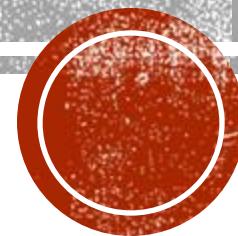


BASI DI DATI I



INTRODUZIONE



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 perseguitamento degli scopi dell'organizzazione).
 - 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.
- 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.e. 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

Sistema Azienda

Sistema Organizzativo

Sistema Informativo

Sistema Informatico

I DATABASE

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,...
- Ricerca nel catalogo elettronico di una biblioteca
- Richiesta di documenti
- Operazioni bancarie
- ...



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 aprendo 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
 - X
 - TikTok
 - ...
- Si ha necessità di memorizzare i post, gli utenti, le foto, i video...
- Tecnologie sempre più complesse e flessibili stanno emergendo per venire incontro alle odierne necessità
 - Big Data Storage Systems (computer Clusters organization)
 - NOSQL (Not Only SQL) Systems
 - Cloud Storaging



DEFINIZIONE DI BASE DI DATI

- 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.
- Una base di dati ha le seguenti proprietà
 - Rappresenta un aspetto del mondo reale (**mini-world**): cambiamenti al mondo reale si riflettono sulla base di dati.
 - È una collezione di dati logicamente coerenti con un significato intrinseco: un assortimento casuale di dati non può essere considerato una base di dati.
 - È progettata, costruita e popolata per uno scopo specifico: possiede utenti specifici e applicazioni specifiche cui questi utenti sono interessati



DATABASE MANAGEMENT SYSTEMS

- Un **DBMS** (Database Management System – Sistema di gestione di basi di dati) è una collezione di programmi che consente la creazione, la gestione, e la manutenzione di una base di dati.
 - È un sistema **General Purpose** che facilita la definizione, la creazione, la manipolazione e la condivisione di una base di dati

DEFINIRE UNA BASE DI DATI

implica specificarne i tipi, le strutture e i vincoli tra i dati (**METADATI**)

COSTRUIRE UNA BASE DI DATI

significa immagazzinare i dati su un mezzo di storing secondario gestito dal DBMS

MANIPOLARE UNA BASE DI DATI

significa interrogare (*query*), aggiornare e creare report sui dati immagazzinati, oltre che accedere al DB tramite app stand-alone o web-app

CONDIVIDERE UNA BASE DI DATI

significa consentire a programmi ed applicazioni di accedere contemporaneamente ai dati

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



IL PROGRAMMA APPLICATIVO

- Un **programma applicativo** accede alla base di dati inviando delle interrogazioni (*query*) al DBMS
 - L'interrogazione comporta l'estrazione di dati dalla base di dati
 - Una serie di letture/scritture su una base di dati può comportare la creazione di una **transazione**.



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**).



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 Kbyte per ogni modulo:
$$100 \times 10^6 \times 100 \times 10^3 \times 2 \times 5 = 10^2 \times 10^6 \times 10^2 \times 10^3 \times 10 = 10^{14} \text{ bytes}$$
- Tenendo traccia delle ultime quattro dichiarazioni, risulterebbe una base di dati di 4×10^{14} bytes= 400 Terabytes
- 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.
- 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



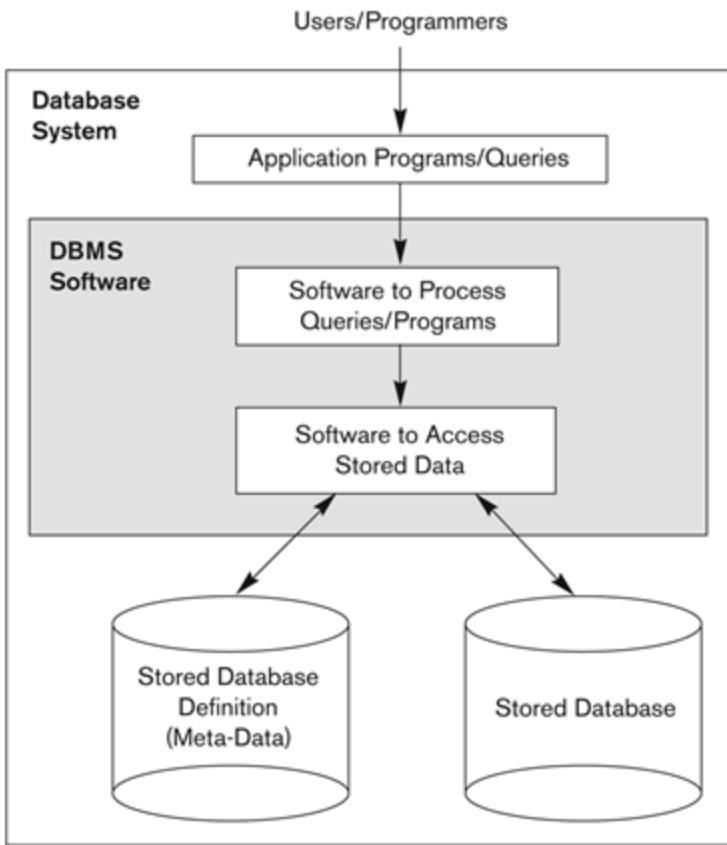
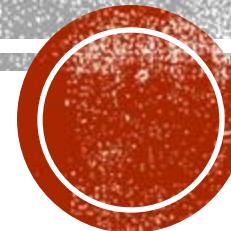


Figure 1.1
A simplified database system environment.





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.
 - INSEGNAMENTO: 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.



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.

Data Element

Un record del file studente contiene dati per rappresentare il **nome**, la **matricola**, l'**anno**, il **corso di laurea**

Tipo di ogni Data Element

Ogni campo è definito come una **stringa di caratteri**

STUDENTE

Nome	Matricola	Anno_Corso	Corso_Laurea
Esposito	N86003223	1	Informatica
Moretti	N86009889	2	Informatica
Gigli	N86009999	F.C.	Informatica
....



DEFINIZIONE DEL DB (2)

INSEGNAMENTO

Nome_Ins	Cod_Ins	CFU	Dipartimento
Basi di Dati	BD	9	Informatica
Object Orientation	OO	9	Informatica
Algoritmi e Strutture Dati	ASD	6	Informatica
....

PREREQUISITI

Cod_Ins	Cod_Prop
BDII	BD
IS	OO
....

VOTAZIONE

Matricola	Cod_Ins	Voto
N86009889	BD	21
N86009112	OO	18
N86008743	ASD	29
N86009889	OO	24

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



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 "Moretti" 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.



MANIPOLAZIONE DEL DB

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

Esempio di query:

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

Esempio di aggiornamenti:

- Cambia l’anno di corso di Moretti in 3
- Inserisci una votazione di 19 per Gigli in Object Orientation
- Elimina Il Prerequisiti tra Object Orientation e Fondamenti di Programmazione



