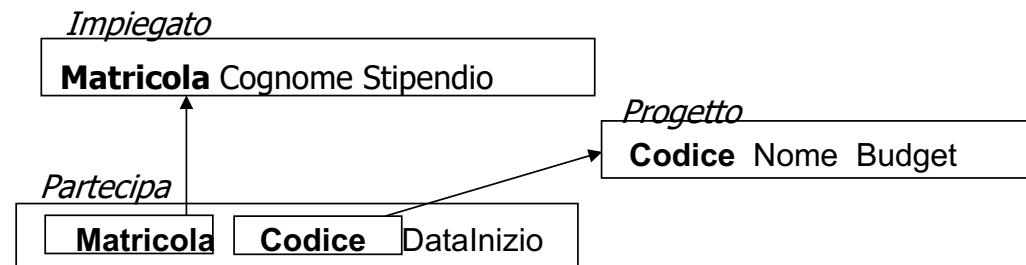
# **DOCUMENTAZIONE DI SCHEMI LOGICI**

- **Risultato della progettazione logica:**
  - Schema logico.
- **Documenti:**
  - Buona parte di quelli ottenuti dalla progettazione concettuale vengono ereditati.
  - Più i documenti per descrivere i vincoli di integrità referenziale introdotti nella traduzione.



# FORMALISMO GRAFICO

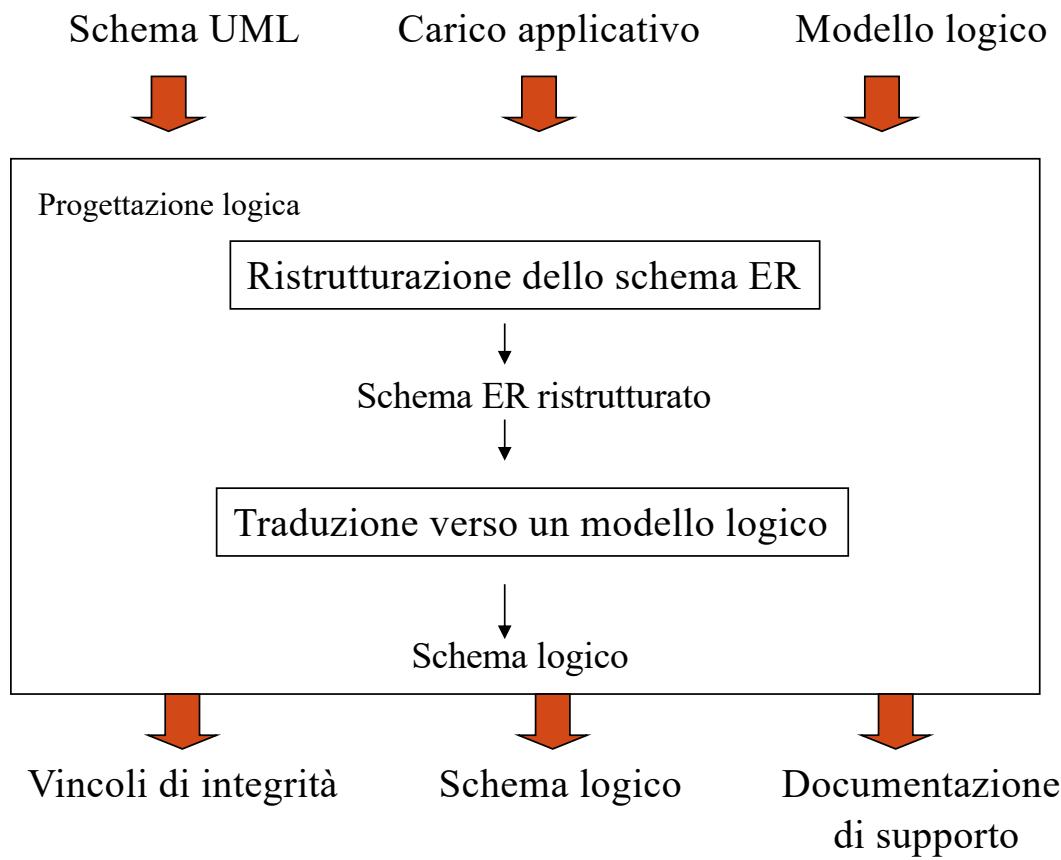
- Le frecce indicano i vincoli di integrità, in grassetto sono indicate le chiavi.
- In alternativa, le chiavi sono indicate con gli attributi sottolineati una volta, e le chiavi esterne con una doppia sottolineatura



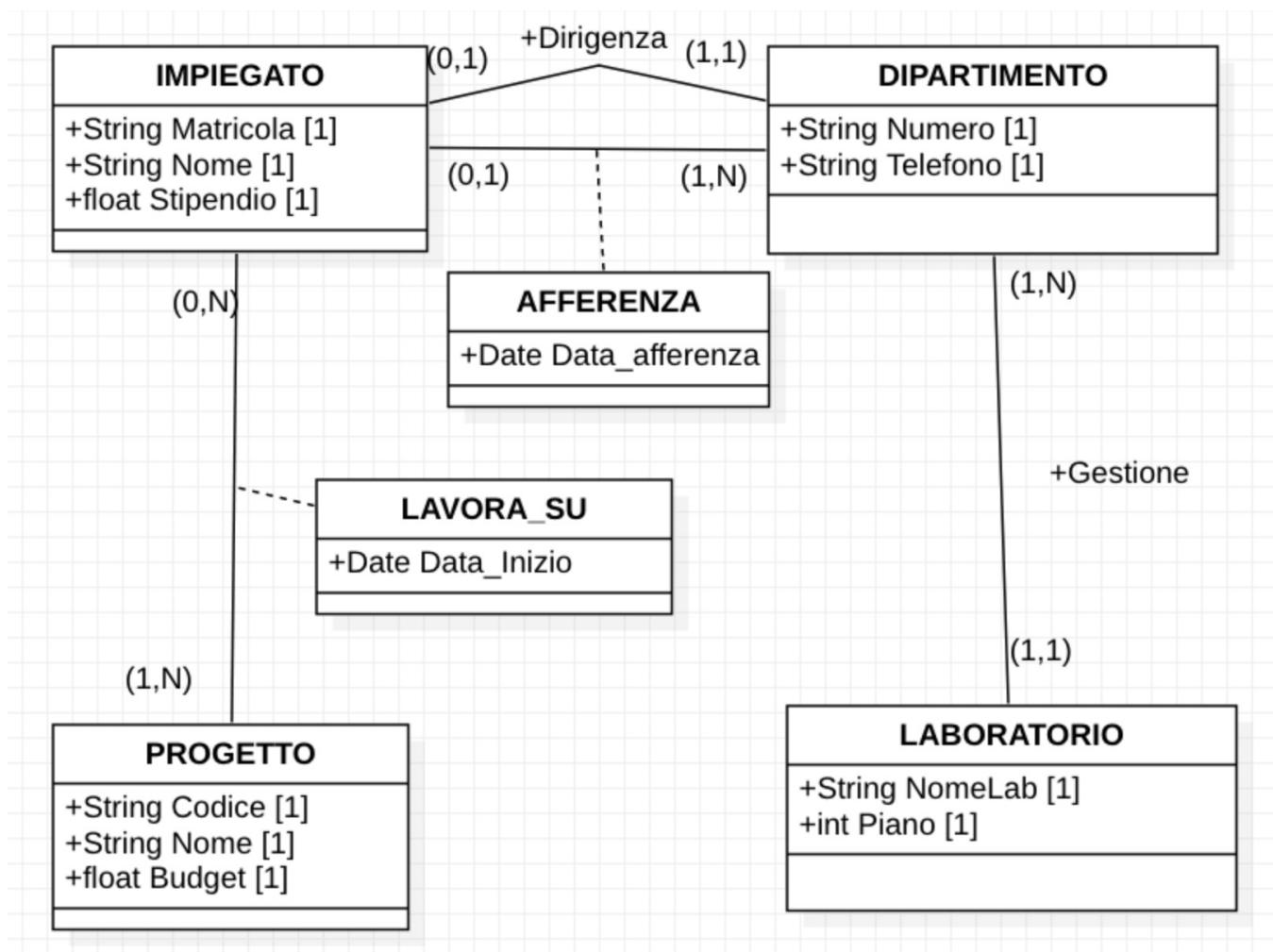
DIPARTIMENTO(Numero, Telefono)  
IMPIEGATO(Matricola, Nome, Stipendio)  
GESTIONE (Matricola\_I, Dipartimento\_N)  
Matricola\_I → IMPIEGATO.Matricola  
Dipartimento\_N → DIPARTIMENTO.Numero



# PROGETTAZIONE LOGICA DI BASI DI DATI



## ESEMPIO

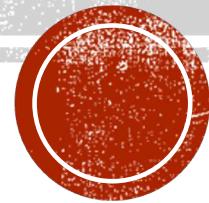




# FINE

Per eventuali domande: (in ordine di preferenza personale)

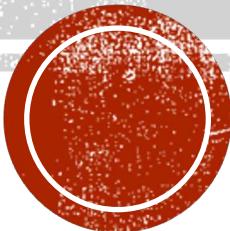
- Ora.
- Chat di Teams
- Mail: [silvio.barra@unina.it](mailto:silvio.barra@unina.it)



# BASI DI DATI I

- Ristrutturazione Class Diagram
- Tavole di analisi
- Passaggi nella ristrutturazione
- Analisi delle ridondanze
- Eliminazione delle generalizzazioni
- Eliminazione attributi multipli
- Eliminazione attributi strutturati
- Partizione/Associazione entità/associazioni
- Identificazione chiavi primarie

# RISTRUTTURAZIONE CLASS DIAGRAM



# OBIETTIVO

- Uno schema logico in grado di descrivere tutte le informazioni contenute nello schema UML prodotto nella fase di progettazione concettuale.



# LA RISTRUTTURAZIONE

- Prima di passare allo schema logico è necessario ristrutturare il class diagram per:
  - Semplificare la traduzione:  
*non tutti i costrutti che abbiamo utilizzato hanno una traduzione naturale nei modelli logici.*
  - Ottimizzare il progetto:  
*rendere più efficiente l'esecuzione delle operazioni.*

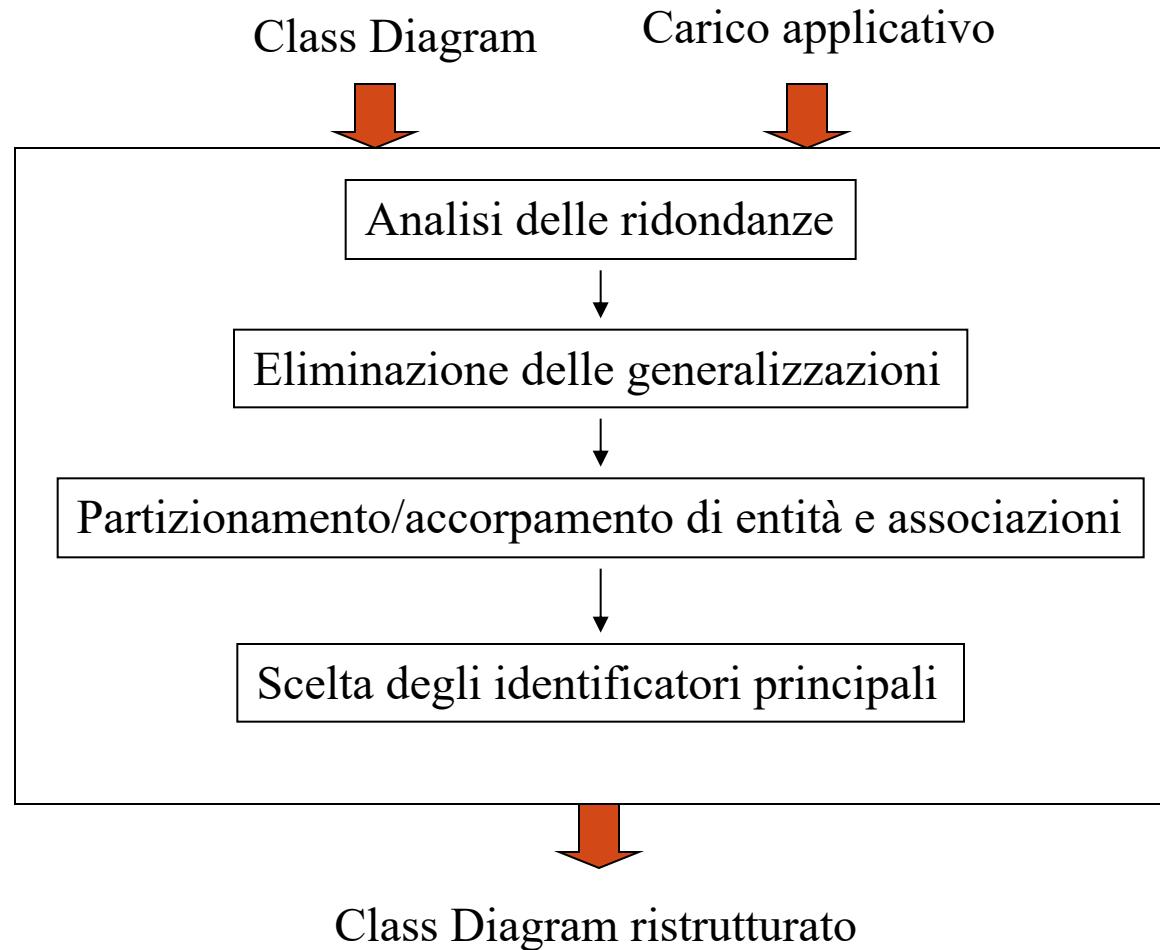


# ATTIVITÀ DI RIORGANIZZAZIONE E TRADUZIONE

- Ristrutturazione del Class Diagram UML:
  - *si basa su criteri di ottimizzazione dello schema.*
- Traduzione verso il modello logico:
  - *fa riferimento ad uno specifico modello logico.*
- I dati in ingresso sono lo schema concettuale ed il carico applicativo (dimensione dei dati e caratteristiche delle operazioni).



# RISTRUTTURAZIONE DI SCHEMI LOGICI



# ANALISI DELLE PRESTAZIONI

- Indici di prestazioni (non valutabili in maniera precisa in sede di progettazione logica):
  - Costo di una operazione (# occorrenze di entità e relazioni visitate per rispondere a una operazione).
  - Occupazione di memoria.

Per studiare questi parametri è necessario conoscere:



- Volume dei dati:
  - # occorrenze di entità e relazioni.
  - Dimensioni di ciascun attributo.
- Caratteristiche delle operazioni:
  - Tipo (interattiva o batch).
  - Frequenza (# medio di esecuzioni in un  $t$ ).
  - Dati coinvolti.



# TAVOLE DI ANALISI



# TAVOLE DI ANALISI

- Tavola dei volumi:

## **concetto tipo volume**

- Vengono riportati tutti i concetti dello schema (entità e relazioni) con il volume previsto a regime.

- Tavola delle operazioni:

## **operazioni tipo frequenza**

- Viene riportata per ogni operazione la frequenza prevista, il tipo (interattiva o batch).

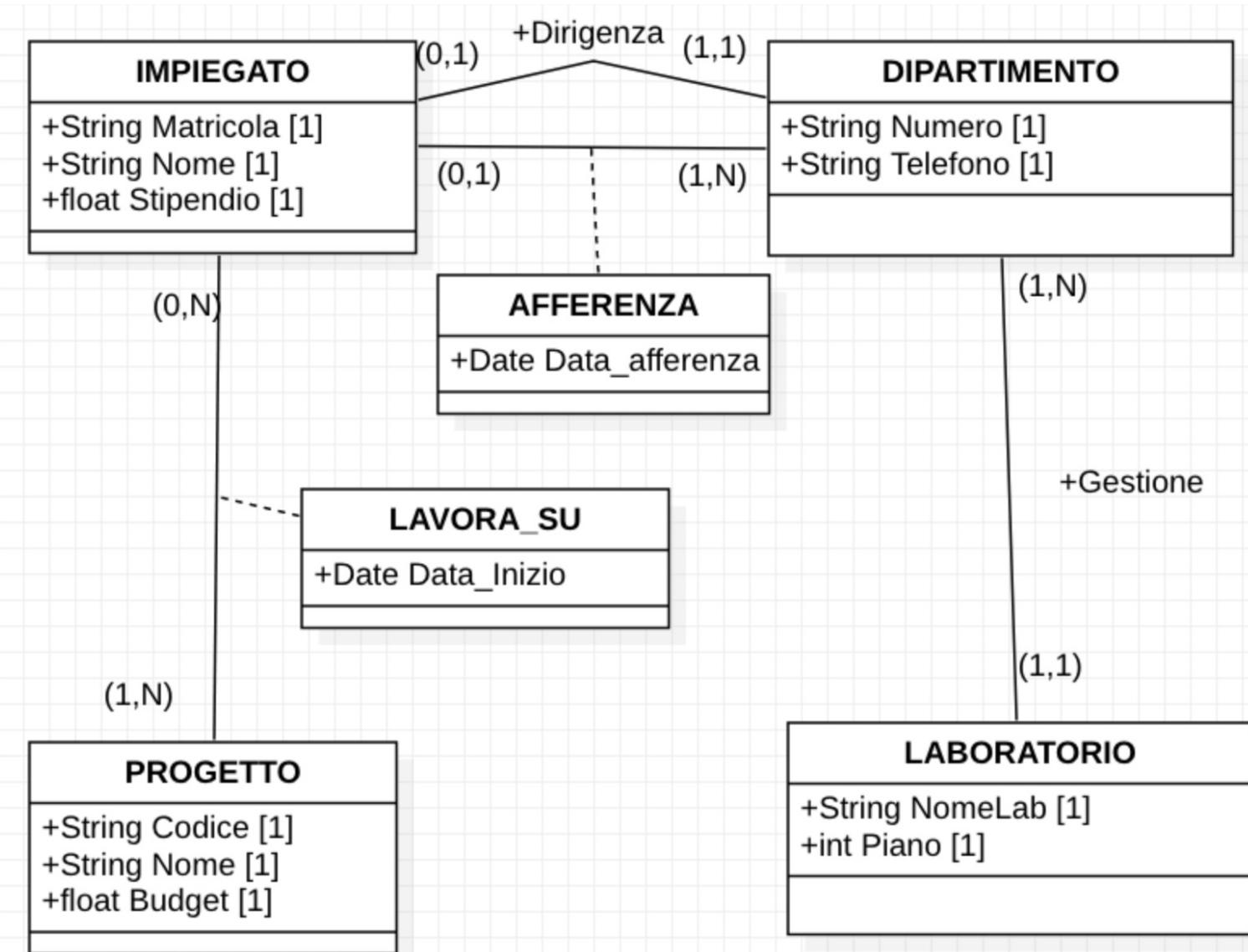
- Tavola degli accessi

## **concetto costrutto accessi tipo**

- Viene riportata, per ogni operazione, il numero di accessi ai concetti coinvolti ed il tipo di accesso (L/S).
- Le operazioni in scrittura (S) sono più onerose di quelle in lettura (L).



# ESEMPIO



# TAVOLA DEI VOLUMI

Concetto	Tipo	Volume
Laboratorio	E	100
Dipartimento	E	80
Impiegato	E	2000
Progetto	E	500
Gestione	R	100
Afferenza	R	1900
Dirigenza	R	80
Lavora_Su	R	6000



# **NUMERO DELLE OCCORRENZE DELLE ASSOCIAZIONI**

- Nella tavola dei volumi.
- Dipende da due parametri:
  - Numero delle occorrenze delle entità coinvolte nelle associazioni.
  - Numero medio di partecipazioni di un'occorrenza di entità alle occorrenze di associazioni.  
**Es:** supponiamo che un impiegato lavori in media a tre progetti.



# OPERAZIONI

1. Assegna un impiegato ad un progetto.
2. Trova i dati di un impiegato, del dipartimento nel quale lavora e dei progetti ai quali partecipa.
3. Trova i dati di tutti gli impiegati di un certo dipartimento.
4. Per ogni sede, trova i suoi dipartimenti con il cognome del direttore e l'elenco degli impiegati del dipartimento.



# TAVOLA DELLE OPERAZIONI

Operazione	Tipo	Frequenza
Op. 1	I	50 al giorno
Op. 2	I	100 al giorno
Op. 3	I	10 al giorno
Op. 4	B	2 a settimana

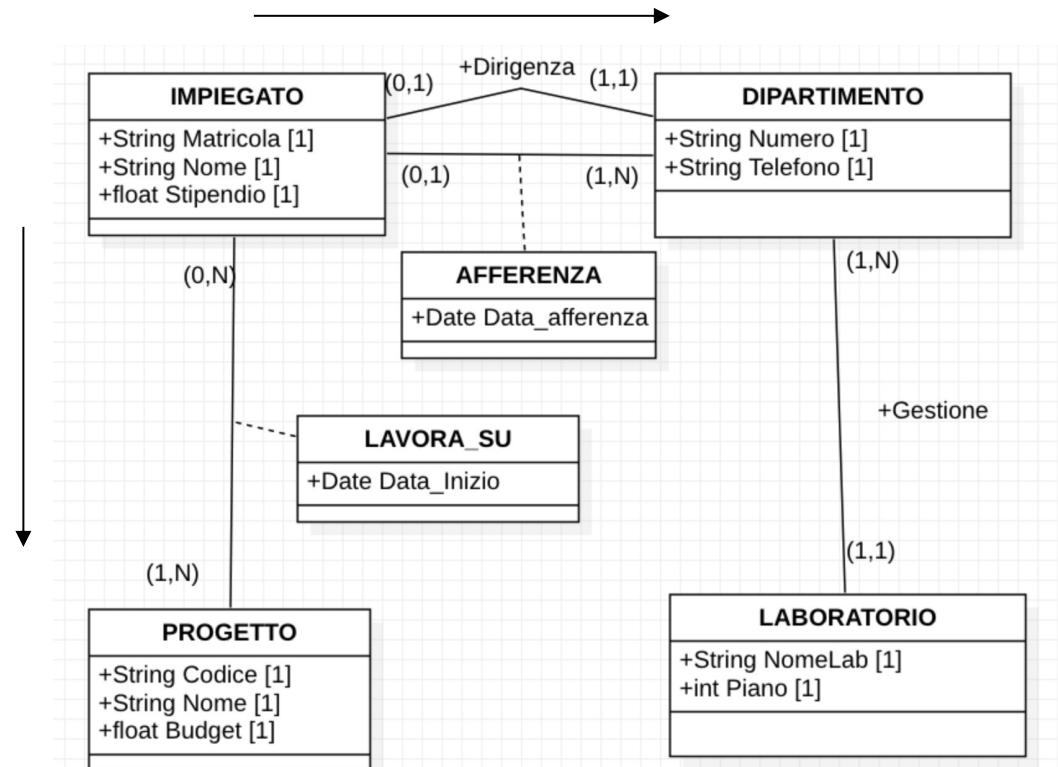


# SCHEMA DI OPERAZIONE

- Per ogni operazione possiamo descrivere graficamente i dati coinvolti attraverso uno **schema di operazione**.
- Descrive il cammino logico da percorrere per accedere alle informazioni di interesse.

## Operazione 2

Trova i dati di un impiegato, del dipartimento nel quale lavora e dei progetti ai quali partecipa.



# COSTO DI UN'OPERAZIONE

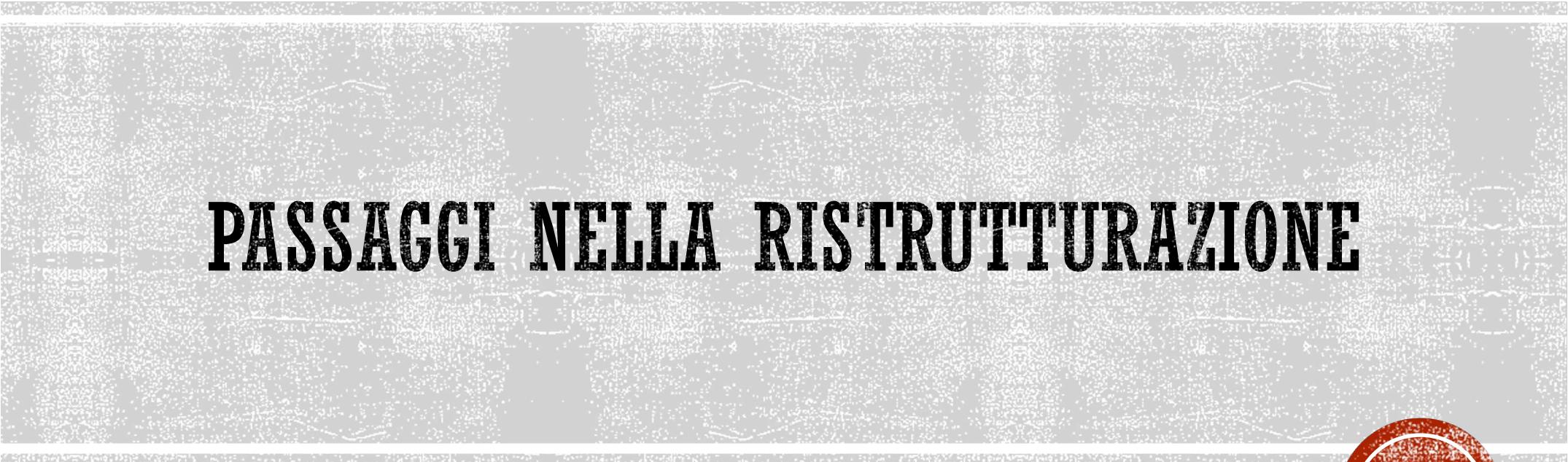
- Si stima il costo di un'operazione contando il numero degli accessi alle occorrenze di entità e di relazioni.



# TAVOLA DEGLI ACCESSI (RIF. OP. 2)

Concetto	Costrutto	Accessi	Tipo
Impiegato	E	1	L
Afferisce	R	1	L
Dipartimento	E	1	L
Partecipa	R	3	L
Progetto	E	3	L





# PASSAGGI NELLA RISTRUTTURAZIONE



# RISTRUTTURAZIONE DEL CLASS DIAGRAM

- La fase di ristrutturazione di uno schema concettuale è suddiviso:
  1. Analisi delle ridondanze.
  2. Eliminazione delle generalizzazioni.
  3. Eliminazione Attributi Multivalore
  4. Eliminazione Attributi Strutturati
  5. Partizionamento/accorpamento di entità e associazioni.
  6. Scelta degli identificatori primari.



# **ANALISI DELLE RIDONDANZE**



# ANALISI DELLE RIDONDANZE

- Una **ridondanza** in uno schema concettuale corrisponde alla presenza di un dato che può essere derivato da altri dati.
- **Vantaggi**
  - Semplificazione delle interrogazioni.
- **Svantaggi**
  - Appesantimento degli aggiornamenti.
  - Maggiore occupazione di spazio.

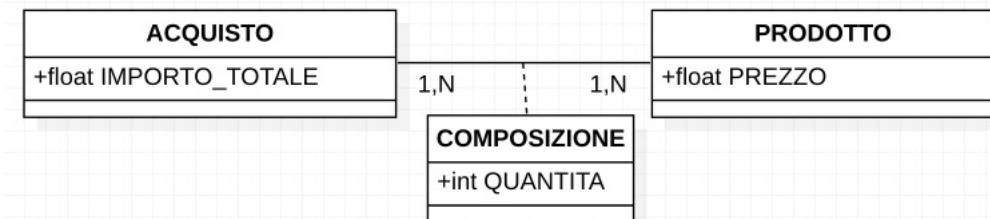


# ATTRIBUTI DERIVABILI

- Da attributi della stessa entità /associazione:



- Da attributi di altre entità/associazioni

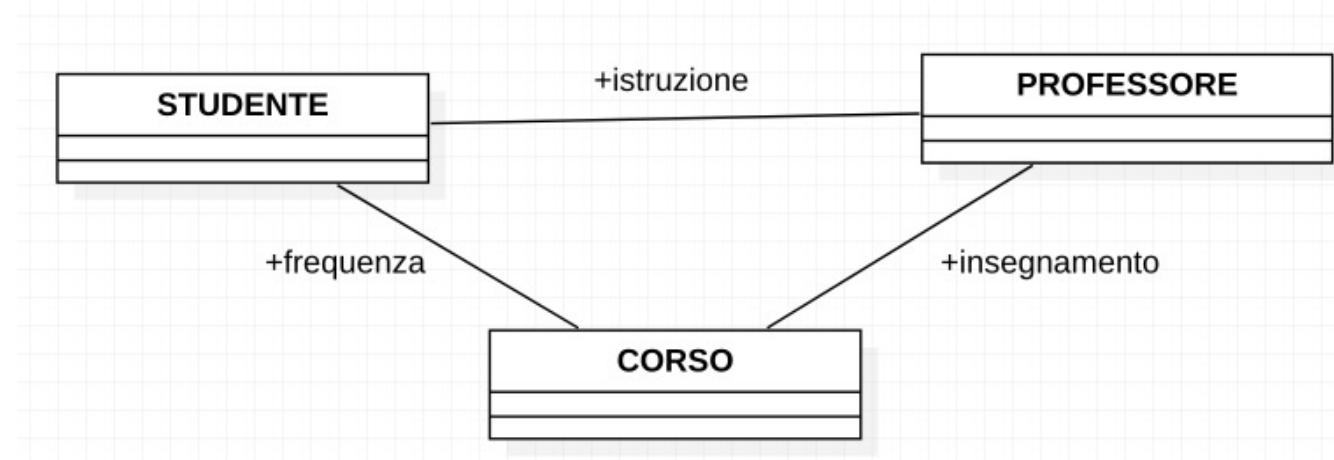


- Dal conteggio di occorrenze



# ATTRIBUTI DERIVABILI (4)

- Da associazioni in presenza di cicli:



- La presenza di cicli non genera necessariamente ridondanze:
  - *Si immagini che l'associazione "istruzione" fosse "è tesista di".*



# INSERIRE UNA RIDONDANZA?

- La decisione va presa confrontando il costo delle operazioni che coinvolgono il dato ridondante e relativa occupazione di memoria nei casi di **presenza/assenza** di ridondanza.



# TAVOLE DI ANALISTI

- Tavola dei volumi:

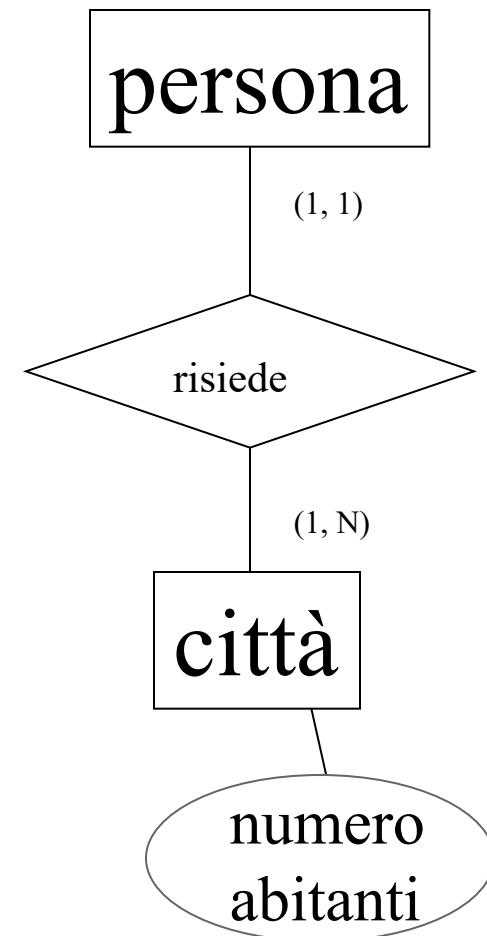
▪ Città	E	200
▪ Persona	E	1000000
▪ Risiede	R	1000000

- *Op<sub>1</sub>: memorizza una nuova persona con la relativa città di residenza.*

- *Op<sub>2</sub>: stampa tutti i dati di una città (incluso il numero di abitanti).*

- Tavola delle operazioni:

▪ Op <sub>1</sub>	I	500/giorno
▪ Op <sub>2</sub>	I	2/giorno



# ***“NUMERO DI ABITANTI” → CITTÀ***

- Proviamo a valutare gli indici di prestazione in caso di presenza del dato ridondante:
- $\text{Mem}(\text{num\_abit}) \approx 4 \text{ byte} \rightarrow \text{dato ridondante} \approx 4 \times 200 = 800 \text{ byte} (\approx 1 \text{ kbyte})$
- Costo:*Un accesso in scrittura ha un costo doppio rispetto ad un accesso in lettura*
  - .



# TAVOLE DEGLI ACCESSI (CON RIDONDANZA)

- Operazione 1: *memorizza una nuova persona con la relativa città di residenza.*

- Persona      E      1      S

(per memorizzare una nuova persona)

- Risiede      R      1      S

(per memorizzare una nuova coppia città-persona)

- Città      E      1      L

(per cercare la città di interesse)

- Città      E      1      S

(incrementa di 1 il numero di abitanti)

- $3S * 500/\text{giorno} + 1L * 500/\text{giorno}$



- 3500 accessi/giorno

(un accesso in scrittura vale il doppio  
di un accesso in lettura)

- Operazione 2: *stampa tutti i dati di una città (incluso il numero di abitanti).*

- Città      E      1      L

- Trascurabile:  $1L * 2/\text{giorno}$



# TAVOLE DEGLI ACCESSI (SENZA RIDONDANZA)

- Op.1: memorizza una nuova persona con la relativa città di residenza.

- Persona E l S
- Risiede R l S

(non si accede a Città per aggiornare il dato derivato)

- $2S * 500/\text{giorno}$



- 2000 accessi/giorno

(contano doppio gli accessi in scrittura)

- Op.2: stampa tutti i dati di una città (incluso il numero di abitanti).

- Città E l L
- Risiede R 5000<sup>1</sup> L

- $1L + 1L * 5000 * 2/\text{giorno}$



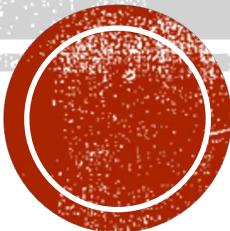
- 10000 accessi/giorno

<sup>1</sup> (#abitanti/città=1000000/200)

- 
- **12.000 accessi vs 3.500 accessi + 1Kbyte**
  - Gli accessi in lettura necessari per calcolare il dato derivato sono molti di più degli accessi in scrittura per mantenerlo aggiornato.



# **ELIMINAZIONE DELLE GENERALIZZAZIONI**



# ELIMINAZIONE DELLE GENERALIZZAZIONI

- I sistemi tradizionali per la gestione di basi di dati non consentono di rappresentare direttamente una generalizzazione.
- È necessario tradurre le generalizzazioni usando altri costrutti dell'UML(classi, associazioni).

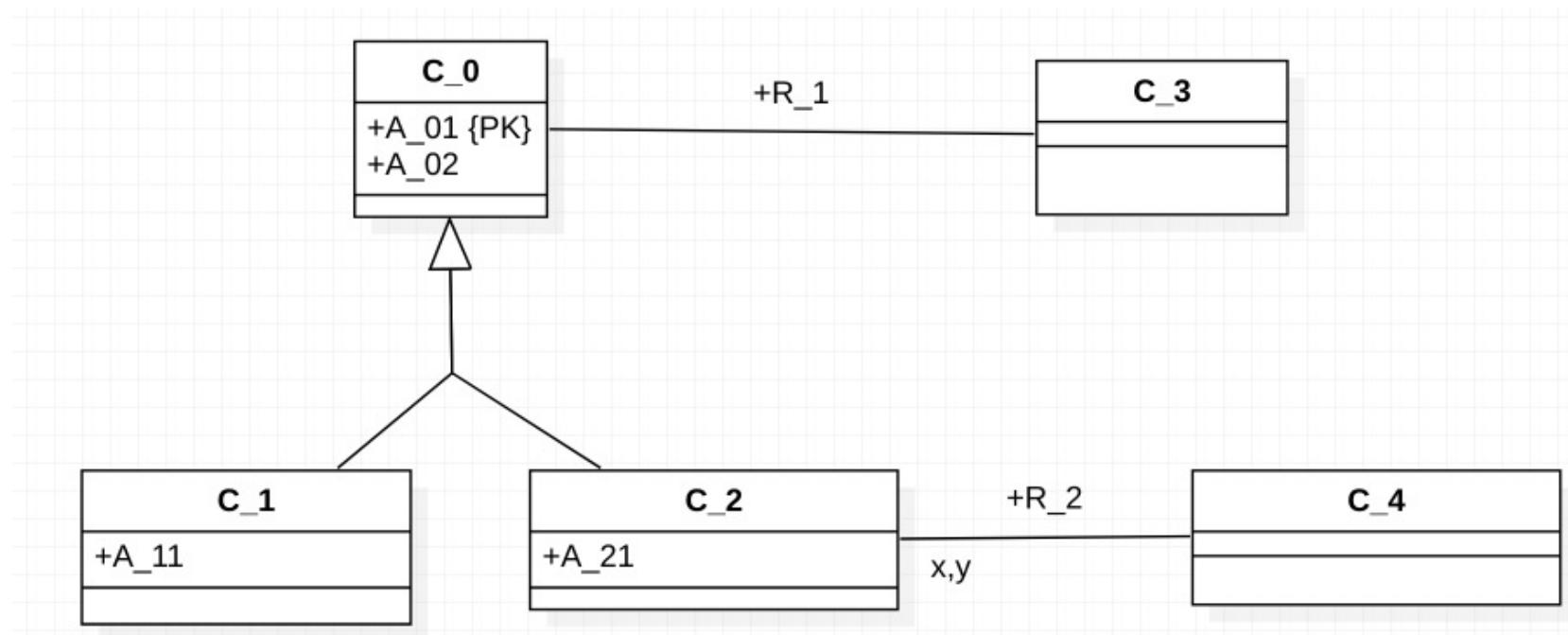


# ELIMINAZIONE DELLE GENERALIZZAZIONI

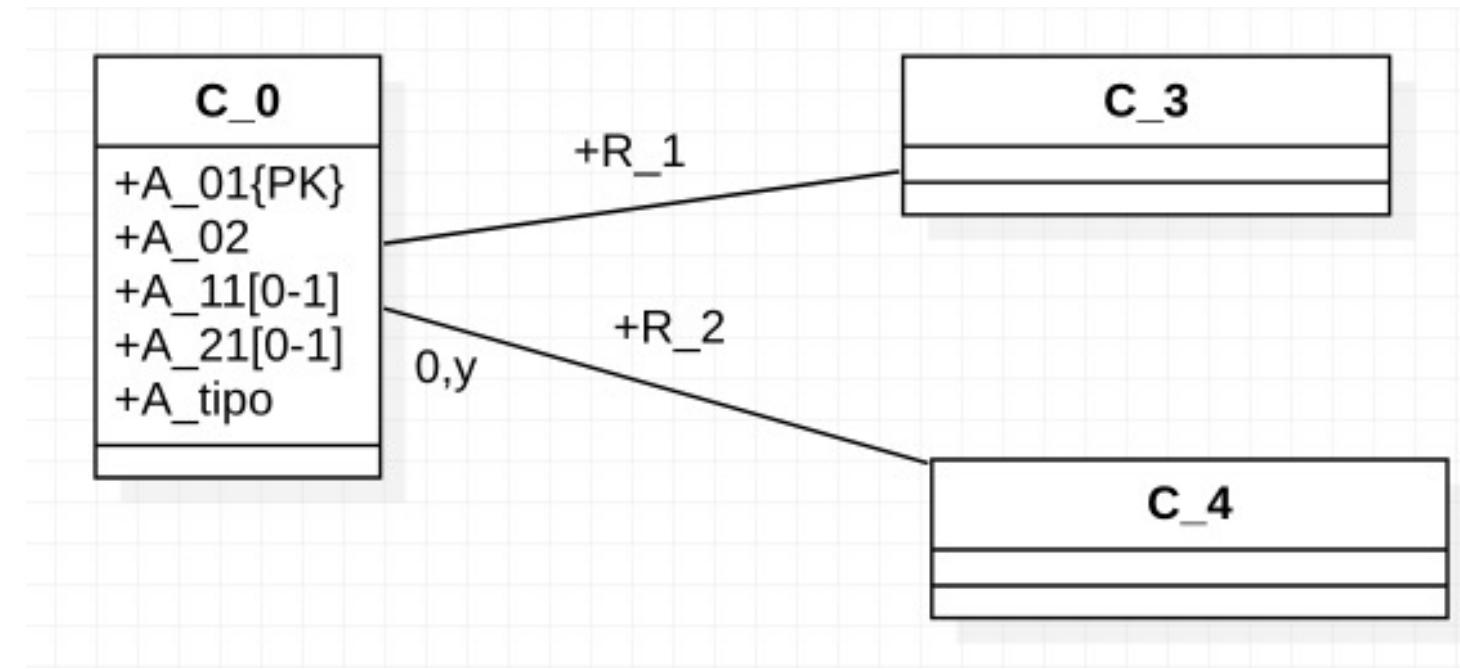
- a) Accorpamento delle figlie della generalizzazione nel padre.
- b) Accorpamento del padre della generalizzazione nelle figlie.
- c) Sostituzione della generalizzazione con associazioni.



# SCHEMA INIZIALE



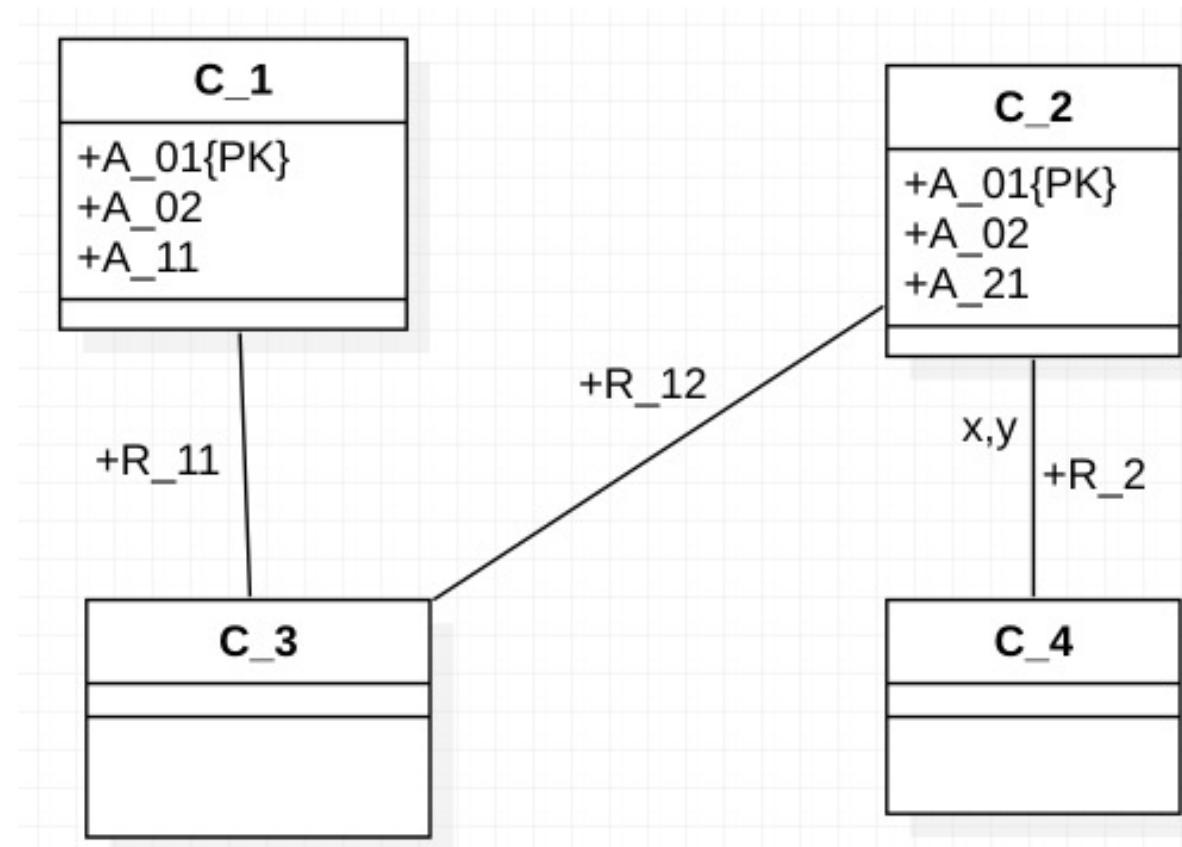
## (A) ACCORPAMENTO DELLE FIGLIE DELLA GENERALIZZAZIONE NEL PADRE



Le entità C<sub>1</sub> e C<sub>2</sub> vengono eliminate e le loro proprietà vengono aggiunte al padre C<sub>0</sub>.

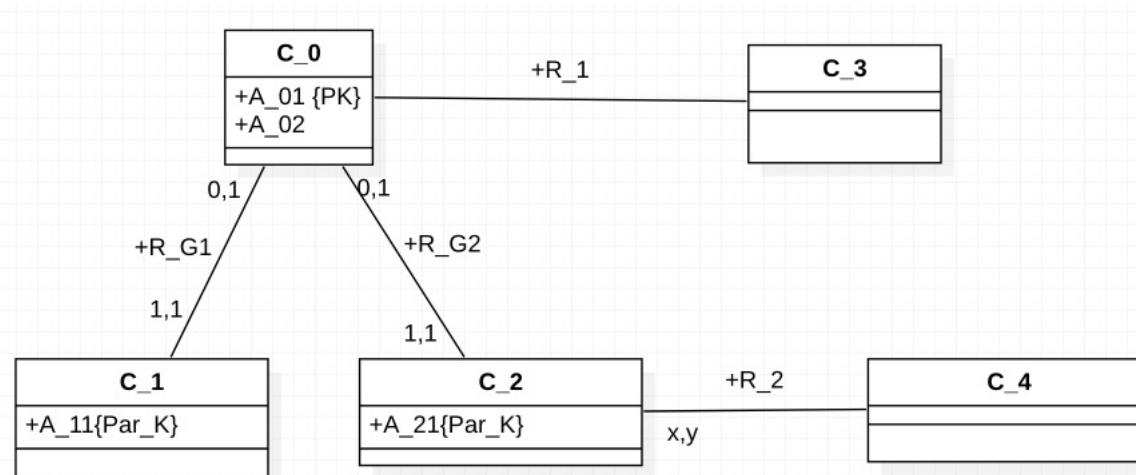


## (B) ACCORPAMENTO DEL PADRE DELLA GENERALIZZAZIONE NELLE FIGLIE



## (C) SOSTITUZIONE DELLA GENERALIZZAZIONE CON ASSOCIAZIONI

- Le entità  $C_1$  e  $C_2$  sono identificate esternamente dall'entità  $C_0$ :
- **Vincoli:** ogni occorrenza di  $C_0$  non può partecipare contemporaneamente a  $R_{G1}$  ed  $R_{G2}$  se la generalizzazione è totale, ogni occorrenza di  $E_0$  deve partecipare o ad un'occorrenza di  $R_{G1}$  o ad un'occorrenza di  $R_{G2}$



- **Vincolo di totalità:** Un elemento di  $C_0$  è associato ad un elemento di  $C_1$  o di  $C_2$ 
  - (**Parziale** se un elemento di  $C_0$  non è associato a nessuno dei figli)
- **Vincolo di disgiunzione:** Un elemento di  $C_0$  non può essere associato a entrambi i figli
  - (**Overlapping** se un elemento può essere associato a entrambi i figli)



# COME SCEGLIERE TRA LE DIVERSE ALTERNATIVE

- L'alternativa **(a)** conviene quando le operazioni non fanno molta distinzione tra le occorrenze e tra gli attributi di  $C_0$ ,  $C_1$ , ed  $C_2$ . Introduciamo valori nulli, ma abbiamo un minor numero di accessi.
- L'alternativa **(b)** è applicabile quando la generalizzazione è totale. Altrimenti le occorrenze di  $C_0$  che non sono occorrenze né di  $C_1$ , né di  $C_2$  non sarebbero rappresentate.
- L'alternativa **(c)** è applicabile quando la generalizzazione non è totale, e ci sono operazioni che fanno distinzione tra entità padre ed entità figlie. Assenza di valori nulli e incremento degli accessi.



## COME SCEGLIERE TRA LE DIVERSE ALTERNATIVE (2)

- La ristrutturazione delle generalizzazioni è un tipico caso per cui il conteggio delle istanze e degli accessi non permette sempre di scegliere la migliore alternativa.
- Infatti, sembrerebbe che l'alternativa **(c)** non conviene quasi mai perché richiede molti più accessi per eseguire le operazioni sui dati.
- Questo tipo di ristrutturazione ha il grosso vantaggio di generare entità con pochi attributi:
  - A livello pratico questo si traduce in strutture logiche di piccole dimensioni per cui un accesso fisico permette di recuperare molti dati (tuple) in una sola volta.

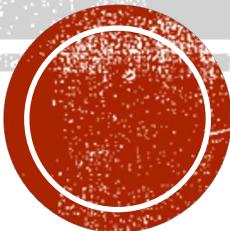


# **GENERALIZZAZIONI A PIÙ LIVELLI**

- Per le generalizzazioni a più livelli si può procedere analizzando una generalizzazione alla volta a partire dal fondo della gerarchia.

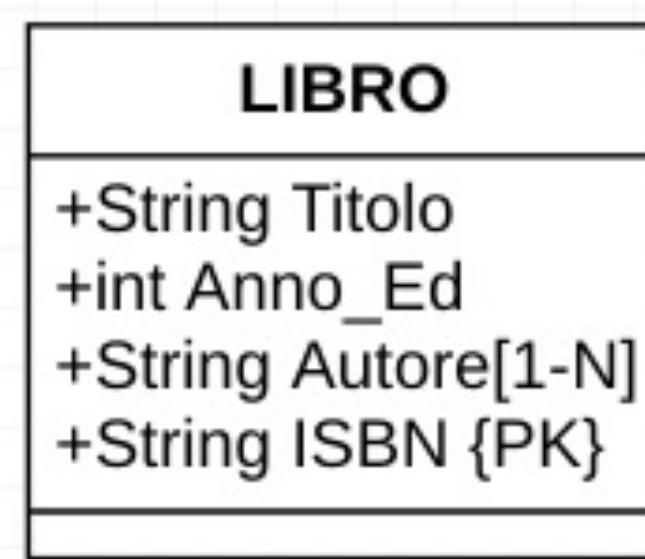


# **ELIMINAZIONE ATTRIBUTI MULTIVALORE**

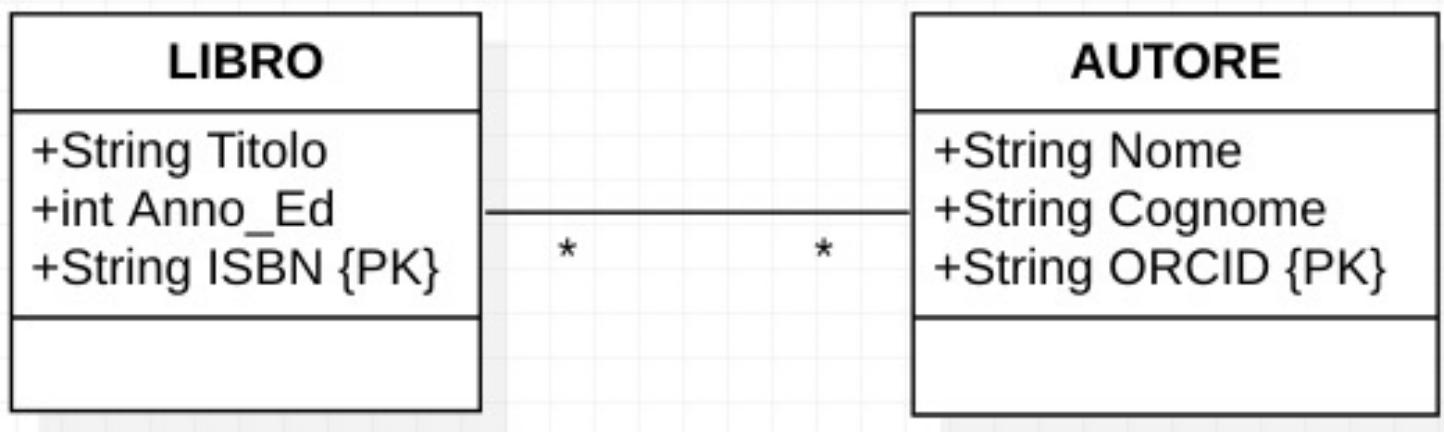


# ELIMINAZIONE ATTRIBUTI MULTIVALORE

- Il modello relazionale non consente questa rappresentazione:



# 1) CREAZIONE DI UNA ENTITÀ ESTERNA ASSOCIATA



## 2) TRATTARE L'ATTRIBUTO MULTIPLO COME FOSSE SINGOLO



Titolo: «Sistemi di basi di dati»

Anno\_Ed: «2015»

ISBN: «6767-8754»

Autori: «Elmasri, Navathe»



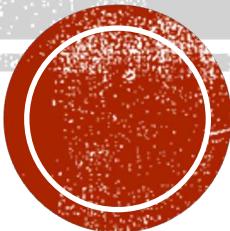
### 3) REPLICARE L'ATTRIBUTO NELLA CLASSE (QUANTE VOLTE??)



- Per alcuni casi potrebbe anche andare bene.
- Persona-> Telefono1, Telefono2
- Per un numero limitato di volte potrebbe essere una soluzione accettabile
  - Andare troppo oltre potrebbe causare un numero eccessivo di campi vuoti



# **ELIMINAZIONE ATTRIBUTI STRUTTURATI**



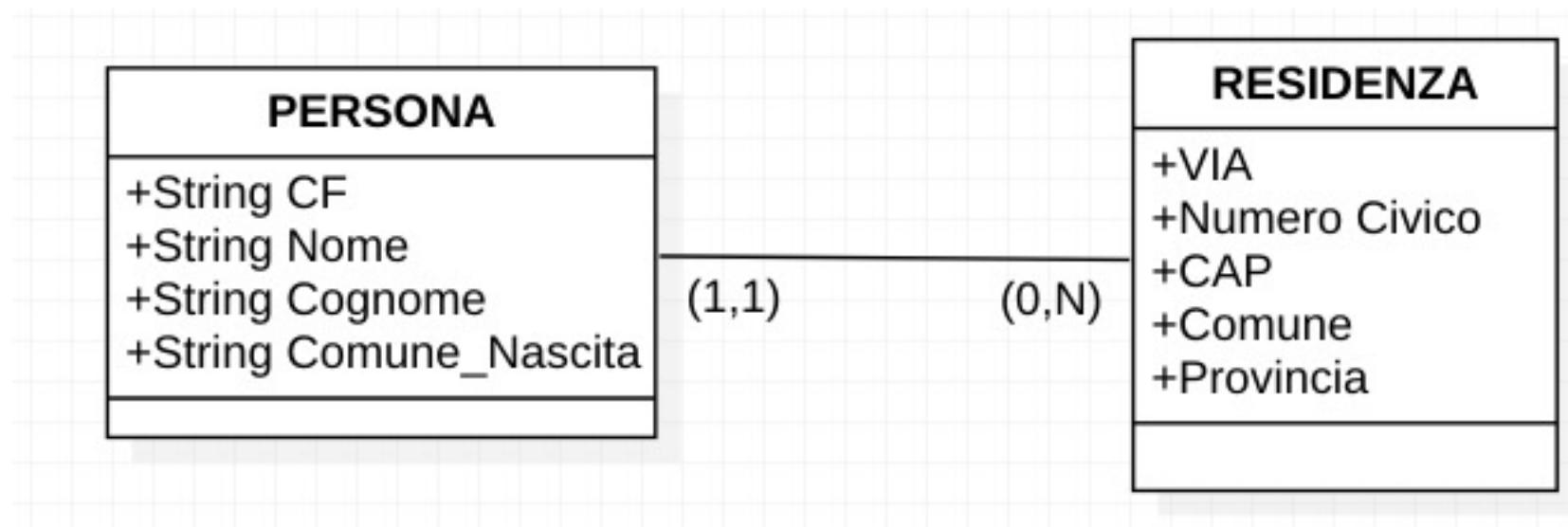
# ELIMINAZIONE ATTRIBUTI STRUTTURATI

PERSONA
+String CF
+String Nome
+String Cognome
+Residenza Residenza
+String Comune_Nascita

RESIDENZA
+VIA
+Numero Civico
+CAP
+Comune
+Provincia



# 1) INTRODUZIONE DI UNA CLASSE PER L'ATTRIBUTO STRUTTURATO



## 2) ESTRAZIONE DEGLI ATTRIBUTI DELLA CLASSE

PERSONA
+String CF
+String Nome
+String Cognome
+String Comune_Nascita
+String Via
+String Numero_Civico
+String CAP
+String Comune
+String Provincia

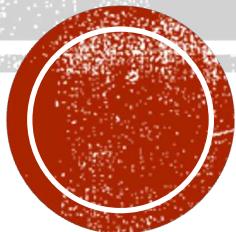


# 3) TRASCURARE LA STRUTTURA DELL'ATTRIBUTO

<b>PERSONA</b>
+String CF
+String Nome
+String Cognome
+String Comune_Nascita
+String Residenza



# **PARTIZIONAMENTO/ACCORPAMENTO DI ENTITÀ/ASSOCIAZIONI**

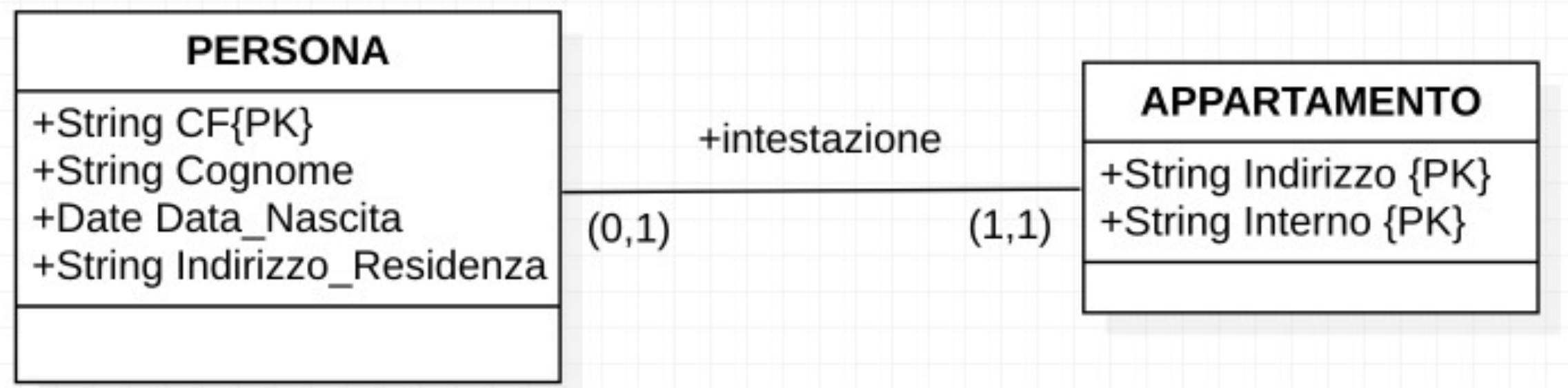


# ACCORPAMENTI DI ENTITÀ

- Accorpamento dei concetti operando sulla struttura.
- È conveniente quando le operazioni accedono a dati presenti su entrambe le entità.
- Si effettua generalmente su associazioni di tipo 1:1.
- Presenza di valori nulli.



# ACCORPAMENTI DI ENTITÀ - ESEMPIO



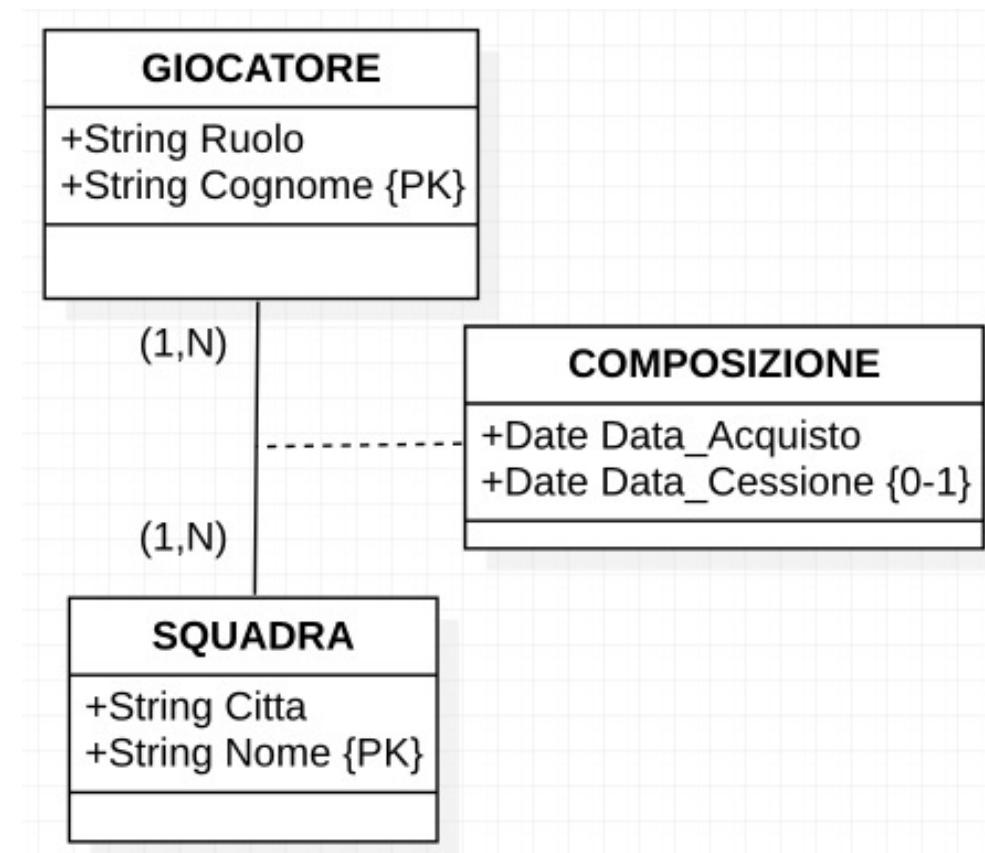
# ACCORPAMENTI DI ENTITÀ – ESEMPIO (2)

- Le operazioni più frequenti su persona richiedono sempre dati relativi all'appartamento che occupa:

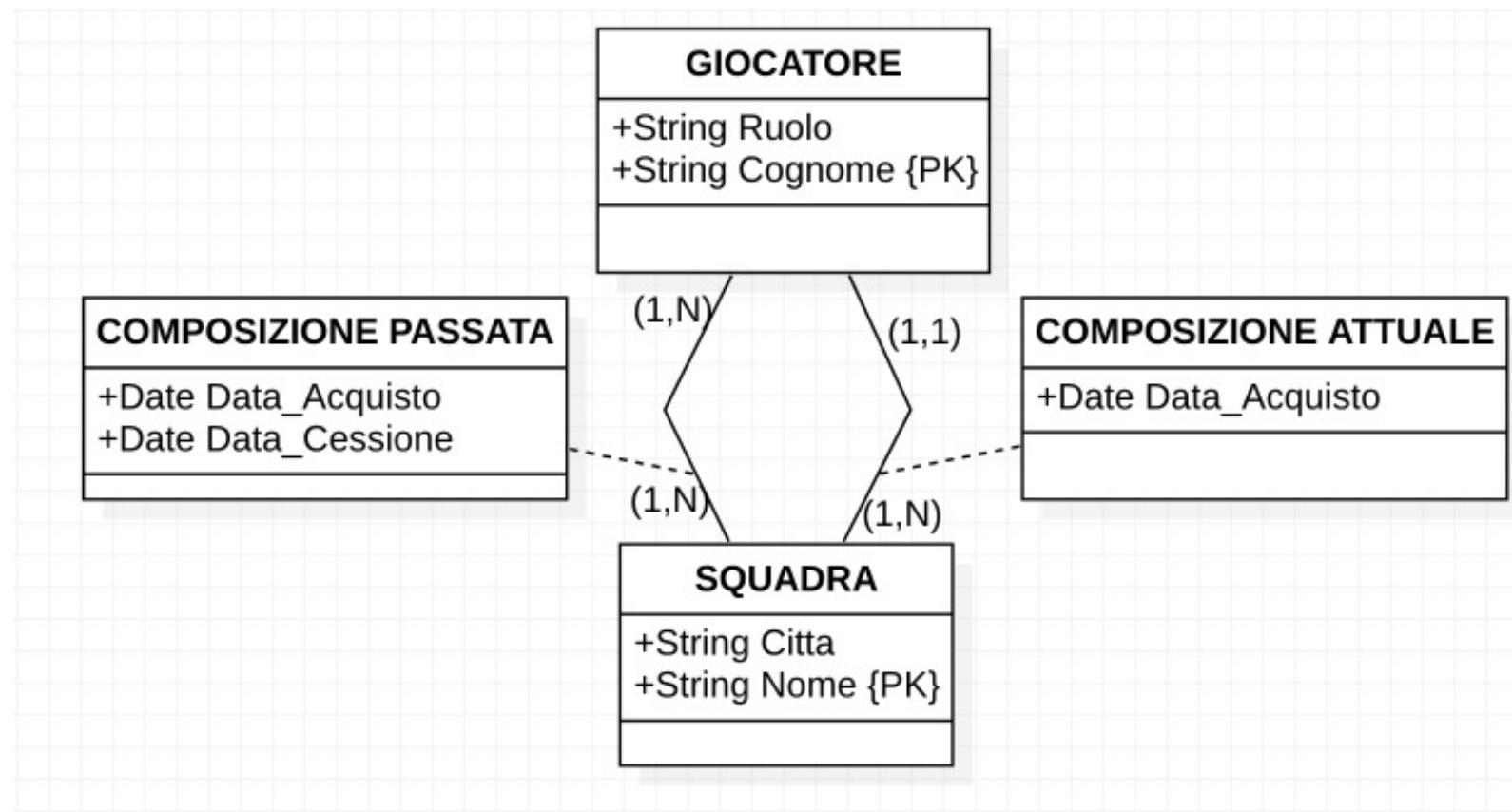
PERSONA
+String CF{PK}
+String Cognome
+String Data_Nascita
+String Indirizzo_Residenza
+String Indirizzo_Appartamento{0-1}
+String Interno_Appartamento{0-1}



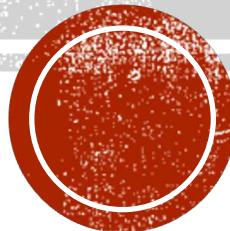
# DECOMPOSIZIONE DI ASSOCIAZIONI - ESEMPIO



# DECOMPOSIZIONE DI ASSOCIAZIONI – ESEMPIO (2)



# **IDENTIFICATORE CHIAVI PRIMARIE**



# IDENTIFICATORI

- Scelta della chiave primaria per la costruzione degli indici per il recupero efficiente dei dati.
- Criteri generali:
  - Escludere attributi con valori nulli.
  - Numero minimo di attributi (dimensioni ridotte degli indici).
  - Identificatore coinvolto in molte operazioni.
  - Velocità di accesso all'indice.
    - Tempo minimo per confrontare i valori.
- Se nessuno degli identificatori candidati soddisfa tali criteri si introduce un ulteriore attributo all'entità:
  - Questo attributo conterrà valori speciali (**codici**) generati appositamente per identificare le occorrenze delle entità.



# IDENTIFICATORI: ESEMPIO

- Nel caso in cui l'entità Studente abbia **due** possibili chiavi:
  1. Matricola (10 caratteri, numerici).
  2. Codice fiscale (16 caratteri, alfanumerici).
- Perché è opportuno selezionare l'attributo *Matricola* come identificatore primario?
  - *La matricola richiede un tempo minore per confrontare due valori tra loro: la velocità di accesso all'indice è ridotta*

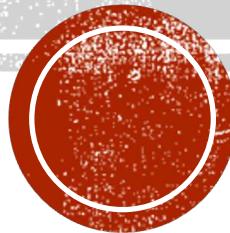




# FINE

Per eventuali domande: (in ordine di preferenza personale)

- Ora.
- Chat di Teams
- Mail: [silvio.barra@unina.it](mailto:silvio.barra@unina.it)

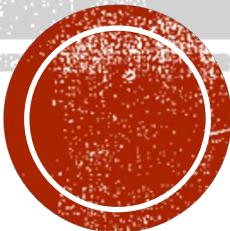


# BASI DI DATI I

- Structured Query Language
- Data Definition Language - SQL



# **STRUCTURED QUERY LANGUAGE (SQL)**



# SQL

- Il linguaggio SQL permette la **definizione**, la **manipolazione** (aggiornamento e recupero) e la **gestione** di basi di dati relazionali.
- È una delle ragioni del successo dei db relazionali in ambito commerciale:  
*essendo uno standard in tutti i DBMS relazionali, gli utenti sono poco propensi a migrare verso data model diversi, quali il gerarchico o il reticolare.*



# SQL – I VANTAGGI DI UNO STANDARD (1)

- SQL è considerato la principale ragione di successo dei database relazionali. Infatti:
  - È uno standard – è possibile migrare da un DBMS relazionale verso un altro DBMS relazionale senza eccessivi problemi.
  - I programmi applicativi usano lo stesso insieme di istruzioni SQL per accedere a DBMS relazionali diversi, senza dover cambiare i sottolinguaggi.
- *I costi di formazione del personale sono ridotti*: la formazione è concentrata su di un solo linguaggio.
- *Produttività*: i tecnici, una volta imparato il linguaggio (e poiché usano solo questo) diventano sempre più esperti e produttivi.
- *Portabilità delle applicazioni*: le applicazioni possono essere spostate da una macchina all'altra, se entrambe usano SQL.



# SQL – I VANTAGGI DI UNO STANDARD (2)

- *Longevità delle applicazioni:* uno standard tende a rimanere in voga per diverso tempo e non c'è necessità di riscrivere le applicazioni.
- *Ridotta dipendenza da un singolo produttore:* quando non è usato un linguaggio proprietario, è più facile usare differenti produttori di DBMS.

Di conseguenza il mercato sarà più competitivo ed i prezzi calano.

- *Comunicazione fra sistemi:* differenti DBMS e programmi applicativi possono comunicare più facilmente.



# **MA HA ANCHE DEGLI SVANTAGGI...**

- Può in qualche modo limitare la creatività e l'innovazione.
- Può non incontrare tutte le necessità di un'industria.
- Può essere difficile da cambiare, poiché molti produttori hanno investito su di esso.
- L'utilizzo di speciali funzionalità aggiunte a SQL da qualche produttore può far perdere i vantaggi di cui alle slide precedenti.



# SQL: UN PO' DI STORIA

- Nel 1970 Codd propone il modello relazionale: iniziano esperimenti e ricerche per la realizzazione di linguaggi relazionali, cioè di linguaggi in grado di realizzare le caratteristiche del modello astratto.
- Il primo risultato è ***SEQUEL*** (*Structured English QUery Language*), definito all'IBM Research:
  - Facile da imparare e utilizzare;
  - Basato su termini inglesi che mascherano i difficili concetti **dell'algebra relazionale**.



# SQL: UN PO' DI STORIA (2)

- Una versione rivista, il **SEQUEL/2**, ridenominata **SQL** (**S**tructured **Q**uery **L**anguage) viene definita nel 1976.
- Il primo prodotto basato su SQL viene chiamato **Oracle** (1979), lanciato dalla Relational Software, Inc.
- Nel 1981 IBM annuncia un prodotto SQL denominato **SQL/Data System**; nel 1983 viene rilasciato il DBMS relazionale **DB2** compatibile con SQL/DS.



# SQL: UN PO' DI STORIA (3)

- Oggi SQL è implementato da tutti i principali fornitori di DBMS, ed è il linguaggio per database più usato al mondo.
- L'ANSI e l'ISO hanno sviluppato una serie di standard per SQL:
  - ANSI SQL-86, SQL-92 (SQL2) ed SQL3.
  - Sfortunatamente ogni DBMS relazionale implementa un suo livello (o **dialetto**) di SQL, che è un'estensione o un sottoinsieme di un livello standard.



# IL LINGUAGGIO SQL

- SQL fornisce *statement* per la definizione di dati, query e aggiornamenti, quindi è sia un **DDL** che un **DML**.
- Fornisce inoltre facility per definire viste e per ricavare indici.
- SQL può essere usato **interattivamente** (con maschere del DBMS) o essere **incorporato (embedded)** in programmi C, Cobol, Java, etc.



# **DEFINIZIONE DI DATI, SCHEMI E VINCOLI IN SQL**



# TERMINOLOGIA SQL

- SQL ha una terminologia diversa da quella classica relazionale:
  - Relazione → Tabella
  - Tupla → Riga
  - Attributo → Colonna



# CONCETTI DI SCHEMA

- Il concetto di **schema** SQL è usato per raggruppare tabelle ed altri costrutti che appartengono alla stessa applicazione di database.
- Uno schema SQL è identificato da un **nome dello schema**, ed include un identificatore di autorizzazione per indicare l'utente proprietario dello schema, così come dei **descrittori** per ogni elemento dello schema.



# CONCETTO DI SCHEMA (2)

- Uno schema include:
  - Tabelle
  - Domini
  - Viste
  - Altri costrutti, quali permessi di autorizzazione, etc.
- La sintassi per creare uno schema è

**CREATE SCHEMA** *nome\_schema* **AUTHORIZATION** *nome\_utente*

- Crea uno schema chiamato *nome\_schema*, il cui proprietario è l'utente con account *nome\_utente*.



# IL COMANDO CREATE TABLE

- **CREATE TABLE** è usato per specificare una nuova relazione, assegnandole un **nome** ed un **insieme di attributi e vincoli**.
- Gli attributi sono specificati da un **nome**, un **tipo di dato** per definire il dominio dei valori, ed eventuali **vincoli**.
- In ultimo si specifica la chiave, i vincoli di integrità di entità e di integrità referenziale.



# ISTANZA DI DATABASE RELAZIONALE

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
John		Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5	
Franklin		Wong	333445555	1955-12-08	638 Vass, Houston, TX	M	40000	888665555	5	
Alicia		Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4	
Jennifer		Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4	
Ramesh		Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5	
Joyce		English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5	
Ahmad		Jabber	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4	
James		Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	null	1	

DEPARTMENT	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE	DEPT_LOCATIONS	DNUMBER	DLOCATION
Research	5	333445555	1968-05-22		Houston		
Administration	4	987654321	1995-01-01		Stafford		
Headquarters	1	888665555	1981-06-19		Bellaire		
					Sugarland		

WORKS_ON	ESSN	PNO	HOURS
123456789	1	32.5	
123456789	2	7.5	
666884444	3	40.0	
453453453	1	20.0	
453453453	2	20.0	
333445555	2	10.0	
333445555	3	10.0	
333445555	10	10.0	
333445555	20	10.0	
999887777	30	30.0	
999887777	10	10.0	
987987987	10	35.0	
987987987	30	5.0	
987654321	30	20.0	
987654321	20	15.0	
888665555	20	null	

PROJECT	PNAME	PNUMBER	PLOCATION	DNUM
ProductX	1	Bellaire	5	
ProductY	2	Sugarland	5	
ProductZ	3	Houston	5	
Computerization	10	Stafford	4	
Reorganization	20	Houston	1	
Newbenefits	30	Stafford	4	

DEPENDENT	ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP
333445555		Alice	F	1986-04-05	DAUGHTER
333445555		Theodore	M	1983-10-25	SON
333445555		Joy	F	1958-05-03	SPOUSE
987654321		Abner	M	1942-02-28	SPOUSE
123456789		Michael	M	1988-01-04	SON
123456789		Alice	F	1988-12-30	DAUGHTER
123456789		Elizabeth	F	1967-05-05	SPOUSE

Un'istanza del database "Company"



# CREATE TABLE: ESEMPIO

```
CREATE TABLE EMPLOYEE
  ( FNAME          VARCHAR(15)      NOT NULL,
    MINIT           CHAR,
    LNAME           VARCHAR(15)      NOT NULL,
    SSN             CHAR(9)         NOT NULL,
    BDATE           DATE,
    ADDRESS         VARCHAR(30),
    SEX              CHAR,
    SALARY          DECIMAL(10,2),
    SUPERSSN        CHAR(9),
    DNO              INT             NOT NULL,
    PRIMARY KEY (SSN),
    FOREIGN KEY (SUPERSSN) REFERENCES EMPLOYEE(SSN),
    FOREIGN KEY (DNO) REFERENCES DEPARTMENT(DNUMBER));
CREATE TABLE DEPARTMENT
  ( DNAME           VARCHAR(15)      NOT NULL,
    DNUMBER          INT             NOT NULL,
    MGRSSN          CHAR(9)         NOT NULL,
    MGRSTARTDATE    DATE,
    PRIMARY KEY (DNUMBER),
    UNIQUE (DNAME),
    FOREIGN KEY (MGRSSN) REFERENCES EMPLOYEE(SSN));
CREATE TABLE DEPT_LOCATIONS
  ( DNUMBER          INT             NOT NULL,
    DLOCATION        VARCHAR(15)      NOT NULL,
    PRIMARY KEY (DNUMBER, DLOCATION),
    FOREIGN KEY (DNUMBER) REFERENCES DEPARTMENT(DNUMBER));
CREATE TABLE PROJECT
  ( PNAME           VARCHAR(15)      NOT NULL,
    PNUMBER          INT             NOT NULL,
    PLOCATION        VARCHAR(15),
    DNUM             INT             NOT NULL,
    PRIMARY KEY (PNUMBER),
    UNIQUE (PNAME),
    FOREIGN KEY (DNUM) REFERENCES DEPARTMENT(DNUMBER));
CREATE TABLE WORKS_ON
  ( ESSN            CHAR(9)         NOT NULL,
    PNO              INT             NOT NULL,
    HOURS            DECIMAL(3,1)    NOT NULL,
    PRIMARY KEY (ESSN, PNO),
    FOREIGN KEY (ESSN) REFERENCES EMPLOYEE(SSN),
    FOREIGN KEY (PNO) REFERENCES PROJECT(PNUMBER));
CREATE TABLE DEPENDENT
  ( ESSN            CHAR(9)         NOT NULL,
    DEPENDENT_NAME  VARCHAR(15)      NOT NULL,
    SEX              CHAR,
    BDATE           DATE,
    RELATIONSHIP    VARCHAR(8),
    PRIMARY KEY (ESSN, DEPENDENT_NAME),
    FOREIGN KEY (ESSN) REFERENCES EMPLOYEE(SSN));
```

Gli statement SQL2 per definire lo schema del database "Company"

# TIPI DI DATI E DOMINI

- Tipi di dati disponibili in SQL2:
  - Numerici
    - Interi (INTEGER o INT, SMALLINT)
    - Reali (FLOAT, REAL, DOUBLE PRECISION)
    - Numeri formattati (DECIMAL(i,j), DEC(i,j), NUMERIC(i,j))
      - i, detta precisione, indica il numero di cifre decimali.
      - j, detta scala, indica il numero di cifre dopo la virgola.
  - Stringhe di caratteri
    - A lunghezza fissa (CHAR(n), CHARACTER(n))
    - A lunghezza variabile (VARCHAR(n) o CHAR VARYING(n))
      - Per default n, il numero massimo di caratteri, è 1.



# TIPI DI DATI E DOMINI (2)

- **Stringhe di bit:**

- A lunghezza fissa (BIT(n))
- A lunghezza variabile (BIT VARYING(n))

- **DATE:**

- Ha dieci posizioni, con componenti YEAR, MONTH e DAY.
- Formato *YYYY-MM-DD*.

- **TIME:**

- Ha (almeno) otto posizioni con componenti HOUR, MINUTE e SECOND.
- Formato *HH:MM:SS*.



# **DOMINI PERSONALIZZATI**



# I VALORI NULL E DEFAULT

- Poiché SQL consente che un attributo abbia valore **null**, se si vuole impedire ciò si usa il vincolo  
**NOT NULL**.
  - Tale vincolo deve sempre essere specificato per la chiave primaria (*vincolo di integrità di entità*).
- È anche possibile specificare un valore di default per un attributo, attraverso la clausola  
**DEFAULT <value>**, dopo la dichiarazione dell'attributo
  - Senza tale clausola il valore di default di un attributo è null.



# ALTRI VINCOLI

- Dopo le specifiche degli attributi, possono essere specificati i **vincoli di tabella**, quali **chiave** ed **integrità referenziale**:
  - La clausola **PRIMARY KEY** specifica uno o più attributi che faranno da chiave primaria.
  - La clausola **UNIQUE** specifica una chiave alternativa.
  - La clausola **FOREIGN KEY** specifica l'integrità referenziale.



## ALTRI VINCOLI (2)

- Il progettista dello schema può specificare l'azione da intraprendere se si viola un vincolo di integrità referenziale, attraverso la cancellazione di una tupla referenziata o attraverso la modifica di un valore di chiave referenziata.
- L'azione referenziale triggered può essere specificata nella clausola **FOREIGN KEY**.
  - Possibili azioni sono **SET NULL**, **CASCADE** e **SET DEFAULT**, qualificate da opzioni **ON DELETE** e **ON UPDATE**.



# ALTRI VINCOLI: ESEMPIO

```
CREATE TABLE EMPLOYEE
(
    ...,
    DNO          INT  NOT NULL  DEFAULT 1,
    CONSTRAINT EMPPK
        PRIMARY KEY (SSN),
    CONSTRAINT EMPSUPERFK
        FOREIGN KEY (SUPERSSN) REFERENCES EMPLOYEE(SSN)
            ON DELETE SET NULL  ON UPDATE CASCADE ,
    CONSTRAINT EMPDEPTFK
        FOREIGN KEY (DNO) REFERENCES DEPARTMENT(DNUMBER)
            ON DELETE SET DEFAULT  ON UPDATE CASCADE );
```

```
CREATE TABLE DEPARTMENT
(
    ...,
    MGRSSN  CHAR(9) NOT NULL DEFAULT '888665555',
    ...,
    CONSTRAINT DEPTPK
        PRIMARY KEY (DNUMBER),
    CONSTRAINT DEPTSK
        UNIQUE (DNAME),
    CONSTRAINT DEPTMGRFK
        FOREIGN KEY (MGRSSN) REFERENCES EMPLOYEE(SSN)
            ON DELETE SET DEFAULT  ON UPDATE CASCADE );
```

```
CREATE TABLE DEPT_LOCATIONS
(
    ...,
    PRIMARY KEY (DNUMBER, DLOCATION),
    FOREIGN KEY (DNUMBER) REFERENCES DEPARTMENT(DNUMBER)
        ON DELETE CASCADE  ON UPDATE CASCADE );
```

Specifica di valori di default e azioni referenziali triggered.



# ALTRI VINCOLI: ESEMPIO

- Nell'esempio, per la chiave esterna SUPERSSN di EMPLOYEE ci sono i vincoli:
  - **SET NULL ON DELETE**
    - Se la tupla dell'impiegato che supervisiona viene cancellata, il valore di SUPERSSN è posto a null in tutte le tuple impiegato che lo referenziano.
  - **CASCADE ON UPDATE**
    - Se il valore SSN di un impiegato che supervisiona è aggiornato, il nuovo valore è riportato in SUPERSSN di tutte le tuple impiegato che referenziano il valore aggiornato.
- Ai vincoli può essere dato un nome (per poterli riutilizzare), usando la keyword **COSTRAINT**.