FASI PER LA PROGETTAZIONE DI UN DATABASE

1. Specifica ed analisti dei Requisiti

1. Progettazione Concettuale

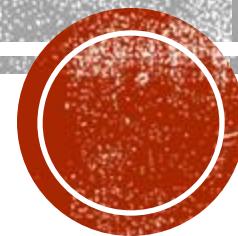
1. Progettazione Logica

1. Progettazione Fisica





UTILIZZO DI UN DB

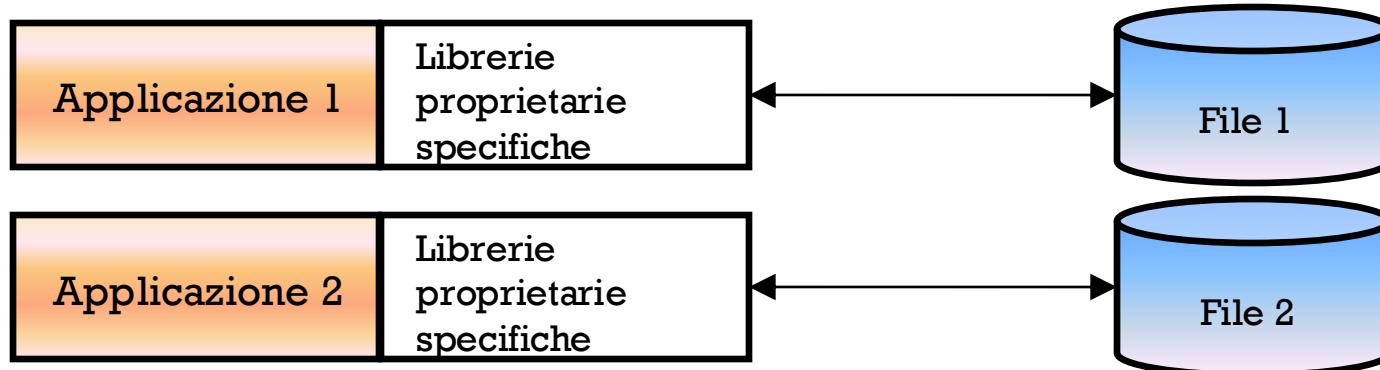


FILE PROCESSING VS DATABASE

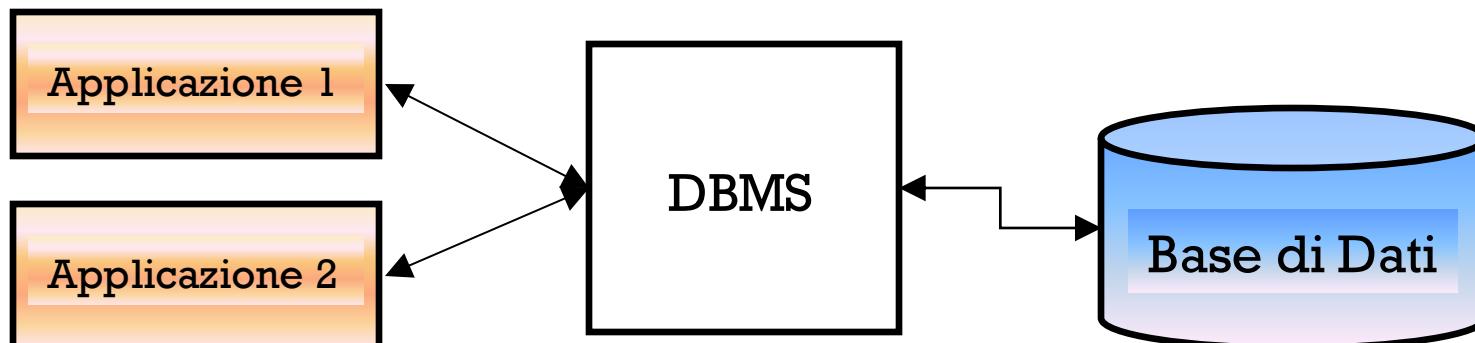
- Vi sono notevoli differenze tra l'approccio con le basi di dati e l'approccio tradizionale basato sull'accesso ai dati in un file.
- Nella tradizionale gestione file
 - Ogni utente definisce ed implementa i file necessari per una specifica applicazione.
 - **Questo è parte della programmazione dell'applicazione stessa.**
 - *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.
- Nell'approccio con le basi di dati
 - Il repository è definito separatamente dall'applicativo ed è indipendente dalle applicazioni/utenti che lo utilizzano



FILE PROCESSING VS DATABASE (SCHEMA)



Approccio File Processing



Approccio DataBase



FILE PROCESSING VS DATABASE (PROPRIETÀ)

- Le principali caratteristiche dell'approccio con basi di dati rispetto all'approccio con gestione file
 - Natura autodescrittiva di un sistema di basi di dati
 - Separazione tra programmi e dati e astrazione dei dati
 - Supporto di viste multiple dei dati
 - Condivisione dei dati e gestione delle transazioni con utenti multipli



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
...



SEPARAZIONE TRA PROGRAMMI E DATI E ASTRAZIONE DEI DATI

- Indipendenza tra programmi e dati
 - Nei sistemi con gestione file, la struttura dei file è memorizzata nei programmi che devono accedervi: una modifica alla struttura comporta una modifica a tutti i programmi che accedono ad essa.
 - Nei sistemi basati su DBMS, la struttura del file è indipendente dal programma: la struttura del file è memorizzata nel catalogo del DBMS.
- Indipendenza tra programmi e operazioni
 - In alcuni sistemi di basi di dati, per esempio i sistemi orientati agli oggetti, gli utenti possono definire le operazioni sui dati nell'ambito della definizione della base di dati
 - Un'operazione è composta da una *signature* e da un'*implementazione*
 - I programmi applicativi possono operare sui dati invocando le operazioni conoscendone la *signature*, indipendentemente da come siano state implementate
- Queste due caratteristiche definisco l'**Astrazione dei Dati**



MODELLI 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.
- Il modello di dati *nasconde dettagli di memorizzazione ed altre informazioni che possono non essere di interesse per molti degli utenti della base di dati.*



SUPPORTO DI VISTE MULTIPLE DEI DATI

- 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 consente la definizione di viste multiple.

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



CONDIVISIONE DEI DATI E GESTIONE DELLE TRANSAZIONE IN AMBIENTI MULTIUTENTE

- Un DBMS multiutente deve consentire l'accesso a più utenti contemporaneamente.
- Essenziale se la base di dati mantiene dati per molteplici applicazioni
- **Controllo della concorrenza**
 - Garantire a più utenti di aggiornare gli stessi dati in maniera controllata, cosicchè il risultato degli aggiornamenti sia corretto.
 - **Online transaction processing (OLTP)**

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



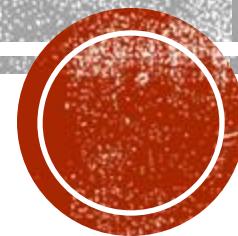
PROPRIETÀ DELLE TRANSAZIONI

- 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
 - UF Occasionali: accedono occasionalmente al DB e lo interrogano attraverso query Languages sofisticati;
 - UF non esperti: rappresentano una parte considerevole di utenza. Usano aggiornamenti e queries standard;
 - UF esperti: ingegneri, scienziati e analisti di affari che hanno familiarità con le facilities del DBMS per richieste complesse;
 - UF indipendenti: 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.



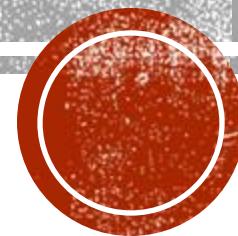
ALTRÉ FIGURE

- **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**
 - Evitare di memorizzare più volte gli stessi dati porta notevoli vantaggi dal punto di vista dell'aggiornamento degli stessi
 - Inoltre, vi è un notevole risparmio di spazio
 - Normalizzazione dei dati
 - Ridondanza controllata
- **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**: file ausiliari che permettono di velocizzare le operazioni di ricerca
 - **Buffering e caching**: memorizzazione di porzioni di DB nel buffer della RAM
 - **Elaborazione ed ottimizzazione delle query**
- Forniscono strumenti di **backup** e **recovery**
- Forniscono interfacce utente multiple
 - **Graphical user interface (GUI)**
- Rappresentazione di relazioni complesse fra dati
- Imposizione di Vincoli di Integrità
 - Integrità referenziale, Vincolo di chiave...



VANTAGGI CONSEQUENZIALI

- 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.
- Potenziale per imporre standard operazionali all'interno di un'organizzazione
- Tempo ridotto per lo sviluppo di applicazioni
- Flessibilità al cambiamento
- Disponibilità di informazioni sempre aggiornate nei sistemi multiutente
- Economie di Scala



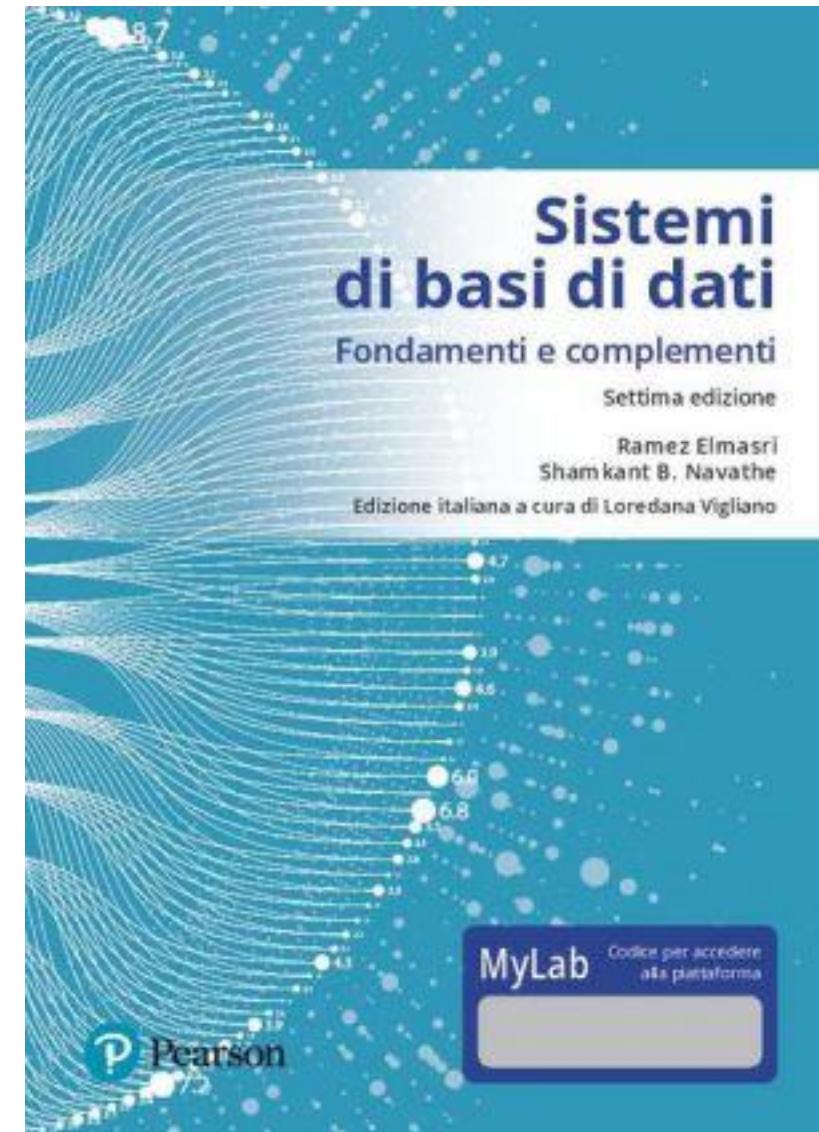
QUANDO NON USARE UN DBMS

- 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.