# CHECK CONSTRAINT

- Il vincolo CHECK serve a controllare che un determinato attributo rispetta determinate condizioni.
- La sintassi è **CHECK(*vincolo*)**

dove vincolo è una determinata condizione che deve essere rispettata

```
CREATE DOMAIN sesso AS CHAR  
    CHECK(sesso = 'm' OR sesso='f')
```

```
CREATE DOMAIN voto AS SMALLINT  
    CHECK(voto> 0 AND voto<=30)
```

```
CONSTRAINT checkLode  
    CHECK(lode=TRUE AND voro=30) OR NOT lode=TRUE
```



```
CREATE Table impiegato(
    CF codice_fiscale,
    fname VARCHAR(15),
    mname CHAR,
    lname VARCHAR(15),
    gender CHAR,
    dno int,
    salary DECIMAL(10,2),
    CONSTRAINT emppk PRIMARY KEY(CF),
    CONSTRAINT fnameNN CHECK(fname IS NOT NULL),
    CONSTRAINT lnameNN CHECK(lname IS NOT NULL),
    CONSTRAINT depFK FOREIGN KEY(dno) REFERENCES dipartimento(dnumber),
    CONSTRAINT genderCorrect CHECK(gender='m' OR gender='f'),
    CONSTRAINT salaryCheck CHECK(salary>1000.00),
    CONSTRAINT cfnumberUNIQUE UNIQUE(CF, dno)
)
```



# RECAP VINCOLI

- CONSTRAINT <nome\_vincolo> <vincolo>
- <vincolo>:= CHECK <espressione booleana>
  - UNIQUE (<lista\_attributi>)
  - PRIMARY KEY (<lista attributi>)
  - FOREIGN KEY (<lista\_attributi\_FK>) REFERENCES <nome\_tabella>(<lista\_att\_PK>)[ON DELETE <azione>][ON UPDATE <azione>]
- <azione>:= NO ACTION | CASCADE | SET DEFAULT | SET NULL



# DAL DDL ALL'SQL

METADATI

Definizione

CREATE TABLE

Modifica

ALTER TABLE

Cancellazione

DROP TABLE

INSERT

UPDATE

DELETE

DATI

SELECT

Interrogazione

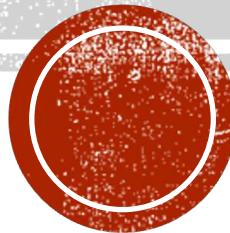




# FINE

Per eventuali domande: (in ordine di preferenza personale)

- Ora.
- Chat di Teams
- Mail: [silvio.barra@unina.it](mailto:silvio.barra@unina.it)



# BASI DI DATI I

- Algebra Relazionale
- Operatore di Selezione
- Operazione di Proiezione

# L'ALGEBRA RELAZIONALE

- Abbiamo visto i concetti per definire la struttura ed i vincoli di un database nel modello relazionale.
- Realizzato lo scheletro di un db, abbiamo bisogno di un insieme di operazione per manipolare i dati.
- **Algebra relazionale: collezione di operazioni usate per manipolare intere relazioni.**



# **PERCHÉ “ALGEBRA”**

- Proprietà di un’algebra, in senso matematico:
  - È basata su operatori e domini dei valori.
- Gli operatori mappano gli argomenti da un dominio ad un altro.
- Quindi un’espressione che coinvolge operatori ed argomenti genera un nuovo valore nel dominio.



# L'ALGEBRA "RELAZIONALE"

- Applichiamo la definizione di algebra al modello relazionale:
  - Dominio: le relazioni;
  - Operatori:
    - Operazioni su insiemi, ereditate dalla teoria matematica degli insiemi: *Unione*, *Intersezione*, *Differenza* e *Prodotto Cartesiano*.
    - Operazioni specificatamente disegnate per database relazionali: *Select*, *Project* e *Join*.
  - Il risultato dell'applicazione di un operatore su una relazione è ancora una relazione.

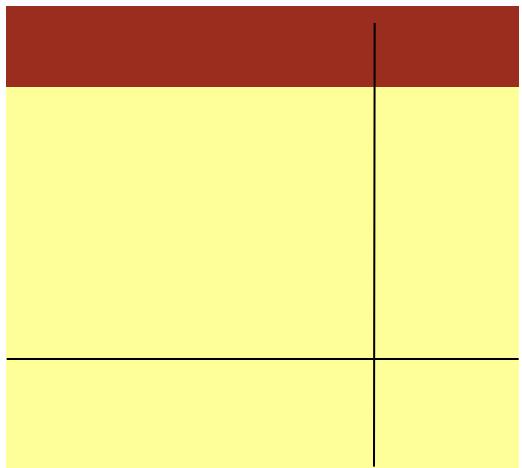


# SELEZIONE E PROIEZIONE

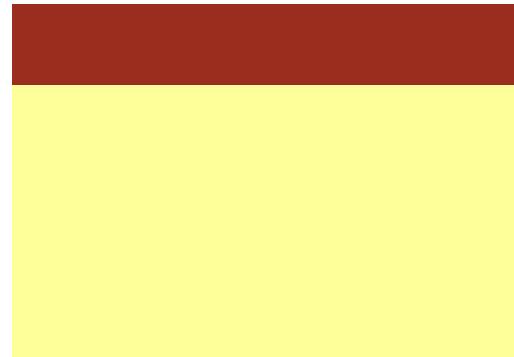
- operatori "ortogonali"
- **selezione:**
  - decomposizione orizzontale
- **proiezione:**
  - decomposizione verticale



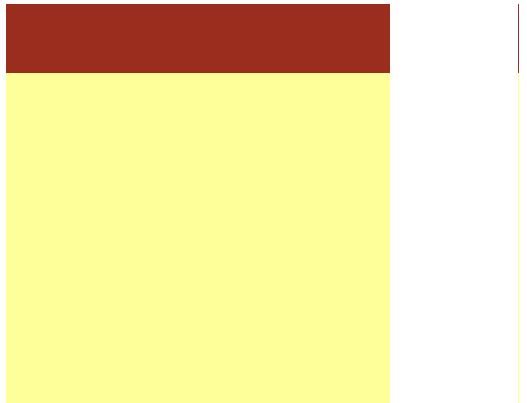
(animazione)



**selezione**



**proiezione**



# **OPERAZIONE DI PROIEZIONE**



# L'OPERATORE SELECT: $\sigma$

- Usato per selezionare un sottoinsieme di tuple in una relazione che soddisfa una **condizione di selezione**.
- Si indica con  $\sigma$ .
- **Esempi:**
  - $\sigma_{dno=4}$  (Employee) restituisce il sottoinsieme degli impiegati che lavora nel dipartimento numero 4.
  - $\sigma_{salary > 30000}$  (Employee) restituisce il sottoinsieme degli impiegati con salario  $> 30000$  \$.



# L'OPERATORE SELECT (2)

- Sintassi:

$\sigma_{<\text{selection condition}>} (<\text{relazione}>)$

- La condizione di selezione è **un'espressione booleana** formata da clausole della forma:

- $<\text{nome\_attributo}> \text{ op\_confronto } <\text{valore costante}>$

oppure

- $<\text{nome\_attributo}> \text{ op\_confronto } <\text{nome\_attributo}>$   
eventualmente concatenate con operatori logici.

- **op\_confronto** è uno degli operatori  $\{=, \neq, <, >, \leq, \geq\}$

- **Esempio:**

- $\sigma_{(dno=4 \text{ and } \text{Salary}>25000 \text{ or } dno=5 \text{ and } \text{Salary}>30000)}(\text{Employee})$



# L'OPERATORE SELECT (3)

- La condizione di selezione è valutata per ogni tupla individualmente: se è vera, la tupla è inserita nella relazione risultante.
- Il **grado** della relazione risultante dopo un'operazione di select è **uguale** a quello della relazione di partenza.
- Il numero di tuple risultanti  $t_r$  è minore o uguale di quelle di partenza  $t_p$ :

$$t_r \leq t_p$$

- Il rapporto  $t_r/t_p$  è detto **selettività** della condizione.



# PROPRIETÀ DELLA SELECT

- L'operatore Select è **unario**.
- L'operatore di Select è commutativo:

$$\sigma < \text{cond}_1 > (\sigma < \text{cond}_2 > (R)) = \sigma < \text{cond}_2 > (\sigma < \text{cond}_1 > (R))$$





# OPERATORE DI PROIEZIONE



# L'OPERATORE PROJECT: $\pi$

- Usato per selezionare **un sottoinsieme delle colonne** di una relazione.
- Sintassi:

$$\pi \langle \text{attribute\_list} \rangle (\langle \text{relazione} \rangle)$$

- La relazione risultante ha gli attributi specificati nella **<attribute\_list>**, nello stesso ordine in cui appaiono nella lista.
- Il grado della relazione risultante da un'operazione di PROJECT è uguale al numero di attributi specificati nella **<attribute\_list>**.



# L'OPERATORE PROJECT (2)

- Se nella <attribute\_list> non è presente una chiave candidata, si potrebbero avere delle tuple duplicate: la PROJECT le rimuove implicitamente.
- Il numero  $t_r$  di tuple risultanti è minore o uguale del numero  $t_p$  di tuple di partenza:
  - Se la lista di attributi include una chiave candidata della relazione, sarà  $t_r=t_p$ .
- $\pi_{<\text{list1}>}(\pi_{<\text{list2}>}(R)) = \pi_{<\text{list1}>}(R)$  se <list2> contiene gli attributi presenti in <list1>; altrimenti la parte sinistra non è corretta
- La commutatività non vale per la PROJECT.



# SELEZIONE E PROIEZIONE

- Combinando selezione e proiezione, possiamo estrarre interessanti informazioni da una relazione.



Matricola Cognome	
7309	Rossi
5998	Neri
5698	Neri

$\pi_{\text{Matricola,Cognome}} (\sigma_{\text{Stipendio} > 50} (\text{Impiegati}))$



# SEQUENZE DI OPERAZIONI

- Per applicare più operazioni una dopo l'altra si può scrivere un'unica espressione dell'algebra relazionale.

- **Esempio:**

- Trovare *nome*, *cognome* e *salario* dei dipendenti che lavorano nel dipartimento n° 5:

$$\pi_{<\text{FNAME}, \text{LNAME}, \text{SALARY}>}(\sigma_{\text{dno}=5}(\text{EMPLOYEE}))$$

- Alternativamente si possono creare risultati intermedi:
  - $\text{DEPS\_EMPS} = \sigma_{\text{dno}=5}(\text{EMPLOYEE})$
  - $\text{RESULT} = \pi_{<\text{FNAME}, \text{LNAME}, \text{SALARY}>}(\text{DEPS\_EMPS})$



# L'OPERAZIONE RENAME

- Per rinominare gli attributi in una relazione che risulta dall'algebra relazionale, semplicemente listiamo i nuovi nomi di attributi in parametri:
  - $\text{TEMP} = \sigma_{\text{dno}=5}(\text{EMPLOYEE})$
  - $R(\text{FirstName}, \text{LastName}, \text{Salary}) = \pi_{< \text{FNAME}, \text{LNAME}, \text{SALARY}>}(\text{TEMP})$
- È un operatore unario.

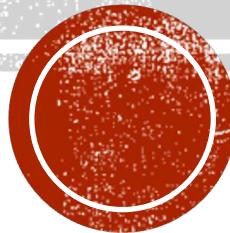




# FINE

Per eventuali domande: (in ordine di preferenza personale)

- Ora.
- Chat di Teams
- Mail: [silvio.barra@unina.it](mailto:silvio.barra@unina.it)



AR (2° parte) + SQL (2° parte)

# BASI DI DATI I

- Algebra Relazionale
- Operazioni Insiemistiche
- Operazioni di Join (prima parte)
- Query di base in SQL
- Renaming, Distinct e Operazioni Insiemistiche

# **OPERAZIONI INSIEMISTICHE**



# OPERAZIONI INSIEMISTICHE

- Una relazione è un insieme di tuple, quindi possiamo applicare le classiche operazioni insiemistiche.
- Il risultato nella combinazione di due relazioni per mezzo di un'operazione su insieme è una nuova relazione.
- Per poter applicare un'operazione insiemistica a due relazioni, queste devono avere la stessa struttura, ovvero essere **union compatibili**...



# UNION COMPATIBILITY

- Due relazioni  $R(A_1, A_2, \dots, A_n)$  e  $S(B_1, B_2, \dots, B_n)$  sono **union compatibili** se hanno lo stesso grado e  
 $\text{dom}(A_i) = \text{dom}(B_i)$  per  $1 \leq i \leq n$ .



# OPERAZIONI AMMISSIBILI

- Su due relazioni  $R$  ed  $S$  *union compatibili*, è possibile effettuare :
  - Unione
  - Intersezione
  - Differenza



# ESEMPIO DI UNIONE

- Trovare il SSN di tutti gli impiegati che lavorano o nel dipartimento n° 5 o supervisionano direttamente un impiegato che lavora nel dipartimento n° 5:
  - $DEPS\_EMPS = \sigma_{dno=5}(EMPLOYEE)$
  - $Result1 = \pi_{SSN}(DEPS\_EMPS)$
  - $Result2 = \pi_{SUPERSSN}(DEPS\_EMPS)$
  - $RESULT = Result1 \cup Result2$

Result1	123456789
	333444555
	666888444
	453453453

Result2	333444555
	888666555

Result	123456789
	333444555
	666888444
	453453453
	888666555



# OPERAZIONI

- **Unione:**
  - Il risultato di questa operazione,  $R \cup S$ , è la relazione che include tutte le tuple che sono in  $R$  o in  $S$ , oppure in  $R$  ed  $S$ . Le tuple duplicate sono eliminate.
- **Intersezione:**
  - Il risultato di questa operazione,  $R \cap S$ , è la relazione che include tutte le tuple che sono sia in  $R$  che in  $S$ .
- **Differenza:**
  - Il risultato di questa operazione,  $R - S$ , è la relazione che include tutte le tuple che sono in  $R$  ma non in  $S$ .
- Si adotta la convenzione che il risultato ha gli stessi nomi di attributi della prima relazione.



# OPERAZIONI: ESEMPI

Date due relazioni S ed I

STUDENT	FN	LN
	Susan	Yao
	Ramesh	Shah
	Johnny	Kohler
	Barbara	Jones
	Amy	Ford
	Jimmy	Wang
	Ernest	Gilbert

INSTRUCTOR	FNANE	LNAME
	John	Smith
	Ricardo	Browne
	Susan	Yao
	Francis	Johnson
	Ramesh	Shah

FN	LN
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert
John	Smith
Ricardo	Browne
Francis	Johnson

FN	LN
Susan	Yao
Ramesh	Shah

$S \cap I$

FN	LN
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

$S - I$

FN	LN
John	Smith
Ricardo	Browne
Francis	Johnson

$I - S$

$S \cup I$



# PROPRIETÀ DELLE OPERAZIONI

- Unione ed intersezione sono commutative, associative e possono essere applicate ad un numero qualsiasi di relazioni
  - $R \cup S = S \cup R$
  - $R \cap S = S \cap R$
  - $R \cup (S \cup T) = (R \cup S) \cup T$
  - $R \cap (S \cap T) = (R \cap S) \cap T$
- La differenza non è commutativa.
  - In generale  $R - S \neq S - R$ .



# PRODOTTO CARTESIANO (CROSS PRODUCT O CROSS JOIN)

- Non è necessario che le relazioni siano union compatibili
  - $R(A_1, A_2, \dots, A_n) \times S(B_1, B_2, \dots, B_m) = Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$
  - In Q si ha una tupla per ogni combinazione di una da R ed una da S.
  - Se R contiene  $n_r$  tuple ed S contiene  $n_s$  tuple, allora  $R \times S$  contiene  $n_r \cdot n_s$  tuple.
- In caso di attributi con lo stesso nome nelle due relazioni, si deve effettuare il rename di uno dei due.



# PRODOTTO CARTESIANO: ESEMPIO

- Vogliamo trovare per ogni impiegato di sesso femminile, una lista dei suoi familiari a carico:
  - $\text{FEMALE\_EMPS} = \sigma_{\text{Sex}=\text{"F"}}(\text{EMPLOYEE})$
  - $\text{EMP NAMES} = \pi_{<\text{FNAME}, \text{LNAME}, \text{SSN}>}(\text{FEMALE\_EMPS})$
  - $\text{EMP\_DEPENDENTS} = \text{EMP NAMES} \times \text{DEPENDENTS}$
  - $\text{ACTUAL\_DEPENDENTS} = \sigma_{\text{SSN}=\text{ESSN}}(\text{EMP\_DEPENDENTS})$
  - $\text{RESULT} = \pi_{<\text{FNAME}, \text{LNAME}, \text{DEPENDENT\_NAME}>}(\text{ACTUAL\_DEPENDENTS})$



# PRODOTTO CARTESIANO: ESEMPIO (2)

FEMALE_EMPS	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4
	Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888865555	4
	Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

EMP NAMES	FNAME	LNAME	SSN
	Alicia	Zelaya	999887777
	Jennifer	Wallace	987654321
	Joyce	English	453453453

EMP_DEPENDENTS	FNAME	LNAME	SSN	ESSN	DEPENDENT_NAME	SEX	BDATE	***
	Alicia	Zelaya	999887777	333445555	Alice	F	1986-04-05	***
	Alicia	Zelaya	999887777	333445555	Theodore	M	1983-10-25	***
	Alicia	Zelaya	999887777	333445555	Joy	F	1958-05-03	***
	Alicia	Zelaya	999887777	987654321	Abner	M	1942-02-28	***
	Alicia	Zelaya	999887777	123456789	Michael	M	1988-01-04	***
	Alicia	Zelaya	999887777	123456789	Alice	F	1988-12-30	***
	Alicia	Zelaya	999887777	123456789	Elizabeth	F	1967-05-05	***
	Jennifer	Wallace	987654321	333445555	Alice	F	1986-04-05	***
	Jennifer	Wallace	987654321	333445555	Theodore	M	1983-10-25	***
	Jennifer	Wallace	987654321	333445555	Joy	F	1958-05-03	***
	Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	***
	Jennifer	Wallace	987654321	123456789	Michael	M	1988-01-04	***
	Jennifer	Wallace	987654321	123456789	Alice	F	1988-12-30	***
	Jennifer	Wallace	987654321	123456789	Elizabeth	F	1967-05-05	***
	Joyce	English	453453453	333445555	Alice	F	1986-04-05	***
	Joyce	English	453453453	333445555	Theodore	M	1983-10-25	***
	Joyce	English	453453453	333445555	Joy	F	1958-05-03	***
	Joyce	English	453453453	987654321	Abner	M	1942-02-28	***
	Joyce	English	453453453	123456789	Michael	M	1988-01-04	***
	Joyce	English	453453453	123456789	Alice	F	1988-12-30	***
	Joyce	English	453453453	123456789	Elizabeth	F	1967-05-05	***

ACTUAL_DEPENDENTS	FNAME	LNAME	SSN	ESSN	DEPENDENT_NAME	SEX	BDATE
	Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28

RESULT	FNAME	LNAME	DEPENDENT_NAME
	Jennifer	Wallace	Abner



# PRODOTTO CARTESIANO

- Operazione binaria
- Contiene sempre un numero di  $n$ -ple pari al prodotto delle cardinalità degli operandi (le  $n$ -ple sono tutte combinabili ).



## Impiegati

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

## Reparti

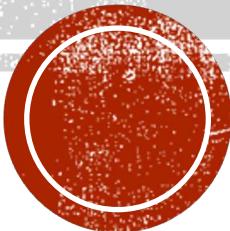
Codice	Capo
A	Mori
B	Bruni

## Impiegati × Reparti

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



# **OPERAZIONI DI JOIN (1° PARTE)**



# JOIN

- La sequenza di operazioni appena vista, riassumibile in  $\sigma_{join-condition}(R \times S)$  è abbastanza frequente: per questo è stata creata un'operazione speciale, chiamata JOIN.
- La JOIN, denotata con  $\bowtie$ , è usata per combinare tuple relate in una sola tupla.
  - $\sigma_{join-condition}(R \times S)$  diventa  $R \bowtie_{join-condition} S$



# JOIN: ESEMPIO

- Supponiamo di voler trovare il nome del manager di ciascun dipartimento:
  - Combiniamo ogni tupla dipartimento con la tupla impiegato il cui SSN fa match col valore MGRSSN nella tupla dipartimento.
- DEPT\_MGR= department  $\bowtie_{MGRSSN=SSN}$  EMPLOYEE
- RESULT=  $\pi_{DNAME, LNAME, FNAME}(\text{DEPT\_MGR})$

DEPT_MGR	DNAME	DNUMBER	MGRSSN	• • •	FNAME	MINIT	LNAME	SSN	• • •
Research	5	333445555	• • •	Franklin	T	Wong	333445555	• • •	• • •
Administration	4	987654321	• • •	Jennifer	S	Wallace	987654321	• • •	• • •
Headquarters	1	888665555	• • •	James	E	Borg	888665555	• • •	• • •



# JOIN: ESEMPIO SU PRODOTTO CARTESIANO

- L'esempio precedente sul prodotto cartesiano può essere risolto rimpiazzando le operazioni:

- $\text{EMP\_DEPENDENTS} = \text{EMPNAME} \times \text{DEPENDENTS}$
- $\text{ACTUAL\_DEPENDENTS} = \sigma_{\text{SSN}=\text{ESSN}}(\text{EMP\_DEPENDENTS})$

Con:

- $\text{ACTUAL\_DEPENDENTS} = \text{EMPNAME} \bowtie_{\text{SSN}=\text{ESSN}} \text{DEPENDENTS}$



# QUERY DI BASE IN SQL



# SQL E GLI INSIEMI

- Occorre fare un'importante distinzione tra SQL ed il modello relazionale formale:
  - SQL consente di avere **più tuple identiche** in tutti gli attributi, quindi in generale una tabella SQL **non è un insieme** di tuple.  
È invece un **multiset** (o bag) di tuple.
  - Alcune relazioni possono essere vincolate ad essere insiemi, usando il vincolo di chiave oppure l'opzione **DISTINCT** con lo statement SELECT...



# SQL, OPERAZIONI SUI DATI

- interrogazione:
  - **SELECT**
- modifica:
  - **INSERT, DELETE, UPDATE**



# IL COMANDO SELECT

- Il comando **SELECT** è l'istruzione di base per recuperare informazioni da un database.
- Il SELECT dell'SQL non ha relazioni con l'operatore di select dell'algebra relazionale.
- La forma di base, detta mapping o blocco di **SELECT FROM WHERE** è formata da tre clausole:

```
SELECT <lista_attributi>
FROM <lista_tabelle>
WHERE <condizione>
```

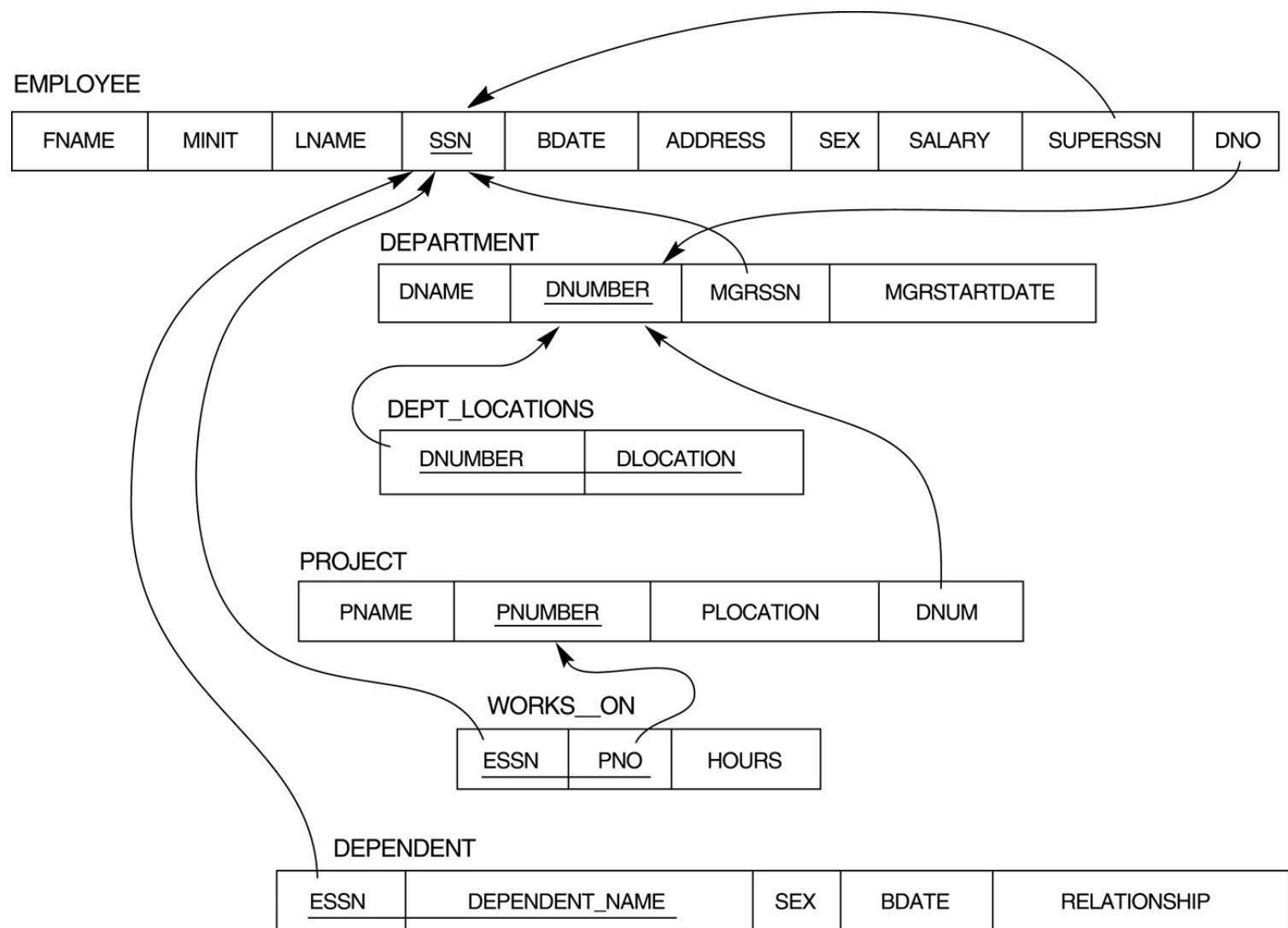


# IL COMANDO SELECT (2)

- **<lista\_attributi>** è una lista di nomi di attributi i cui valori devono essere recuperati dalla query.
- **<lista\_tabelle>** è una lista di nomi di relazioni richiesti per elaborare la query.
- **<condizione>** è un'espressione booleana di ricerca che identifica la tupla da ritrovare.



# MAPPING DELLO SCHEMA COMPANY



# IL COMANDO SELECT: ESEMPIO

- Trovare la data di nascita e l'indirizzo dell'impiegato di nome '*John B. Smith*':

```
SELECT      BDATE, ADDRESS  
FROM        EMPLOYEE  
WHERE       FNAME='JOHN' AND MINIT='B' AND  
           LNAME='SMITH';
```

- BDATE e ADDRESS sono detti anche **Attributi di proiezione**.
- Equivalente nell'algebra relazionale:

$$\pi_{\langle \text{BDATE}, \text{ADDRESS} \rangle} (\sigma_{\text{FNAME}='JOHN' \text{ AND } \text{MINIT}='B'} (\text{EMPLOYEE})) \\ \text{AND } \text{LNAME}='SMITH'$$


# IL COMANDO SELECT: ESEMPIO (2)

- Trovare cognome, nome e indirizzo di tutti gli impiegati del dipartimento '*Research*':

```
SELECT    FNAME, LNAME, ADDRESS  
FROM      EMPLOYEE, DEPARTMENT  
WHERE    DNAME='Research' AND DNUMBER=DNO;
```

- È simile alla sequenza SELECT-PROJECT-JOIN dell'algebra relazionale, ed è perciò detta query *select-project-join*.
- Equivalente nell'algebra relazionale:

$$\pi_{\langle \text{FNAME}, \text{LNAME}, \text{ADDRESS} \rangle} (\sigma_{\text{DNAME}=\text{'Research'}} (\text{EMPLOYEE} \bowtie_{\text{DNO}=\text{DNUMBER}} \text{DEPARTMENT}))$$


## Maternità

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

## Paternità

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

## Persone

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



# SELEZIONE E PROIEZIONE

- Nome e reddito delle persone con gli anni minore o uguale a trenta:

$$\pi_{\langle \text{nome, reddito} \rangle}(\sigma_{\text{eta} \leq 30}(\text{Persone}))$$

```
SELECT nome, reddito  
FROM persone  
WHERE eta <= 30
```

- Abbreviazione di:

```
SELECT p.nome AS nome, p.reddito AS reddito  
FROM persone p  
WHERE p.eta <= 30
```



(animazione)

## Persone

Nome	Reddito
Andrea	21
Aldo	15
Filippo	30



# **SELEZIONE, PROIEZIONE E JOIN**

- Istruzioni SELECT con una sola relazione nella clausola FROM permettono di realizzare:
  - selezioni, proiezioni, ridenominazioni.
- Con più relazioni nella FROM si realizzano join (e prodotti cartesiani).



# SQL E ALGEBRA RELAZIONALE

- $R1(A1, A2) \ R2(A3, A4)$

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

- proiezione (**SELECT**)
- prodotto cartesiano (**FROM**)
- selezione (**WHERE**)



# SQL E ALGEBRA RELAZIONALE (2)

- Consideriamo gli schemi  $R1(A1, A2)$   $R2(A3, A4)$ :

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

$$\pi_{\langle A1, A4 \rangle} (\sigma_{A2=A3} (R_1 \times R_2))$$


# PROIEZIONE SENZA SELEZIONE

- Nome e reddito di tutte le persone :

$\pi_{<\text{nome, reddito}>}(\text{Persone})$

```
SELECT nome, reddito  
FROM persone
```

- Abbreviazione di:

```
SELECT p.nome AS nome, p.reddito AS reddito  
FROM persone p
```



# ABBREVIAZIONI

- Dato uno schema R(A,B); tutti gli attribuiti di R:

$$\pi_{A, B}(R)$$

```
SELECT *
FROM R
```

- equivale (intuitivamente) a:

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



# ESEMPIO

- I padri di persone che guadagnano più di 22:

$$\pi_{\text{Padre}} (\text{Paternita} \bowtie_{\text{Figlio}=\text{Nome}} \sigma_{\text{Reddito} > 22} (\text{Persone}))$$

```
SELECT      DISTINCT Padre  
FROM        Persone, Paternita  
WHERE       Figlio = Nome AND Reddito > 22
```

Padre
Luigi
Luigi

Padre
Luigi



# IL COMANDO SELECT: ESEMPIO

- Per ogni progetto localizzato a '*Strafford*', listare il n° di progetto, il n° di dipartimento di controllo, ed il cognome, l'indirizzo e la data di nascita del manager del dipartimento:

```
SELECT      PNUMBER, DNUM, LNAME, ADDRESS, BDATE  
FROM        PROJECT, DEPARTMENT, EMPLOYEE  
WHERE       DNUM=DNUMBER AND MGRSSN=SSN AND PLOCATION='Stafford'
```



# **RENAMING, DISTINCT E OPERAZIONI INSIEMISTICHE**



# NOMI DI ATTRIBUTI E RENAMING

- In SQL lo stesso nome può essere usato per più attributi solo se questi appartengono a relazioni diverse.
- Se una query coinvolge tali relazioni, occorre **qualificare** il nome dell'attributo con il nome della relazione per evitare ambiguità.
- **Esempio:** Employee.SSN



# NOMI DI ATTRIBUTI E RENAMING (2)

- Si può avere ambiguità anche nel caso di query che riferiscono due volte alla stessa relazione:
  - **Esempio:** Per ogni impiegato, trovare il suo nome il suo cognome e quello del suo diretto superiore:

```
SELECT E.FNAME, E.LNAME, S.FNAME, S.LNAME
FROM EMPLOYEE AS E, EMPLOYEE AS S
WHERE E.SUPERSSN=S.SSN;
```
- Abbiamo dichiarato nomi di relazione alternativi **E** ed **S**, detti **alias**, per la relazione EMPLOYEE .



# NOMI DI ATTRIBUTI E RENAMING (3)

- È anche possibile rinominare gli attributi della relazione nella query, dando loro degli alias scrivendo:  
**EMPLOYEE AS E(FN, MI, LN, SSN, BD, ADDR, SEX, SAL, SSSN, DNO)**
- nella clausola **FROM**.
- *Quella che abbiamo visto è un esempio di query ricorsiva.*



# MANCANZA DEL WHERE

- Omettere la clausola WHERE equivale a **WHERE TRUE**, cioè tutte le tuple della relazione specificata nella clausola FROM fanno parte del risultato.
- Se più di una relazione è specificata nella clausola **FROM**, allora il risultato sarà il *prodotto cartesiano* delle relazioni.



# MANCANZA DEL WHERE: ESEMPI

- *Esempio:* Selezionare tutti i SSN:

```
SELECT SSN  
FROM EMPLOYEE;
```

- *Esempio:* Selezionare tutte le combinazioni Employee.SSN e Department.Dname:

```
SELECT SSN, DNAME  
FROM EMPLOYEE, DEPARTMENT;
```



# IL CARATTERE JOLLY “\*”

- Per recuperare tutti gli attributi delle tuple selezionate, si usa il carattere jolly \*:

- **Esempio:** Trovare tutti i valori degli attributi degli impiegati che lavorano per il dipartimento n°5:

```
SELECT *  
FROM EMPLOYEE  
WHERE DNO=5;
```

- **Esempio:** Trovare tutti gli attributi di Employee e gli attributi di Department per cui lavora ogni impiegato del dipartimento ‘*Research*’:

```
SELECT *  
FROM EMPLOYEE, DEPARTMENT  
WHERE DNAME='Research' AND DNO=DNUMBER;
```



# DUPLICAZIONI DI TUPLE IN SQL

- SQL non tratta relazioni come insiemi: *tuple duplicate possono apparire più di una volta.*
- SQL non elimina le duplicazioni per le seguenti ragioni:
  - è un'operazione costosa (l'implementazione richiederebbe l'ordinamento e poi l'eliminazione);
  - l'utente può essere interessato alle duplicazioni;
  - con funzioni di aggregazione siamo interessati a non eliminarle.



# LA CLAUSOLA DISTINCT

- Se le duplicazioni non sono volute, lo si specifica con la clausola **DISTINCT**:

- **Esempi:**

- Trovare i salari di tutti gli impiegati:

```
SELECT SALARY  
FROM EMPLOYEE;
```

- Trovare i salari distinti degli impiegati:

```
SELECT DISTINCT SALARY  
FROM EMPLOYEE;
```



# **DISTINCT, ATTENZIONE**

- cognome e filiale di tutti gli impiegati

Cognome	Filiale
Neri	Napoli
Neri	Milano
Rossi	Roma

$\pi_{<\text{Cognome}, \text{Filiale}>} (\text{Impiegati})$



**SELECT**  
Cognome, Filiale  
**FROM** Impiegati

Cognome	Filiale
Neri	Napoli
Neri	Milano
Rossi	Roma
Rossi	Roma

**SELECT DISTINCT**  
Cognome, Filiale  
**FROM** Impiegati

Cognome	Filiale
Neri	Napoli
Neri	Milano
Rossi	Roma



# OPERAZIONI INSIEMISTICHE

- SQL incorpora le seguenti operazioni insiemistiche (è *richiesta la union-compatibilità*):
  - UNION
  - EXCEPT
  - INTERSECT
- **EXCEPT** restituisce tutti i valori distinti della query a sinistra dell'operando non presenti nella query a destra.
- Usando tali operazioni le tuple duplicate sono eliminate (a meno che non venga richiesto il contrario con la clausola **ALL**).

} SQL 2



# OPERAZIONI INSIEMISTICHE - ESEMPIO

- Fare una lista dei numeri di progetti per i progetti che coinvolgono un impiegato il cui cognome è ‘Smith’ come lavoratore **oppure** come manager del dipartimento che controlla il progetto:

```
(SELECT PNUMBER
      FROM PROJECT,WORKS_ON,EMPLOYEE
     WHERE PNUMBER=PNO AND ESSN=SSN AND
          LNAME='Smith');

UNION

(SELECT PNUMBER
      FROM PROJECT,DEPARTMENT,EMPLOYEE
     WHERE DNUM=DNUMBER AND MGRSSN=SSN AND
          LNAME='Smith')
```



# CONFRONTO TRA SOTTOSTRINCHE

- Per il confronto tra stringhe si usa l'operatore **LIKE**
- *Caratteri jolly:*
  - '%' rimpiazza qualsiasi numero di caratteri;
  - '\_' rimpiazza un singolo carattere;
- **Esempio:** Trovare tutti gli impiegati il cui indirizzo è a '*Houston, Texas*':

```
SELECT FNAME, LNAME  
FROM EMPLOYEE  
WHERE ADDRESS LIKE '%HOUSTON, TEXAS%';
```



# **“LIKE” - ESEMPIO**

- Le persone che hanno un nome che inizia per '*A*' e ha una '*d*' come terza lettera:

```
SELECT *
FROM persone
WHERE nome LIKE 'A_d%';
```



# CONFRONTO TRA SOTTOSTRINGHE - *ESEMPI*

- Trovare tutti gli impiegati nati negli anni '50. Il formato di data è *YYYY-MM-DD*:

```
SELECT FNAME, LNAME  
FROM EMPLOYEE  
WHERE BDATE LIKE '__ 5 _____';
```

- Mostrare i salari risultanti se a tutti gli impiegati che lavorano sul progetto '*Product X*' viene concesso un aumento del 10%:

```
SELECT FNAME, LNAME, 1.1*SALARY  
FROM EMPLOYEE, WORKS_ON, PROJECT  
WHERE ESSN=SSN AND PNO=PNUMBER AND PNAME='Product X';
```



# GESTIONE DEI VALORI NULLI

## Impiegati

Matricola	Cognome	Filiale	Età
5998	Neri	Milano	45
9553	Bruni	Milano	NULL

- Gli impiegati la cui età è NULL o potrebbe essere maggiore di 40.

$\sigma$  Età > 40 OR Età IS NULL (Impiegati)



# GESTIONE DEI VALORI NULLI (2)

- Gli impiegati la cui età è NULL o potrebbe essere maggiore di 40.

$\sigma$  Età > 40 OR Età IS NULL (Impiegati)

```
SELECT *
FROM Impiegati
WHERE eta > 40 OR eta IS NULL
```

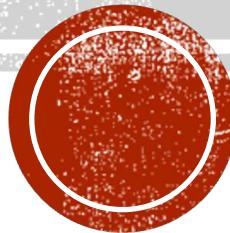




# FINE

Per eventuali domande: (in ordine di preferenza personale)

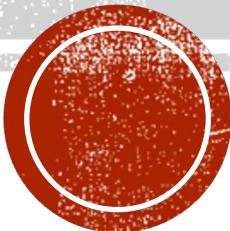
- Ora.
- Chat di Teams
- Mail: [silvio.barra@unina.it](mailto:silvio.barra@unina.it)



# BASI DI DATI I

- Tutte le operazioni di Join (prima parte)
- Prima parte
- Seconda parte

# **OPERAZIONI DI JOIN (1° PARTE)**



# JOIN

- La sequenza di operazioni appena vista, riassumibile in  $\sigma_{join-condition} (R \times S)$  è abbastanza frequente: per questo è stata creata un'operazione speciale, chiamata JOIN.
- La JOIN, denotata con  $\bowtie$ , è usata per combinare tuple relate in una sola tupla.
  - $\sigma_{join-condition} (R \times S)$  diventa  $R \bowtie_{join-condition} S$



# JOIN: ESEMPIO

- Supponiamo di voler trovare il nome del manager di ciascun dipartimento:
  - Combiniamo ogni tupla dipartimento con la tupla impiegato il cui SSN fa match col valore MGRSSN nella tupla dipartimento.
  - DEPT\_MGR= department  $\bowtie_{MGRSSN=SSN}$  EMPLOYEE
  - RESULT=  $\pi_{DNAME, LNAME, FNAME}(\text{DEPT\_MGR})$

DEPT_MGR	DNAME	DNUMBER	MGRSSN	• • •	FNAME	MINIT	LNAME	SSN	• • •
Research	5	333445555	• • •	Franklin	T	Wong	333445555	• • •	• • •
Administration	4	987654321	• • •	Jennifer	S	Wallace	987654321	• • •	• • •
Headquarters	1	888665555	• • •	James	E	Borg	888665555	• • •	• • •



# **JOIN: ESEMPIO SU PRODOTTO CARTESIANO**

- L'esempio precedente sul prodotto cartesiano può essere risolto rimpiazzando le operazioni:

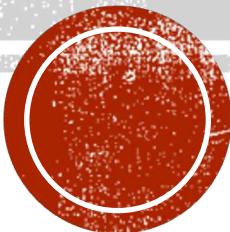
- $\text{EMP\_DEPENDENTS} = \text{EMPNAME} \times \text{DEPENDENTS}$
- $\text{ACTUAL\_DEPENDENTS} = \sigma_{\text{SSN}=\text{ESSN}}(\text{EMP\_DEPENDENTS})$

Con:

- $\text{ACTUAL\_DEPENDENTS} = \text{EMPNAME} \bowtie_{\text{SSN}=\text{ESSN}} \text{DEPENDENTS}$



# **OPERAZIONI DI JOIN (2° PARTE)**



# NOZIONI SULLA JOIN

- Sintassi :

Date le relazioni  $R(A_1, A_2, \dots, A_n)$  e  $S(B_1, B_2, \dots, B_m)$

Scriveremo  $R \bowtie_{\text{join\_condition}} S$

- Il risultato della join ha  $n+m$  attributi:

$Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$

- La condizione della join è della forma:

$\langle \text{condition} \rangle \text{ AND } \langle \text{condition} \rangle \text{ AND } \dots \text{ AND } \langle \text{condition} \rangle$   
dove ogni condizione è della forma  $A_i \theta B_j$ ,

con  $A_i \in R, B_j \in S, \text{dom}(A_i) = \text{dom}(B_j)$  e  $\theta \in \{=,,\neq,<,>,\leq,\geq\}$

- Tuple con attributi di join **null** non appaiono nel risultato.



# TIPI DI JOIN

- Una join con  $\theta$  generica è detta **THETA JOIN**
- Il join più comune ha come  $\theta$  l'operatore di uguaglianza ed è detto **EQUIJOIN**:
  - Nei risultati dell'EQUIJOIN abbiamo sempre una coppia di valori di attributi identici.
- Poiché tale ripetizione è superflua, esiste una nuova operazione, chiamata **NATURAL JOIN** e denotata da  $*$ , che elimina il secondo attributo superfluo.
  - È in sostanza una EQUIJOIN con la rimozione degli attributi con valori uguali superflui



## Impiegati

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

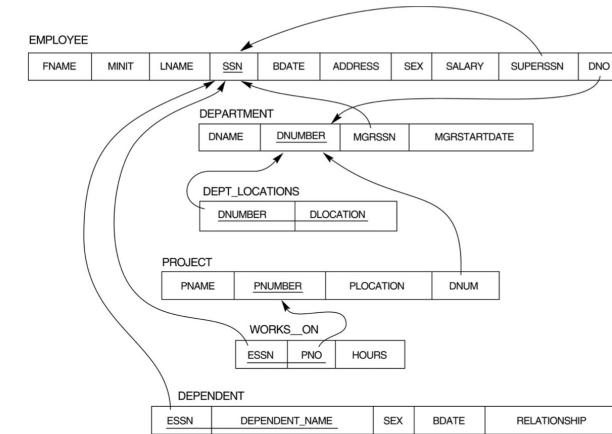
## Reparti

Codice	Capo
A	Mori
B	Bruni

Impiegati  $\triangleright\triangleleft_{\text{Reparto}=\text{Codice}}$  Reparti

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

# NATURAL JOIN

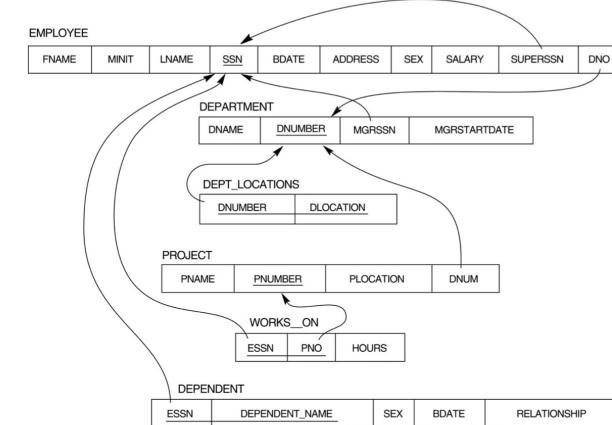


- La definizione standard di Natural join richiede che i due attributi di join (per ogni coppia) abbiano lo stesso nome. Se ciò non vale, occorre prima un'operazione di renaming.
  - $\text{DEPT} = \pi_{(\text{DNAME}, \text{DNUM}, \text{MGRSSN}, \text{MGRSTARTDATE})} (\text{DEPARTMENT})$
  - $\text{PROJ\_DEPT} = \text{PROJECT} * \text{DEPT}$ 
    - L'attributo DNUM è detto “**attributo di join**”

PROJ_DEPT	PNAME	PNUMBER	PLOCATION	DNUM	DNAME	MGRSSN	MGRSTARTDATE
	ProductX	1	Bellaire	5	Research	333445555	1988-05-22
	ProductY	2	Sugarland	5	Research	333445555	1988-05-22
	ProductZ	3	Houston	5	Research	333445555	1988-05-22
	Computerization	10	Stafford	4	Administration	987654321	1995-01-01
	Reorganization	20	Houston	1	Headquarters	888665555	1981-06-19
	Newbenefits	30	Stafford	4	Administration	987654321	1995-01-01



# NATURAL JOIN (2)



- Qualora gli attributi di join abbiano lo stesso nome, non è necessario il renaming.
- **Esempio:**
  - DEPT\_LOCS = DEPARTMENT \* DEPT\_LOCATIONS

DEPT_LOCS	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE	LOCATION
	Headquarters	1	888665555	1981-06-19	Houston
	Administration	4	987654321	1995-01-01	Stafford
	Research	5	333445555	1988-05-22	Bellaire
	Research	5	333445555	1988-05-22	Sugarland
	Research	5	333445555	1988-05-22	Houston

- In generale il Natural Join verifica l'uguaglianza di tutte le coppie di attributi.



# NATURAL JOIN (3)

Definizione più generale:

- $Q = R^*(\langle \text{list}_1 \rangle)(\langle \text{list}_2 \rangle)S$ 
  - $\langle \text{list}_1 \rangle$ =lista di attributi da R
  - $\langle \text{list}_2 \rangle$ =lista di attributi da S
- Le liste sono usate per le uguaglianze tra coppie di attributi corrispondenti;
  - le condizioni sono combinate in AND.
  - Solo la lista corrispondente ad attributi della prima relazione,  $\langle \text{list}_1 \rangle$  è mantenuta nel risultato.



# NATURAL JOIN (4)

- Se nessuna combinazione di tuple soddisfa la condizione di join, la relazione risultante avrà zero tuple:
  - Date le relazioni  $R$  (con  $n_r$  tuple) ed  $S$  (con  $n_s$  tuple),
    - $n_q Q = R \bowtie_{\text{joincond}} S$  avrà da 0 a  $n_r \cdot n_s$  tuple.
    - $n_q / (n_r \cdot n_s)$  è detta **join selectivity**.
- Se non esiste alcuna **<joincondition>**, tutte le tuple compariranno nella relazione risultante, ed il join diventa un prodotto cartesiano, detto **CROSS JOIN**.



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

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

- Ogni  $n$ -pla contribuisce al risultato:
  - join **completo**.



# UN JOIN NON COMPLETO

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

Impiegato	Reparto	Capo
Neri	B	Mori
Bianchi	B	Mori



# UN JOIN VUOTO

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

Impiegato   Reparto   Capo



# UN JOIN COMPLETO, CON N · M MAPPE

Impiegato	Reparto
Rossi	B
Neri	B

Reparto	Capo
B	Mori
B	Bruni

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



# CARDINALITÀ DEL JOIN

- Il join di  $R_1$  e  $R_2$  contiene un numero di  $n$ -ple compreso fra zero e il prodotto di  $|R_1|$  e  $|R_2|$ .
- Se il join coinvolge una chiave di  $R_2$ , allora il numero di  $n$ -ple è compreso fra zero e  $|R_1|$ .
- Se il join coinvolge una chiave di  $R_2$  e un vincolo di integrità referenziale, allora il numero di  $n$ -ple è pari a  $|R_1|$ .



# JOIN, UNA DIFFICOLTÀ

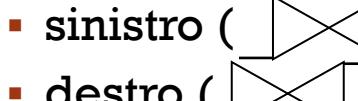
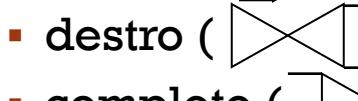
Impiegato	Reparto	Reparto	Capo
Rossi	A	B	Mori
Neri	B	C	Bruni
Bianchi	B		

Impiegato	Reparto	Capo
Neri	B	Mori
Bianchi	B	Mori

- Alcune  $n$ -ple non contribuiscono al risultato: vengono “tagliate fuori”.



# JOIN ESTERNO

- Il join **esterno** estende, con valori nulli, le  $n$ -ple che verrebbero tagliate fuori da un join (**interno**).
- Esiste in tre versioni:
  - sinistro (  **LEFT** ),
  - destro (  **RIGHT** ),
  - completo (  **FULL** ).



# JOIN ESTERNO (2)

- **Sinistro**: mantiene tutte le  $n$ -ple del *primo operando*, estendendole con valori nulli, se necessario.
- **Destro**: ... *del secondo operando* ...
- **Completo**: ... *di entrambi gli operandi* ...



## Impiegati

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

## Reparti

Reparto	Capo
B	Mori
C	Bruni

Impiegati  $\bowtie_{\text{LEFT}}$  Reparti

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



Impiegati		Reparti	
Impiegato	Reparto	Reparto	Capo
Rossi	A	B	Mori
Neri	B	C	Bruni
Bianchi	B		

Impiegati  $\bowtie_{\text{RIGHT}}$  Reparti

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



## Impiegati

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

## Reparti

Reparto	Capo
B	Mori
C	Bruni

Impiegati  $\bowtie_{\text{FULL}}$  Reparti

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



# L'OPERAZIONE DI DIVISIONE



# L'OPERAZIONE DI DIVISIONE

- L'operazione di divisione è indicata con  $\div$  è un tipo particolare di interrogazione che talvolta si presenta nelle applicazioni di basi di dati.
- Un esempio è “*trova i nomi degli impiegati che lavorano a tutti i progetti su cui lavora ‘John Smith’*”.
- La divisione  $\div$ , di  $r1$  per  $r2$ , con  $r1$  su  $R_1(X_1X_2)$  e  $r2$  su  $R_2(X_2)$ , è (il più grande) insieme di tuple con schema  $X_1$  tale che, facendo il prodotto cartesiano con  $r2$ , ciò che si ottiene è una relazione contenuta in  $r1$ .



# ESEMPIO

Voli

Codice	Data
AZ427	21/07/2001
AZ427	23/07/2001
AZ427	24/07/2001
TW056	21/07/2001
TW056	24/07/2001
TW056	25/07/2001

Linee

Codice
AZ427
TW056

Voli ÷ Linee

Data
21/07/2001
24/07/2001

(Voli ÷ Linee) ▷◁ Linee

Codice	Data
AZ427	21/07/2001
AZ427	24/07/2001
TW056	21/07/2001
TW056	24/07/2001

La divisione trova le date con voli per tutte le linee



# UN INSIEME COMPLETO DI OPERAZIONI

- Si può provare che  $\{\sigma, \pi, \cup, -, \times\}$  è un **insieme completo**, cioè tutte le altre operazioni possono essere espresse come combinazioni di queste.

- **Esempi:**

- Intersezione:

- $R \cap S = (R \cup S) - ((R - S) \cup (S - R))$

- Join:

- $R \bowtie_{<\text{condition}>} S = \sigma_{<\text{condition}>} (R \times S)$

- Divisione:

- $R \div S = T1 \leftarrow \pi_{<\text{attribute of } R - \text{attribute of } S>} (R)$

- $T2 \leftarrow \pi_{<\text{attribute of } R - \text{attribute of } S>} ((S \times T1) - R)$

- $T \leftarrow T1 - T2$



# ALGEBRA RELAZIONALE: LIMITI

- Ci sono però interrogazioni interessanti non esprimibili:
  - Calcolo di valori derivati: possiamo solo **estrarre** valori, non calcolarne di nuovi; calcoli di interesse:
    - a livello di  $n$ -pla o di singolo valore (conversioni somme, differenze, etc.);
    - su insiemi di  $n$ -ple (somme, medie, etc.).
  - Interrogazioni inerentemente **ricorsive**, come la **chiusura transitiva**.



# FUNZIONI AGGREGATE E DI RAGGRUPPAMENTO

- *Non sono presenti nell'algebra relazionale base.*
- Operano su un insieme di dati per restituire come risultato una relazione con un solo valore.
- Sono funzioni applicate a collezioni di valori numerici:
  - SUM, AVERAGE, MAXIMUM, MINIMUM, COUNT.
- *Notazione:*

$\langle \text{attributi di raggruppamento} \rangle \mathcal{F}_{\langle \text{lista funzioni} \rangle}(R)$

- dove  $\langle \text{attributi di raggruppamento} \rangle$  è una lista di attributi della relazione R e raggruppa le tuple presenti nella relazione sulla base dei loro valori,
- e  $\langle \text{lista funzioni} \rangle$  è una lista di coppie  $\langle \text{funzione} \rangle \langle \text{attributo} \rangle$ :
  - $\langle \text{funzione} \rangle$  è una delle funzioni SUM,...
  - $\langle \text{attributo} \rangle$  è un attributo della relazione R.



# ESEMPIO

- $T(N\_D, N\_IMP, STIP\_MEDIO) \leftarrow \text{COUNT}_{N\_D} \text{COUNT}_{N\_IMP} \text{AVERAGE}_{STIP\_MEDIO}(\text{IMPIEGATO})$
- $R \leftarrow \sigma_{N\_D, N\_IMP, STIP\_MEDIO}(T)$

R	N_D	N_IMP	STIP_MEDIO
5	4	33250	
4	3	31000	
1	1	55000	

- $S \leftarrow \text{COUNT}_{S} \text{COUNT}_{COUNT\_SSN} \text{AVERAGE}_{STIPENDIO}$

S	COUNT_SSN	AVERAGE_STIPENDIO
8		35125



# EQUIVALENZA DI ESPRESSIONI

- Due espressioni sono **equivalenti** se producono lo stesso risultato qualunque sia l'istanza attuale della base di dati.
- L'equivalenza è importante perché i DBMS cercano di eseguire espressioni equivalenti a quelle date, ma meno “costose”.
- Trasformazioni di equivalenza.

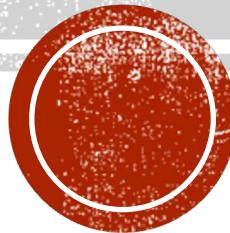




# FINE

Per eventuali domande: (in ordine di preferenza personale)

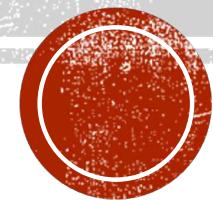
- Ora.
- Chat di Teams
- Mail: [silvio.barra@unina.it](mailto:silvio.barra@unina.it)



# BASI DI DATI I

- Query complesse in SQL
- Aggiornamenti in SQL

# **QUERY COMPLESSE IN SQL**



# QUERY ANNIDATE E CONFRONTO DI INSIEMI

- La query usata per mostrare la **UNION** può essere così riformulata:

```
SELECT DISTINCT PNUMBER
FROM PROJECT
WHERE PNUMBER IN
  (SELECT PNUMBER
   FROM PROJECT, DEPARTMENT, EMPLOYEE
   WHERE DNUM=DNUMBER AND MGRSSN=SSN AND LNAME= 'Smith')
```

**OR**

```
PNUMBER IN
  (SELECT PNO
   FROM WORKS_ON, EMPLOYEE
   WHERE ESSN=SSN AND LNAME='Smith');
```



## QUERIES ANNIDATE E CONFRONTO DI INSIEMI (2)

- La prima query seleziona il numero dei progetti che hanno uno '*Smith*' come manager, mentre la seconda seleziona il numero di progetto dei progetti che hanno uno '*Smith*' come lavoratore.
- Nella query esterna selezioniamo una tupla PROJECT se il valore PNUMBER compare nel risultato di una delle query annidate.



# L'OPERATORE IN

- L'operatore **IN** confronta un valore con un insieme di tuple union-compatibili.
- L'operatore **IN** permette di specificare valori multipli nella clausola WHERE.

- **Esempio:**

```
SELECT DISTINCT ESSN
  FROM WORKS_ON
 WHERE (PNO, HOURS) IN
       (SELECT PNO, HOURS
        FROM WORKS_ON
       WHERE ESSN='123456789');
```



# ALTRI OPERATORI

## ▪ **ANY (o SOME)**

- confronta un singolo valore (attributo) **v** con un multiset **V**, restituendo TRUE se **v** è uguale a qualche valore in **V**.
- **ANY e SOME** sono equivalenti. Possono essere combinati con { $>$ ,  $\geq$ ,  $<$ ,  $\leq$ ,  $<>$ }.
- $= ANY$  è equivalente ad usare l'operatore **IN**.

## ▪ Anche **ALL** può essere combinato con questi operatori.

- $v > ALL V$  è TRUE se il valore **v** è maggiore di tutti i valori in **V**.



# **OPERATORE ALL - ESEMPIO**

- Trovare tutti i nomi degli impiegati il cui salario è maggiore del salario di tutti gli impiegati del dipartimento 5:

```
SELECT LNAME, ENAME  
      FROM EMPLOYEE  
 WHERE SALARY > ALL (SELECT SALARY  
                         FROM EMPLOYEE  
                        WHERE DNO=5);
```



# **OPERATORE ANY - ESEMPIO**

```
SELECT LNAME, ENAME  
FROM EMPLOYEE  
WHERE SALARY > ANY (SELECT SALARY  
                 FROM EMPLOYEE);
```

- Trovare tutti i nomi degli impiegati tranne quelli il con il salario minimo.



# CASI DI AMBIGUITÀ

- **Ambiguità nei nomi di attributi:**
  - Si ha, se esistono attributi con lo stesso nome, uno in una relazione nella clausola FROM della query esterna e l'altro in una relazione della clausola FROM della query interna.
- **Regola:** un riferimento a un attributo non qualificato riferisce alla relazione dichiarata nella query annidata più interna.



# CASI DI AMBIGUITÀ - ESEMPIO

- Trovare il nome di ogni impiegato che ha una persona a carico con lo stesso nome e lo stesso sesso dell'impiegato:

```
SELECT E.ENAME, E.LNAME  
FROM EMPLOYEE AS E  
WHERE E.SSN IN (SELECT ESSN  
         FROM DEPENDENT  
         WHERE ESSN=E.SSN  
         AND DEPENDENT_NAME = E.FNAME  
         AND SEX=E.SEX);
```



È necessario qualificarlo altrimenti farebbe riferimento alla relazione DEPENDENT.



# CASI DI AMBIGUITÀ (2)

- In generale, una query scritta con blocchi annidati **SELECT...FROM...WHERE** e con operatori di confronto **=** o **IN** può essere sempre espressa come un singolo blocco.
- La precedente query può anche essere scritta come:

```
SELECT E.FNAME, E.LNAME
FROM EMPLOYEE AS E, DEPENDENT AS D
WHERE E.SSN=D.SSN AND E.SEX=D.SEX AND
E.FNAME=D.DEPENDENT_NAME
```



# L'OPERATORE **CONTAINS**

- L'implementazione originale SQL su **systemR** prevedeva un operatore **CONTAINS** per confrontare due insiemi.
- È stato poi eliminato per motivi di efficienza.



# L'OPERATORE CONTAINS - ESEMPIO

- Ritrovare il nome e cognome di ciascun impiegato che lavora su tutti i progetti controllati dal dipartimento 5:

```
SELECT FNAME, LNAME  
FROM EMPLOYEE AS E  
WHERE (( SELECT PNO  
          FROM WORKS_ON  
          WHERE E.SSN=ESSN)  
        CONTAINS  
        ( SELECT PNUMBER  
          FROM PROJECT  
          WHERE DNUM=5));
```



# **EXIST E NOT EXIST**

- **EXISTS e NOT EXISTS:**

- Per verificare se il risultato di una query annidata correlata è vuota.

- **Esempio:** Ritrovare il nome degli impiegati che non hanno persone a carico:

```
SELECT FNAME, LNAME  
FROM EMPLOYEE  
WHERE NOT EXISTS ( SELECT *  
    FROM DEPENDENT  
    WHERE SSN=ESSN);
```



# INSIEMI ESPLICITI

- Trovare il SSN di tutti gli impiegati che lavorano sui progetti 1, 2 o 3:

```
SELECT DISTINCT ESSN  
FROM WORKS_ON  
WHERE PNO IN (1, 2, 3);
```

- Si può anche testare se un valore è **NULL**:

- = e ≠ sono scritti come 'IS' e 'IS NOT' per confronti con NULL.
- **Esempio:** Trovare il nome di tutti gli impiegati che non hanno supervisori:

```
SELECT FNAME, LNAME  
FROM EMPLOYEE  
WHERE SUPERSSN IS NULL;
```



# LA KEYWORD AS

- È possibile rinominare qualsiasi attributo che compare in una query con la keyword **AS**:

```
SELECT E.LNAME AS EMPLOYEE_NAME,
      S.LNAME AS SUPERVISOR_NAME
FROM EMPLOYEE AS E, EMPLOYEE AS S
WHERE E.SUPERSSN=S.SSN;
```



# TABELLE JOINED

- Il concetto di tabella joined compare con SQL2 per specificare un'operazione di **JOIN** nella clausola FROM.
- Possono essere usati i seguenti tipi di join:
  - **NATURAL JOIN**
  - **INNER JOIN**
  - **LEFT OUTER JOIN / RIGHT OUTER JOIN**
  - **FULL OUTER JOIN**



# TABELLE JOINED - ESEMPIO

- Trovare il nome e l'indirizzo di ogni impiegato che lavora per il Dipartimento '*Research*':

```
SELECT FNAME, LNAME, ADDRESS  
FROM (EMPLOYEE JOIN DEPARTMENT ON  
DNO=DNUMBER)  
WHERE DNAME='Research';
```



# AGGREGAZIONE E RAGGRUPPAMENTO

- Le funzioni di aggregazione e di raggruppamento sono diffusissime nella gestione di basi di dati. SQL incorpora le seguenti funzioni:
  - **COUNT**: conteggio tuple.
  - **COUNT(DISTINCT ...)**: conteggio di tuple distinte.
  - **SUM**: somma dei valori di un attributo in una tabella.
  - **MAX**: valore massimo tra gli attributi di una tabella.
  - **MIN**: valore minimo tra gli attributi di una tabella.
  - **AVG**: valore medio tra gli attributi di una tabella.
  - **STD**: deviazione standard tra gli attributi di una tabella.



# AGGREGAZIONE E RAGGRUPPAMENTO

- **Esempio:** Trovare la somma dei salari di tutti gli impiegati, il massimo, il minimo e la media dei salari:

```
SELECT SUM(SALARY), MAX(SALARY),  
       MIN(SALARY), AVG(SALARY)  
FROM EMPLOYEE;
```



# COUNT - ESEMPI

- Conta il numero di impiegati:

```
SELECT COUNT(*)  
FROM EMPLOYEE;
```

- Restituisce il numero di tuple nel risultato della query (\*):

```
SELECT COUNT(*) AS Conteggio  
FROM EMPLOYEE, DEPARTMENT  
WHERE DNO=DNUMBER AND DNAME='Research';
```



# COUNT - ESEMPI (2)

- Conta il numero di valori di stipendi distinti:

```
SELECT COUNT (DISTINCT SALARY)  
FROM EMPLOYEE;
```

- Elencare il nome ed il cognome degli impiegati che hanno due o più persone a carico:

```
SELECT LNAME, FNAME  
FROM EMPLOYEE  
WHERE ( SELECT COUNT(*)  
        FROM DEPENDENT  
        WHERE SSN=ESSN)>=2;
```



# ORDINAMENTO DI TUPLE

- Per ordinare le tuple nel risultato della query si usa la clausola **ORDER BY**.
- **Esempio:** Ritrovare una lista di impiegati e dei progetti su cui lavorano, ordinati per dipartimento, e nell'ambito di ciascun dipartimento, alfabeticamente per cognome e nome:

```
SELECT DNAME, FNAME, LNAME, PNAME  
FROM DEPARTMENT, EMPLOYEE, WORKS_ON, PROJECT  
WHERE DNUMBER=DNO AND SSN=ESSN AND  
PNO=PNUMBER  
ORDER BY DNAME, LNAME, FNAME;
```



# ORDINAMENTO DI TUPLE (2)

- L'ordine di default è crescente:
  - **ASC** per crescente.
  - **DESC** decrescente.
- **Esempio:** per avere un ordine decrescente di dipartimento e crescente per nome e cognome:

...

**ORDER BY DNAME DESC, LNAME ASC, FNAME ASC;**



# GROUP BY

- Raggruppiamo le tuple che hanno lo stesso valore per alcuni attributi.
- **Esempio:**  

```
SELECT DNO, COUNT(*), AVG(SALARY)
  FROM EMPLOYEE
 GROUP BY DNO;
```
- Le tuple sono divise in gruppi, ogni gruppo ha lo stesso valore per DNO.
- Le funzioni COUNT e AVG sono applicate ad ogni gruppo di queste tuple.



# GROUP BY (2)

- Risultato:

DNO	COUNT(*)	AVG(SALARY)
1	4	23000
4	3	25000
3	4	22000



# GROUP BY (3)

- **Esempio:** Per ogni progetto, visualizzare il numero del progetto, il nome del progetto ed il numero di impiegati che lavorano su quel progetto:

```
SELECT pnumber, pname, COUNT(*)  
FROM project, works_on  
WHERE pnumber = pno  
GROUP BY pnumber;
```



# GROUP BY (4)

- **Esempio:** Per ogni progetto visualizzare il numero del progetto, il nome del progetto ed il numero di impiegati del dipartimento n.5 che lavorano su quel progetto:

```
SELECT pnumber, pname, COUNT(*)  
FROM project, works_on, employee  
WHERE pnumber = pno AND ssn = essn AND dno = 5  
GROUP BY pnumber, pname;
```



# GROUP BY (5) – USO DI HAVING

- **Esempio:** Per ogni progetto su cui lavorano più di due impiegati, visualizzare il numero del progetto, il nome del progetto ed il numero di impiegati che lavorano su quel progetto:

```
SELECT pnumber, pname, COUNT(*)  
FROM project, works_on  
WHERE pnumber = pno  
GROUP BY pnumber, pname  
HAVING COUNT(*) > 2;
```



# GROUP BY (6) – USO DI HAVING (2)

- **Esempio:** Determinare, per ogni dipartimento che ha più di 6 impiegati, il numero totale degli impiegati il cui stipendio è maggiore di \$40.000:

```
SELECT dname, COUNT(*)  
FROM department, employee  
WHERE dnumber = dno AND salary > 40000  
GROUP BY dname  
HAVING COUNT(*) > 6;
```



# RIEPILOGO DELLE INTERROGAZIONI

**SELECT** <elenco attributi e funzioni>  
**FROM** <elenco delle tavelle>  
[**WHERE** <condizioni>]  
[**GROUP BY** <attributo o attributi di raggruppamento>]  
[**HAVING** <condizione di raggruppamento>]  
[**ORDER BY** <elenco attributi>]



# AGGIORNAMENTI IN SQL



# **AGGIORNAMENTI IN SQL**

- In SQL sono previsti tre comandi per modificare il database:
  - **INSERT**
  - **DELETE**
  - **UPDATE**



# IL COMANDO INSERT

- Il comando **INSERT INTO** inserisce nuove righe in una relazione.

- Sintassi:

```
INSERT INTO Target [(FieldName,...)]  
VALUES (Value1,...);
```

oppure

```
INSERT INTO Target [(FieldName,...)]  
SELECT FieldName,...  
FROM TableExpression;
```



# IL COMANDO INSERT - ESEMPIO

- Aggiungere una nuova tupla alla relazione '*Employee*':

```
INSERT INTO EMPLOYEE  
VALUES ('Richard', 'K', 'Marini', '654765876',  
'30-DEC-52', '98 Oak Forest, Katy, TX', 'M', 37000, '987654321', 4);
```



# IL COMANDO INSERT (2)

- È possibile non assegnare valori a tutti gli attributi.
  - In tal caso, questi avranno il valore di **default** o **NULL**.
- 
- **Esempio:**
    - `INSERT INTO EMPLOYEE (FNAME, LNAME, SSN)  
VALUES ('Richard', 'Marini', '654765876');`



# IL COMANDO INSERT - ESEMPIO

- Creare una tabella temporanea che ha nome, numero di impiegati e salari totali per ciascun dipartimento:

```
CREATE TABLE DEPTS_INFO ( DEPT_NAME VARCHAR(15),  
                           NO_OF_EMPS INTEGER,  
                           TOTAL_SAL INTEGER);
```

```
INSERT INTO DEPTS_INFO (DEPT_NAME, NO_OF_EMPS, TOTAL_SAL)  
SELECT DNAME, COUNT(*), SUM(SALARY)  
FROM DEPARTMENT, EMPLOYEE  
WHERE DNUMBER=DNO  
GROUP BY DNAME;
```

- *Eventuali aggiornamenti successivi non influenzano la tabella originale. Per aggiornarle, è invece necessario definire una view.*



# USO DI *AUTO\_INCREMENT*

- L' attributo **AUTO\_INCREMENT** può essere usato per generare un identificatore unico per le nuove righe:

```
CREATE TABLE animals (
    id INT NOT NULL AUTO_INCREMENT,
    name CHAR(30) NOT NULL,
    PRIMARY KEY (id)
) AUTO_INCREMENT=5;
```

```
INSERT INTO animals (name) VALUES
('dog'),('cat'),('penguin'),('wolf'),('whale'),('ostrich');
```

```
SELECT * FROM animals;
```

<b>id</b>	<b>name</b>
5	dog
6	cat
7	penguin
8	wolf
9	whale
10	ostrick



# IL COMANDO DELETE

- Il comando **DELETE** rimuove una o più tuple da una relazione.
- Sintassi:

```
DELETE
FROM TableName
WHERE Criteria;
```



# IL COMANDO DELETE - *ESEMPI*

```
DELETE FROM EMPLOYEE  
WHERE LNAME='Brown';
```

```
DELETE FROM EMPLOYEE  
WHERE DNO IN (SELECT DNUMBER  
         FROM DEPARTMENT  
         WHERE DNAME='Research');
```



# IL COMANDO UPDATE

- Il comando **UPDATE** permette di modificare valori in una relazione:

```
UPDATE PROJECT  
SET PLOCATION='Bellaire', DNUM=5  
WHERE PNUMBER=10;
```

```
UPDATE EMPLOYEE  
SET SALARY=SALARY * 1.1  
WHERE DNO IN (SELECT DNUMBER  
FROM DEPARTMENT  
WHERE DNAME='Research');
```



# VISTE IN SQL

- Le viste sono tabelle ‘virtuali’ derivate da tabelle esistenti nel db.
- Possono essere definite per nascondere dei dati da alcune tabelle (es. per questioni di privacy), per combinare più tabelle, per creare report, etc.
- Sintassi:  
**CREATE VIEW** *ViewName*  
**AS** *SelectStatement*;



# VISTE IN SQL - ESEMPIO

```
CREATE VIEW WORKS_ON1
AS SELECT FNAME, LNAME, PNAME, HOURS
FROM EMPLOYEE, PROJECT, WORKS_ON
WHERE SSN=ESSN AND PNO=PNUMBER;
```

**WORKS\_ON1**

FNAME	LNAME	PNAME	HOURS
-------	-------	-------	-------



# VISTE IN SQL - ESEMPIO (2)

```
CREATE VIEW DEPTS_INFO (DEPT_NAME,  
    NO_OF_EMPS, TOTAL_SAL)  
    SELECT DNAME, COUNT(*), SUM(SALARY)  
    FROM DEPARTMENT, EMPLOYEE  
    WHERE DNUMBER=DNO  
    GROUP BY DNAME;
```

**DEPT\_INFO**

DEPT_NAME	NO_OF_EMPS	TOTAL_SAL
-----------	------------	-----------





**FINE**

Per eventuali domande: (in ordine di preferenza personale)

- Ora.
- Chat di Teams
- Mail: [silvio.barra@unina.it](mailto:silvio.barra@unina.it)



# BASI DI DATI I

- RECAP su SQL
- Assertion



# STRUTTURA DI UN'INTERROGAZIONE

**SELECT** <elenco attributi e funzioni>

**FROM** <elenco delle tabelle o viste>

[**WHERE** <condizioni> ]

[**GROUP BY** <attributo o attributi di raggruppamento>]

[**HAVING** <condizione di raggruppamento>]

[**ORDER BY** <elenco attributi>]



# NELLA CLAUSOLA WHERE

- WHERE attributo op.rel. Valore
- WHERE attributo op.rel. Attributo
- WHERE attributo IS [NOT] NULL
- WHERE attributo IN (vall, val2, ...)
- WHERE attributo BETWEEN vall AND val2
- WHERE attributo LIKE ‘pattern’
- WHERE salario < 30000 (<>, =, <, <=, >, >=)
- WHERE M.salario > E.salario
- WHERE salario IS NOT NULL
- WHERE sesso IN ('M', 'F')
- WHERE salario BETWEEN 300 AND 400
- WHERE nome LIKE '\_Mario%



# DOVE ANNIDARE LE QUERY

**SELECT** <elenco attributi e funzioni> NO

**FROM** <elenco delle tavelle o viste> SI

[**WHERE** <condizioni> ] SI

[**GROUP BY** <attributo o attributi di raggruppamento>] NO

[**HAVING** <condizione di raggruppamento>] SI

[**ORDER BY** <elenco attributi>] NO



# RAGIONARE SULLA QUERY

- Le query possono essere di diverse complessità
- In tutti i casi, la richiesta è chiara e ci è molto semplice andare ad identificare ciò di cui abbiamo bisogno all'interno dello schema logico
- Risulta però complicato andare a ragionare su COME costruire la query che restituiscia effettivamente l'output richiesto
- È SEMPRE importante andare a ragionare bene sulla richiesta, ed eventualmente trovare un altro modo per riscriverla, in modo che risulti più semplice andare a trasformarla in una query.



# ESEMPIO

- Schema logico

STUDENTE (MATRICOLA, NOME COGNOME)  
ESAME(MATRICOLA, CORSO)

- Query

Gli studenti che hanno sostenuto almeno tutti gli esami sostenuti dallo studente con matricola 100

PROVIAMO A RISCRIVERLA?



# ESEMPIO

- Schema logico

STUDENTE (MATRICOLA, NOME COGNOME)  
ESAME(MATRICOLA, CORSO)

- Query

**Gli studenti che hanno sostenuto almeno tutti gli esami sostenuti dallo studente con matricola 100**

**Trovo tutti gli studenti per cui NON ESISTA un esame che hanno sostenuto e che lo studente 100 non ha sostenuto**

**L'insieme degli esami sostenuti dallo studente 100 e non sostenuti dallo studente x è vuoto**



# ESEMPIO

```
SELECT *  
FROM Studenti AS S  
WHERE NOT EXISTS
```

```
(SELECT E.CORSO  
FROM Esami AS E
```

```
WHERE E.Matricola = '100' AND E.CORSO NOT IN
```

```
(SELECT E2.CORSO  
FROM Esami AS E2  
WHERE E2.Matricola = S.Matricola))
```



# ESEMPIO

```
SELECT *
FROM Studenti AS S
WHERE NOT EXISTS
    (SELECT E.CORSO
     FROM Esami AS E
     WHERE E.Matricola = '100' AND E.CORSO NOT IN
          (SELECT E2.CORSO
           FROM Esami AS E2
           WHERE E2.Matricola = S.Matricola))
```

Insieme dei corsi sostenuti  
dallo studente S.Matricola



# ESEMPIO

Insieme dei corsi sostenuti da 100, ma non sostenuti dallo studente S.Matricola

```
SELECT *  
FROM Studenti AS S  
WHERE NOT EXISTS
```

```
(SELECT E.CORSO  
FROM Esami AS E  
WHERE E.Matricola = '100' AND E.CORSO NOT IN
```

```
(SELECT E2.CORSO  
FROM Esami AS E2  
WHERE E2.Matricola = S.Matricola))
```



# ESEMPIO

```
SELECT *
FROM Studenti AS S
WHERE NOT EXISTS
    (SELECT E.CORSO
     FROM Esami AS E
     WHERE E.Matricola = '100' AND E.CORSO NOT IN
```

Insieme degli studenti S per cui l'insieme degli esami sostenuti da 100, ma non sostenuti da S, è vuoto.

```
(SELECT E2.CORSO
     FROM Esami AS E2
     WHERE E2.Matricola = S.Matricola))
```



# ATTENZIONE ALLA VISIBILITÀ DEI NOMI

```
SELECT *  
FROM Studenti AS S  
WHERE NOT EXISTS
```

```
(SELECT E.CORSO  
FROM Esami AS E
```

```
WHERE E.Matricola = '100' AND E.CORSO NOT IN
```

```
(SELECT E2.CORSO  
FROM Esami AS E2  
WHERE E2.Matricola = S.Matricola))
```



# ASSERTION

- Vincoli che conosciamo
  - Vincoli di dominio: i valori dell'attributo devono appartenere al relativo dominio
  - Vincoli di integrità referenziale: FOREIGN KEY
  - Vincoli intrarelazionale: NOT NULL, UNIQUE, PRIMARY KEY
- Altri vincoli
  - Vincoli interrelazionali: vincoli che coinvolgono attributi di più tabelle
  - Nello standard di SQL viene messo a disposizione un costrutto che si chiama ASSERTION
  - La Assertion è un vincolo che la base di dati deve rispettare

CREATE ASSERTION <nome vincolo> espressione booleana



# ASSERTION

*CREATE ASSERTION <nome\_vincolo> espressione booleana*

[NOT] EXIST (SELECT

<Cerco la violazione del vincolo>

)



# ESEMPIO

*ALBERO(CodAlbero, CodRadice)*

*NODE(CodNodo, Label, CodAlbero)*

*ARCO(CodArco, NodoSorgente, NodoTarget, Label, CodAlbero)*

## Vincoli

- *Sorgente e target di un albero sono diversi*



# ESEMPIO

*ALBERO(CodAlbero, CodRadice)*

*NODE(CodNodo, Label, CodAlbero)*

*ARCO(CodArco, NodoSorgente, NodoTarget, Label, CodAlbero)*

## Vincoli

- *Sorgente e target di un albero sono diversi (vincolo intrarelazionale, in particolare è un vincolo di ennupla) → non ho bisogno di un vincolo di asserzione, ma posso inserire un vincolo nella tabella.*



# ESEMPIO

*ALBERO(CodAlbero, CodRadice)*

*NODE(CodNodo, Label, CodAlbero)*

*ARCO(CodArco, NodoSorgente, NodoTarget, Label, CodAlbero)*

## Vincoli

1) *Sorgente e target di un albero sono diversi (vincolo intrarelazionale, in particolare è un vincolo di ennupla) → non ho bisogno di un vincolo di asserzione, ma posso inserire un vincolo nella tabella.*

*ALTER TABLE ARCO*

*ADD CONSTRAINT C1*

*CHECK NodoSorgente <> Nodo Target*



# ESEMPIO

*ALBERO(CodAlbero, CodRadice)*

*NODE(CodNodo, Label, CodAlbero)*

*ARCO(CodArco, NodoSorgente, NodoTarget, Label, CodAlbero)*

## Vincoli

2) *L'identificativo della radice è un nodo dell'albero*



# ESEMPIO

*ALBERO(CodAlbero, CodRadice)*

*NODE(CodNodo, Label, CodAlbero)*

*ARCO(CodArco, NodoSorgente, NodoTarget, Label, CodAlbero)*

## Vincoli

2) L'identificativo della radice è un nodo dell'albero (interrelazionale)

*CREATE ASSERTION A1*

*CHECK NOT EXIST (SELECT \* FROM ALBERO AS T*

*WHERE T.CodRadice NOT IN*

*(SELECT N.CodNodo FROM NODO N*

*WHERE N.CodNodo= T.CodRadice )*



# ALTRI VINCOLI INTERRELAZIONALI

- Un nodo ha un solo padre
- Una radice non ha padri
- Tutti i nodi sono connessi

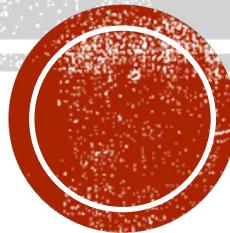




# FINE

Per eventuali domande: (in ordine di preferenza personale)

- Ora.
- Chat di Teams
- Mail: [silvio.barra@unina.it](mailto:silvio.barra@unina.it)

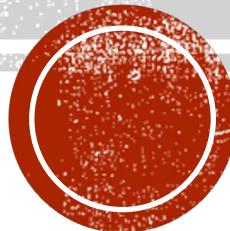


# BASI DI DATI I

- DML and Trigger



# DML & TRIGGER



# DML: DATA MANIPULATION LANGUAGE

- In SQL sono previsti tre comandi per modificare il database:
  - **INSERT**: popolamento tabelle
  - **DELETE**: rimozione di righe
  - **UPDATE**: cambiamento di valori nelle righe delle tabelle



# **DELETE**

- Il comando **DELETE** rimuove una o più tuple da una relazione.

**DELETE**

**FROM** <*NOME TABELLA*>  
**WHERE** <*CONDIZIONE*>;

- Cancella da <*NOME TABELLA*> tutte le righe che soddisfano <*CONDIZIONE*>
- Se la clausula **WHERE** è omessa allora è come se ci fosse la clausula

**WHERE TRUE**

e vengono cancellate tutte le righe della tabella

- **NB:** Il comando DELETE non rimuove la tabella
  - Il comando DROP <Nome Tabella> lo fa



# IL COMANDO DELETE - *ESEMPI*

```
DELETE FROM EMPLOYEE  
WHERE LNAME='Brown';
```

```
DELETE FROM EMPLOYEE  
WHERE DNO IN (SELECT DNUMBER  
                 FROM DEPARTMENT  
                 WHERE DNAME='Research');
```



# IL COMANDO INSERT

- Il comando **INSERT INTO** inserisce nuove righe in una relazione.