INFO SULLA LEZIONE

- Tutto il capitolo 1 tranne l'1.7
- L'1.7 tratta delle applicazioni delle basi di dati
 - Applicazioni con sistemi reticolari e gerarchici
 - Basi di dati relazionali e ORDBMS
 - XML per l'interscambio su WEB
 - Sistemi di archiviazione per Big Data e NOSQL

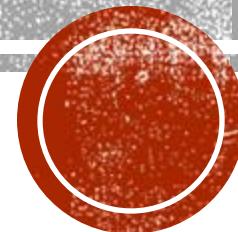




FINE

Per eventuali domande: (in ordine di preferenza personale)

- Ora.
- Chat di Teams
- Mail: silvio.barra@unina.it



BASI DI DATI I

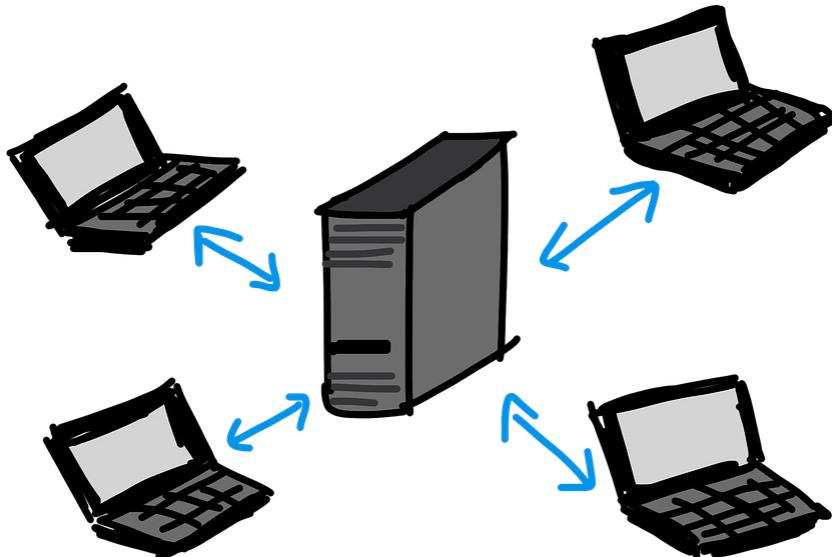
DBMS: DAL MONOLITICO AL DISTRIBUITO

- La maggior parte dei sistemi informativi sta sempre più adottando soluzioni distribuite, contrapponendosi al sistema monolitico che prevedeva un sistema HW e SW estremamente integrate l'uno con l'altra
- L'evoluzione dei DBMS ha avuto una analoga evoluzione
 - Dal monolitico al modulare/distribuito con architetture di tipo client-server
 - Gli attuali ambienti di cloud computing consistono di migliaia di server dislocati geograficamente che gestiscono enormi moli di dati (Big Data) per conto degli utenti



CLIENT-SERVER

- Nelle architetture client-server si hanno
 - Un modulo **client** progettato per essere eseguito su un pc dell'utente
 - Dovendo gestire l'interazione con l'utente, è spesso fornito di GUI user-friendly
 - Un modulo **server** che si occupa di permettere l'accesso agli utenti ad una serie di servizi, quali la memorizzazione, l'elaborazione e la generazione di informazioni



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** (modello dei dati) che consiste di:
 - Una **struttura** della base di dati
 - **Es:** i tipi dei dati, i vincoli che specificano restrizioni sui dati per renderli validi e le relazioni che intercorrono tra i dati
 - Le **operazioni** per manipolare le strutture
 - **Es:** Inserzione, cancellazione, modifica, ritrovamento di un oggetto.
 - Un **comportamento dinamico (o aspetto)** per permettere all'utente di definire operazioni sui dati
 - Un'operazione CALCOLA_MEDIA_VOTO che calcola la media dei voti e che può essere applicata a un oggetto studente.



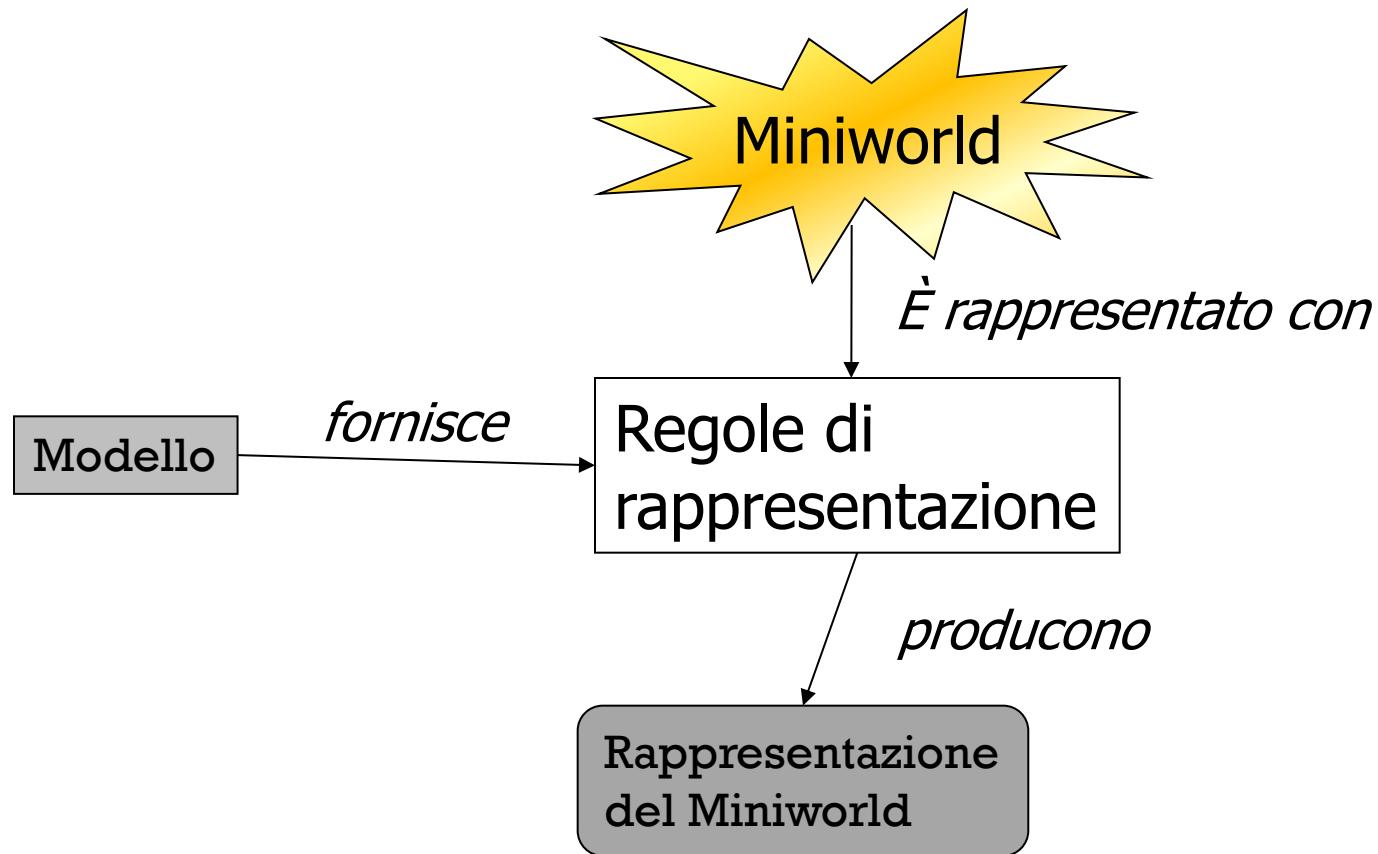
COSA DEVE FARE UN MODELLO

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.

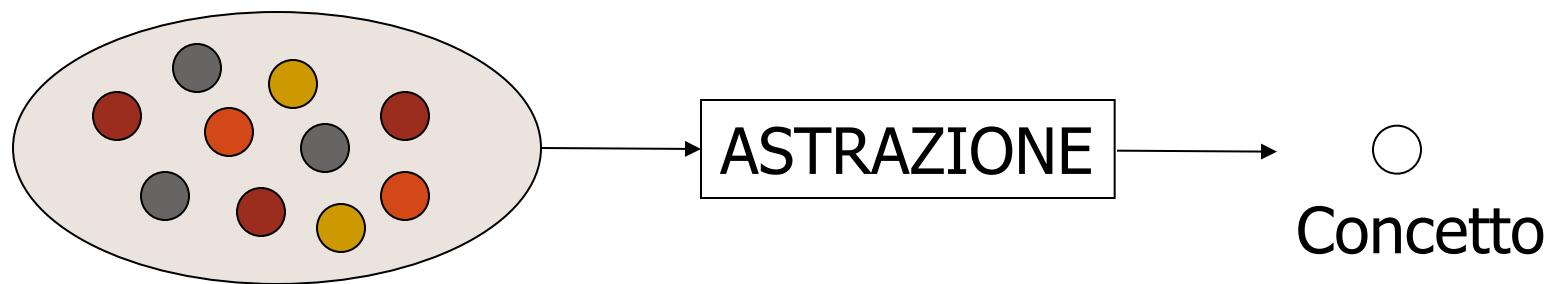


COS'È UN MODELLO (SCHEMA)



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 implementazione** forniscono concetti comprensibili agli utenti finali, ma che non sono troppo lontani dal modo in cui i dati sono fisicamente organizzati.
- 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 CONCETTUALI (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 (Entity-Relationship).



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 encapsulati 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.) - *legacy*
 - 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.) - *legacy*
- Tali modelli rappresentano i dati usando strutture di record. Per tale motivo sono chiamati anche **modelli dei dati record-based**.
- I **modelli dei dati 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



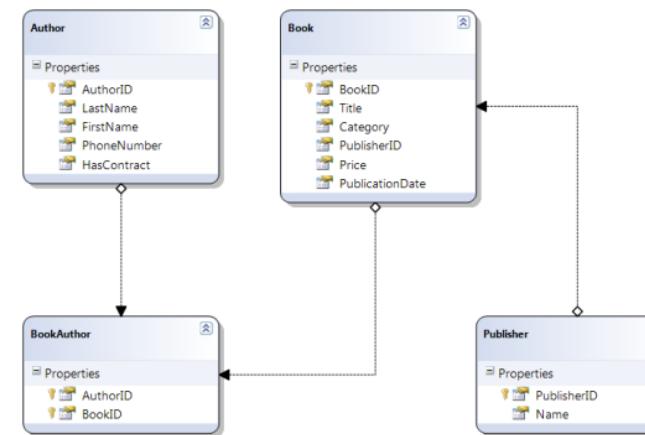
reticolare

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

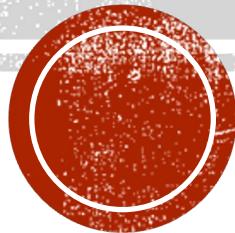


DATA MODEL FISICI

- 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.
 - L'**indice** è un esempio di percorso di accesso che garantisce l'accesso diretto ai dati tramite un termine indice o una parola chiave
 - Gli indici possono essere poi organizzati in maniera lineare, gerarchica o in altra maniera



SCHEMI, ISTANZE E STATO DI UNA BASE DI DATI



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.
- La descrizione è detta **schema** della base di dati e corrisponde alla struttura logica del database.
 - Talvolta chiamata **intensione della base di dati**
- Un'**istanza** è il contenuto del database in un particolare istante di tempo.
 - Talvolta chiamata **estensione della base di dati**



SCHEMI DI BASI DI DATI

- **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	Corso_Laurea
------	-----------	------------	--------------

INSEGNAMENTO

Nome_Ins	Cod_Ins	CFU	Dipartimento
----------	---------	-----	--------------

PREREQUISITI

Cod_Ins	Cod_Prop
---------	----------

VOTAZIONE

Matricola	Cod_Ins	Voto
-----------	---------	------

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



ISTANZE DI DATABASE

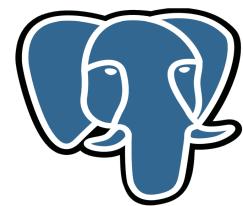
- **I dati reali in un DB in uno specifico istante di tempo costituiscono l'istanza della base di dati**
 - Al contrario dello schema questi possono cambiare frequentemente.
 - Es: Il DB UNIVERSITA' cambia ogni volta che si inserisce un nuovo studente o si registra un nuovo esame per uno studente.
- In un certo stato della base di dati, ogni costrutto ha il proprio insieme corrente di istanze
 - Es: Il costrutto STUDENTE conterrà l'insieme delle singole entità (*record*) di ogni studente
- 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



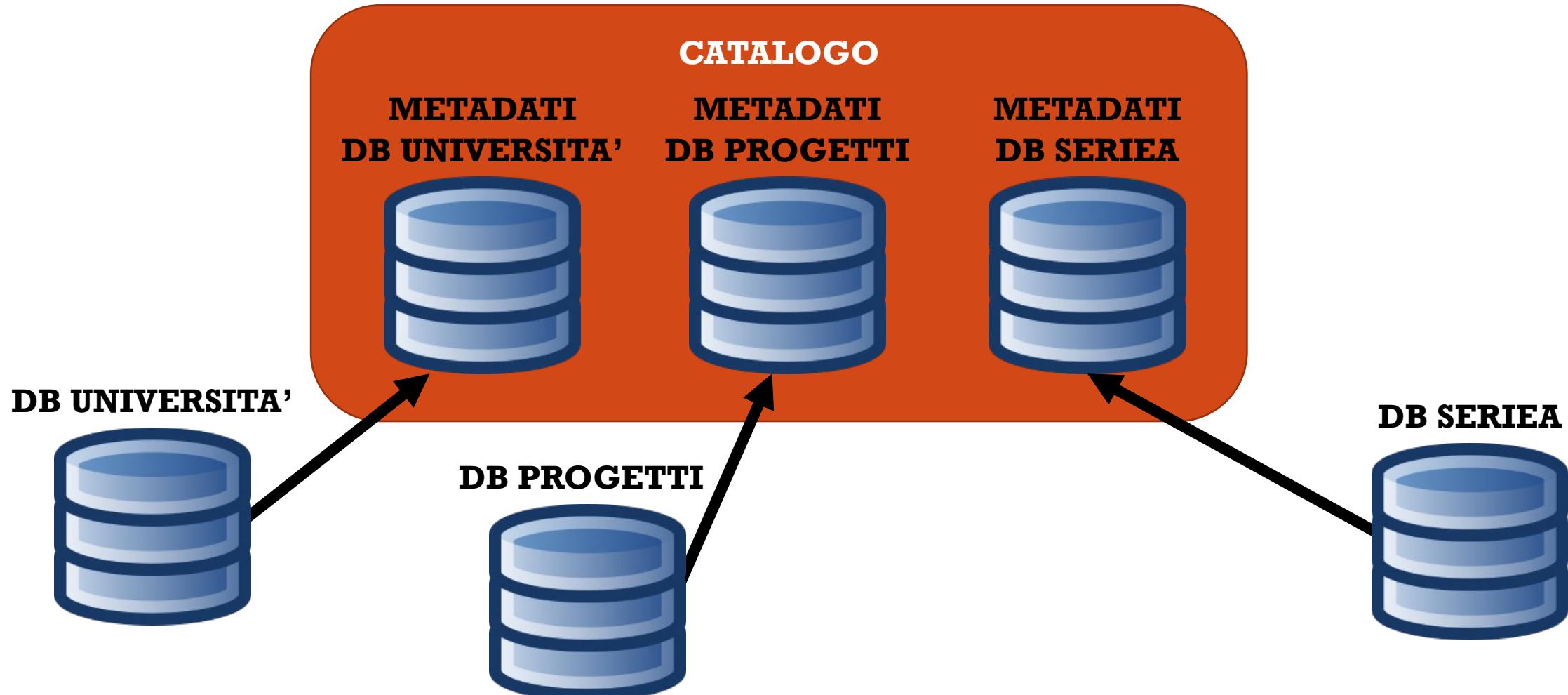
L' IMPORTANZA DELLO SCHEMA DI DATABASE

- La distinzione tra **schema** e **istanza** è di notevole importanza
 - 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 (popolazione della base di dati).
 - Da questo momento in poi ogni inserimento, cancellamento o modifica porterà il DB in un nuovo stato.
- 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.





PostgreSQL



EVOLUZIONI

- Lo schema della base di dati non dovrebbe cambiare frequentemente, ma ad ogni modo può accadere che alcune modifiche siano necessarie, soprattutto quando:
 - Cambiano i requisiti dell'applicazione.
 - Cambia il mini-world che stiamo rappresentando.
- In tali casi, lo schema deve cambiare.
 - **Evoluzione dello schema della base di dati.**
- La maggior parte dei moderni DBMS include operazioni per permettere l'evoluzione dello schema **MENTRE** la base di dati è operativa.