**INSERT INTO** <Nome Tabella> [ElencoAttributi]  
**VALUES** (*ListaValori*);

oppure

**INSERT INTO** <NomeTabella> [ElencoAttributi]  
**SELECT** *ElencoAttributi*  
**FROM** *Espressione*;



# IL COMANDO INSERT (2)

**INSERT INTO** <Nome Tabella> [ElencoAttributi]  
**VALUES** (*ListaValori*);

1. Se l'elenco degli attributi viene omesso, allora si intendono riempire i campi degli attributi in ordine di definizione della tabella
2. Se specifico l'elenco degli attributi, allora l'elenco deve includere tutti gli attributi TOTALI (NOT NULL). Decidiamo noi l'ordine.
3. Gli attributi PARZIALI non presenti, assumeranno il valore NULL o il valore di DEFAULT, se questo è specificato

---

**INSERT INTO** EMPLOYEE (FNAME, LNAME, SSN)  
**VALUES** ('Richard', 'Marini', '654765876');

---

**INSERT INTO** EMPLOYEE  
**VALUES** ('Richard', 'K', 'Marini', '654765876', '30-DEC-52', '98 Oak Forest, Katy, TX',  
'M', 37000, '987654321', 4);



# IL COMANDO INSERT - ESEMPIO

- Creare una tabella temporanea che ha nome, numero di impiegati e salari totali per ciascun dipartimento:

```
CREATE TABLE DEPTS_INFO ( DEPT_NAME VARCHAR(15),  
                           NO_OF_EMPS INTEGER,  
                           TOTAL_SAL INTEGER);
```

```
INSERT INTO DEPTS_INFO (DEPT_NAME, NO_OF_EMPS, TOTAL_SAL)  
SELECT DNAME, COUNT(*), SUM(SALARY)  
FROM DEPARTMENT, EMPLOYEE  
WHERE DNUMBER=DNO  
GROUP BY DNAME;
```

- *Eventuali aggiornamenti successivi non influenzano la tabella originale. Per aggiornarla, è invece necessario definire una view.*



# IL COMANDO UPDATE

- Il comando **UPDATE** permette di modificare valori in una relazione:

**UPDATE** <NomeTabella> [alias]

**SET** <NomeAttributo1>= <Espressione> [, <NomeAttributoN>=<EspressioneN>]

[**WHERE** <condizione>]

- Si lavora su NomeTabella
- Si modificano le righe che soddisfano <condizione>
- Per ogni riga si modificano gli attributi della clausola SET con il valore calcolato dall'espressione



# **IL COMANDO UPDATE - ESEMPIO**

**UPDATE PROJECT**

**SET PLOCATION='Bellaire', DNUM=5  
WHERE PNUMBER=10;**

**UPDATE EMPLOYEE**

**SET SALARY=SALARY \* 1.1  
WHERE DNO IN (SELECT DNUMBER  
FROM DEPARTMENT  
WHERE DNAME='Research');**



# ESEMPIO

*Magazzino (CodArticolo, Descrizione, Quantità)*

*Ordine (CodOrdine, Data, PIVA, DataInvio)*

*CompOrdine(CodOrdine\*, CodArticolo\*, Quantità, Prezzo)*

*Carrello(CodCarrello, Data, PIVA, Completo)*

*CompCarrello(CodCarrello\*, CodArticolo\*, Quantità, Prezzo)*

- Operazioni che devono essere fatte

Quando il carrello è completo

a) Trasformo il carrello in ordine

1. Creare un nuovo ordine con la struttura del carrello
2. Rimuovere il carrello

b) Aggiornare le scorte nel magazzino



# ESEMPIO

*Magazzino (CodArticolo, Descrizione, Quantità)*

*Ordine (CodOrdine, Data, PIVA, DataInvio)*

*CompOrdine(CodOrdine\*, CodArticolo\*, Quantità, Prezzo)*

*Carrello(CodCarrello, Data, PIVA, Completo)*

*CompCarrello(CodCarrello\*, CodArticolo\*, Quantità, Prezzo)*

- Operazioni che devono essere fatte

Quando il carrello è completo

- a) Trasformo il carrello in ordine

1. Creare un nuovo ordine con la struttura del carrello
2. Rimuovere il carrello

- b) Aggiornare le scorte nel magazzino

**Che operazioni di Manipolazione  
devo fare?**

**In che ordine devo fare le operazioni  
di manipolazione?**



# ESEMPIO (2)

Quando il carrello è completo

a) Trasformo il carrello in ordine

1. Creare un nuovo ordine con la struttura del carrello  
(INSERT in ORDINE e COMPOORDINE)
2. Rimuovere il carrello  
(DELETE in CARRELLO e COMPCARRELLO)

b) Aggiornare le scorte nel magazzino

(UPDATE su MAGAZZINO)



# ESEMPIO (3)

- Creare un nuovo ordine con la struttura del carrello (Lavoriamo con il carrello con il codice 'XYZ')

1) **INSERT INTO** Ordine (CodO, Data, PIVA, DataInvio)

```
(SELECT C.CodO, Data, PIVA, NULL  
FROM Carrello C  
WHERE C.CodO = 'XYZ')
```

Oppure

**INSERT INTO** Ordine (CodO, Data, PIVA)

```
(SELECT C.CodO, Data, PIVA  
FROM Carrello C  
WHERE C.CodO = 'XYZ')
```



# ESEMPIO (4)

- Creare un nuovo ordine con la struttura del carrello (Lavoriamo con il carrello con il codice 'XYZ')

2) **INSERT INTO** CompOrdine

```
(SELECT *  
FROM CompCarrello C  
WHERE C.CodC = 'XYZ')
```



# ESEMPIO (5)

- Creare un nuovo ordine con la struttura del carrello (Lavoriamo con il codice 'XYZ')

## 2) **DELETE**

**FROM** CompCarrello C  
**WHERE** C.CodC = 'XYZ'

## **DELETE**

**FROM** Carrello C  
**WHERE** C.CodC = 'XYZ'



# ESEMPIO (6)

- Aggiornare le scorte nel magazzino

- Per ogni articolo nell'ordine 'XYZ' devo rimuoverne la quantità venduta nel magazzino

- b) **UPDATE** Magazzino M

```
SET M.Quantità = M.Quantità - (SELECT H.Quantità
```

```
FROM CompOrdine H
```

```
WHERE H.CodO = 'XYZ' AND H.CodA = M.CodA)
```

```
WHERE M.CodA IN (SELECT C.CodA
```

```
FROM CompOrdine C
```

```
WHERE C.CodO = 'XYZ')
```



# TRANSACTION

- Sequenza di operazioni effettuate tra un

**BEGIN TRANSACTION**

***INSERT...***

***UPDATE...***

***DELETE...***

***procedure...***

**COMMIT**

**END TRANSACTION**

Parlammo già delle transaction...



# TRIGGER

- Un trigger è una di procedura/funzione di manutenzione della base di dati
- A differenza delle procedure standard, che sono chiamate in maniera esplicita, un trigger non è invocato.
- La loro attivazione è scatenata da una serie di eventi di diversa natura che avvengono nella base di dati
- Questi eventi scatenano una serie di operazioni che servono per mantenere la base di dati
  - Gli eventi che ci interessano, in particolare sono quelli che vanno ad alterare la base di dati
    - INSERT
    - UPDATE
    - DELETE



# CREAZIONE TRIGGER

- Quando voglio creare un trigger, chiaramente devo andare a definire i parametri del trigger:
  1. Quale evento scatena la reazione?
    - a) INSERT
    - b) DELETE
    - c) UPDATE
  2. Quando voglio scatenare la reazione? Prima o dopo l'evento?
    - a. Dopo: C'è stato, ad esempio, un UPDATE e voglio propagare l'evento
    - b. Prima: Ci sarà, ad esempio, una DELETE e voglio eseguire dei comandi prima della cancellazione
  3. Condizioni di Filtraggio che mi permettono di definire precisamente quale è l'evento che scatena la reazione
  4. Operazione globale/per ciascuna riga
  5. Reazione



# SIGNATURE TRIGGER

**CREATE TRIGGER** <nomeTrigger>

[ **AFTER/BEFORE/INSTEAD OF**] <operazione>

[**FOR EACH ROW**]

**WHERE** <condizione>

**BEGIN**

<CORPO DELLA PROCEDURA>

**END**

*INSERT ON <tabella>  
DELETE ON <tabella>  
UPDATE OF <ATTRIBUTO> ON <tabella>*

# TORNANDO ALL'ESEMPIO DI PRIMA

**CREATE TRIGGER** creaOrdine

**AFTER UPDATE ON** CARRELLO **OF** Completo

**FOR EACH ROW**

**WHEN OLD.Completo = 'Incompleto' AND NEW.Completo = 'Completo'**

**BEGIN**

**INSERT INTO** Ordine (CodO, Data, PIVA)

**(SELECT** C.CodO, Data, PIVA

**FROM** Carrello C

**WHERE** C.CodC = 'NEW.CodC');



```
INSERT INTO CompOrdine  
(SELECT *  
FROM CompCarrello C  
WHERE C.CodC = NEW.CodC);  
DELETE  
FROM CompCarrello C  
WHERE C.CodC = NEW.CodC;  
DELETE  
FROM Carrello C  
WHERE C.CodC = NEW.CodC;  
UPDATE Magazzino M  
SET M.Quantità = M.Quantità - (SELECT H.Quantità  
                 FROM CompOrdine H  
                 WHERE H.CodO = 'XYZ' AND H.CodA = M.CodA)  
WHERE M.CodA IN (SELECT C.CodA  
                 FROM CompOrdine C  
                 WHERE C.CodO = NEW.CodC);  
END;
```



# ESEMPIO DI APPLICAZIONE DI UN TRIGGER

Tabella (**ID**, Numero)

```
CREATE TRIGGER incremento  
AFTER UPDATE ON Tabella OF Numero  
FOR EACH ROW  
WHEN NEW.Numero > OLD.Numero  
BEGIN  
    UPDATE Tabella T  
    SET T.Numero = T.Numero+1  
    WHERE NEW.ID = T.ID  
END
```

```
UPDATE Tabella  
SET Numero = Numero+1  
WHERE Tabella.ID = '100'
```



# ESEMPIO

Tabella (**ID**, Numero)

```
CREATE TRIGGER incremento  
AFTER UPDATE ON Tabella OF Numero  
FOR EACH ROW  
WHEN NEW.Numero > OLD.Numero  
BEGIN  
    UPDATE Tabella T  
    SET T.Numero = T.Numero+1  
    WHERE NEW.ID = T.ID  
END
```

```
UPDATE Tabella  
SET Numero = Numero+1  
WHERE Tabella.ID = '100'
```

Quale sarà il valore di Numero,  
dopo l'esecuzione della UPDATE?



# ESEMPIO

Tabella (**ID**, Numero)

```
CREATE TRIGGER incremento  
AFTER UPDATE ON Tabella OF Numero  
FOR EACH ROW  
WHEN NEW.Numero > OLD.Numero  
BEGIN  
    UPDATE Tabella T  
    SET T.Numero = T.Numero+1  
    WHERE NEW.ID = T.ID  
END
```

```
UPDATE Tabella  
SET Numero = Numero+1  
WHERE Tabella.ID = '100'
```

Quale sarà il valore di Numero,  
dopo l'esecuzione della UPDATE?

In tal caso avremo un loop infinito.



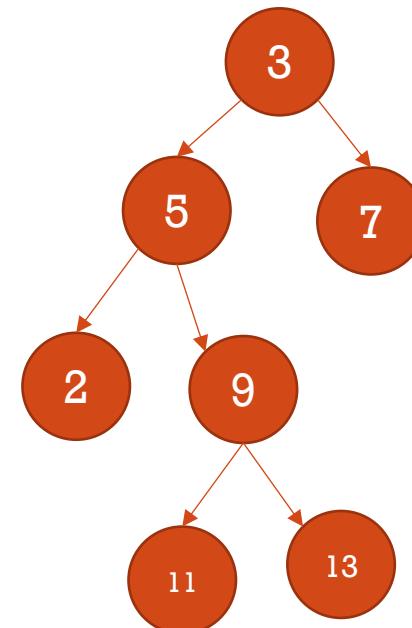
# ALTRÒ ESEMPIO

Tree (CodT, Radice)

Nodo (CodT, CodN, Etichetta)

Arco (CodT, NodoS, NodoD)

Quando incremento il valore di un nodo  
devo incrementare tutti i discendenti della  
stessa quantità.



Ipotizziamo inoltre che vogliamo propagare solo gli incrementi positivi.



**CREATE TRIGGER** propagazione

**AFTER UPDATE** Nodo **ON** Etichetta

**FOR EACH ROW**

**WHEN NEW.Etichetta – OLD.Etichetta > 0**

**BEGIN**

**UPDATE** Nodo N

**SET** N.Etichetta = N.Etichetta + (**NEW.Etichetta – OLD.Etichetta**)

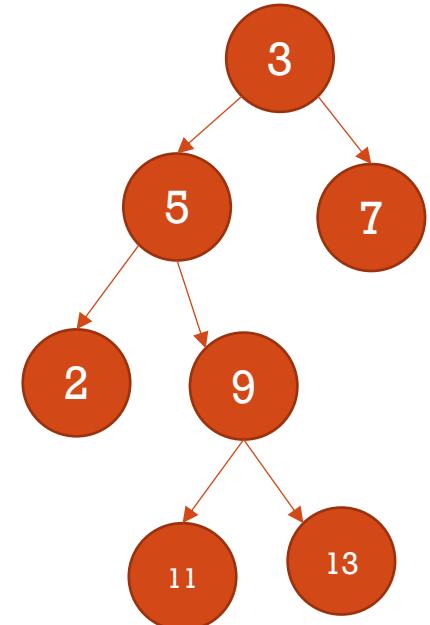
**WHERE** N.CodN **IN** (**SELECT** A.NodoS

**FROM** Arco A

**WHERE** A.CodT = **NEW.CodT**

**AND** A.NodoD = **NEW.CodN**)

**END**



**CREATE TRIGGER** propagazione

**AFTER UPDATE** Nodo **ON** Etichetta

**FOR EACH ROW**

**WHEN NEW.Etichetta – OLD.Etichetta > 0**

**BEGIN**

**UPDATE** Nodo N

**SET** N.Etichetta = N.Etichetta + (**NEW.Etichetta – OLD.Etichetta**)

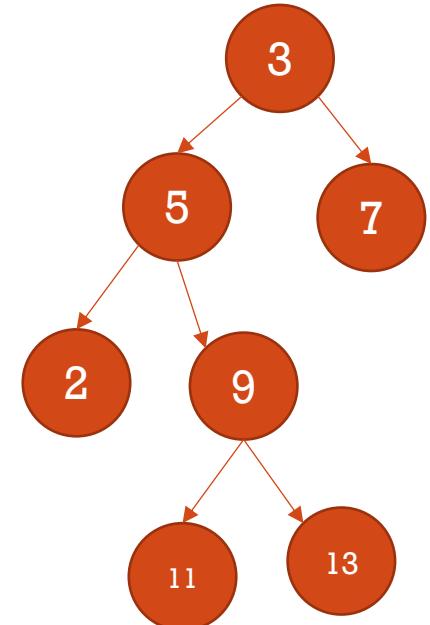
**WHERE** N.CodN **IN** (**SELECT** A.NodoS

**FROM** Arco A

**WHERE** A.CodT = **NEW.CodT**

**AND** A.NodoD = **NEW.CodN**)

**END**



Update sui figli  
di **NEW.CodN**



# BLOCCARE UN TRIGGER

1. Elimino il trigger

**DROP TRIGGER** <nometrigger>

2. Disabilo il trigger (un trigger si considera attivato alla sua azione)

**ALTER TRIGGER** <nometrigger> **DISABLE** (per disattivarlo)

**ALTER TRIGGER** <nometrigger> **ENABLE** (per riattivarlo)



# CLAUSOLA BEFORE

- La clausola BEFORE viene utilizzata quando faccio ad esempio degli inserimenti e voglio controllare i dati che vengono inseriti.
- ESEMPIO – *Chiavi sintetiche*

**CREATE SEQUENCE** *set\_id*

**START WITH** 1000

**INCREMENT BY** 1

**MAX VALUE** 10000



# CLAUSOLA BEFORE 2

Esempio (AutoIncID, Nome, Cognome)

```
INSERT INTO Esempio  
values(NULL, 'Ciro', 'Esposito')
```



# **CLAUSOLA BEFORE 3**

Esempio (AutoIncID, Nome, Cognome)

```
INSERT INTO Esempio  
values(NULL, 'Ciro', 'Esposito')
```

```
CREATE TRIGGER setKey  
BEFORE INSERT on Esempio  
FOR EACH ROW  
BEGIN  
    NEW.AutoIncID := set_id.NEXTVAL  
END
```



# CLAUSOLA BEFORE 3

Esempio (AutoIncID, Nome, Cognome)

```
INSERT INTO Esempio  
values(NULL, 'Ciro', 'Esposito')
```

```
CREATE TRIGGER setKey  
BEFORE INSERT on Esempio  
FOR EACH ROW  
BEGIN  
    NEW.AutoIncID := set_id.NEXTVAL  
END
```

```
CREATE TRIGGER setKey  
BEFORE INSERT on Esempio  
FOR EACH ROW  
BEGIN  
    NEW.AutoIncID := set_id.NEXTVAL;  
    NEW.Nome := UPPER(NEW.Nome);  
    NEW.Cognome := LOWER(NEW.Cognome);  
END
```



# CLAUSOLA INSTEAD OF

- La clausola **INSTEAD OF** non viene utilizzata sulle tabelle, ma è usata esclusivamente per operazioni di manipolazione sulle viste.
- Le Viste sono tabelle virtuali (non memorizzate)
  - Vanno bene per operazioni di lettura (**SELECT**)
  - Sono problematiche se devo fare variazioni di dati (**INSERT, UPDATE e DELETE**)



# ESEMPIO

- Studenti (Matricola, CF, Nome, Cognome, DataN, Residenza)

CF, Nome e Cognome sono attributi totali. DataN e Residenza sono parziali.

**CREATE VIEW** Studenti2 **AS**

**SELECT** Matricola, CF, Nome, Cognome

**FROM** Studenti

1- La vista contiene il campo chiave

2- Contiene gli attributi totali

3- Basata su una sola tabella

**DELETE FROM** Studenti2

**WHERE** Matricola = '340'

**INSERT INTO** Studenti2

**VALUES**('260', 'GLLCRR88M07H703Z', 'Ciro', 'Gallo', )

**UPDATE** Studenti2

**SET** Nome='Gianluca'

**WHERE** CF='GNCFRR91M09H333Q'



# ESEMPIO

- Studenti (Matricola, CF, Nome, Cognome, DataN, Residenza)

CF, Nome e Cognome sono attributi totali. DataN e Residenza sono parziali.

**CREATE VIEW** Studenti2 **AS**

**SELECT** Matricola, CF, Nome, Cognome  
**FROM** Studenti

- 1- La vista contiene il campo chiave
- 2- Contiene gli attributi totali
- 3- Basata su una sola tabella

**DELETE FROM** Studenti2

**WHERE** Matricola = '340'

**INSERT INTO** Studenti2

**VALUES**('260', 'GLLCRR88M07H703Z', 'Ciro', 'Gallo', )

**UPDATE** Studenti2

**SET** Nome='Gianluca'

**WHERE** CF='GNCFRR91M09H333Q'

Queste tre operazioni si ripercuotono  
sulla tabella originale



# ESEMPIO

Studente (Matricola, Nome, Cognome)  
Anagrafe (Matricola, CF, Residenza, DataN)

Relazione 1:1 tra Studente e Anagrafe

```
CREATE VIEW Info AS  
SELECT *  
FROM Studente INNER JOIN Anagrafe
```



```
CREATE TRIGGER Info_Stud  
INSTEAD OF INSERT ON Info OR DELETE ON Info OR UPDATE OF ATTRIBUTO ON...  
FOR EACH ROW  
BEGIN  
    IF INSERTING (TG_OP = INSERT)  
        INSERT INTO Studente (Matricola, Nome, Cognome)  
        VALUES(NEW.Matricola, NEW.Nome, NEW.Cognome)  
        INSERT INTO Anagrafe (Matricola, CF, Residenza, DataN)  
        VALUES(NEW.Matricola, NEW.CF, NEW.Residenza, NEW.DataN)  
    END IF  
    IF DELETING (TG_OP = 'DELETE')  
        DELETE FROM Anagrafe WHERE Matricola = OLD.Matricola  
        DELETE FROM Studente WHERE Matricola = OLD.Matricola  
    END IF  
    IF UPDATING (TG_OP = UPDATE)  
    ....  
    END IF  
END
```

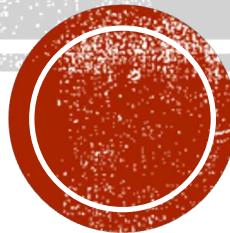




# FINE

Per eventuali domande: (in ordine di preferenza personale)

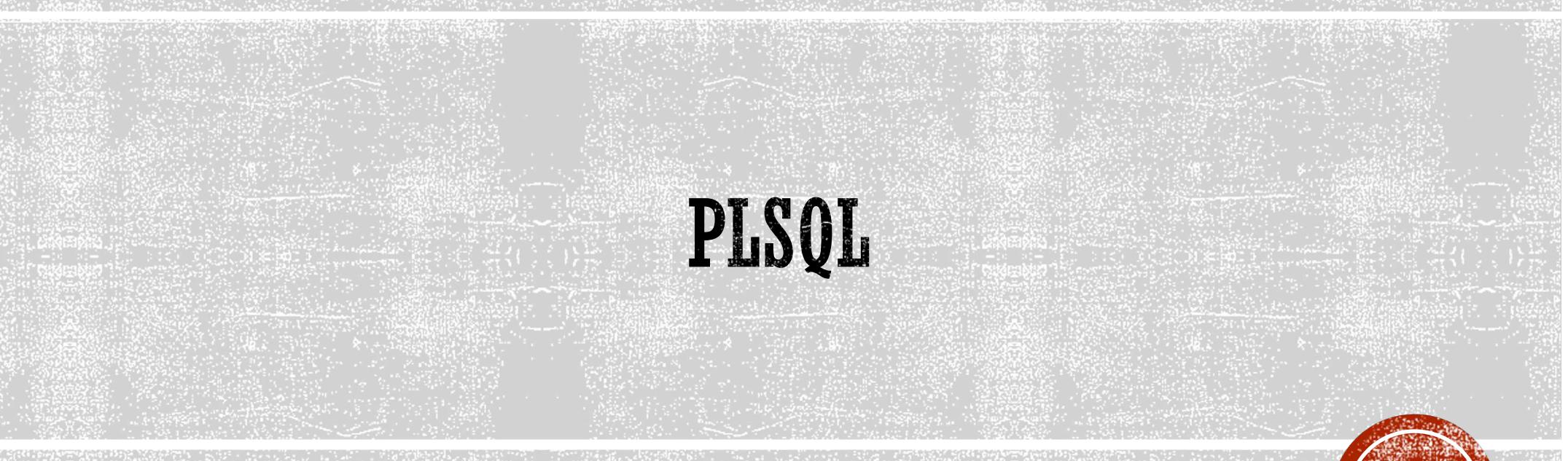
- Ora.
- Chat di Teams
- Mail: [silvio.barra@unina.it](mailto:silvio.barra@unina.it)



# BASI DI DATI I

- PL/SQL
- SQL dinamico





**PLSQL**



# BLOCK STRUCTURE

- I programmi PL/SQL sono divisi in strutture chiamate **blocks**
- Ogni block contiene istruzioni che possono essere
  - PL/SQL
  - Pure SQL

```
[DECLARE
    declaration_statements
]
BEGIN
    executable_statements
[EXCEPTION
    exception_handling_statements
]
END;
/
```

Il blocco DECLARE è opzionale.

Contiene le variabili che saranno utilizzate nel seguito del programma.

Il blocco BEGIN/END contiene tutte le istruzioni che saranno effettivamente eseguite come istruzioni, cicli, ...

Il blocco EXCEPTION è opzionale.

Contiene le istruzioni da eseguire per gestire le eccezioni

Ogni block PL/SQL termina con uno slash (/)



# ESEMPIO

```
SET SERVEROUTPUT ON

DECLARE
    v_width  INTEGER;
    v_height INTEGER := 2;
    v_area   INTEGER := 6;
BEGIN
    -- set the width equal to the area divided by the height
    v_width := v_area / v_height;
    DBMS_OUTPUT.PUT_LINE('v_width = ' || v_width);
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.PUT_LINE('Division by zero');
END;
/
```



# TIPI E VARIABILI

- Sono gli stessi che vengono utilizzati per la definizione delle tabelle
  - Allineamento perfetto tra i tipi delle variabili e le colonne del DB

```
v_product_id      INTEGER;
v_product_type_id INTEGER;
v_name            VARCHAR2(30);
v_description     VARCHAR2(50);
v_price           NUMBER(5, 2);
```

*DECLARE*

```
<nome_variabile> <tipo_del_dato> [:= valore]
<nome_variabile> <tabella>.<attributo>%TYPE
```



# OPERAZIONI DI SELECT

- Una operazione di SELECT può restituire tre tipi di output
  1. Una sola riga
  2. Nessuna riga
  3. Un insieme di righe

Nel **primo caso**, posso andare a recuperare i valori restituiti dichiarando nel blocco *DECLARE* le variabili restituite ed associandole all'output della SELECT tramite la keyword **INTO**.

```
DECLARE
    Vnome, Vcognome VARCHAR(20);
BEGIN
    SELECT S.Nome, S.Cognome INTO Vnome, Vcognome
    FROM Studente S
    WHERE S.Matricola = 'N8600001941'
```



# OPERAZIONI DI SELECT

- Una operazione di SELECT può restituire tre tipi di output
  1. Una sola riga
  2. Nessuna riga
  3. Un insieme di righe

Nel **secondo e terzo caso**, mi trovo nella situazione in cui non so quante righe sono state restituite.

Devo pertanto definire un **CURSOR** per andare a scorrere il ResultSet ottenuto.



# COSTRUTTI CONDIZIONALI

```
IF condition1 THEN  
    statements1  
ELSIF condition2 THEN  
    statements2  
ELSE  
    statements3  
END IF;
```

*condition1 & condition2 sono espressioni booleane  
statements1, statements2, statements3 sono istruzioni PLSQL*

```
    IF v_count > 0 THEN  
        v_message := 'v_count is positive';  
        IF v_area > 0 THEN  
            v_message := 'v_count and v_area are positive';  
        END IF  
    ELSIF v_count = 0 THEN  
        v_message := 'v_count is zero';  
    ELSE  
        v_message := 'v_count is negative';  
    END IF;
```



# COSTRUTTI ITERATIVI

- Abbiamo 3 tipi di costrutti iterativi

1. *LOOP semplici* che vengono eseguiti finché non si chiude esplicitamente il ciclo
2. *WHILE* che cicla finché è verificata una determinata condizione
3. *FOR* che cicla per un predeterminato numero di volte



# LOOP SEMPLICI

*LOOP*

*statements*

*END LOOP*

Per terminare il ciclo si può usare:

- EXIT per terminare il loop immediatamente
- EXIT WHEN per terminare il loop quando una determinata condizione occorre

```
v_counter := 0;  
LOOP  
    v_counter := v_counter + 1;  
    EXIT WHEN v_counter = 5;  
END LOOP;
```



# LOOP SEMPLICI - CONTINUE

```
v_counter := 0;  
LOOP  
    -- after the CONTINUE statement is executed, control returns here  
    v_counter := v_counter + 1;  
    IF v_counter = 3 THEN  
        CONTINUE; -- end current iteration unconditionally  
    END IF;  
    EXIT WHEN v_counter = 5;  
END LOOP;
```

---

```
v_counter := 0;  
LOOP  
    -- after the CONTINUE WHEN statement is executed, control returns here  
    v_counter := v_counter + 1;  
    CONTINUE WHEN v_counter = 3; -- end current iteration when v_counter = 3  
    EXIT WHEN v_counter = 5;  
END LOOP;
```



# WHILE

*WHILE condition LOOP*

*statements*

*END LOOP;*

```
v_counter := 0;  
WHILE v_counter < 6 LOOP  
    v_counter := v_counter + 1;  
END LOOP;
```



# CICLI FOR

- Un ciclo for è eseguito per un determinato numero di volte
- Si imposta il numero di volte specificando il *lower bound* e l' *upper bound* per la variabile di loop
- La variabile è incrementata/decrementata ogni volta che si fa un nuovo giro nel loop.

```
FOR loop_variable IN [REVERSE] lower_bound..upper_bound LOOP  
    statements  
END LOOP;
```



# CICLI FOR (2)

- *loop\_variable* è la variabile che regola il loop. È possibile utilizzare una variabile che già esiste o se ne utilizza una apposta per il loop (creata nel caso in cui la variabile specificata non esiste). La variabile è incrementata di 1 ogni volta che si passa attraverso il loop. Se non definita al di fuori del loop, la variabile non è più visible all'uscita del FOR.
- *REVERSE* indica se la variabile deve essere incrementata o decrementata. In ogni caso, il lower bound deve essere specificato prima dell'upper bound.

```
FOR v_counter2 IN 1..5 LOOP  
    DBMS_OUTPUT.PUT_LINE(v_counter2);  
END LOOP;
```

```
FOR v_counter2 IN REVERSE 1..5 LOOP  
    DBMS_OUTPUT.PUT_LINE(v_counter2);  
END LOOP;
```



# CURSORI

- Un cursore viene dichiarato nel blocco *DECLARE*

**DECLARE**

```
Vmatricola Studente.Matricola%TYPE;  
Vnome Studente.Nome%TYPE;  
cursor scansiona_cognome IS  
    SELECT S.Nome, S.Matricola  
    FROM Studente S  
    WHERE S.Cognome = 'Russo'
```

**BEGIN**

```
OPEN scansiona_cognome  
FETCH scansiona_cognome INTO Vnome, Vmatricola;  
...  
...  
CLOSE scansiona_cognome
```

- *La dichiarazione del cursore non esegue l'interrogazione*



# CURSORI

```
DECLARE
    cursor scansiona_cognome IS
        SELECT S.Nome, S.Matricola
        FROM Studente S
        WHERE S.Cognome = 'Russo'
    rigacorrente scansiona_cognome%ROWTYPE
BEGIN
    FETCH scansiona_cognome INTO rigacorrente
    ...rigacorrente.Matricola...
    ...rigacorrente.Nome...
```

Posso evitare di definire le variabili che vanno ad ospitare gli attributi recuperati dal cursore definendo un tipo **%ROWTYPE** collegato al cursore



# ESEMPIO CURSOR

```
LOOP
    -- fetch the rows from the cursor
    FETCH v_product_cursor
    INTO v_product_id, v_name, v_price;

    -- exit the loop when there are no more rows, as indicated by
    -- the Boolean variable v_product_cursor%NOTFOUND (= true when
    -- there are no more rows)

    EXIT WHEN v_product_cursor%NOTFOUND;

    -- use DBMS_OUTPUT.PUT_LINE() to display the variables
    DBMS_OUTPUT.PUT_LINE(
        'v_product_id = ' || v_product_id || ', v_name = ' || v_name ||
        ', v_price = ' || v_price
    );
END LOOP;
```



```

-- This script displays the product_id, name, and price columns
-- from the products table using a cursor

SET SERVEROUTPUT ON

DECLARE
    -- step 1: declare the variables
    v_product_id products.product_id%TYPE;
    v_name      products.name%TYPE;
    v_price     products.price%TYPE;

    -- step 2: declare the cursor
    CURSOR v_product_cursor IS
        SELECT product_id, name, price
        FROM products
        ORDER BY product_id;
    BEGIN
        -- step 3: open the cursor
        OPEN v_product_cursor;

        LOOP
            -- step 4: fetch the rows from the cursor
            FETCH v_product_cursor
            INTO v_product_id, v_name, v_price;

            -- exit the loop when there are no more rows, as indicated by
            -- the Boolean variable v_product_cursor%NOTFOUND (= true when
            -- there are no more rows)
            EXIT WHEN v_product_cursor%NOTFOUND;

            -- use DBMS_OUTPUT.PUT_LINE() to display the variables
            DBMS_OUTPUT.PUT_LINE(
                'v_product_id = ' || v_product_id || ', v_name = ' || v_name ||
                ', v_price = ' || v_price
            );
        END LOOP;

        -- step 5: close the cursor
        CLOSE v_product_cursor;
    END;
/

```

## ESEMPIO CON il FOR LOOP per i CURSORI

```

-- This script displays the product_id, name, and price columns
-- from the products table using a cursor and a FOR loop

SET SERVEROUTPUT ON

DECLARE
    CURSOR v_product_cursor IS
        SELECT product_id, name, price
        FROM products
        ORDER BY product_id;
    BEGIN
        FOR v_product IN v_product_cursor LOOP
            DBMS_OUTPUT.PUT_LINE(
                'product_id = ' || v_product.product_id ||
                ', name = ' || v_product.name ||
                ', price = ' || v_product.price
            );
        END LOOP;
    END;
/

```

# ECCEZIONI

- Le eccezioni sono utilizzate per gestire errori a tempo di esecuzione negli statement PL/SQL.
- Il blocco *EXCEPTION* è in carica di recuperare le eccezioni e gestirle come specificato (se specificato)
  - Altrimenti ogni eccezione ha un gestore di default che la prende in consegna



Exception	Error	Description	SELF_IS_NULL	ORA-30625	An attempt was made to call a MEMBER method on a null object. That is, the built-in parameter SELF (which is always the first parameter passed to a MEMBER method) is null.
ACCESS_INTO_NULL	ORA-06530	An attempt was made to assign values to the attributes of an uninitialized database object. (You'll learn about objects in Chapter 13.)			
CASE_NOT_FOUND	ORA-06592	None of the WHEN clauses of a CASE statement was selected, and there is no default ELSE clause.	STORAGE_ERROR	ORA-06500	The PL/SQL module ran out of memory or the memory has been corrupted.
COLLECTION_IS_NULL	ORA-06531	An attempt was made to call a collection method (other than EXISTS) on an uninitialized nested table or varray, or an attempt was made to assign values to the elements of an uninitialized nested table or varray. (You'll learn about collections in Chapter 14.)	SUBSCRIPT_BEYOND_COUNT	ORA-06533	An attempt was made to reference a nested table or varray element using an index number larger than the number of elements in the collection.
CURSOR_ALREADY_OPEN	ORA-06511	An attempt was made to open an already open cursor. The cursor must be closed before it can be reopened.	SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	An attempt was made to reference a nested table or varray element using an index number that is outside the legal range (-1 for example).
DUP_VAL_ON_INDEX	ORA-00001	An attempt was made to store duplicate values in a column that is constrained by a unique index.	SYS_INVALID_ROWID	ORA-01410	The conversion of a character string to a universal rowid failed because the character string does not represent a valid rowid.
INVALID_CURSOR	ORA-01001	An attempt was made to perform an illegal cursor operation, such as closing an unopened cursor.	TIMEOUT_ON_RESOURCE	ORA-00051	A timeout occurred while the database was waiting for a resource.
INVALID_NUMBER	ORA-01722	An attempt to convert a character string into a number failed because the string does not represent a valid number.  Note: In PL/SQL statements, VALUE_ERROR is raised instead of INVALID_NUMBER.	TOO_MANY_ROWS	ORA-01422	A SELECT INTO statement returned more than one row.
LOGIN_DENIED	ORA-01017	An attempt was made to connect to a database using an invalid user name or password.	VALUE_ERROR	ORA-06502	An arithmetic, conversion, truncation, or size-constraint error occurred.  For example, when selecting a column value into a character variable, if the value is longer than the declared length of the variable, PL/SQL aborts the assignment and raises VALUE_ERROR.  Note: In PL/SQL statements, VALUE_ERROR is raised if the conversion of a character string into a number fails. In SQL statements, INVALID_NUMBER is raised instead of VALUE_ERROR.
NO_DATA_FOUND	ORA-01403	A SELECT INTO statement returned no rows, or an attempt was made to access a deleted element in a nested table or an uninitialized element in an "index by" table.	ZERO_DIVIDE	ORA-01476	An attempt was made to divide a number by zero.
NOT_LOGGED_ON	ORA-01012	An attempt was made to access a database item without being connected to the database.			
PROGRAM_ERROR	ORA-06501	PL/SQL had an internal problem.			
ROWTYPE_MISMATCH	ORA-06504	The host cursor variable and the PL/SQL cursor variable involved in an assignment have incompatible return types.  For example, when an open host cursor variable is passed to a stored procedure or function, the return types of the actual and formal parameters must be compatible.			

# L'ECCEZIONE OTHERS

- Se non sappiamo quale eccezione può essere lanciata e pertanto non sappiamo quale specificare all'interno del blocco, possiamo sempre affidarci all'eccezione *OTHERS*
- L'eccezione OTHERS matcha con tutte le eccezioni.

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(1 / 0);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('An exception occurred');
END;
/
```

An exception occurred



# PROCEDURE

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
    procedure_body
END procedure_name;
```

- *IN* indica che il parametro deve essere settato ad un valore quando la procedura è eseguita
- *OUT* indica che il valore del parametro è impostato ad un valore nel corpo della procedura
- *IN OUT* indica che il parametro può avere un valore quando è eseguita la procedura e che questo può cambiare nel corpo



# ESEMPIO

```
CREATE PROCEDURE update_product_price(
    p_product_id IN products.product_id%TYPE,
    p_factor      IN NUMBER
) AS
    v_product_count INTEGER;
BEGIN
    -- count the number of products with the supplied product_id
    -- (the count will be 1 if the product exists)
    SELECT COUNT(*)
    INTO v_product_count
    FROM products
    WHERE product_id = p_product_id;

    -- if the product exists (v_product_count = 1) then
    -- update that product's price
    IF v_product_count = 1 THEN
        UPDATE products
        SET price = price * p_factor
        WHERE product_id = p_product_id;
        COMMIT;
    END IF;
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
END update_product_price;
/
```

- La procedura accetta in input due parametri, entrambi in IN mode; pertanto i valori per i due parametri devono essere impostati quando la procedura è chiamata.
- E soprattutto non possono essere modificati nel corpo della procedura

# ATTENZIONE

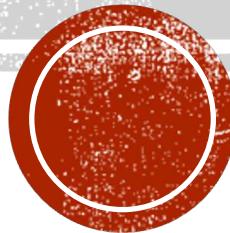
- NON SI PUO' UTILIZZARE UNA SELECT PURA ALL'INTERNO DEL CORPO DELLA PROCEDURA
  - O la si collega ad un cursore
  - O si usa la parola chiave INTO
- NON SI PUO' UTILIZZARE UNA SELECT INTO PER LE QUERY CHE RESTITUISCONO PIU' DI UNA RIGA



# STRING FUNCTIONS

ASCII	ASCII('A')	65	Returns an ASCII code value of a character.	REGEXP_LIKE	REGEXP_LIKE( 'Year of 2017','\d+' )	true	Match a string based on a regular expression pattern.
CHR	CHR('65')	'A'	Converts a numeric value to its corresponding ASCII character.		REGEXP_REPLACE	REGEXP_REPLACE( 'Year of 2017','\d+', 'Dragon' )	'Year of Dragon'
CONCAT	CONCAT('A','BC')	'ABC'	Concatenate two strings and return the combined string.	REGEXP_SUBSTR	REGEXP_SUBSTR( 'Number 10', '\d+' )	10	Extract substrings from a string using a pattern of a regular expression.
CONVERT	CONVERT( 'Ã‰ ï', 'US7ASCII', 'WE8ISO8859P1' )	'A E I'	Convert a character string from one character set to another.		REPLACE	REPLACE('JACK AND JOND','J','BL');	'BLACK AND BLOND'
DUMP	DUMP('A')	Typ=96 Len=: 65	Return a string value (VARCHAR2) that includes the datatype code, length measured in bytes, and internal representation of a specified expression.	RPAD	RPAD('ABC',5,'*')	'ABC**'	Return a string that is right-padded with the specified characters to a certain length.
INITCAP	INITCAP('hi there')	'Hi There'	Converts the first character in each word in a specified string to uppercase and the rest to lowercase.		RTRIM	RTRIM(' ABC ')	' ABC'
INSTR	INSTR( 'This is a playlist', 'is' )	3	Search for a substring and return the location of the substring in a string	SOUNDEX	SOUNDEX('sea')	'S000'	Return a phonetic representation of a specified string.
LENGTH	LENGTH('ABC')	3	Return the number of characters (or length) of a specified string		SUBSTR	SUBSTR('Oracle Substring', 1, 6 )	'Oracle'
LOWER	LOWER('Abc')	'abc'	Return a string with all characters converted to lowercase.	TRANSLATE	TRANSLATE('12345', '143', 'bx')	'b2x5'	Replace all occurrences of characters by other characters in a string.
LPAD	LPAD('ABC',5,'*')	'**ABC'	Return a string that is left-padded with the specified characters to a certain length.		TRIM	TRIM(' ABC ')	'ABC'
LTRIM	LTRIM(' ABC ')	'ABC '	Remove spaces or other specified characters in a set from the left end of a string.	UPPER	UPPER('Abc')	'ABC'	Convert all characters in a specified string to uppercase.
REGEXP_COUNT	REGEXP_COUNT('1 2 3 abc','\d')	3	Return the number of times a pattern occurs in a string.				
REGEXP_INSTR	REGEXP_INSTR('Y2K problem','\d+')	2	Return the position of a pattern in a string.				

# SQL DINAMICO



# SQL DINAMICO

- Il Dynamic SQL consente di eseguire comandi SQL prodotti a runtime.
- Le differenze principali con l'SQL statico è che mentre nell'SQL statico a parte i parametri, la struttura del comando rimane la stessa, nell'SQL dinamico è l'intero comando ad essere prodotto a tempo di esecuzione.

SQL

- CREATE, ALTER, DROP
- INSERT, UPDATE, DELETE
- COMMIT, ROLLBACK

PL/SQL

```
CREATE PROCEDURE nome_procedura
IS {or AS} -- la clausola IS sostituisce DECLARE definizioni;
BEGIN
    corpo_procedura;
EXCEPTION
    gestione delle eccezioni
END;
/
```



# SQL DINAMICO

- Il Dynamic SQL consente di eseguire comandi SQL prodotti a runtime.
- Le differenze principali con l'SQL statico è che mentre nell'SQL statico a parte i parametri, la struttura del comando rimane la stessa, nell'SQL dinamico è l'intero comando ad essere prodotto a tempo di esecuzione.