ARCHITETTURA DEI R-DBMS

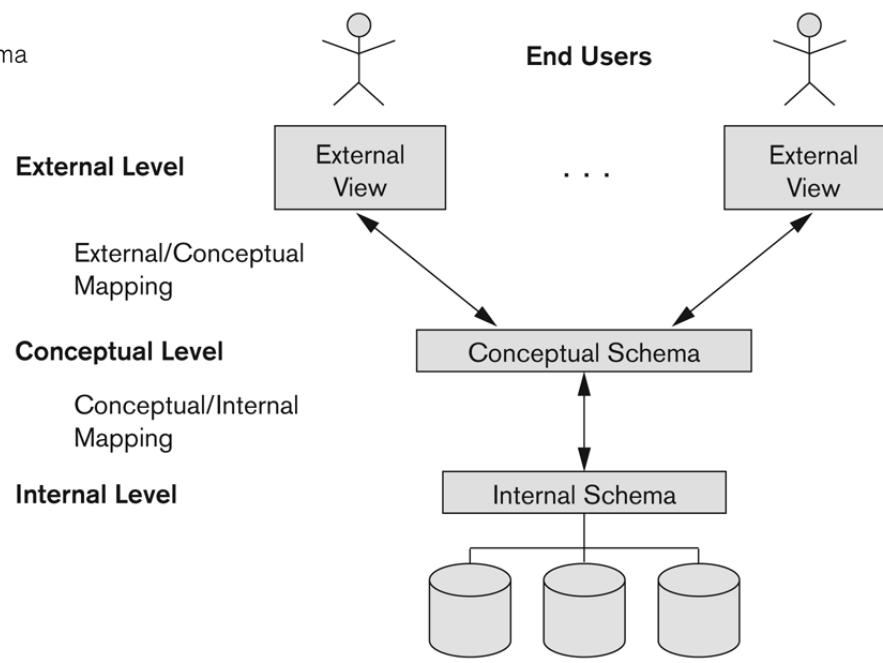


L'ARCHITETTURA A TRE LIVELLI

(ARCHITETTURA THREE-SCHEMA)

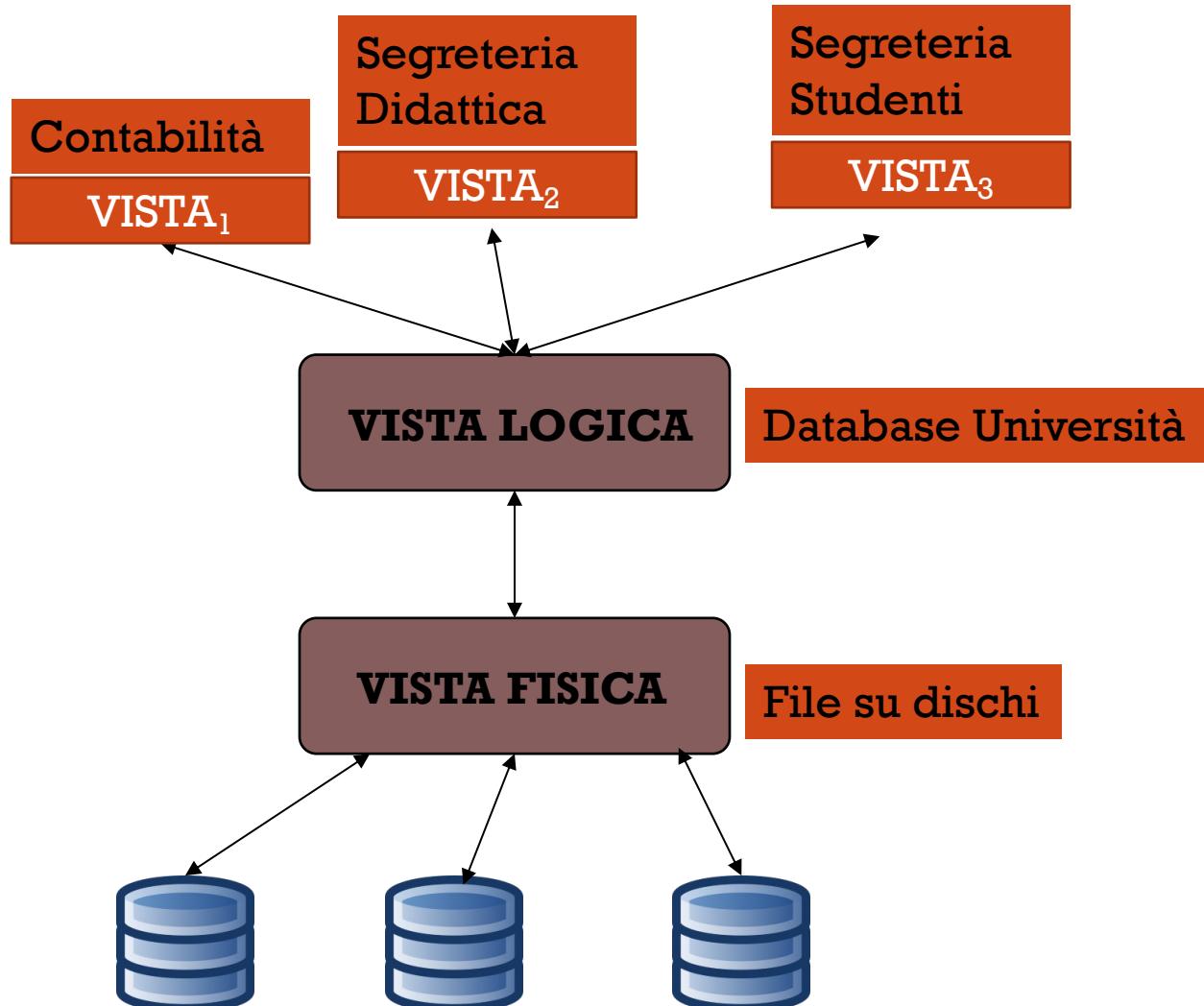
Figure 2.2

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);
 - Isolamento tra programmi e dati.
- 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, rimuovendo o modificando un tipo di record, un data item o un vincolo).
- 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.



SCHEMI DI DATABASE: *ESEMPIO DB UNIVERSITÀ*

STUDENTE
Nome

Matricola

Anno_Corso

Corso_Laurea

INSEGNAMENTO
Nome_Ins

Cod_Ins

CFU

Dipartimento

PREREQUISITI
Cod_Ins

Cod_Prop

VOTAZIONE
Matricola

Cod_Ins

Voto

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



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.



REALIZZAZIONE INDIPENDENZA LOGICA E FISICA

- L'**indipendenza fisica** dei dati si realizza quasi sempre, nella maggior parte delle basi di dati e nella maggior parte degli ambienti
 - I dettagli di basso livello sono effettivamente nascosti agli utenti.
 - Le applicazioni ignorano dettagli di memorizzazione, posizionamento, compressione, ...
- L'**indipendenza logica** è più complicata da realizzare
 - Apportare modifiche alla struttura dei dati e ai vincoli, senza incidere sulle applicazioni è alquanto impegnativo.
- Con un'architettura a più livelli, è necessario aggiungere ulteriori informazioni nel catalogo del DBMS su come mappare le richieste tra i diversi livelli.
- L'indipendenza dei dati avviene perché se uno schema ad un determinato livello L_i muta, non muta lo schema al livello L_{i+1} (o L_{i-1}), bensì muta il mapping tra i livelli.



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 multi-livello 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

- Abbiamo visto in precedenza, quanti e quali sono gli utenti tipici di un DBMS
 - Amministratori, progettisti, utenti finali, operatori, sviluppatori, ...
- *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.



DATA DEFINITION LANGUAGE (DDL)

- Nei DBMS dove NON C'È una netta separazione tra livelli, progettisti e DBA usano un **data definition language (DDL)** per definire lo schema concettuale e lo schema interno.
- Il compilatore del DDL (contenuto nel DBMS) ha la funzione di:
 - Elaborare le istruzioni DDL
 - Identificare le descrizioni dei costrutti dello schema.
 - Memorizzare la descrizione dello schema nel catalogo del DBMS.



STORAGE DEFINITION LANGUAGE (SDL)

- 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.
- Nella maggior parte dei DBMS non c'è uno specifico linguaggio con il ruolo di SDL, ma i DBA procedono definendo una serie di parametri e specifiche relative alla memorizzazione



VIEW DEFINITION LANGUAGE (VDL)

- Nelle architetture three-schema viene usato un **view definition language (VDL)** per specificare le viste utente e i loro mapping sullo schema concettuale.
- Nella maggior parte dei DBMS, il **DDL** è utilizzato sia per la definizione degli schemi esterni che per gli schemi concettuali
- Negli R-DBMS, si utilizza l'SQL come linguaggio per la definizione delle viste, come output di una determinata interrogazione



DATA MANIPULATION LANGUAGE (DML)

- 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 (o NONPROCEDURALE)

- 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*)).
- I DML di alto livello (come SQL) sono detti **set-at-a-time** o **set-oriented** perché possono specificare e ritrovare molti record in un 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.

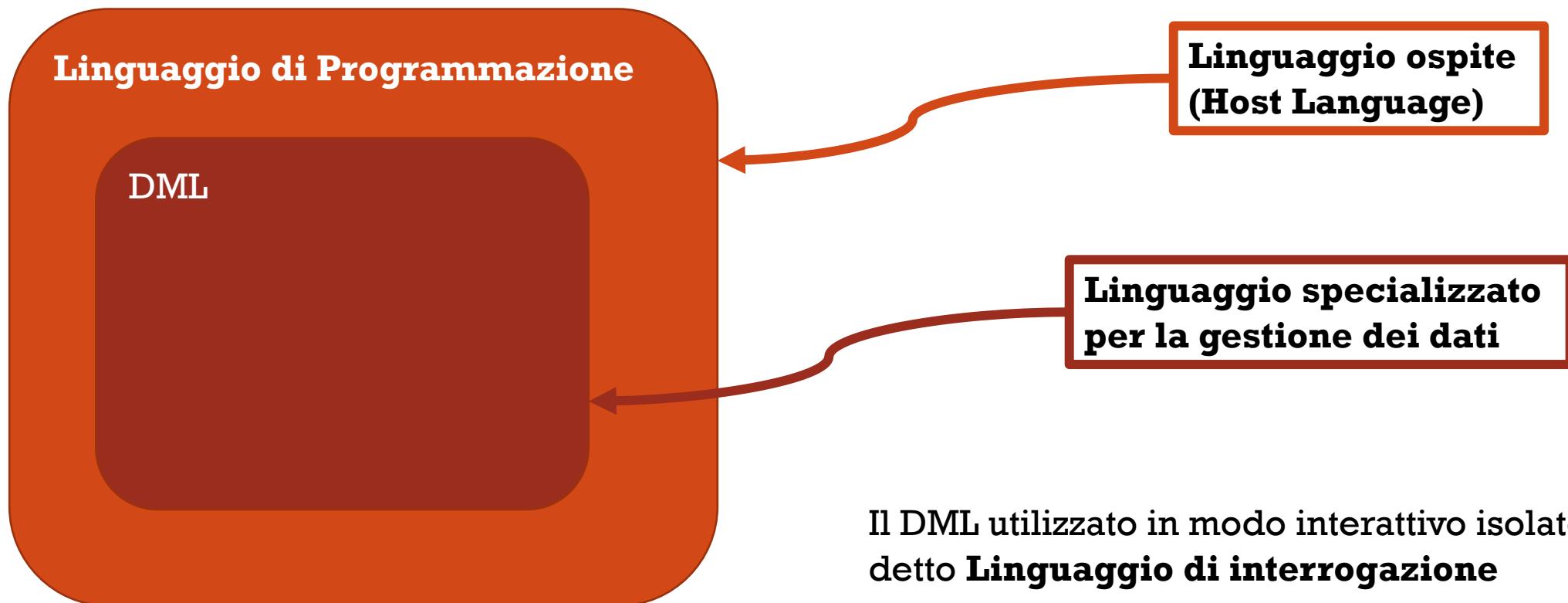


I DML DI BASSO LIVELLO (PROCEDURALE)

- **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.



DML + LINGUAGGI DI PROGRAMMAZIONE



SQL, UN LINGUAGGIO INTERATTIVO

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

Corsi

Corso	Docente	Aula
Basi di dati	Rossi	DS3
Sistemi	Neri	N3
Reti	Bruni	N3
Controlli	Bruni	G