## SQL DINAMICO

```
CREATE PROCEDURE CANCELLA_TABELLA (table_name IN VARCHAR2)
AS
    sql_istr VARCHAR2(100);
BEGIN
    sql_istr := 'DROP TABLE ' || table_name;
    EXECUTE IMMEDIATE (sql_istr );
EXCEPTION
    gestione delle eccezioni
END;
/
```



# MODALITÀ DI INTERAZIONE

- SQL Dinamico mette a disposizione due modalità di interazione
  - 1. La gestione dell'interrogazione avviene in due fasi
    - PREPARE in cui vi è una fase di preparazione del comando
    - EXECUTE in cui il comando è mandato in esecuzione
  - 2. L'interrogazione è eseguita immediatamente O in un parametro di tipo stringa che contiene il comando O in un comando specificato direttamente come parametro della EXECUTE IMMEDIATE



# MODALITA' 1: PREPARE & EXECUTE

- Il comando PREPARE analizza l'istruzione SQL e ne prepara una traduzione  
**PREPARE <nome comando> FROM <comando SQL>**
- Dove **<nome comando>** è il nome associato da **PREPARE** alla traduzione del comando SQL
- Il comando SQL può contenere dei parametri in ingresso rappresentati da ?  
**PREPARE comandoSQL**  
**FROM 'SELECT nome FROM studente WHERE matricola = ?'**



# MODALITÀ 1: PREPARE & EXECUTE

- L'esecuzione del comando **PREPARE** associa alla variabile **comandoSQL** la traduzione dell'interrogazione, con un parametro in ingresso che rappresenta la matricola dello studente.
- L'esecuzione della query avviene tramite il comando **EXECUTE**

**EXECUTE <nome comando> [INTO <variabiliTarget>] [USING <lista parametri>]**

- **<variabiliTarget>** contiene l'elenco dei parametri in cui deve essere scritto il risultato del comando
- **<lista parametri>** specifica i valori che devono essere assunti dai parametri variabili



# MODALITA' 1: PREPARE & EXECUTE

**EXECUTE comandoSQL INTO nomeStudente USING matr\_studente**

dove **matr\_studente** è una variabile in cui è inserita la matricola dello studente  
*N86001941*

- Pertanto l'esecuzione del comando effettua la query

**SELECT nome**

**FROM studente**

**WHERE matricola = 'N86001941'**



# MODALITÀ 2: EXECUTE IMMEDIATE

- Nella seconda modalità, quella che verrà utilizzata più frequentemente, la **prepare** non è eseguita esplicitamente con un comando apposta, bensì viene effettuata contestualmente alla preparazione.
- Il comando che si utilizza è

**EXECUTE IMMEDIATE <nome comando> [INTO <listaTarget>] [USING <listaPar>]**

```
comandoSQL := “DELETE FROM Studente WHERE Matricola='N86001941' ”;
EXECUTE IMMEDIATE comandoSQL;
```



# ESEMPIO

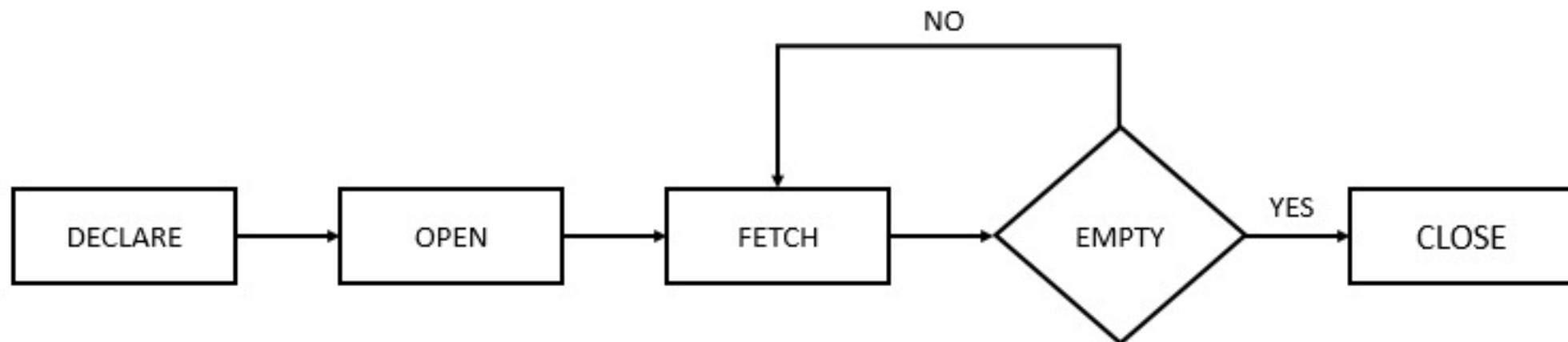
```
DECLARE
    sql_stmt      VARCHAR2(200);
    plsql_block   VARCHAR2(500);
    emp_id        NUMBER(4) := 7566;
    salary         NUMBER(7,2);
    dept_id       NUMBER(2) := 50;
    dept_name     VARCHAR2(14) := 'PERSONNEL';
    location       VARCHAR2(13) := 'DALLAS';
    emp_rec        emp%ROWTYPE;
BEGIN
    EXECUTE IMMEDIATE 'CREATE TABLE bonus (id NUMBER, amt NUMBER)';
    sql_stmt := 'INSERT INTO dept VALUES (:1, :2, :3)';
    EXECUTE IMMEDIATE sql_stmt USING dept_id, dept_name, location;
    sql_stmt := 'SELECT * FROM emp WHERE empno = :id';
    EXECUTE IMMEDIATE sql_stmt INTO emp_rec USING emp_id;
    plsql_block := 'BEGIN emp_pkg.raise_salary(:id, :amt); END;';
    EXECUTE IMMEDIATE plsql_block USING 7788, 500;
    sql_stmt := 'UPDATE emp SET sal = 2000 WHERE empno = :1
                RETURNING sal INTO :2';
    EXECUTE IMMEDIATE sql_stmt USING emp_id RETURNING INTO salary;
    EXECUTE IMMEDIATE 'DELETE FROM dept WHERE deptno = :num'
                    USING dept_id;
    EXECUTE IMMEDIATE 'ALTER SESSION SET SQL_TRACE TRUE';
END;
```



# CLASSICO CURSOR

Quando il comando SQL è esplicito

```
CURSOR <nome_cursore> IS <comando SQL>
OPEN <nome_cursore>;
FETCH <nome_cursore> INTO <lista parametri>;
CLOSE <nome_cursore>;
```



# CURSOR PER L'SQL DINAMICO

- Un cursore può anche non essere legato obbligatoriamente ad una query in particolare.
- Ciò significa che il cursore potrebbe essere aperto per una qualsiasi query
- Il vantaggio principale di questo tipo di cursori è il seguente:
  1. Innanzitutto la possibilità di utilizzare un cursore per una query costruita dinamicamente

```
comandoSQL := 'SELECT * FROM studente WHERE ' || nomeAttributo || '=' || nomeValore;  
OPEN nomeCursore FOR comandoSQL;
```

2. Inoltre, è possibile restituire una variabile di tipo cursore come output di una function



```
CREATE OR REPLACE FUNCTION get_direct_reports(
    in_manager_id IN employees.manager_id%TYPE)
RETURN SYS_REFCURSOR
AS
    c_direct_reports SYS_REFCURSOR;
BEGIN
    OPEN c_direct_reports FOR
        SELECT
            employee_id,
            first_name,
            last_name,
            email
        FROM
            employees
        WHERE
            manager_id = in_manager_id
        ORDER BY
            first_name,
            last_name;
    RETURN c_direct_reports;
END;
```

```
DECLARE
    c_direct_reports SYS_REFCURSOR;
    l_employee_id employees.employee_id%TYPE;
    l_first_name employees.first_name%TYPE;
    l_last_name employees.last_name%TYPE;
    l_email      employees.email%TYPE;
BEGIN
    -- get the ref cursor from function
    c_direct_reports := get_direct_reports(46);

    -- process each employee
    LOOP
        FETCH
            c_direct_reports
        INTO
            l_employee_id,
            l_first_name,
            l_last_name,
            l_email;
        EXIT
        WHEN c_direct_reports%notfound;
            dbms_output.put_line(l_first_name || ' ' || l_last_name || ' - ' ||
        END LOOP;
    -- close the cursor
    CLOSE c_direct_reports;
END;
/
```

# FUNCTION

- Una function è simile ad una procedura, con la differenza che viene definito un valore di ritorno della funzione.

```
CREATE [OR REPLACE] FUNCTION function_name
  [(parameter_name [IN | OUT | IN OUT] type [, ...])]  
RETURN type  
{IS | AS}  
BEGIN  
  function_body  
END function_name;
```



# ESEMPIO DI UNA FUNZIONE

```
CREATE FUNCTION circle_area (
    p_radius IN NUMBER
) RETURN NUMBER AS
    v_pi    NUMBER := 3.1415926;
    v_area NUMBER;
BEGIN
    -- circle area is pi multiplied by the radius squared
    v_area := v_pi * POWER(p_radius, 2);

    RETURN v_area;
END circle_area;
/
```



# CALL DI UNA PROCEDURE VS CALL DI UNA FUNCTION

- È possibile chiamare la procedura direttamente dalla console del DBMS

**CALL <nome\_procedura( [lista\_parametri])>;**

- Mentre in JDBC

```
CallableStatement cs = null;
cs = this.con.prepareCall("{call SHOW_SUPPLIERS}");
ResultSet rs = cs.executeQuery();
```

- Per quanto riguarda la function è necessario chiamarla all'interno di una select

```
SELECT <nomefunzione([lista_parametri])>
FROM nome_tabella;
```

- Quando non è possibile definire una tabella su cui si deve effettuare la funzione, si usa la tabella **dual**, la quale rappresenta una dummy table (esclusivamente per ORACLE).



# FUNCTIONS IN JDBC

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Types;
public class CallingFunctionsExample {
    public static void main(String args[]) throws SQLException {
        //Registering the Driver
        DriverManager.registerDriver(new com.mysql.jdbc.Driver());
        //Getting the connection
        String mysqlUrl = "jdbc:mysql://localhost/mydatabase";
        Connection con = DriverManager.getConnection(mysqlUrl, "root", "password");
        System.out.println("Connection established.....");
        //Preparing a CallableStatement to call a function
        CallableStatement cstmt = con.prepareCall("{? = call getDob(?)}");
        //Registering the out parameter of the function (return type)
        cstmt.registerOutParameter(1, Types.DATE);
        //Setting the input parameters of the function
        cstmt.setString(2, "Amit");
        //Executing the statement
        cstmt.execute();
        System.out.print("Date of birth: "+cstmt.getDate(1));
    }
}
```

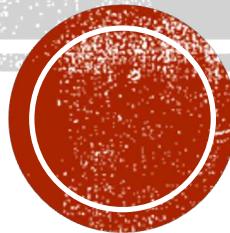




# FINE

Per eventuali domande: (in ordine di preferenza personale)

- Ora.
- Chat di Teams
- Mail: [silvio.barra@unina.it](mailto:silvio.barra@unina.it)





DIE UNIVERSITÀ DEGLI STUDI DI  
TI. NA POLI FEDERICO II  
DIPARTIMENTO DI INGEGNERIA ELETTRICA  
E DELLE TECNOLOGIE DELL'INFORMAZIONE

A.A. 2021/2022

# BASI DI DATI I

- Java DataBase Connectivity

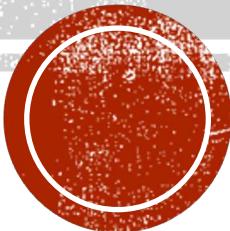
Prof. Adriano Peron  
Prof. Silvio Barra

# COS'È JDBC ?

- E' un insieme di classi ed interfacce che forniscono un'API standard per operare con Basi di Dati Relazionali in pure JAVA .
- JDBC è un trademark SUN (dal 2010 Oracle)
  - Acronimo di Java Database Connectivity?
- Definito nel 1996 (JDBC 1.0),
  - Molto migliorato nel 1998 per Java 2 (JDBC 2.0),
  - Ottimizzato nel 2003 (JDBC 3.0).
- E' una base per altri package di più alto livello



# **ARCHITETTURE CLIENT-SERVER**



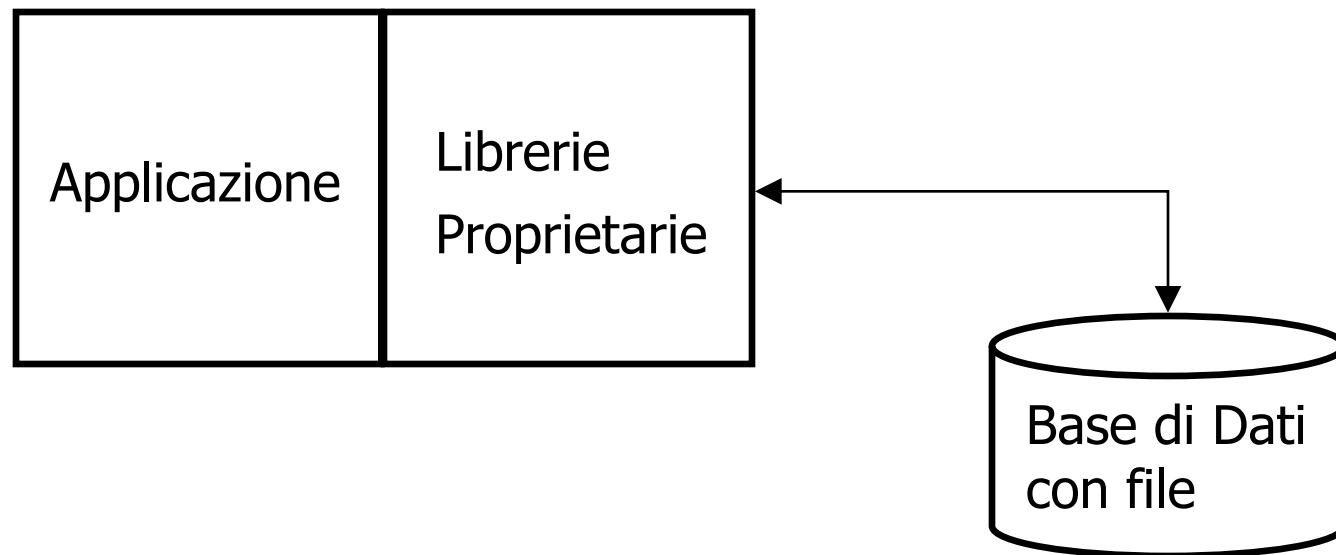
# ACCESSO DIRETTO

- **Vantaggi**

- **Efficiente per soluzioni piccolissime**

- **Svantaggi**

- Sviluppo molto lento
- Non flessibile/scalabile
- Non condivisibile
- Distoglie dalla ‘business logic’



# UTILIZZO DI DBMS

- Un DataBase Management System è un'applicazione general-purpose, la cui unica funzione è manipolare basi di dati
- Fornisce un modo per memorizzare informazioni in strutture dati efficienti, scalabili e flessibili,
- Permette ricerche da più ‘direzioni’

# UTILIZZO DI DBMS (2)

- Permette la definizione di architetture client-server (o 2-tiered)
  - Il client conosce come gestire i dati
  - Il DBMS gestisce lo storage ed il retrieval dei dati
    - Lo sviluppatore scrive solo la “business logic”
- Esempi: Access, Oracle, PostgreSQL, SQL Server, Informix, ecc...

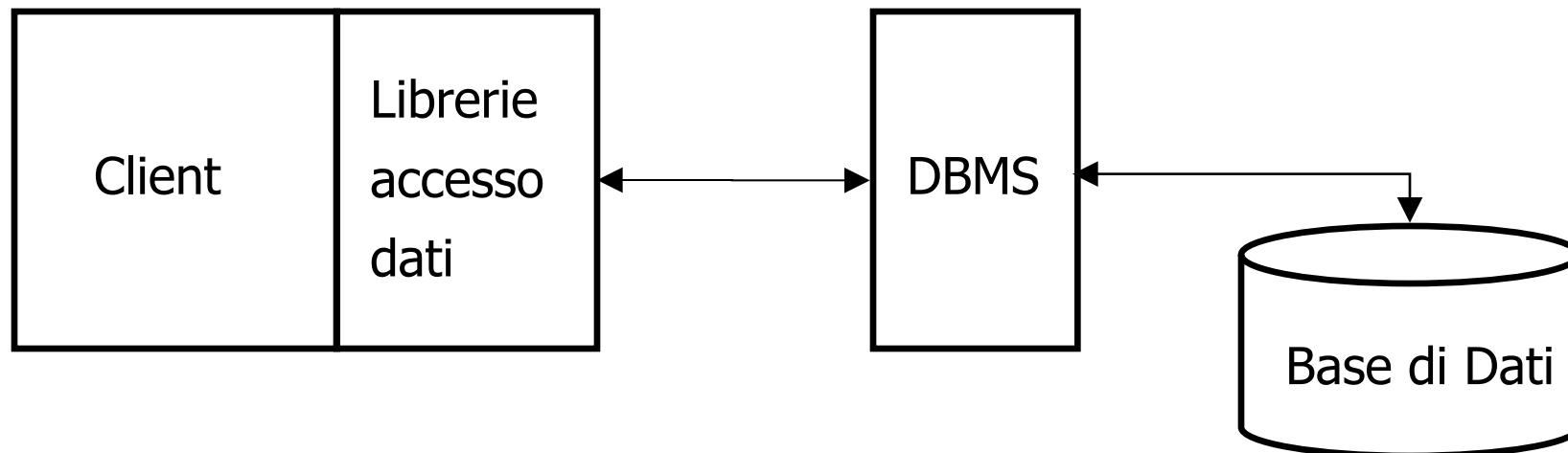
# ARCHITETTURA 2-TIER

- Vantaggi

- Molto più efficiente
- Sviluppo rapido
- Scalabile/flessibile

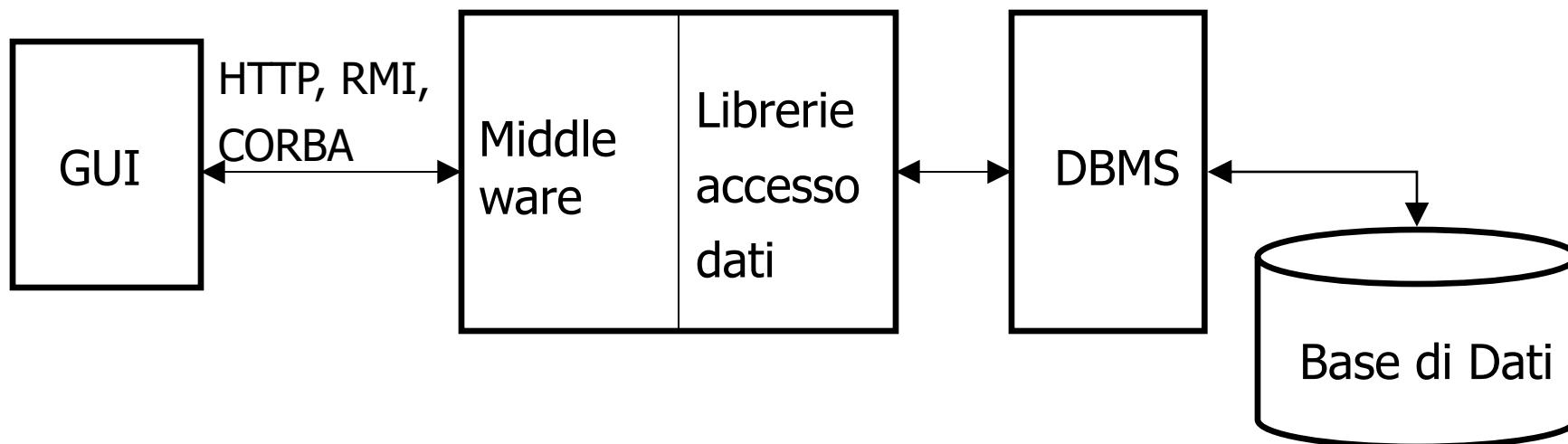
- Svantaggi

- Sicurezza
- # licenze per DBMS
- GUI legata a business logic
- Non utilizzabile sul Web



# ARCHITETTURA 3-TIER

- Vantaggi
  - Policy sicurezza
  - GUI indipendente dai dati
  - Orientata al Web
- Svantaggi
  - Maggiore complessità



# COME INTERAGIRE CON SQL

- Usare SQL interattivamente...

The screenshot shows a SQL development environment with two main windows. The top window is titled 'Centro comandi' and contains a toolbar, a menu bar with 'Centro comandi', 'Interattivo', 'Editare', 'Strumenti', and 'Aiuto', and several tabs: 'Interattivo' (selected), 'Script', 'Risultati dell'interrogazione', and 'Plan di accesso'. Below the tabs, it shows a connection named 'CSITE61 - DB2 - SAMPLE'. The 'Cronologia dei comandi' pane displays the SQL query: 'SELECT LastName, Salary, Job, Birthdate FROM Employee WHERE Salary > 30000 ORDER BY S...'. The 'Comando' pane shows the same query again. To the right of the command panes are buttons for 'SQL Assist' and 'Accodare allo script'. The bottom window is titled 'Risultati dell'interrogazione' and displays the query results in a grid:

LASTNAME	SALARY	JOB	BIRTHDATE
HAAS	52750.00	PRES	1933-08-24
LUCCHESI	46500.00	SALESREP	1929-11-05
THOMPSON	41250.00	MANAGER	1948-02-02
GEYER	40175.00	MANAGER	1925-09-15
KWAN	38250.00	MANAGER	1941-05-11
PULASKI	36170.00	MANAGER	1953-05-26
STERN	32250.00	MANAGER	1945-07-07

Non è adatto ai nostri scopi!

# COME INTERAGIRE CON SQL (2)

- Inserire SQL nel codice...

```
System.out.println("Retrieve some data from the database...");  
Statement stmt = con.createStatement();  
ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE");  
// display the result set  
while (rs.next()) {  
    String a = rs.getString(1);  
    String str = rs.getString(2);  
    System.out.print(" empno= " + a);  
    System.out.print(" firstname= " + str);  
    System.out.print("\n");  
}  
rs.close();  
stmt.close();
```

Come gestire la connessione con il DBMS?

# COME INTERAGIRE COL DBMS

- **Embedded SQL**
  - Il programma colloquia direttamente col DBMS
  - Gli statement SQL sono compilati utilizzando un precompilatore specifico del DBMS.
  - SQL diviene parte integrante del codice.
- **Call Level Interface**
  - Il programma accede al DBMS per mezzo di un'interfaccia standard
  - Gli statement SQL non sono compilati, ma inviati al DBMS a run-time.
  - Il DBMS può essere sostituito
  - La stessa interfaccia può interagire con più DBMS

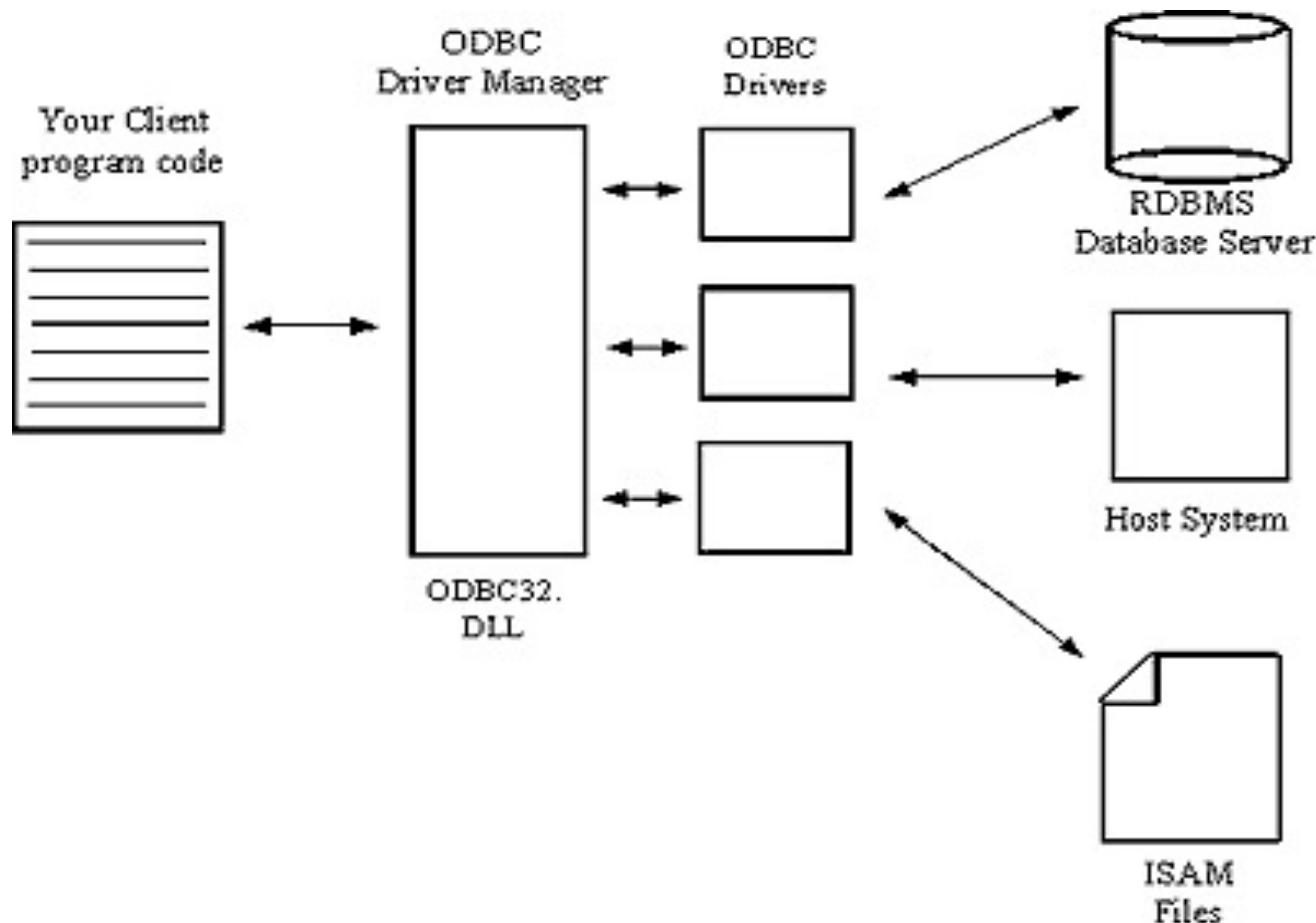
# ESEMPI DI CLI

- **ODBC**
  - Open Database Connectivity
- **OLE-DB**
  - Object Linking and Embedding for Databases
- **ADO**
  - Active Data Objects
- **UDA**
  - Universal Data Access
- **JDBC**

# ODBC

- API standard definita da Microsoft nel 1992
- La prima implementazione di un CLI
- Permette l'accesso a dati residenti in DBMS diversi (Access, MySQL, DB2, Oracle, ...)
  - Supportato da praticamente tutti i DBMS in commercio
- Gestisce richieste SQL convertendole in un formato comprensibile al particolare DBMS
  - Permette agli sviluppatori di formulare richieste SQL a DB distinti senza dover conoscere le interfacce di programmazione proprietarie di ogni singolo DB

# ARCHITETTURA ODBC



◆ Esempio di sorgente dati in Windows

# **ODBC (SVANTAGGI)**

- Utilizza interfacce C
- E' procedurale (NO O-O)
- Ha poche funzioni con molti parametri
- E' abbastanza ostico

# JDBC

# MOTIVAZIONI DI JDBC

- I Db sono il ‘core system’ di gran parte dei sistemi client-server
- 
- Quasi la totalità delle entità necessita di accedere ad un DB
  - Si vuole che tale accesso sia :
    1. Cross-plataform, cioè scritto in JAVA
    2. Capace di interagire con qualsiasi RDBMS

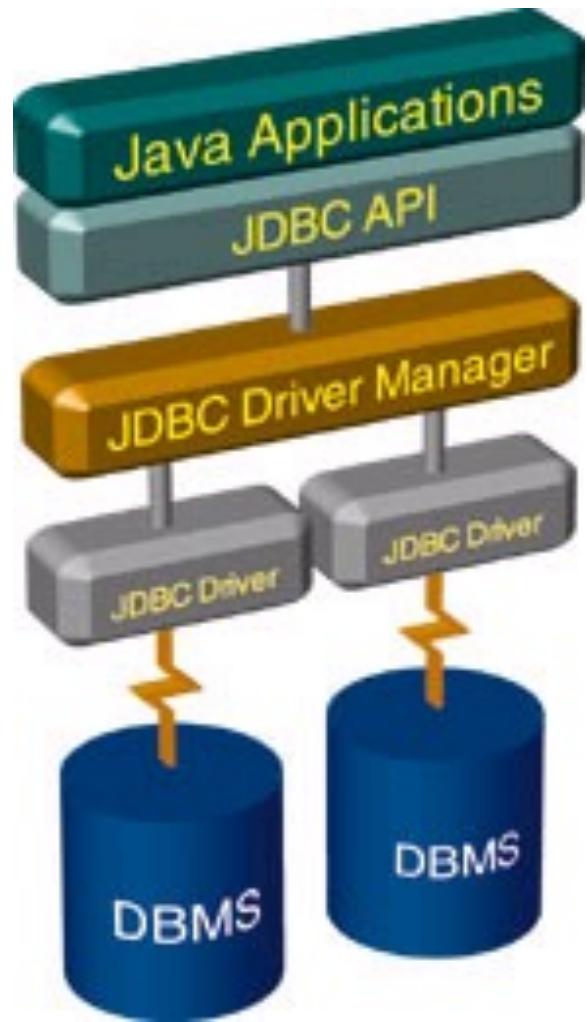
# COSA FA JDBC

- JDBC permette ad oggetti JAVA di dialogare con database relazionali.
- Effettua tre operazioni fondamentali
  - Stabilisce una connessione con il database
  - Invia statements SQL
  - Processa i risultati

# PUNTI CHIAVE DI JDBC

- Non necessita di Driver preinstallati (a differenza di ODBC)
- Utilizza la sintassi degli URL per la localizzazione di risorse
- Una classe speciale (il DriverManager) è responsabile di attivare il driver giusto per la connessione ad un RDBMS
- Mappa i tipi SQL in tipi JAVA

# ARCHITETTURA DI JDBC



# JDBC DRIVER

# DRIVER TIPO I

Wrapper da JDBC ad ODBC

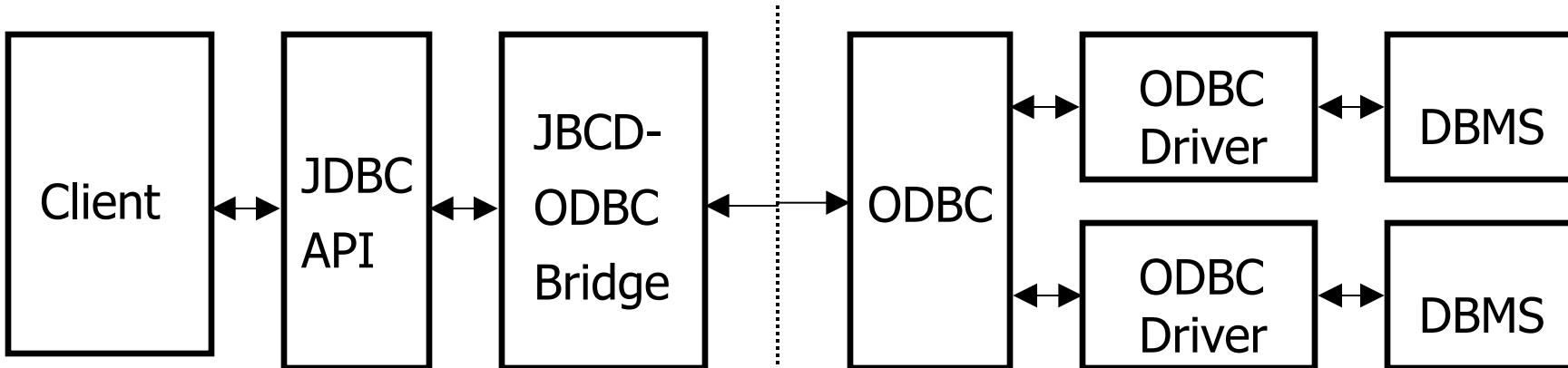
Traduce le JDBC calls in ODBC calls

## ◆ Vantaggi

- E' incluso nel JDK (gratis)
- Funziona con qualsiasi DBMS ODBC
- Facile da configurare

## ◆ Svantaggi

- Molto lento
- Utilizzabile solo per sviluppo codice



# DRIVER TIPO II

Wrapper da JDBC a driver nativo DBMS

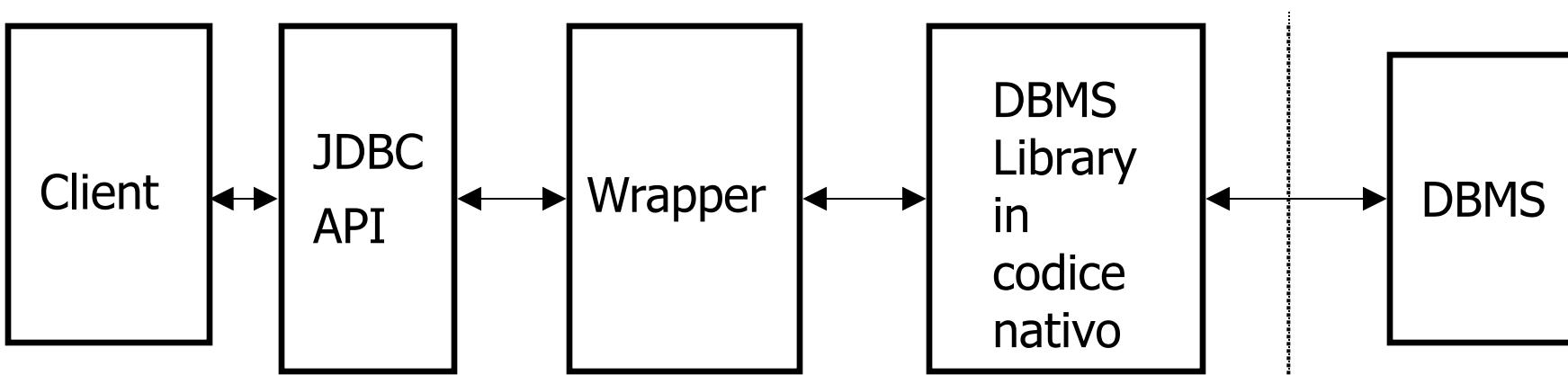
Traduce le JDBC calls in chiamate JNI al codice C/C++ del DBMS  
Driver

## ◆ Vantaggi

- E' di facile realizzazione

## ◆ Svantaggi

- Un bug nel driver può portare a crash del sistema
- Difficile da configurare
- N.a. al Web



# DRIVER TIPO III

## Driver Pure Java per DB MiddleWare

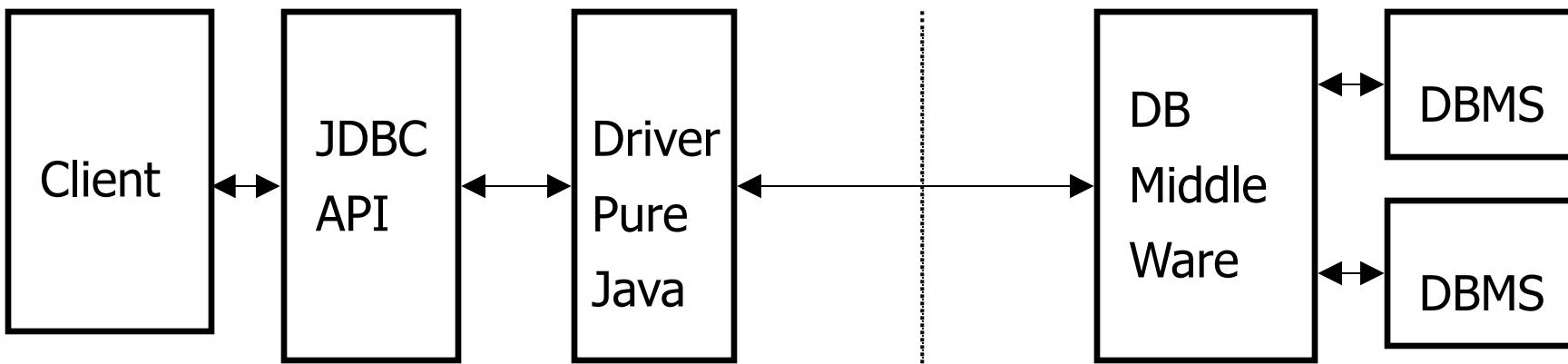
Traduce le JDBC calls in un net-protocol indipendente dal DBMS,  
convertito poi nel DBMS-protocol dal server

### ◆ Vantaggi

- Pure Java
- 1 driver client per N DBMS  
lato server

### ◆ Svantaggi

- Difficile da configurare
- I DBMS vendors non  
realizzano questo tipo di  
driver



# DRIVER TIPO IV

## Driver Pure Java per DBMS

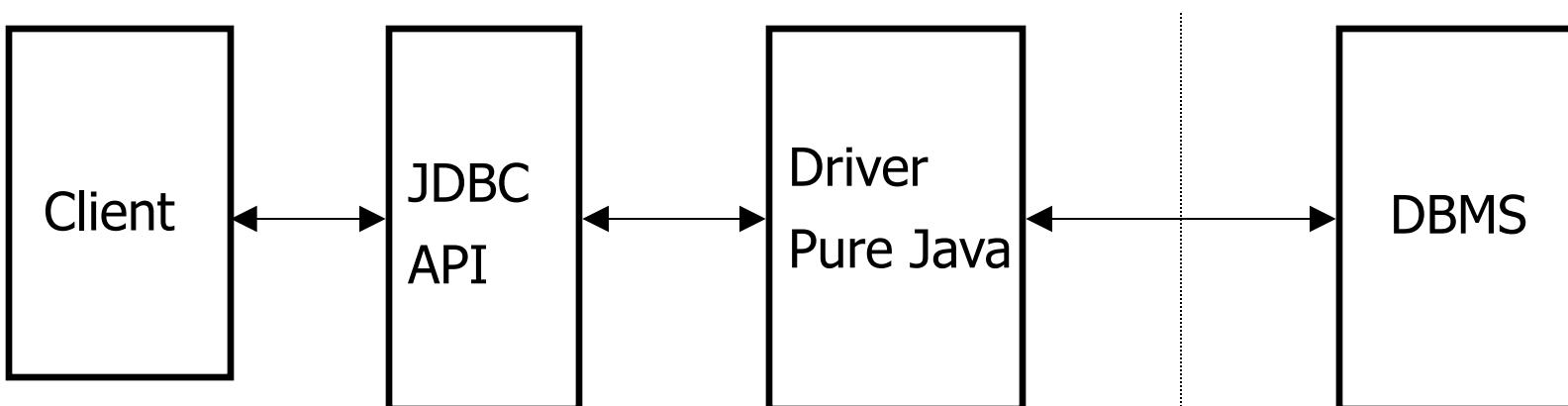
Traduce le JDBC calls in un net-protocol comprensibile dal DBMS

### ◆ Vantaggi

- Massima velocità
- Facili da configurare
- Pure Java

### ◆ Svantaggi

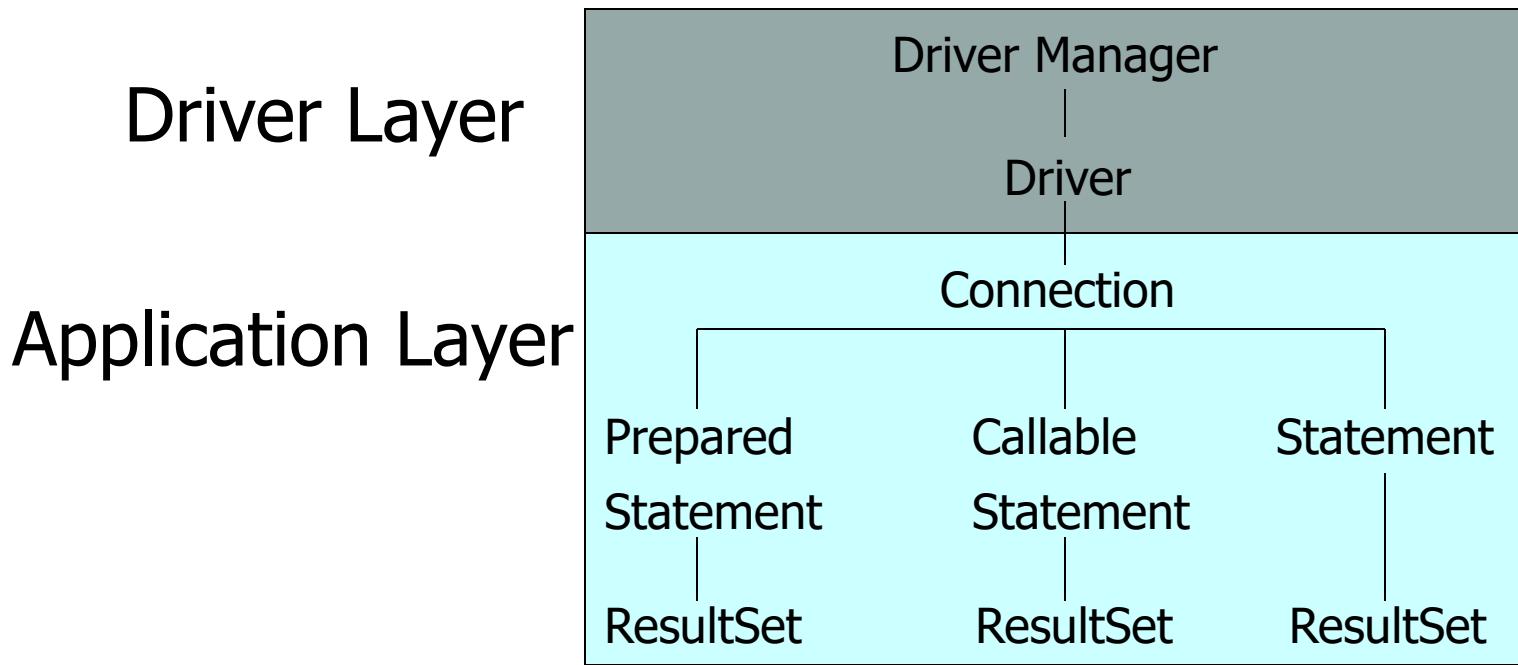
- Non supportato da tutti i DBMS
- Richiede 1 Driver per ogni DBMS, lato client



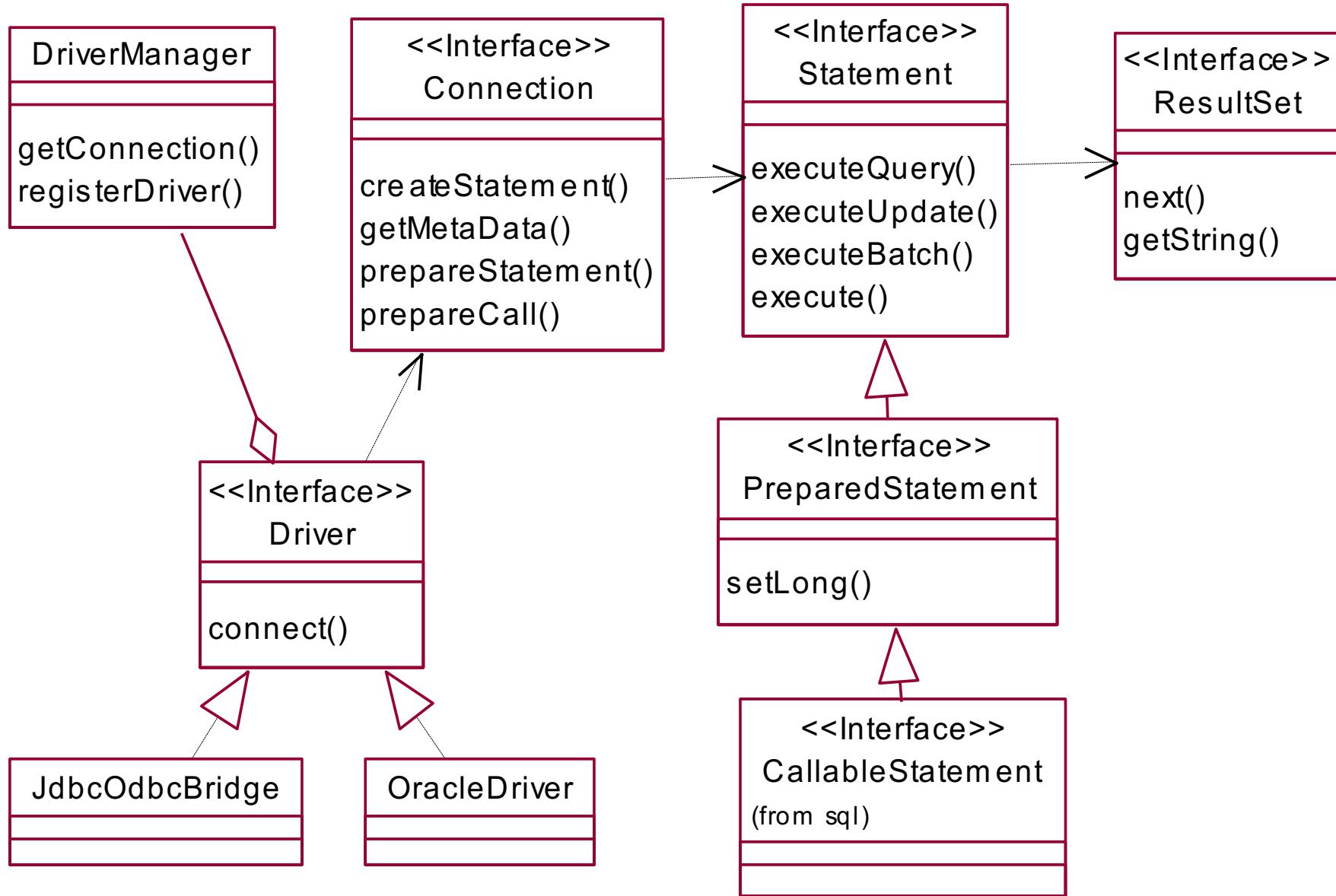
# JDBC API

# STRUTTURA DI JDBC

- JDBC è costituito da 2 'layer' principali:
  - JDBC driver (verso il DBMS)
  - JDBC API (verso l'applicativo)



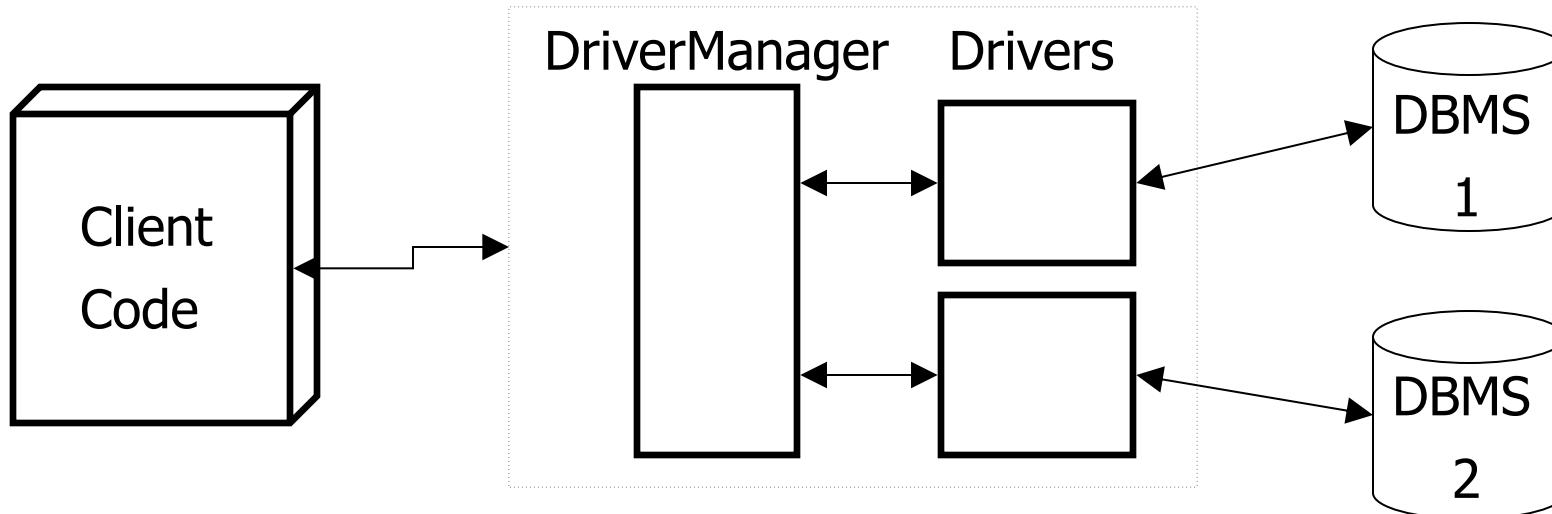
# CLASS DIAGRAM IN UML



# PUNTUALIZZAZIONE

- Tutte le classi fondamentali di JDBC sono interfacce
  - Non sono implementate in JDBC
  - Sono implementate dallo specifico driver

Esempio di Interface-Implementation pattern nell'O-O design.



# JDBC API

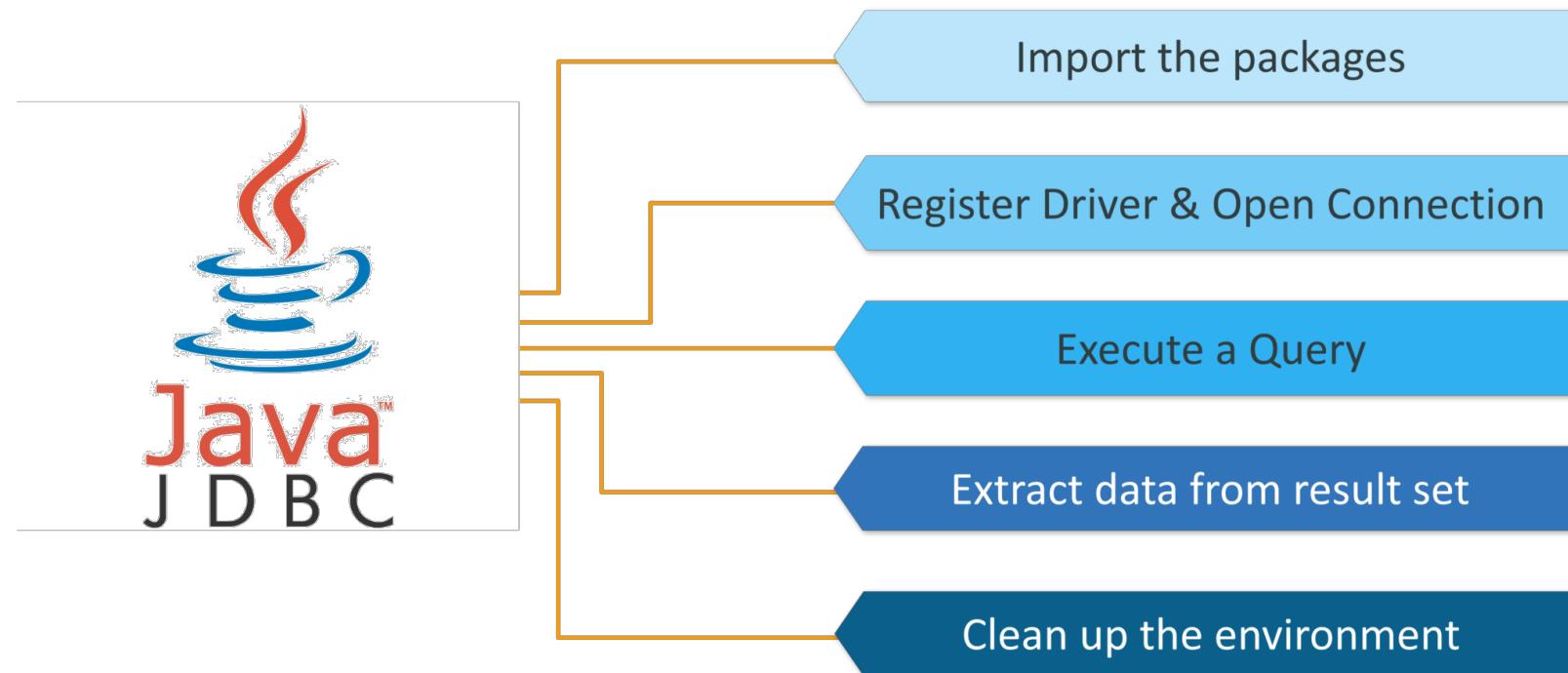
- Data Types
- API di base
  - 1. Caricare un driver
  - 2. Ottenere una connessione
  - 3. Eseguire uno Statement
  - 4. Gestire i risultati
  - 5. Rilasciare le risorse
- API avanzate
  - Prepared e Callable Statements
  - Eccezioni e Warning
  - Transazioni e Isolamento

# DATA TYPES

JDBC definisce un insieme di tipi SQL nella classe `java.sql.Types`

JDBC Types Mapped to Java Types	
JDBC Type	Java Type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	<code>java.math.BigDecimal</code>
DECIMAL	<code>java.math.BigDecimal</code>
BIT	<code>boolean</code>
TINYINT	<code>byte</code>
SMALLINT	<code>short</code>
INTEGER	<code>int</code>
BIGINT	<code>long</code>
REAL	<code>float</code>
FLOAT	<code>double</code>
DOUBLE	<code>double</code>
BINARY	<code>byte[]</code>
VARBINARY	<code>byte[]</code>
LONGVARBINARY	<code>byte[]</code>
DATE	<code>java.sql.Date</code>
TIME	<code>java.sql.Time</code>
TIMESTAMP	<code>java.sql.Timestamp</code>

# BASIC JDBC



# CONNESSIONE.JAVA (POSTGRES)