Aule

Nome	Edificio	Piano
DS1	OMI	Terra
N3	OMI	Terra
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 ;
```



INTERFACCE PER I DATABASE



INTERFACCE MENU-BASED

- Presentano all'utente liste di opzioni, dette menù, che guidano l'utente nella formulazione di una richiesta.

La query è composta un passo alla volta scegliendo opzioni da una lista di menu che viene visualizzata dal sistema.

- Non c'è necessità di memorizzare i comandi specifici e la sintassi di un query language.
- I *menu pull-down* sono spesso usati nelle interfacce browsing, che consentono all'utente di guardare i contenuti del DB in maniera non strutturata.

INTERFACCE MOBILE

- Interfacce di accesso ai propri dati tramite mobile devices
- Le app che le implementano solitamente sono fornite di interfacce integrate che permettono una scelta limitata di opzioni rispetto alle interfacce Web.
- Permettono comunque operazioni sensibili quali pagamenti, prenotazioni, ...



INTERFACCE FORM-BASED

- Un'interfaccia form-based visualizza un form per ciascun utente. Gli utenti possono:
 - Riempire tutte le entrate del form completamente per inserire nuovi dati.
 - Riempire solo alcune entrate, nel qual caso il DBMS ritroverà dati che fanno matching per il resto delle entrate.
- I form sono in genere progettati e programmati per utenti naive (non esperti).
- Il form specification language, che molti DBMS hanno, aiuta i programmatore a specificare i form.
 - **Es:** *SQL*Forms* è un Form-based Language che permette di definire interrogazioni per mezzo di form progettati insieme allo schema della base di dati
 - **Es:** *Oracle Forms* è un componente della suite Oracle per la progettazione e lo sviluppo di applicazioni tramite form



Ricerca avanzata

Trova pagine web che contengono...

tutte queste parole:

Digita le parole importanti: labrador retriever nero

questa esatta parola o frase:

Racchiudi le parole esatte tra virgolette: "labrador retriever"

una qualunque di queste parole:

Digita OR tra tutte le parole che vuoi: miniatura OR standard

nessuna di queste parole:

Anteponi il segno - (meno) alle parole da escludere:
-roditore, - "Jack Russell"

numeri da:

 a

Inserisci due punti (...) tra i numeri e aggiungi un'unità di misura:
10..35 kg, € 300..€ 500, 2010..2011

Poi limita i risultati per...

lingua:

Trova le pagine nella lingua selezionata.

area geografica:

Trova le pagine pubblicate in un'area geografica specifica.

ultimo aggiornamento:

Trova le pagine aggiornate nel periodo di tempo specificato.

sito o dominio:

Cerca in un sito (come wikipedia.org) o visualizza soltanto i risultati relativi a un dominio, come .edu, .org o .gov

termini che compaiono:

Cerca i termini nell'intera pagina, nel titolo della pagina, nell'indirizzo web o nei link che rimandano alla pagina desiderata.

SafeSearch:

Indica a SafeSearch se filtrare i contenuti sessualmente esplicativi.

tipo di file:

Trova le pagine nel formato che preferisci.

diritti di utilizzo:

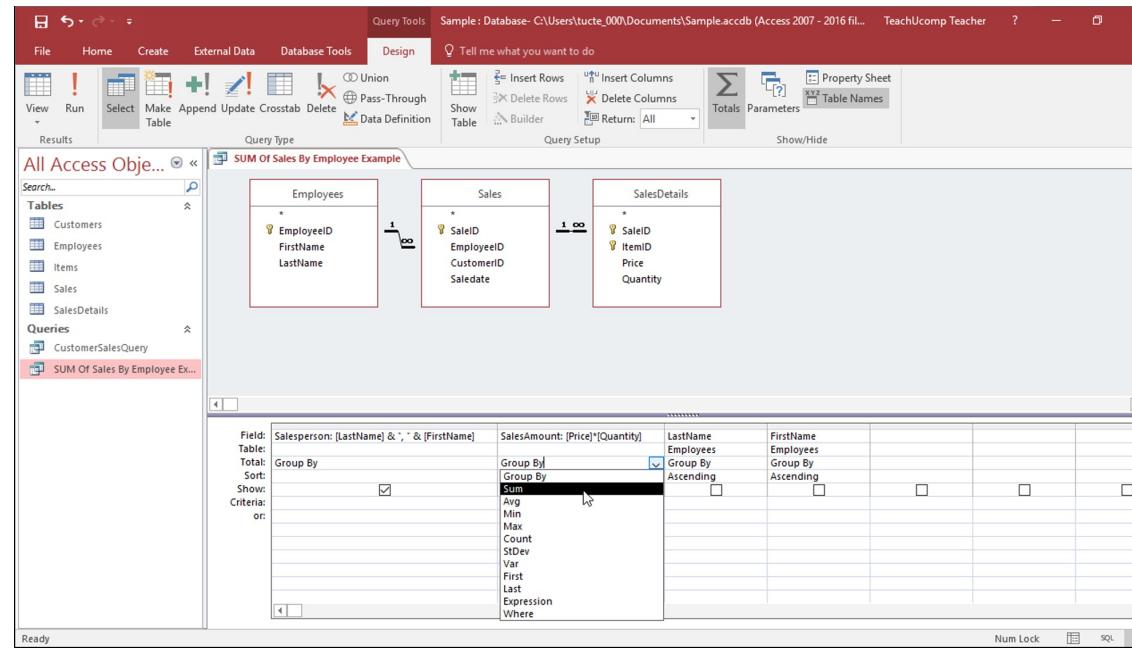
Trova le pagine che puoi utilizzare liberamente.

Ricerca avanzata



INTERFACCIE GRAFICHE (GUI)