```
import java.sql.*;  
  
class Connessione {  
    public static void main(String args[]) throws Exception {  
        try {  
            Class.forName("org.postgres.Driver");  
  
            String url = "jdbc:postgresql://localhost:5432/nometabella";  
            Connection conn = DriverManager.getConnection(url, "nomeutente", "password");  
            System.out.println("Connessione OK \n");  
            conn.close();  
        }  
        catch (ClassNotFoundException e) {  
            System.out.println("DB driver not found \n");  
            System.out.println(e);  
        }  
        catch(SqlException e) {  
            System.out.println("Connessione Fallita \n");  
            System.out.println(e);  
        }  
    }  
}
```



# CONNESSIONE.JAVA (MYSQL)

```
import java.sql.*;  
  
class Connessione {  
    public static void main(String args[]) throws Exception {  
        try {  
            Class.forName("com.mysql.cj.jdbc.Driver");  
  
            String url = "jdbc:mysql://localhost:3306/nometabella";  
            Connection con = DriverManager.getConnection(url, »nomeutente», »password»);  
            System.out.println("Connessione OK \n");  
            con.close();  
        }  
        catch (ClassNotFoundException e) {  
            System.out.println("DB driver not found \n");  
            System.out.println(e);  
        }  
        catch(SqlException e) {  
            System.out.println("Connessione Fallita \n");  
            System.out.println(e);  
        }  
    }  
}
```



# PREPARAZIONE

- Importare le classi:

```
import java.sql.*;
```

- Usare i blocchi **try ... catch**:

- Il primo blocco **try** contiene il metodo **Class.forName**, dal package **java.Lang**. Questo metodo lancia un **ClassNotFoundException**, in maniera tale da permettere al blocco **catch** di gestire subito l'eccezione.
  - Può essere gestita una sola volta nel costruttore.
- Il secondo blocco **try** contiene i metodi JDBC, che lanciano tutti un'eccezione del tipo **SQLException**, così il blocco **catch** può gestire solamente oggetti di quel tipo.



# STEP 1 – CARICARE IL DRIVER

```
import java.sql.*;  
.  
.  
try  
{  
    Class.forName(NOME_DRIVER) ;  
}  
catch(ClassNotFoundException)  
{// Driver non trovato !  
}
```

# CARICAMENTO DEI DRIVER

```
// per un database MySQL
String driver = "com.mysql.cj.jdbc.Driver";

// se il database è Oracle
... driver = "oracle.jdbc.OracleDriver";

// se il driver è JDBC-ODBC (tipo 1)
... driver = "sun.jdbc.odbc.JdbcOdbcDriver";

// se il database è PostgreSQL
... driver = »org.postgresql.Driver»;

// il metodo forName forza il caricamento del driver
Class.forName(driver); // lancia una ClassNotFoundException
```



# IL DRIVER MANAGER

- Il *DriverManager* mantiene una lista di classi *Driver* che registrano se stessi invocando il metodo *registerDriver()*.

# CARICARE IL DRIVER

- L'inizializzatore statico del driver, chiamato dalla JVM al caricamento della classe, si registra nel Driver Manager

```
public class MyDriver
{
    static
    {
        new MyDriver();
    }
    public MyDriver()
    {
        java.sql.DriverManager.register(this);
    }
}
```

# STEP 2 - OTTENERE UNA CONNESSIONE

- L'oggetto Connection rappresenta un canale di comunicazione con un DB.
- Una sessione di connessione include istruzioni SQL che vengono eseguite e processate ritornando i valori attraverso la connessione.

# OTTENERE UNA CONNESSIONE

```
Connection con =  
    DriverManager.getConnection(  
        URL_MY_DATABASE);
```

- E' richiesta una connessione al Driver Manager, senza specificare quale particolare Driver debba istanziarla
  - E' l'unico modo per ottenere una completa indipendenza dal DBMS!

# GLI URL IN JDBC

- E' una stringa formata da nodi, separati da ':' o '/'
- Formato:
  - protocollo:sottoprotocollo:databasename
- Il parametro "Databasename" non ha una particolare sintassi, ma è interpretato dal driver
- Esempi
  - jdbc:odbc:myDB
  - jdbc:oracle:@mywebsite:1521:myDB
  - jdbc:cloudscape:myDB
  - jdbc:cloudscape:rmi:myDB;create=true

# QUALE DRIVER CREA LA CONNESSIONE?

- L'URL specifica un sottoprotocollo ed il data source-database system
  - Es. jdbc:odbc:MyDataSource
- Il Driver Manager trova il driver appropriato chiamando il metodo acceptURL (URL) di ogni driver caricato:
- Il primo driver che risponde true, stabilisce una connessione.
- E' possibile in questo modo scegliere il driver JDBC appropriato a run-time!

# L'INTERFACCIA DRIVER

<<Interface>>

Driver

```
connect(url : String, info : java.util.Properties) : Connection  
acceptsURL(url : String) : boolean  
get PropertyInfo(url : String, info : java.util.Properties) : DriverPropertyInfo[]  
getMajorVersion() : int  
getMinorVersion() : int  
jdbcCompliant() : boolean
```

# LA CLASSE DRIVERMANAGER

## DriverManager

```
getConnection(url : String, info : java.util.Properties) : Connection  
getConnection(url : String, user : String, password : String) : Connection  
getConnection(url : String) : Connection  
getDriver(url : String) : Driver  
registerDriver(driver : java.sql.Driver) : void  
getDrivers() : java.util.Enumeration
```

# L'INTERFACCIA CONNECTION

<<Interface>>

Connection

createStatement() : Statement

getMetaData() : DatabaseMetaData

prepareStatement(sql : String) : PreparedStatement

prepareCall(sql : String) : CallableStatement

# STEP 3 - ESEGUIRE UNO STATEMENT

- Lo Statement è l'oggetto che ‘trasporta’ le istruzioni SQL sulla connection, verso il DB.
- È unico per tutta la durata della connessione.
  - Per fare una nuova query, semplicemente si cambia la stringa SQL al suo interno

# ESEGUIRE UNO STATEMENT

```
try
{
    Statement st = con.createStatement();
    ResultSet rs = st.executeQuery("SELECT nome FROM studenti");
}
catch (SQLException sqe)
{
    // Problema
}
```

- Lo scopo principale della classe Statement è eseguire istruzioni SQL.

# ESEGUIRE UNO STATEMENT

- Si utilizza `executeQuery()` per gli statement che restituiscono tuple
  - Comandi DQL
  - Restituisce un `ResultSet`
- Si utilizza `executeUpdate()` per gli statement che non restituiscono tuple
  - Comandi DDL/DML
  - Restituisce il numero di righe modificate
- JDBC 2.0 ha introdotto `executeBatch()` per eseguire più statement in sequenza (per motivi di efficienza)

# ESEGUIRE UNO STATEMENT (2)

- Se non si conosce il tipo di query (es. L'utente la inserisce a run-time), si usa la execute()
  - Restituisce true se è disponibile un ResultSet
  - Si deve chiamare la getResult() per recuperare le informazioni
- Ad ogni Statement è associato un unico ResultSet

# L'INTERFACCIA STATEMENT

<<Interface>>

Statement

```
executeQuery(sql : String) : ResultSet  
executeUpdate(sql : String) : int  
executeBatch()  
execute(sql : String) : boolean  
getWarnings() : SQLWarning  
getResultSet() : ResultSet  
getUpdateCount() : int  
getMoreResults() : boolean
```

# **PREPARED STATEMENT**

- E' un SQL Statement precompilato (il DBMS la salva nella propria cache)
- Utilizzabile con query molto simili nella struttura, ma che cambiano spesso parametri
- Migliora le prestazioni se la query è eseguita molte volte

```
PreparedStatement updateEsami =  
    con.prepareStatement("UPDATE Studenti SET esami  
    = ? WHERE Matricola LIKE ?");  
updateEsami.setInt(1, 13);  
updateEsami.setString(2, "011/245389");  
updateEsami.executeUpdate();
```

# CALLABLE STATEMENT

- E' la classe JDBC per supportare le stored procedures
  - Utilizzate per encapsulare un insieme di operazioni o query da eseguire sul database server
- L'utilizzo è richiesto solo per le stored procedure che restituiscono dei valori.

# STEP 4 – GESTIRE I RISULTATI

- In genere l'esecuzione di una query porta alla restituzione di dati da parte del DBMS
- Un ResultSet contiene tutte le tuple che soddisfano la condizione nell'istruzione SQL inviata usando lo Statement
- I dati sono gestiti in un'apposita struttura dati
- Fornisce dei metodi per accedere ai dati che contiene.

# GESTIRE I RISULTATI

```
while(rs.next())  
{  
    System.out.println("Nome: " +  
rs.getString("nome"));  
  
    System.out.println("Esami: " + rs.getInt("esami"));  
}
```

- I risultati di una query sono salvati in un oggetto RecordSet

# RESULTSET

- Ha un ‘cursore’ che punta al record corrente
- Il cursore del ResulSet è posizionato prima della riga iniziale, dopo l’esecuzione di un metodo executeXXX()
- Dispone di molti metodi di navigazione (metodi per modificare la posizione del cursore)

# RESULTSET (2)

- Permette di recuperare i valori tramite i metodi `getXXXX(colonna)`
- La colonna può essere specificata o con il nome o con il numero:
  - Il nome delle colonne è case-insensitive
  - La numerazione delle colonne parte da 1

```
rs.getString("nome");
```

```
rs.getString(1);
```

# NAVIGAZIONE NEL RESULTSET

- Operazioni disponibili sul ResultSet
  - `first()`, `last()`, `next()`
  - `previous()`, `beforeFirst()`,  
`afterLast()`
  - `absolute(int)`, `relative(int)`
    - JDBC 1.0 permetteva una navigazione solo in avanti

# DATI NEL RESULTSET

- Il driver JDBC converte il tipo di dato del DB in quello richiesto con il metodo `getXXX()`.
  - Ad esempio se il tipo di dato nel DB è di tipo VARCHAR e la richiesta è di tipo String, JDBC effettua la conversione.
- Per determinare se un valore restituito è di tipo JDBC NULL, bisogna prima leggere la colonna e poi usare il metodo `ResultSet.wasNull` per scoprire se il valore ritornato è NULL.
  - Il metodo restituisce un valore che indica “null” a seconda del tipo di dato analizzato. Abbiamo :
    - Il valore null di Java per `getString`, `getDate`, `getTime` ed altri;
    - Il valore 0 (zero) per `getByte`, `getInt` ed altri numerici;
    - Il valore false per il tipo booleano.

# MODIFICHE AL RESULTSET

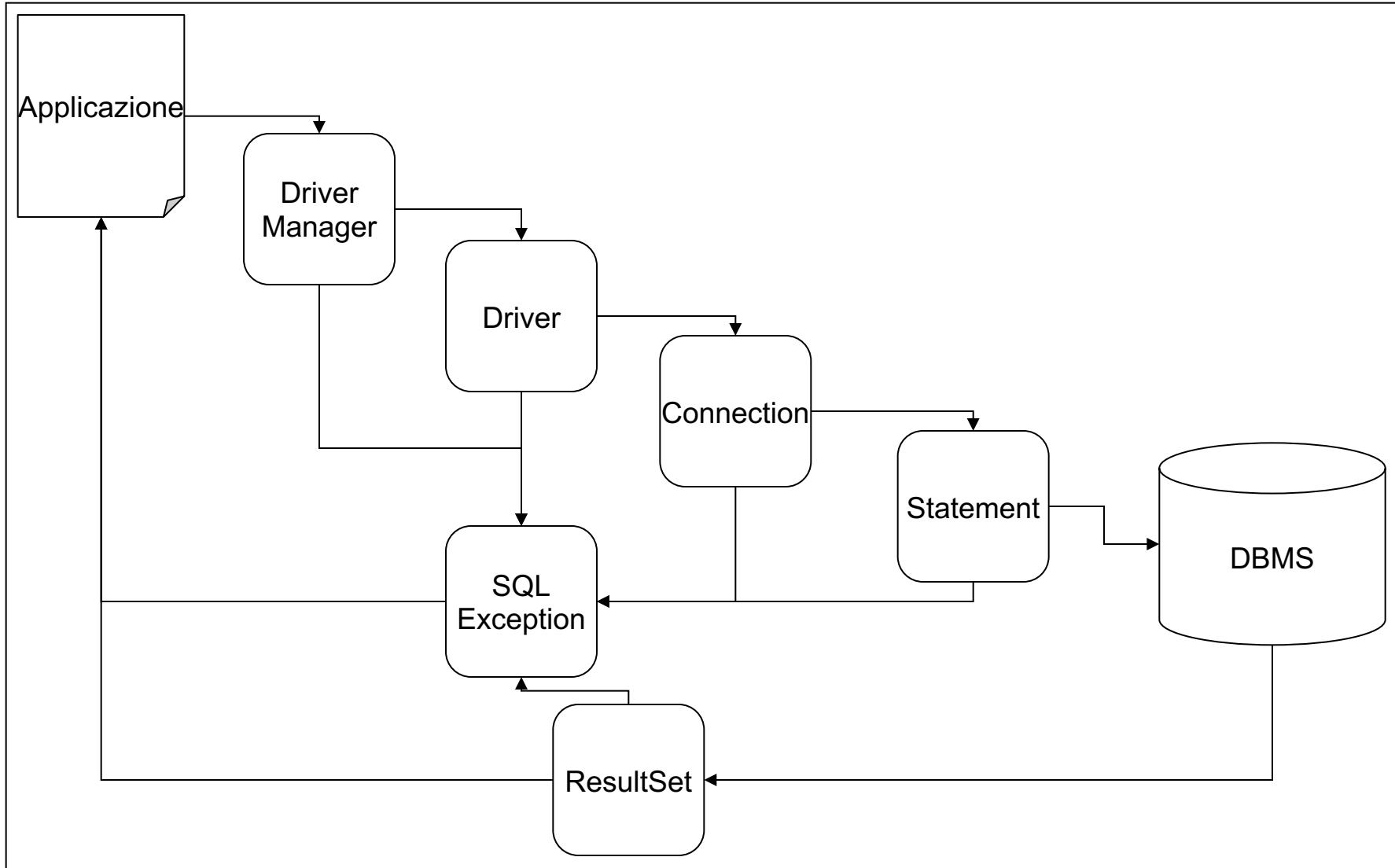
- Si possono aggiungere o modificare le righe di un ResultSet:
  - `rs.update( 3, "new value");`
  - `rs.updateRow();`
- Si possono eliminare delle righe

# STEP 5 – RILASCIARE LE RISORSE

```
rs.close();  
st.close();  
con.close();
```

# ESEMPIO COMPLETO

```
import java.sql.*;  
  
...  
  
try  
    Class.forName(NOME_DRIVER);  
catch(ClassNotFoundException){/* Driver non trovato ! */}  
  
try  
{  
    Connection con = DriverManager.getConnection(URL_MY_DATABASE);  
    Statement st = con.createStatement();  
    ResultSet rs = st.executeQuery("SELECT nome FROM studenti");  
    while(rs.next())  
        System.out.println("Nome: " + rs.getString("nome"));  
  
    catch(SQLException sqe){/* Problema */}  
  
    rs.close();  
    st.close();  
    con.close();  
    ...
```



# JDBC API AVANZATE

# PROGRAMMI DINAMICI

- Non tutti i programmi sono a conoscenza della struttura del DB su cui operano.
- I ‘table viewer’, ad esempio, scoprono a run-time lo schema del database.
- Sono necessarie delle classi per accedere alla struttura del DB:
  - La classe `DatabaseMetaData`, istanziata da `Connection`, restituisce informazioni generiche sul database.
  - La classe `ResultSetMetaData`, istanziata da `ResultSet`, restituisce le informazioni sulla struttura dello specifico `ResultSet`

# DATABASEMETADATA

- L'interfaccia DatabaseMetaData è utilizzata per reperire informazioni sulle sorgenti di dati.
- L'interfaccia mette a disposizione circa 150 metodi classificati nelle seguenti categorie :
  - informazioni generali circa la sorgente dati (Es. Nome del DBMS, ver. del Driver...);
  - aspetti e capacità supportate dalla sorgente dati;
  - limiti della sorgente dati;
  - cosa gli oggetti SQL contengono e gli attributi di questi oggetti (Es. Tutte le tabelle del DB);
  - transazioni offerte dalla sorgente dati.
- DatabaseMetaData md = con.getMetaData();

# RESULTSETMETADATA

- Tra le informazioni troviamo:
  - Numero di colonne (getColumnCount())
  - Nome di una colonna (getColumnName())
  - Tipo di una colonna (getColumnTypeName())
- `ResultSetMetaData md = rs.getMetaData();`

# LE TRANSAZIONI

- Sono insiemi di Statement raggruppati in un'unità logica inscindibile
- Ogni statement deve andare a buon fine (commit), altrimenti l'intera transazione viene annullata (roll back)
- Passi
  - Inizia transazione
  - esegui statements
  - commit o rollback della transazione

# API JDBC PER LE TRANSAZIONI

- Di default ogni singola operazione è una transazione
- I metadati del database specificano il livello di isolamento supportato dal DBMS
- Per eseguire più statement in un'unica transazione si usa:

```
con.setAutoCommit(false);
// esegui gli statements
con.commit();
```

# LIVELLI DI ISOLAMENTO

- Servono per gestire transazioni concorrenti
- Dipendono dall'oggetto Connection
  - `con.setTransactionIsolation(...)`
    - TRANSACTION\_NONE
      - **Transazioni disabilitate o non supportate dal DBMS**
    - TRANSACTION\_READ\_UNCOMMITTED
      - Le altre transazioni possono vedere i risultati di una transazione non completata
      - Se una transazione fa il roll back, le altre applicazioni possono restare con dati non validi
      - I dati sono sempre disponibili per la lettura

# LIVELLI DI ISOLAMENTO (2)

- TRANSACTION \_ READ \_ COMMITTED
  - Se un'altra transazione sta scrivendo dati, il reader resta in attesa
  - Le righe lette dal reader non sono bloccate
- TRANSACTION \_ REPEATABLE \_ READ
  - Se un'applicazione effettua una sequenza di read, otterrà gli stessi risultati
  - Se un'altra transazione sta scrivendo dati, il reader resta bloccato in attesa
  - Le righe lette dal reader sono bloccate finché non effettua una commit()
- TRANSACTION \_ SERIALIZABLE
  - L'intera tabella è bloccata durante una lettura

# ECCEZIONI E WARNING

# PROBLEMI CON I DB

- Sebbene Java sia fortemente tipato, non è possibile effettuare, durante la compilazione, controlli sul corretto utilizzo dei valori provenienti da un DB
- Quasi tutte le istruzioni viste in precedenza possono portare a potenziali problemi (Es. Perdita connessione al db).
  - Alcuni problemi sono ‘recuperabili’, altri no.

# ECCEZIONI E WARNING

- In JDBC, oltre alle classiche eccezioni Java, esistono anche i warning
  - Un'eccezione causa una brutale terminazione del metodo in corso da parte della JVM
  - Un warning viene concatenato all'oggetto che lo ha generato, permettendo la prosecuzione del metodo corrente
- E' a discrezione dell'implementatore del driver stabilire quale problema generi un'eccezione e quale un warning

# SQL EXCEPTION

- E' una sottoclasse di Java.lang.Exception
- Aggiunge informazioni sul tipo di errore proveniente dal database
- Più eccezioni possono essere concatenate

# SQLEXCEPTION

SQLException

vendorCode : int

SQLException(reason : String, SQLState : String, vendorCode : int)

SQLException(reason : String, SQLState : String)

SQLException(reason : String)

SQLException()

getSQLState() : String

getErrorCode() : int

getNextException() : SQLException

setNextException(ex : SQLException) : void

# SQL WARNING

- E' una sottoclasse di SQLException
- Non richiede il catch
- Utilizzato quando il problema non è tanto grave da richiedere un'eccezione
- Esaminabile con il metodo getWarnings() di Connection, Statement, ResultSet.

# SQL WARNING

## SQLWarning

```
SQLWarning(reason : String, SQLstate : String, vendorCode : int)
SQLWaming(reason : String, SQLstate : String)
SQLWaming(reason : String)
SQLWarning()
getNextWaming() : SQLWarning
setNextWarning(w : SQLWarning) : void
```

# **CONCLUSIONI**

# IL FUTURO

- ORDBMS o ODMG?
- SQL3 (Dicembre '99) aggiunge il supporto agli oggetti nei RDBMS
  - Nuovo Paradigma: Object Relational Database Management System (ORDBMS)
- Object Data Management Group è un consorzio di sviluppatori di DBMS.
  - Il binding ODMG per Java fornisce un supporto nativo alla memorizzazione di oggetti.

# ODMG BINDING

- In JDBC lo sviluppatore ha la responsabilità di mappare gli oggetti Java in tabelle del DB e viceversa.
- I bindings ODMG permettono allo sviluppatore di rendere trasparente la persistenza di oggetti Java.

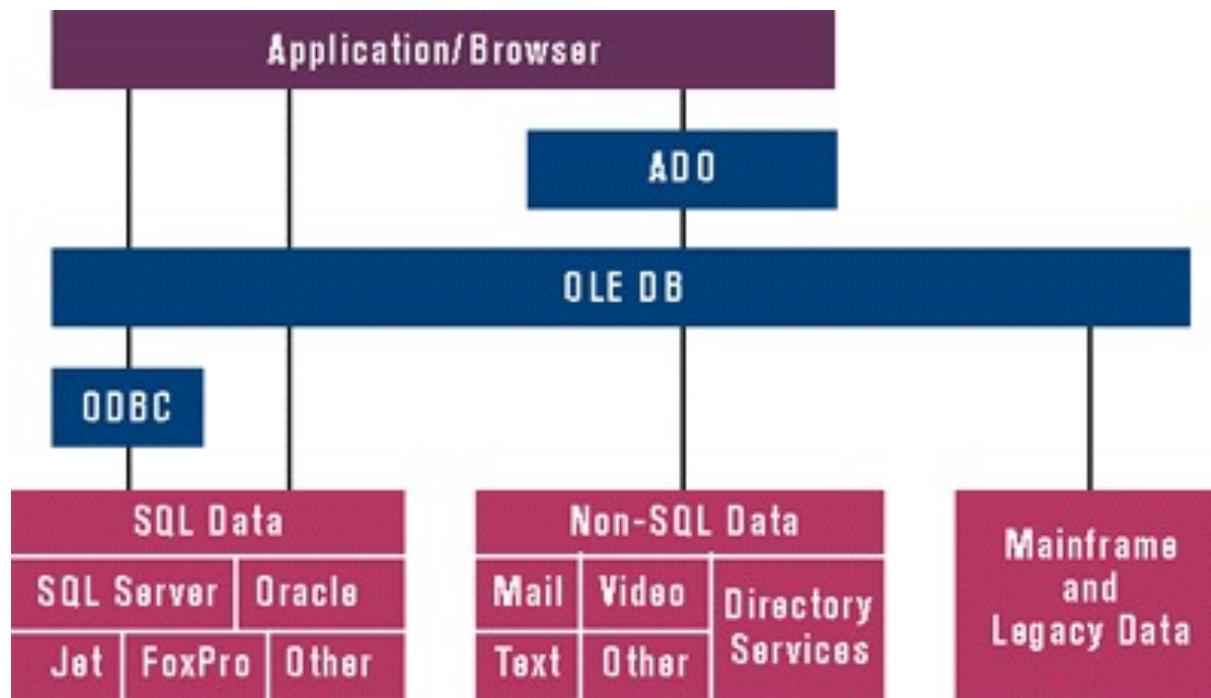
# JAVA BLEND

- Il binding ODMG è implementato nel package “Java Blend”, costruito su JDBC.
- Java Blend crea a run-time oggetti da un database relazionale, che corrispondono ad oggetti Java
  - Le tuple di una tabella diventano oggetti
  - Le chiavi esterne diventano riferimenti...

# IL FUTURO DELLE CLI

- ODBC non può supportare le nuove caratteristiche di SQL3
  - Nessun supporto agli oggetti
- OLE-DB, ADO e JDBC supportano le nuove caratteristiche
  - Sono basati sul modello ad oggetti
- Microsoft punta su UDA, che permette di accedere con la stessa interfaccia a qualsiasi fonte dati (non necessariamente RDBMS)

# ARCHITETTURA UDA



UDA Architecture

# RIFERIMENTI

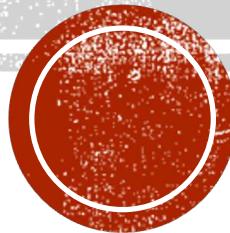
- **JDBC**
  - <http://www.javasoft.com/products/jdbc/index.html>
  - <http://java.sun.com/docs/books/tutorial/jdbc>
  - <http://java.sun.com/products/jdbc/datasheet.html>
- **JDBC e Architetture**
  - <http://www.mokabyte.it>
- **UDA, ADO, OLE-DB**
  - <http://www.microsoft.com/Mind/0498/uda/UDA.htm>
- **DBMS e SQL**
  - Fundamentals of Database Systems
    - Elmasri – Navathe, Addison Wesley

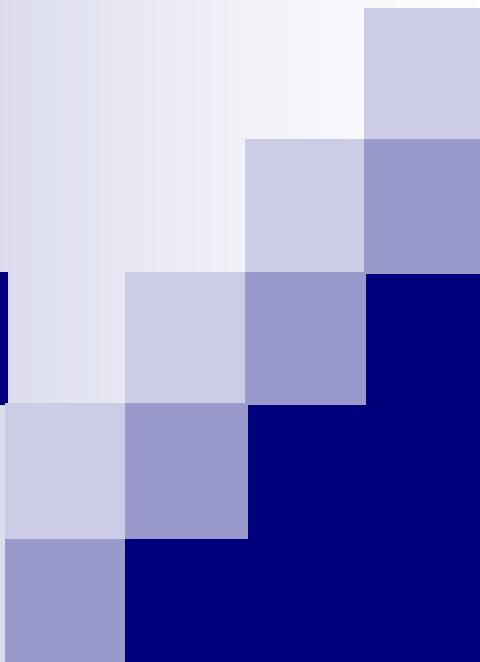


# FINE

Per eventuali domande: (in ordine di preferenza personale)

- Ora.
- Chat di Teams
- Mail: [silvio.barra@unina.it](mailto:silvio.barra@unina.it)





JDBC

L'interfaccia ResultSet

# L'interfaccia ResultSet

- Un ResultSet è un oggetto JDBC che contiene il risultato di una query di selezione
  - Sono le righe e le colonne che soddisfano la condizione dell'istruzione SQL
- Gli oggetti ResultSet sono il punto di connessione tra le tabelle di un database e le applicazioni Java/JDBC
  - Possono rappresentare
    - Una singola tabella (o parte di essa)
    - Più tabelle (risultato di una JOIN)

# L'interfaccia ResultSet

- ResultSet dispone di metodi per accedere alle righe e alle colonne rappresentate nell'oggetto
- In base all'implementazione del driver e alle esigenze delle applicazioni un ResultSet può essere:
  - Dal punto di vista dell'accesso:
    - “forward only” : accesso sequenziale alle righe (l'unico consentito con JDBC 1.0)
    - “scrollabile”: accesso casuale (introdotto con JDBC 2.0)
  - Dal punto di vista della modifica:
    - “solo lettura”
    - “modificabile”

# L'interfaccia ResultSet

- L'interfaccia ResultSet incapsula l'operatore *cursor*
- Il cursor si occupa di gestire le righe provenienti da una query di selezione
- Con i cursori è possibile mantenere allineati i dati tra DBMS e ResultSet
- I driver possono
  - usare una propria implementazione dei cursori a livello client
  - sfruttare quella presente in molti DBMS (cursori lato server)

# Creare un ResultSet

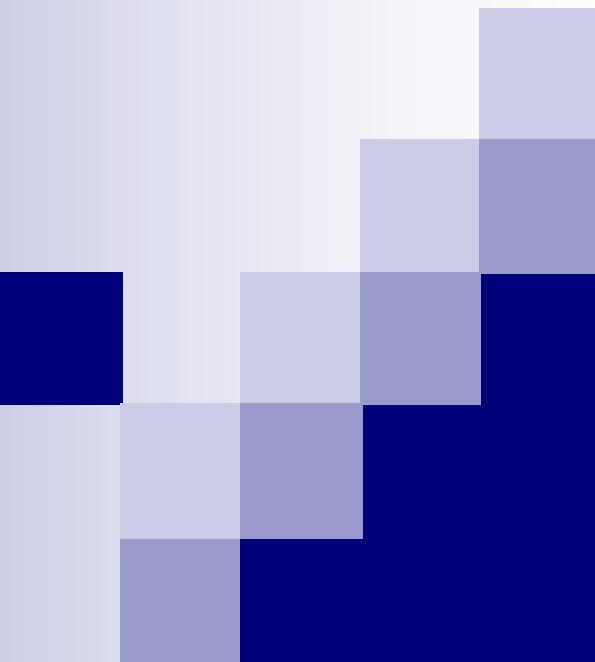
- Gli oggetti *ResultSet* sono costruiti a partire da oggetti *Statement*, *PreparedStatement* e *CallableStatement* che lavorano su query di selezione
- Per default gli oggetti *ResultSet* sono creati di tipo *forward only* e *non modificabili*
  - Una volta visualizzate le righe di un *ResultSet* non è più possibile accedervi a meno di non rieseguire la query e creare un nuovo *ResultSet*
- Esempio: dopo aver creato un oggetto *Statement* è possibile creare un *ResultSet* con il metodo `executeQuery`

# Esempio con Statement

```
Statement st = con.createStatement();  
  
Resultset rs =  
    st.executeQuery("SELECT * FROM articolo " +  
    "WHERE prezzo<50");  
  
while(rs.next()) {  
    System.out.println("Codice = " +  
        rs.getString("codice"));  
    System.out.println("Descrizione = " +  
        rs.getString("descrizione"));  
    System.out.println("Prezzo = " +  
        rs.getString("prezzo"));  
}  
}
```

# Esempio con PreparedStatement

```
PreparedStatement pst =  
    con.prepareStatement("SELECT * FROM articolo " +  
    "WHERE prezzo<50");  
  
Resultset rs = pst.executeQuery();  
  
while(rs.next()) {  
    System.out.println("Codice = " +  
        rs.getString("codice"));  
    System.out.println("Descrizione = " +  
        rs.getString("descrizione"));  
    System.out.println("Prezzo = " +  
        rs.getString("prezzo"));  
}  
/* si noti che viene usato executeQuery senza parametri perchè con  
   PreparedStatement la query è già impostata in fase di costruzione  
   dell'oggetto*/
```



# Navigazione nel RecordSet

# Muoversi tra le righe

- Un cursore può essere visto come un puntatore alla riga corrente nel *ResultSet*
  - Quando il *ResultSet* è creato con `executeQuery`, il cursore sarà prima della prima riga
  - La chiamata al metodo `next()` provoca il posizionamento sulla prima riga (e quindi la possibilità di accedere ai dati)
  - `next()` risponde true finchè il cursore punta ad una riga valida.
  - `next()` restituisce falso non appena il cursore si posiziona dopo l'ultima riga
- Ogni tentativo di accedere agli attributi quando il cursore si trova prima della prima riga o dopo l'ultima riga provocherà una `SQLException`

# RecordSet navigabili

- Con la versione 2.0 di JDBC sono stati introdotti metodi (diversi da *next*) che consentono il movimento tra le righe in tutte le direzioni e in modo casuale
- Se vogliamo *ResultSet* “scrollabili” è necessario che l’oggetto *Statement* di partenza ne sia informato
- Occorre utilizzare un’altra versione di *createStatement* che ha come argomento:
  - il tipo di *ResultSet* e
  - il tipo di concorrenza

```
Statement st = con.createStatement  
        (ResultSet.TYPE_SCROLL_SENSITIVE,  
         ResultSet.CONCUR_UPDATABLE);
```

```
ResultSet rs = st.executeQuery("SELECT * FROM articolo");
```

# Metodi per la navigazione

- *boolean next ()*
  - muove il puntatore alla riga successiva
  - restituisce false se viene raggiunta la fine del *ResultSet* e il cursore punterà dopo l'ultima riga
  
- *boolean previous ()*
  - muove il puntatore alla riga precedente
  - restituisce false se viene raggiunto l'inizio del *ResultSet* e il cursore punterà prima della prima riga

# Metodi per la navigazione

- *boolean first()*
  - muove il puntatore alla prima riga
  - restituisce false se il *ResultSet* non ha nessuna riga
  
- *boolean last()*
  - muove il puntatore all'ultima riga
  - restituisce false se il *ResultSet* non ha nessuna riga

# Metodi per la navigazione

- *boolean absolute(int n)* con  $n \neq 0$ 
  - Se  $n > 0$  il puntatore punta all' $n$ -esima riga del *ResultSet* (a partire dalla prima)
  - Se  $n < 0$  il puntatore si sposterà  $n$  righe indietro a partire dalla fine del *ResultSet*
  - Se si raggiunge l'inizio o la fine del *ResultSet* restituisce false
- *boolean relative(int n)*
  - Se  $n > 0$  il puntatore si sposterà di  $n$  righe avanti nel *ResultSet* a partire dall'attuale
  - Se  $n < 0$  il puntatore si sposterà  $n$  righe indietro a partire dall'attuale
  - $n = 0$  è consentito ma non ha effetto
  - Se si raggiunge l'inizio o la fine del *ResultSet* il metodo restituisce false

# Metodi per la navigazione

- *void afterfirst()*
  - muove il puntatore immediatamente prima della prima riga
  
- *void afterlast()*
  - muove il puntatore immediatamente dopo dell'ultima riga

# Navigazione: Esempio

```
Statement st =
con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_READ_ONLY);

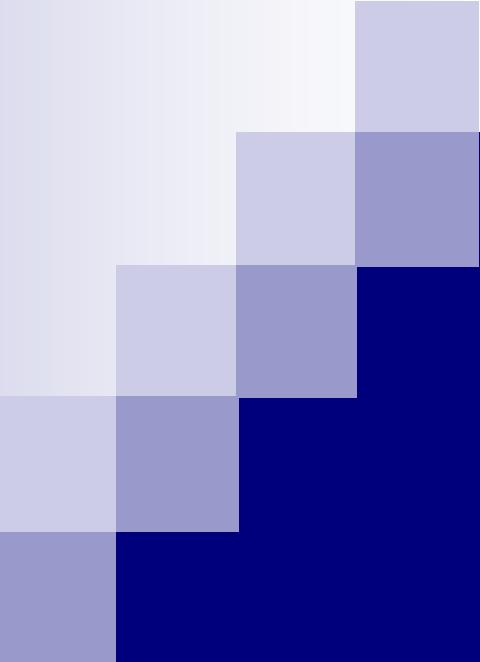
ResultSet rs =
    st.executeQuery("SELECT * FROM articolo WHERE prezzo < 50");

rs.afterLast();
while(rs.previous()) {
    System.out.println("Codice = " + rs.getString("codice"));
    System.out.println("Descrizione = " + rs.getString("descrizione"));
    System.out.println("Prezzo = " + rs.getFloat("prezzo"));
}

rs.absolute(5);
while(rs.next()) {
    System.out.println("Codice = " + rs.getString("codice"));
    System.out.println("Descrizione = " + rs.getString("descrizione"));
    System.out.println("Prezzo = " + rs.getString("prezzo"));
}
```

# Metodi per controllare il cursore

- `boolean isFirst()`
  - Restituisce true se il puntatore è sulla prima riga del *ResultSet*
- `boolean isBeforeFirst()`
  - Restituisce true se il puntatore è immediatamente prima della prima riga del *ResultSet*
- `boolean isLast()`
  - Restituisce true se il puntatore è sull'ultima riga del *ResultSet*
- `boolean isAfterLast()`
  - Restituisce true se il puntatore è immediatamente dopo l'ultima riga del *ResultSet*
- `int getRow()`
  - Restituisce il numero della riga corrente



Accesso ai campi

# Metodi *getXXX*

- I metodi *getXXX* operano sulla riga puntata dal cursore e consentono di leggere i valori associati alle colonne del *ResultSet* (cioè agli attributi)
  - *XXX* rappresenta il tipo Java da usare per gestire il valore
    - `getString`, `getInt`, `getFloat`, ecc...

# Metodi getXXX

- E' possibile accedere ad una colonna specificandone il nome come parametro di `getXXX`
- Esempio:

```
String codice = rs.getString("codice");
```

dove `codice` è una stringa non case-sensitive

# Metodi getXXX

- E' possibile accedere ad una colonna anche specificando la posizione della colonna all'interno del *ResultSet*
- Esempio:  
Se la query è:

```
SELECT codice, genere, prezzo,  
      FROM articolo
```

allora

```
rs.getString(1); ←→ rs.getString("codice");
```

- Se la query utilizza \* (non esplicita i nomi delle colonne), allora la numerazione segue l'ordine presente nella tabella
- Se la query è una JOIN allora conviene usare gli indici invece dei nomi perché ci potrebbero essere colonne con lo stesso nome

# Metodi getXXX

- Sappiamo che i tipi SQL sono mappati in tipi JDBC/SQL ognuno dei quali corrisponde a uno o più tipi Java
- I metodi getXXX restituiscono un tipo o classe Java, mentre i dati delle tabelle hanno una rappresentazione legata al tipo JDBC/SQL
- Esempio: Data l'istruzione:  

```
String codice = rs.getString("codice");
```

l'applicazione cercherà di convertire il valore dell'attributo codice in una String Java

# Metodi getXXX

## ■ Esempio: l'istruzione

```
int short = rs.getShort ("codice");
```

può fallire se *codice* nel database:

- non è di tipo numerico (es: CHAR, o VARCHAR) e il valore non può essere convertito in un numero (contiene caratteri alfanumerici),
  - se di tipo numerico ma con un valore superiore al range consentito per gli short
- 
- E' responsabilità del programmatore assicurarsi che le conversioni non falliscano e che non ci sia perdita di precisione



# Tipi di ResultSet

# Tipi di ResultSet

## A. *ResultSet.TYPE\_FORWARD\_ONLY*

- Non scrollabile: il cursore si può muovere solo in avanti con il metodo *next* dalla prima all'ultima
- Vantaggio:
  - Assicura max portabilità sia tra le diverse versioni sia di JDBC, sia tra i DBMS
- Svantaggio:
  - A. Limitazione delle funzionalità

## B. *ResultSet.TYPE\_SCROLL\_INSENSITIVE*

- Scrollabile (con i metodi *previous*, *first*, *last*, ecc)
- Non si accorge di eventuali modifiche che avvengono alla tabella associata per opera di altri processi

## c. *ResultSet.TYPE\_SCROLL\_SENSITIVE*

- Scollabile (con i metodi *previous*, *first*, *last*, ecc)
- Si accorge di eventuali modifiche alla tabella associata (qualsiasi modifica alle tabelle dopo la creazione del *ResultSet* sarà visibile)

## ■ Warning:

- Non tutti i driver supportano ResultSet di Tipo B e C
- `boolean supportResultSetType(int type)`  
dell'interfaccia *DatabaseMetaData* restituisce vero se il ResultSet del tipo specificato è supportato dal driver corrente

# Tipi di Concorrenza

## A. *ResultSet.CONCUR\_READ\_ONLY*

- E' di sola lettura non può fare modifiche sul database
- Con JDBC 1.0 era il solo tipo di concorrenza consentita
- Vantaggio:
  - Concorrenza illimitata: un lock di sola lettura non impedisce che  $n$  client possono accedere contemporaneamente alla stessa riga della tabella
- Svantaggio:
  - A. Limitazione delle funzionalità

## B. *ResultSet.CONCUR\_UPDATABLE*

- I ResultSet possono essere utilizzati per aggiornare i dati presenti nelle tabelle del DBMS
- Introdotto con JDBC 2.0
- Vantaggio:
  - È possibile direttamente da programma inserire, cancellare aggiornare righe nelle tabelle senza ricorrere a INSERT INTO, DELETE FROM o UPDATE
- Svantaggio:
  - Riduce il livello di concorrenza:
    - Con un lock write-only una riga potrebbe essere inaccessibile ad altri processi

# ResultSet: Esempio

- Con oggetti Statement

```
Statement st = con.createStatement  
        (ResultSet.TYPE_SCROLL_INSENSITIVE,  
         ResultSet.CONCUR_READ_ONLY);  
  
ResultSet rs = st.executeQuery("SELECT * FROM  
articolo");
```

- Con oggetti PreparedStatement

```
PreparedStatement prepareStatement  
        (String sql, int ResultSetType, int ResultSetConcurrency);
```

- Con oggetti CallableStatement

```
CallableStatement prepareCall  
        (String sql, int ResultSetType, int ResultSetConcurrency);
```

# ResultSet: Esempio

- Con oggetti PreparedStatement

```
PreparedStatement pst =  
con.prepareStatement  
(“SELECT * FROM articolo WHERE prezzo < ?”,  
ResultSet.TYPE_SCROLL_SENSITIVE,  
ResultSet.CONCUR_UPDATABLE);  
  
pst.setInt(1, 50);  
ResultSet rs = pst.executeQuery();
```



# ResultSet Aggiornabili

# Metodi *updateXXX*

- Operano sulla riga puntata dal cursore e consentono di inserire un nuovo valore in un attributo
  - Hanno due parametri:
    - il primo specifica l'attributo (il nome della colonna o l'indice)
    - il secondo il nuovo valore

```
void updateXXX(int columnIndex, XXX newValue);  
void updateXXX(String columnName, XXX newValue);
```

- XXX rappresenta il tipo o classe Java
  - Valgono le stesse regole di conversione date per *getXXX*

```
void updateInt(int columnIndex, int newValue);  
void updateInt(String columnName, int newValue);
```

# Metodi *updateXXX*

- Permettono opzionalmente di salvare le modifiche sul db.
- Il cambiamento effettivo nel database avverrà solo dopo aver invocato il metodo *updateRow* *che opera sulla riga corrente!*

# Metodi *updateXXX*: Esempio

```
Statement st = con.createStatement
    (ResultSet.TYPE_SCROLL_INSENSITIVE,
     ResultSet.CONCUR_UPDATABLE);

ResultSet rs =      st.executeQuery
    ("SELECT codice, descrizione, prezzo" +
     "FROM articolo " +
     "WHERE codice = 'AB 0777'");

if (rs.next()) {
    rs.updateFloat("prezzo", (float) 28.79);
    rs.updateRow();
}

/* in alternativa
rs.updateFloat(3, (float) 28.79);*/
```

# Metodo *cancelRowUpdates*: Esempio

```
Statement st = con.createStatement
    (ResultSet.TYPE_SCROLL_INSENSITIVE,
     ResultSet.CONCUR_UPDATABLE);

ResultSet rs =      st.executeQuery
    ("SELECT * FROM articolo " +
     "WHERE codice = 'AB 0777'");

if (rs.next()) {
    rs.updateFloat("prezzo", (float) 28.79);
    rs.updateString("codice", "AB 0989");

    rs.cancelRowUpdates();
}

/* prima di invocare updateRow è possibile riportare il ResultSet ai
   valori precedenti all'aggiornamento */
```

# Cancellazione di righe

- Il metodo *deleteRow* dell'interfaccia *ResultSet* elimina la riga puntata dal cursore
- Ha effetto sia sul *ResultSet*, sia sulla tabella associata nel db

```
rs.absolute(5);
```

*/\*posiziona il cursore sulla quinta riga del ResultSet\*/*

```
rs.deleteRow();
```

# Inserimento di righe

- Concetto di “*insert row*”
  - Riga che presenta le stesse colonne del *ResultSet* ma non ne fa parte
  - È usato come buffer per memorizzare valori di una riga
  - Il metodo *insertRow* renderà effettivo l'inserimento della riga sia nel *ResultSet* sia nel database
- Passi da seguire:
  - 1) Invocare il metodo *moveToInsertRow* per spostare il cursore sulla *insert row*
  - 2) Usare il metodo *updateXXX* per inserire un valore per tutte le colonne presenti nel *ResultSet*
  - 3) Invocare il metodo *insertRow* per rendere effettivo l'inserimento
  - 4) Invocare il metodo *moveToCurrentRow* per muovere il cursore su una riga del *ResultSet*

# Inserimento righe: Esempio

```
Statement st = con.createStatement
    (ResultSet.TYPE_SCROLL_INSENSITIVE,
     ResultSet.CONCUR_UPDATABLE);

ResultSet rs =
    st.executeQuery("SELECT * FROM articolo");

rs.moveToInsertRow();

rs.updateString("codice", "AB 0989");
rs.updateString("descrizione", "TV COLOR SONY 20p");
rs.updateFloat("prezzo", (float) 300);

rs.insertRow();
rs.moveToCurrentRow();
/* sposterà il cursore dalla insert row alla riga puntata prima della chiamata
moveToInsertRow*/
```

# Inserimento righe: Warning

- È possibile usare i metodi `getXXX` quando il cursore punta alla *insert row* ma solo se a quell'attributo è già stato assegnato un valore tramite `updateXXX`, altrimenti porterà un risultato indefinito
- Prima di invocare `insertRow` è necessario che a tutti gli attributi sia stato assegnato un valore, altrimenti si verifica un'eccezione
- L'invocazione di `insertRow` provoca l'aggiunta della riga sia nel *ResultSet* sia nel database

# ResultSet aggiornabili: Considerazioni

- Non tutte le query producono ResultSet aggiornabili anche se specificata la concorrenza **CONCUR\_UPDATABLE**
- Dipende dall'implementazione del DBMS
- Comunque conviene attenersi alle seguenti regole sulla query :
  - deve coinvolgere una sola tabella (il risultato di una JOIN in genere non è aggiornabile)
  - non deve contenere la clausola GROUP BY
  - deve selezionare gli attributi appartenenti alla chiave primaria

# Inserimento Righe

- Affinchè vada a buon fine è necessario che la query:
  - Selezioni tutti gli attributi creati con il vincolo NOT NULL
  - Selezioni tutti gli attributi che non hanno un valore di default

# Altri metodi dell'interfaccia Resultset

- `Statement getStatement () ;`
  - Restituisce l'oggetto `Statement` che ha prodotto la query
- `int getFetchSize () ;`
- `void setFetchSize (int rows) ;`
  - Ogni volta che si muove il cursore non si accede al database: per migliorare le prestazioni saranno lette n righe e messe in una cache. Questi metodi restituiscono e impostano il numero di righe che devono essere lette dal database (meglio usare il valore predefinito per il driver)
- `void close () ;`
  - Consente di rilasciare le risorse occupate dall'oggetto `ResultSet` senza attendere la chiusura automatica (quando l'oggetto `Statement` da cui è stato creato viene chiuso oppure non si fa più riferimento all'oggetto)



# JDBC

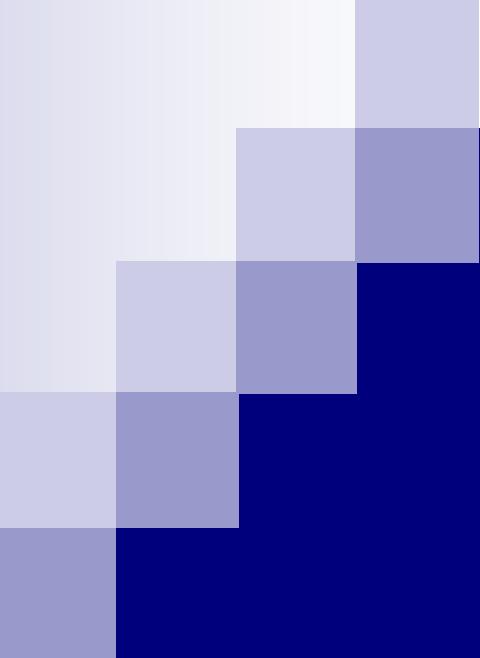
Query semplici  
e  
Query parametriche

# Creare applicazioni indipendenti dal DBMS

- Le API JDBC consentono alle applicazioni Java di inviare query SQL al server di database e di gestirne i risultati
- I DBMS in commercio non sempre rispettano in pieno lo standard ANSI
  - “per difetto”: es. non forniscono Stored Procedure, o OUTER JOIN
  - “per eccesso”: aderiscono allo standard ma estendono SQL con funzionalità proprie

# Creare applicazioni indipendenti dal DBMS

- Ogni istruzione può arrivare al DBMS sottostante senza che JDBC effettui alcun controllo, sarà il DBMS a segnalare situazione di errore
- Affinchè un driver JDBC possa essere definito *JDBC-compliant* è necessario che supporti lo standard ANSI-SQL 2 Entry Level 1992
- Se si utilizzano solo query ANSI-SQL 2, l'applicazione funzionerà con tutti i driver JDBC-compliant e quindi con i più diffusi DBMS



# Statement SQL con JDBC

# Query con JDBC

- Tre diverse classi per inviare le query:

- Statement

- Un oggetto Statement può essere creato con il metodo *createStatement* di Connection
    - E' di solito usato per inviare query semplici che non utilizzano parametri

- PreparedStatement

- Un oggetto PreparedStatement può essere creato con il metodo *prepareStatement* di Connection
    - Estende le potenzialità dell'interfaccia Statement da cui deriva
      - Consente di specificare query parametriche (IN)
      - E' più efficiente: la query sarà pre-compilata per usi futuri

# Query con JDBC (2)

## □ CallableStatement

- Un oggetto CallableStatement può essere creato con il metodo *prepareCall* di Connection
- Viene utilizzato per invocare le Stored Procedure che risiedono sul server di database
- Possono essere utilizzati parametri IN, OUT e INOUT

# Interfaccia Statement

- Stabilità la connessione, si può creare un oggetto Statement:  
*Statement Connection.createStatement();*
- Tre diversi metodi per inviare uno statement SQL:
  - *executeUpdate*
  - *executeQuery*
  - *execute*

# *executeUpdate*

- `int executeUpdate(String sql);`
- Consente di inviare query SQL di tipo DDL e DML
  - CREATE TABLE, DROP TABLE, INSERT, UPDATE; DELETE
- Il nome deriva dal fatto che è utilizzato con query di aggiornamento
  - Restituisce il numero di righe aggiornate
  - Per le istruzioni DDL che non operano su righe (CREATE TABLE) restituisce 0
- Con un insieme di chiamate a *executeUpdate* possiamo costruire lo schema di un DB, tavelle e vincoli d'integrità.
- E' possibile inviare statement per assegnare diritti amministrativi

# Esempio:creazione tabella

```
import java.sql.*;
public class ExecuteQuery {
    //Oggetto Connection
    Connection con = null;
    Statement st = null;
    public static void main(String[] args) {
        ExecuteQuery eq = new ExecuteQuery();
        // Connessione al database
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            String url = "jdbc:odbc:DSNTEST";
            eq.con = DriverManager.getConnection(url);
            eq.st = con.createStatement();
            System.out.println("Connessione OK");
        }
        catch (Exception e) {
            System.out.println("Connessione Fallita");
            e.printStackTrace();
            System.exit(1);
        }
        //Creazione della tabella Articolo
        eq.createTableArticolo();
    }
}
```

# Esempio:creazione tabella

```
private void createTableArticolo() {  
    try {  
  
        String sqlCommand =  
            "Create Table articolo " +  
            "(codice CHAR(15) CONSTRAINT v PRIMARY KEY," +  
            " descrizione CHAR(50), " +  
            " genere      CHAR(20), " +  
            " prezzo      INTEGER)";  
        st.executeUpdate(sqlCommand);  
    }  
    catch (SQLException e) {  
        System.out.println("Non posso creare tab articolo");  
        e.printStackTrace();  
        System.exit(1);  
    }  
}
```

# Esempio:aggiungere righe

```
//esecuzione di più query con lo stesso Statement
private void addArticolo() {
    try {

        String sqlCommand =
            "INSERT INTO articolo VALUES ( " +
            /* Codice */ "'566-78-09'," +
            /* Descrizione*/ "'TV Color Sony 16p'," +
            /* Genere */      "'TV Color'," +
            /* Prezzo */       "400" +
            ")";
        st.executeUpdate(sqlCommand);
        sqlCommand =
            "INSERT INTO articolo VALUES ( " +
            /* Codice */ "'565-79-91'," +
            /* Descrizione*/ "'Eriksson GF 768'," +
            /* Genere */      "'Telefono'," +
            /* Prezzo */       "200" +
            ")";
        st.executeUpdate(sqlCommand);
    }
    catch (SQLException e) {
        System.out.println("Errore nell'inserimento");
        e.printStackTrace();
        System.exit(1);
    }
}
```

# Esempio: modificare righe

```
//aumenta del 20% il prezzo degli articoli con codice che
//inizia per 566
private void updateArticolo() {
    try {

        String sqlCommand =
            "UPDATE articolo " +
            "SET prezzo=prezzo*1.2 " +
            "WHERE codice LIKE '566%';

        int num = st.executeUpdate(sqlCommand);
        System.out.println("Articoli Aggiornati: " + num);
    }
    catch (SQLException e) {
        System.out.println("Non posso aggiornare la
tabella");
        e.printStackTrace();
        System.exit(1);
    }
}
```

# Esempio: eliminare righe

```
//elimina dalla tabella gli articoli di genere 'TV
color'
private void deleteArticolo() {
    try {

        String sqlCommand =
            "DELETE FROM articolo " +
            "WHERE genere = 'TV color'";

        int num = st.executeUpdate(sqlCommand);
        System.out.println("Articoli Eliminati: " + num);
    }

    catch (SQLException e) {
        System.out.println("Non posso eliminare le righe");
        e.printStackTrace();
        System.exit(1);
    }
}
```

# *executeQuery*

- Nell'interfaccia Connection è definito il metodo

```
Resultset executeQuery(String sql);
```

- Consente di inviare query di selezione che restituiscono un insieme di righe
  - SELECT, JOIN
- Vincolo: solo query costanti, non possono essere specificati parametri
- L'istruzione SQL è specificata dall'argomento String sql.
- Il risultato sarà inserito in un oggetto Resultset, e potrà essere gestito nel modo che l'applicazione vorrà: potrà essere visualizzato, inserito in una tabella, inviate via rete ad un altro sistema, ecc

# Esempio: recuperare righe

```
//recupera dalla tabella Articolo tutte le righe il cui
// genere sia 'TV Color' e il prezzo inferiore a 200 euro
private void selectArticolo() {
    try {
        String sqlCommand =
            "SELECT * FROM articolo WHERE" +
            "genere='TV Color' AND " +
            "prezzo<200";

        Resultset rs = st.executeQuery(sqlCommand);

        while(rs.next()) {
            System.out.println(
                rs.getString("codice") + " " +
                rs.getString("descrizione") + " " +
                rs.getString("prezzo"));
        }
    }
    catch (SQLException e) {
        e.printStackTrace();
        System.exit(1);
    }
}
```

# Ulteriori metodi di Statement: Rilasciare le risorse

- `void close();`
  - consente di rilasciare le risorse allocate all'oggetto Statement,
  - quando ci si rende conto che un serve più all'applicazione senza dovere aspettare l'attivazione del Garbage Collector
  - ogni successivo tentativo di utilizzare un oggetto Statement fallirà

# Ulteriori metodi di Statement:

- `void setMaxRows (int max) ;`
  - consente di limitare il numero di righe restituite da una query di selezione
  - Esempio di necessità: seleziona gli abbonati Esposito a Napoli → crash del S.O., applicazione “out of memory”, comunque non utile all’utente
  - Utilizzando questo metodo l’applicazione segnalerà all’utente il caso in cui la query restituisce un numero di righe superiore a quello consentito → l’utente potrà riformulare la query
  - `int getMaxRows () ;`  
Consente di conoscere il valore max impostato da `setMaxRows`, (per default max = 0 → No limite)

# Esempio: limitare le righe

```
/*il metodo prende come parametri un oggetto Connection e una query di
selezione, se la risposta raggiungerà il limite di 30 occorrenze sarà
visualizzato un messaggio */
void executeLimitedQuery(Connection con, String sql) {
    try {
        Statement st = con.createStatement();
        st.setMaxRows(30);

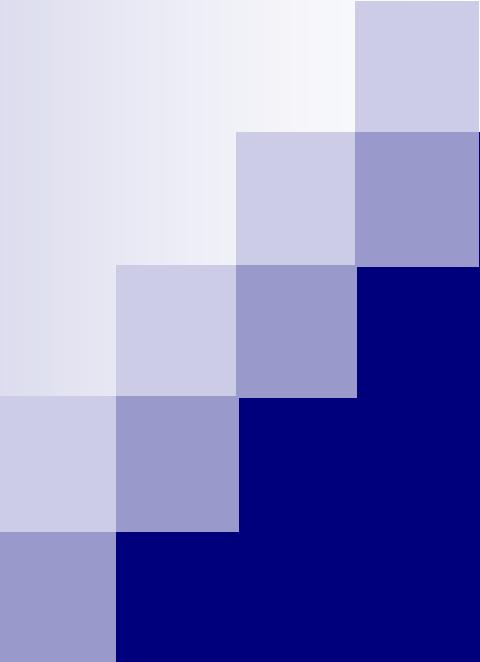
        Resultset rs = st.executeQuery(sql);
        int i=1;

        for(i=1;rs.next(); i++) {
            System.out.println(
                rs.getString("codice") + " " +
                rs.getString("descrizione") + " " +
                rs.getString("prezzo"));

            if (i==30)
                System.out.println("Attenzione Raggiunto il Limite");
        }
    } catch (SQLException e) {
        e.printStackTrace();
        System.exit(1);
    }
}
```

# Ulteriori metodi di Statement:

- `void setQueryTimeout(int seconds);`
  - consente di limitare il periodo di tempo che intercorre tra l'invio della query e la risposta del DBMS
  - Esempio di necessità: evitare lunghe attese quando la rete ha malfunzionamento o la query è troppo complessa
  - `int getQueryTimeout();`  
Consente di conoscere il valore max impostato da `setQueryTimeout`, per default seconds = 0 → non esistono limiti di tempo



# Query Parametriche

# Query parametriche: l'interfaccia PreparedStatement

- L'interfaccia PreparedStatement è lo strumento che consente la gestione di query parametriche
- Rappresenta estensione di Statement → ne eredita tutte le funzionalità.
- Differenze con l'interfaccia Statement:
  - Le istruzioni SQL contenute in un oggetto PreparedStatement possono contenere parametri input, perché l'interfaccia fornisce un insieme di metodi per assegnare valori reali a tali parametri
  - Le istanze di PreparedStatement contengono query SQL precompilate
    - prestazioni più elevate, soprattutto per eseguire la query più volte (Access non supporta query precompilate!)

# Query parametriche

```
SELECT codice, descrizione, prezzo  
FROM articolo  
WHERE genere = ? AND prezzo <?
```

- I valori rappresentati da ? non sono noti ma l'applicazioni li fornirà prima di inviare la query
- Gli oggetti PreparedStatement forniscono metodi per ricevere una query parametrica e sostituire i ? con valori reali
- Ciò non accade con gli oggetti Statement, ma è possibile simulare tale comportamento
  - costruendo run-time il comando SQL quando si hanno a disposizione i dati completi

# Esempio: Query parametriche

- Supponiamo di avere una maschera per la ricerca degli articoli:
  - Utilizzando la Combo-Box possiamo selezionare il genere dell'articolo
  - Utilizzando un Text-Field possiamo inserire il prezzo max
- Nella lista dovrà apparire l'insieme di tutti gli articoli del tipo scelto e con un prezzo minore di quello inserito.
  - Se non è stato inserito alcun prezzo o un valore non consentito, saranno visualizzati tutti gli articoli di quel genere
- Quando l'utente preme Cerca l'applicazione avrà i dati necessari per creare la stringa e inviare la query.

# Esempio: query parametriche

```
private void inviaQuery() {  
  
    // Composizione del comando SQL  
  
    int prezzo = 0;  
    try {  
        prezzo = Integer.parseInt(TxtPrezzo.getText());  
        //TxtPrezzo rappresenta l'istanza del Text-Field  
        // contenente il prezzo  
    }  
    catch (Exception e) {  
        txtPrezzo.setText(" ");  
    }  
    String articolo = cmbArticolo.getSelectedItem();  
    //cmbArticolo rappresenta l'istanza del Combo-Box  
    // relativa al genere  
    String sqlSelect =  
        "SELECT codice, descrizione, prezzo " +  
        "FROM articolo";
```

# Esempio: query parametriche

```
String sqlWhere = "";
if (prezzo >0)
    sqlWhere =
        "WHERE prezzo<" + prezzo +
        " AND genere=''" + articolo +
        "'";
else
    sqlWhere =
        "WHERE genere ='" + articolo + "'",
String sqlCommand = sqlSelect + " " +
sqlWhere;
```

# Esempio: query parametriche

```
// Invio Query
Statement st = null;
try {
    st = con.createStatement();
    Resultset rs = st.executeQuery(sqlCommand);
}

// Visualizzazione dei Risultati

lstArticolo.removeAll();
while(rs.next()) {
    String strRiga =
        rs.getString("codice") + " " +
        rs.getString("descrizione") + " " +
        rs.getInt("prezzo"));

    lstArticolo.add(strRiga);
}

catch (SQLException e) {
    e.printStackTrace();
    Return;
}
}
```

# Interfaccia PreparedStatement

- Riscriviamo l'esempio precedente utilizzando al posto di Statement l'interfaccia PreparedStatement
  - L'applicazione costruisce una query parametrica concatenando il comando SQL con i parametri provenienti dall'interfaccia utente
  - Il codice rimane invariato tranne per il metodo *inviaQuery*

# Query parametriche: l'interfaccia PreparedStatement

- Ogni oggetto di tipo PreparedStatement è associato ad un'unica query parametrica
  - L'associazione avviene nel momento in cui l'oggetto è costruito
  - Poi sarà possibile impostare i parametri ed eseguire la query più volte
- Creare un oggetto PreparedStatement:
  - Si utilizza il metodo prepareStatement dell'interfaccia Connection che prende in input la stringa che rappresenta la query (Selezione o DML)
- Esempio:

```
PreparedStatement pst =  
    con.prepareStatement (  
        "SELECT * " +  
        "FROM articolo " +  
        "WHERE genere =? AND prezzo < ?"  
    );
```

# Passaggio dei parametri

- L'interfaccia *PreparedStatement* mette a disposizione un insieme di metodi *setXXX* per assegnare alla query i valori per gli attributi (sostituendo i vari ? con i valori effettivi)
  - *setXXX* per ogni tipo base di Java, per la classe String e per le date
  - Tutti i metodi *setXXX* hanno in input
    - la posizione ordinale del parametro all'interno del comando SQL e
    - il valore da assegnare

# Esempio: setInt e setString

```
// Preparazione della Query
PreparedStatement st = null;

String querySQL =
    "SELECT codice, descrizione, prezzo " +
    "FROM articolo" +
    "WHERE genere = ? AND prezzo < ? ";

/* assegnazione dei valori ai parametri tramite i metodi setString e
SetInt di PreparedStatement*/
try {
    st = con.prepareStatement(querySQL);
    st.setString(1, genere);
    st.setInt(2, prezzo);
```

# Interfaccia PreparedStatement

- L'esecuzione di una query “preparata” sarà delegata ai metodi della classe `executeXXX`

- È tecnicamente possibile usare i metodi dell'interfaccia Statement:

```
int executeUpdate(String sql);  
ResultSet executeQuery(String sql);  
Boolean execute(String sql);
```

- Non si utilizzano perché non ha senso specificare di nuovo la stringa `sql` visto che l'oggetto è già associato ad una query. I metodi utilizzati sono sempre tre e non vogliono parametri:

```
int executeUpdate(); /*query di aggiornamento*/  
ResultSet executeQuery();/* query di selezione*/  
Boolean execute(); /*per quelle che restituiscono più di un  
ResultSet*/
```

# Esempio: query parametriche

```
private void inviaQuery() {  
  
    /* Trasferimento del calore di genere e prezzo dalla GUI alle  
    variabili locali genere e prezzo */  
  
    int prezzo = 0;  
  
    try {  
        prezzo = Integer.parseInt(TxtPrezzo.getText());  
        /*TxtPrezzo rappresenta l'istanza del Text-Field contenente il prezzo*/  
    }  
    catch (Exception e) {  
        txtPrezzo.setText(" ");  
        prezzo = 999999;  
    }  
    /* se la conversione del prezzo non va a buon fine, prezzo è forzato a un valore molto alto*/  
}  
  
String genere = cmbArticolo.getSelectedItem();  
/*cmbArticolo rappresenta l'istanza del Combo-Box relativa al genere*/
```

# Esempio: query parametriche

```
// Preparazione della Query
PreparedStatement st = null;

String querySQL =
    "SELECT codice, descrizione, prezzo " +
    "FROM articolo" +
    "WHERE genere = ? AND prezzo < ? ";

/* Assegnazione dei valori ai parametri tramite i metodi setString e SetInt di
PreparedStatement*/
try {
    st = con.prepareStatement(querySQL);
    st.setString(1, genere);
    st.setInt(2, prezzo);

/* Invocazione del metodo executeQuery*/
    ResultSet rs = st.executeQuery();
}
```

# Esempio: query parametriche

```
/* Visualizzazione dei Risultati (uguale al caso Statement) */

lstArticolo.removeAll();
while(rs.next()) {
    String strRiga =(
        rs.getString("codice") + " " +
        rs.getString("descrizione") + " " +
        rs.getInt("prezzo"));

    lstArticolo.add(strRiga);
}

catch (SQLException e) {
    e.printStackTrace();
    Return;
}
}
```

# Query precompilate

- Per come è scritto il codice di *inviaQuery* l'applicazione non trae vantaggio dal fatto che la query è precompilata:
  - Il metodo è invocato ad ogni pressione del tasto “Cerca” e ad ogni chiamata sarà costruito un oggetto *PreparedStatement* diverso
  - Ciò è dovuto al fatto che la creazione dell'oggetto *PreparedStatement* e l'esecuzione della query avvengono nello stesso metodo
  - Per ovviare:
    - l'istanza di *PreparedStatement* (*st*) può essere dichiarato come attributo privato della classe e non del metodo → l'oggetto *PreparedStatement* sarà costruito assieme alla classe (nel costruttore) e non ad ogni chiamata di *inviaQuery*

# Query parametriche: l'interfaccia PreparedStatement

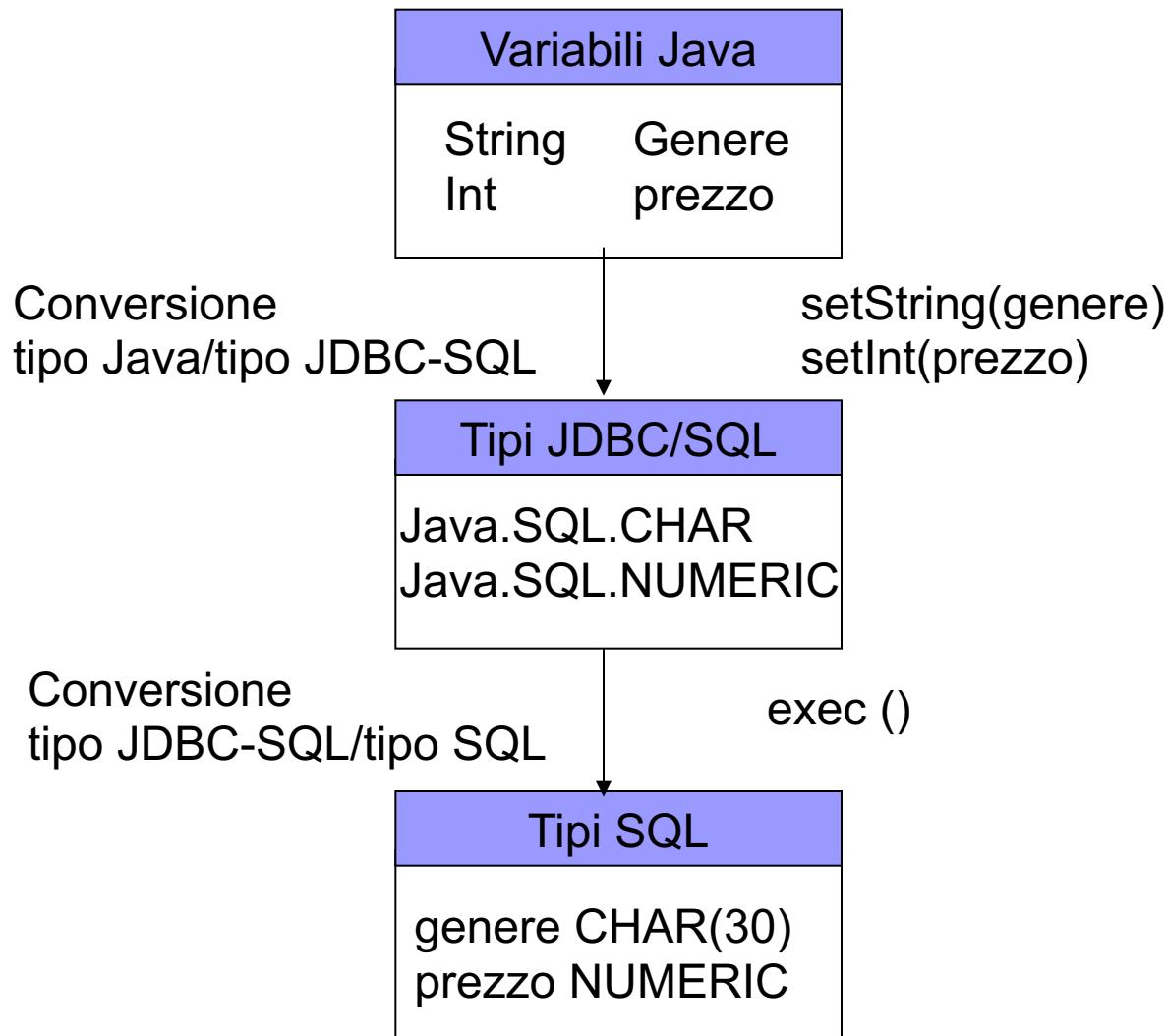
- Esempio Query DML:

```
PreparedStatement pst =  
    con.prepareStatement(  
        "UPDATE articolo " +  
        "SET prezzo = prezzo*1.2 " +  
        "WHERE genere =?") ;
```

# Conversione da tipo Java a tipo SQL

- Una variabile Java passata come argomento a uno dei metodi della famiglia `setXXX` deve subire una serie di operazioni preliminari per poter essere correttamente interpretato da un DBMS:
  - I metodi `setXXX` a partire dalle variabili effettuano una conversione tra il tipo Java e il tipo JDBC/SQL
  - I tipi JDBC/SQL definiti nella classe `java.sql.Types` fanno da ponte verso i tipi SQL veri e propri. Ciò è necessario per garantire l'indipendenza dai tipi SQL definiti dai vari DBMS
  - Il driver JDBC converte i valori JDBC/SQL nei corrispondenti tipi SQL del DBMS usato

# Conversione da tipo Java a tipo SQL



# SetObject

- I metodi *setXXX* di *PreparedStatement* presuppongono che il tipo di un attributo sia noto al momento della compilazione
- In alcune circostanze il tipo è noto solo a tempo di esecuzione
- Come ovviare?
  - Utilizzando il metodo *setObject*

```
void setObject (int parameterIndex, Object x);
```

consente di impostare il parametro x come oggetto generico.  
Successivamente il driver JDBC convertirà x nel tipo che il DBMS si aspetta.  
Si verificherà un *ClassCastException* nel caso x non sia di un tipo compatibile con quello del relativo attributo della tabella

# Esempio: setObject

```
void query(Object prezzo) throws SQLException {  
  
    String sqlCommand =  
        "SELECT * FROM articolo" +  
        "WHERE prezzo > ? ";  
  
    PreparedStatement st =  
    con.prepareStatement(sqlCommand);  
    st.setObject(1, prezzo);  
  
/*prima di inviare la query, JDBC cercherà di convertire l'oggetto generico  
prezzo nel tipo SQL NUMERIC, se questo non è possibile si verificherà  
un'eccezione*/  
  
    st.execute();  
  
}
```

# setObject

- Al metodo setObject non è possibile passare argomenti appartenenti a tipi java fondamentali
  - Errore: `st.setObject(1, 34000);`
  - 34000 è di tipo fondamentale (int)
  - Per ovviare usare le classi wrapper:  
`st.setObject(1, new Integer(34000));`

# SetObject: altra versione del metodo

- Il parametro aggiuntivo è l'indicazione esplicita del tipo JDBC/SQL che si intende utilizzare per la conversione

```
void setObject (int parameterIndex, Object x, intsqlType);
```

- Esempio

```
void query(Object prezzo) throws SQLException {  
  
    String sqlCommand =  
        "SELECT * FROM articolo" +  
        "WHERE prezzo > ? ";  
    PreparedStatement st = con.prepareStatement(sqlCommand);  
    st.setObject(1, prezzo, Types.Numeric);  
/*si comunica al driver che si intende convertire il parametro prezzo nel tipo  
NUMERIC*/  
    st.execute();  
}
```

# SetNull

- Serve ad impostare su Null il valore dei parametri

```
void setNull (int parameterIndex, intsqlType);
```

- Esempio:

```
/*recuperare dalla tabella articolo tutti gli articoli per i quali non è stato  
ancora impostato il prezzo*/
```

```
void query(Object prezzo) throws SQLEception {  
  
    String sqlcommand =  
        "SELECT * FROM articolo" +  
        "WHERE prezzo > ? ";  
    PreparedStatement st = con.prepareStatement(sqlCommand);  
    st.setNull(1, Types.Numeric);  
/*è obbligatorio specificare il tipo JDBC/SQL*/  
    st.execute();  
}
```

# SetBinaryStream: Input da Stream

- Serve ad assegnare un valore ad un parametro prelevandolo da file
- Utile soprattutto quando la quantità di informazione da trasferire è elevata
- Signature:

```
void setBinaryStream (int  
parameterIndex, inputStream x, int  
length);
```

- Dove x è il file da cui prelevare
- Occorre specificare la lunghezza perché alcuni server di database devono conoscere a priori la quantità esatta di caratteri prima di eseguire la query

# SetBinaryStream: Input da Stream

## ■ Esempio:

/\*le informazioni assegnate al parametro *stuff* provengono dal  
file “tmp/data” e dovranno essere compatibili con il tipo SQL  
di *stuff*\*/

```
File file = new File("/tmp/data");
Int fileLength = file.length();
InputStream fin = new FileInputStream(file);

PreparedStatement st = con.prepareStatement(
    UPDATE Table5 SET stuff = ? WHERE index = 4");

st.setBinaryStream(1, file, fileLength);
st.executeUpdate();
}
```

## ■ Se lo stream è di tipo ASCII o UNICODE è possibile usare getAsciiStream o getUnicodeStream