- Tipicamente un'interfaccia grafica dispone su video uno schema in forma diagrammatica.
- L'utente specifica una query manipolando il diagramma, tramite dispositivi di puntamento, come il mouse.



INTERFACCE IN LINGUAGGIO NATURALE

- Accettano richieste scritte in linguaggio naturale e cercano di “*capire*”.
- Hanno un proprio schema specifico, che è simile allo schema concettuale del DB.
- Nell'interpretare le richieste l'interfaccia fa riferimento alle parole nello schema e a un insieme di parole standard (*dizionario*).
- se l'interpretazione ha successo, l'interfaccia genera una query high-level che corrisponde alla richiesta in linguaggio naturale e la sottomette al DBMS altrimenti inizia un dialogo con l'utente per chiarire la richiesta.
 - **Es:** Motori di ricerca come Google, AskJeeves, ...
 - **Es:** SIRI



INTERFACCE BASATE SU KEYWORD

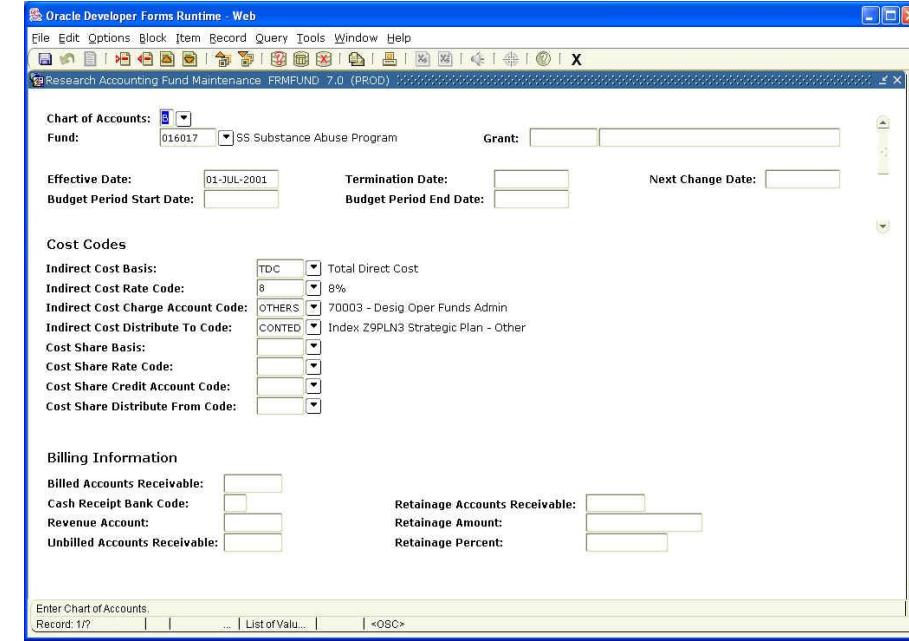
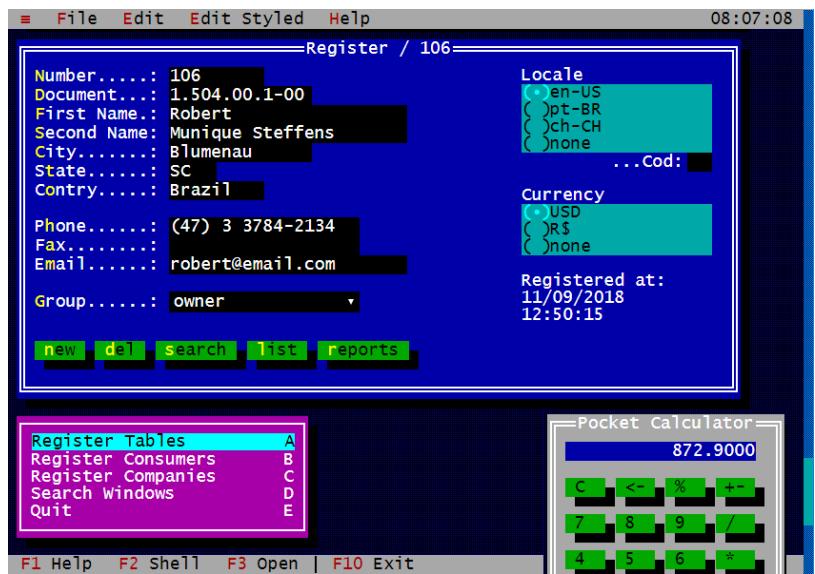
Google yahoo!

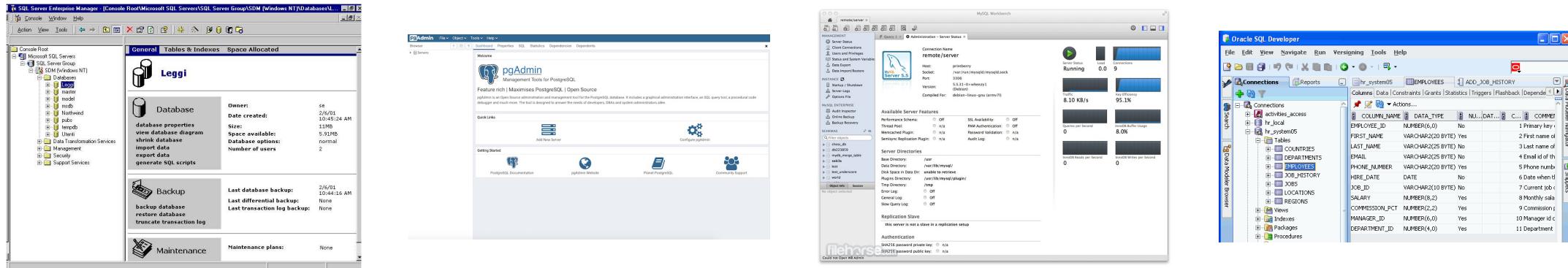
Bing Baidu 百度



INTERFACCE PER UTENTI PARAMETRICI

- Sono progettate per utenti naivi (es: impiegati di banca), così da comprendere un piccolo insieme di comandi abbreviati.
- Riguardano l'esecuzione ripetuta di un piccolo insieme di operazioni.
- Viene implementata una interfaccia speciale per ciascuna classe nota di utenti non esperti.

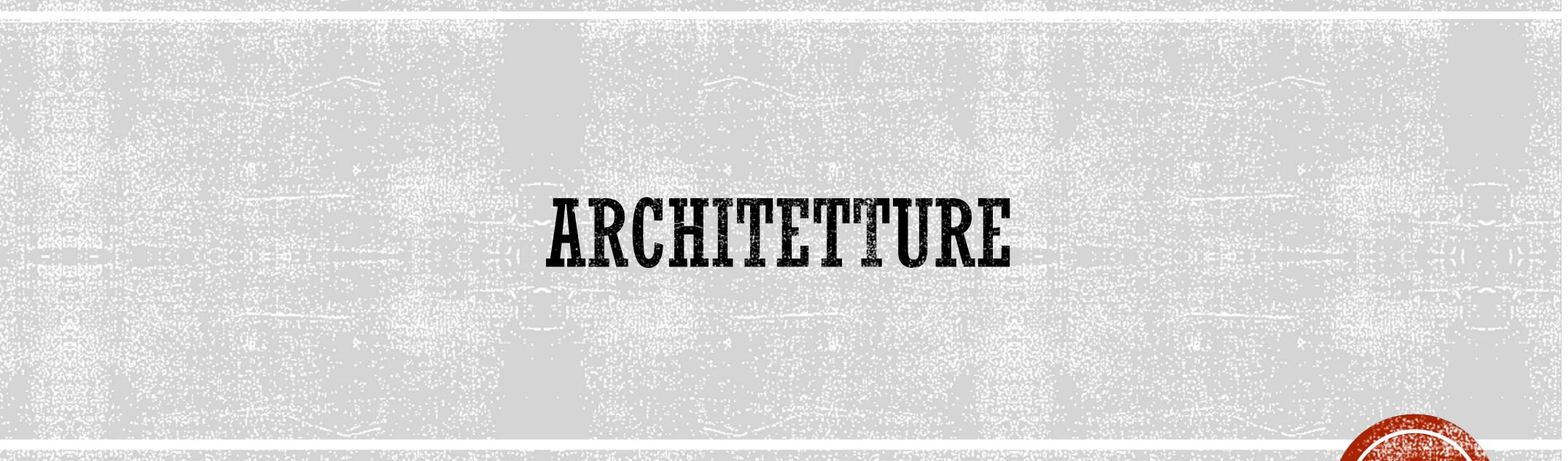




INTERFACCE PER I DBA

- La maggior parte dei DBMS contengono comandi privilegiati che possono essere usati solo dallo staff del DBA (es: creare account, settare dei parametri, cambiare uno schema, riorganizzare la struttura di memorizzazione del DB).





ARCHITETTURE



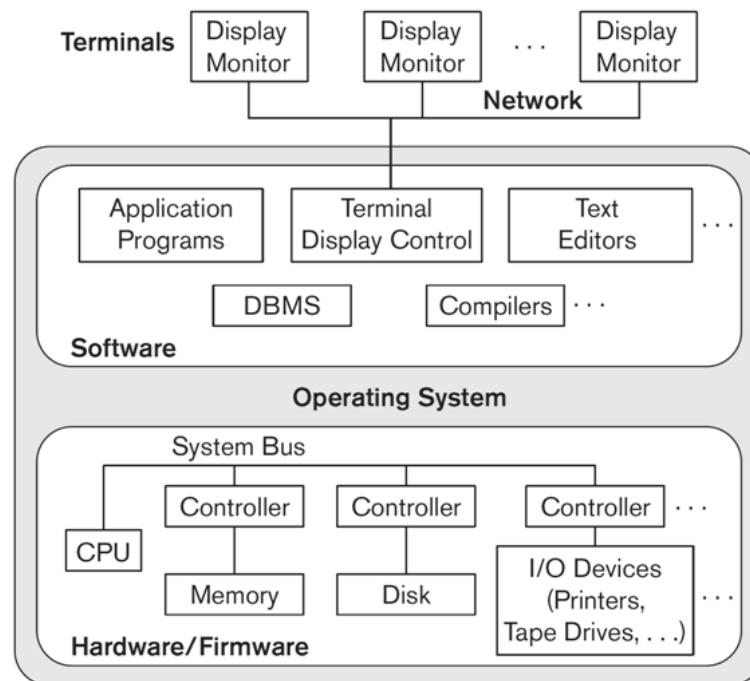
ARCHITETTURE PER I DBMS

- Le architetture per i DBMS hanno seguito un'evoluzione molto simile a quelle dei moderni sistemi di elaborazione
- In un primo momento si avevano sistemi mainframe, ai quali gli utenti accedevano con dei terminali «dumb», che fornivano solo operazioni di visualizzazione
- In un secondo momento, si è andato verso architetture modulari (client-server), dove una macchina funge da fornitore di servizi e le altre accedono e forniscono strumenti di visualizzazione (e alcune permettono anche l'elaborazione)
 - A due livelli
 - A tre livelli
 - A *n* livelli



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.



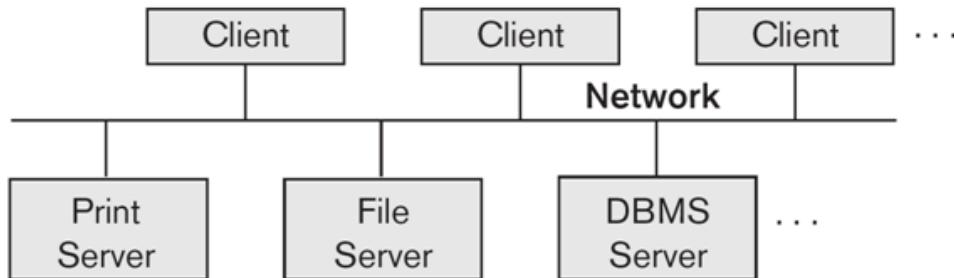
ARCHITETTURA CLIENT/SERVER BASE

- 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.

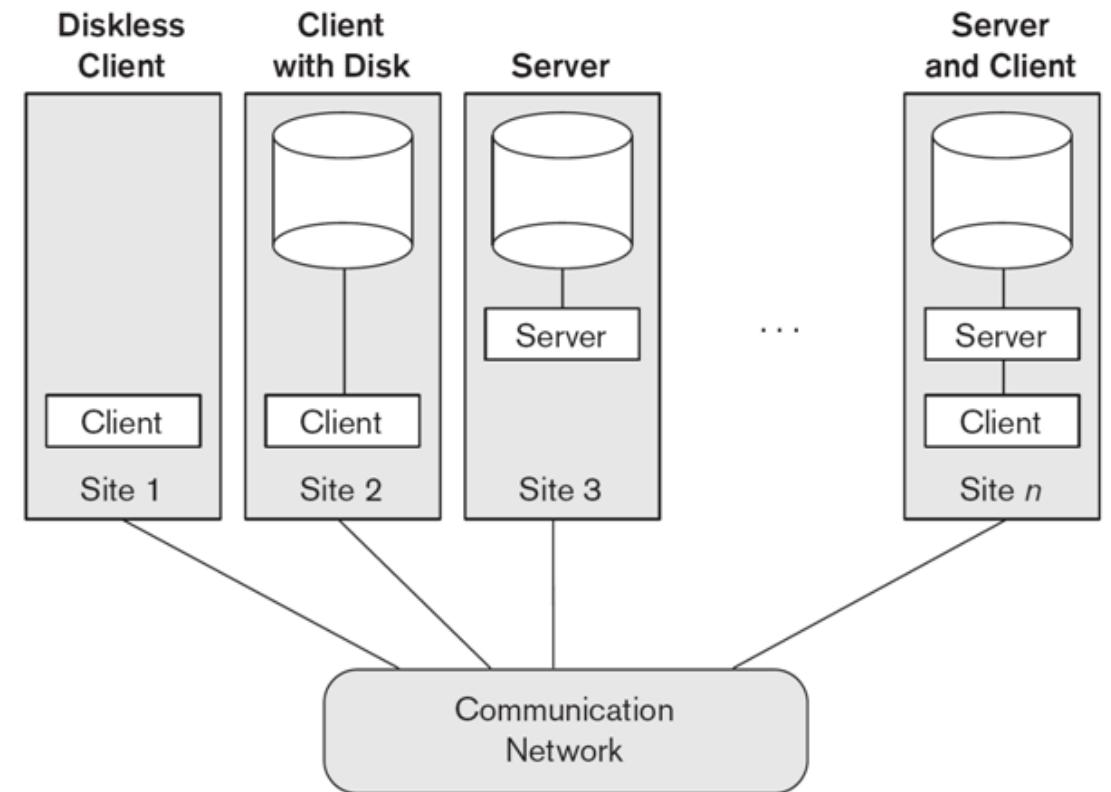


ARCHITETTURA CLIENT/SERVER TWO-TIER

- Architettura logica:



- Architettura fisica:



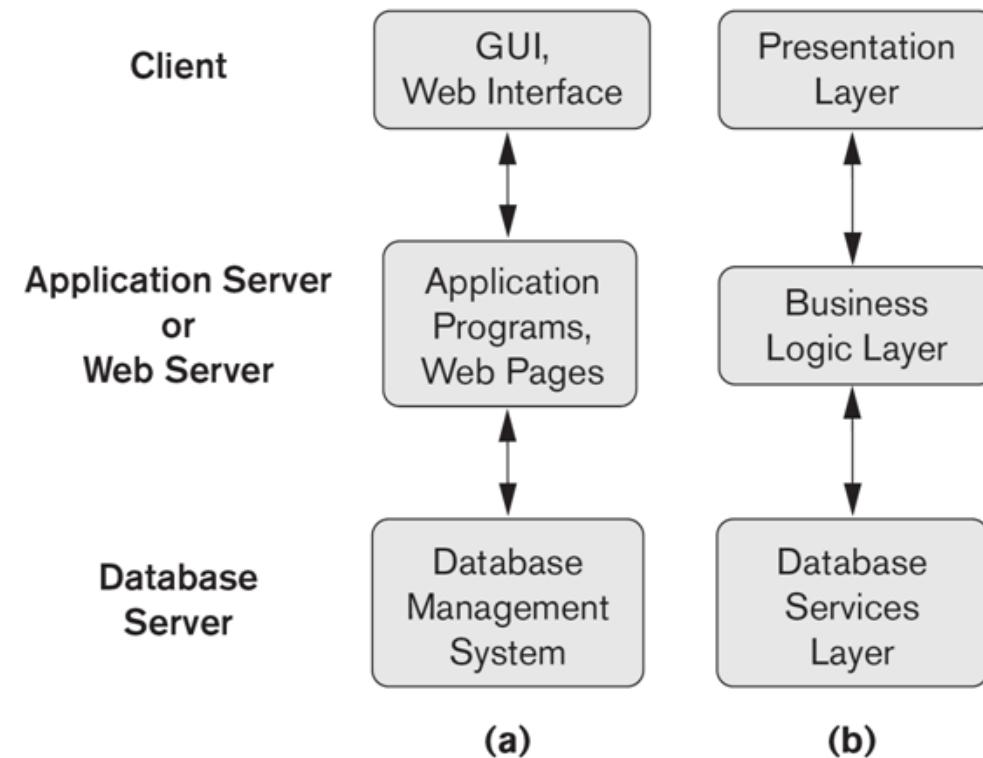
ARCHITETTURA CLIENT/SERVER TWO-TIER PER DBMS

- Applicativi e interfacce sono state le prime componenti ad essere spostati sul lato client.
 - Quando è richiesto l'accesso a un DBMS, viene stabilita una connessione
- Le funzionalità transazionali e di interrogazione rimasero lato server
 - Server delle interrogazioni/transazioni, Server SQL (nei R-DBMS)
- Open Database Connectivity (ODBC)
 - Fornisce una Application Programming Interface (API) che 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.



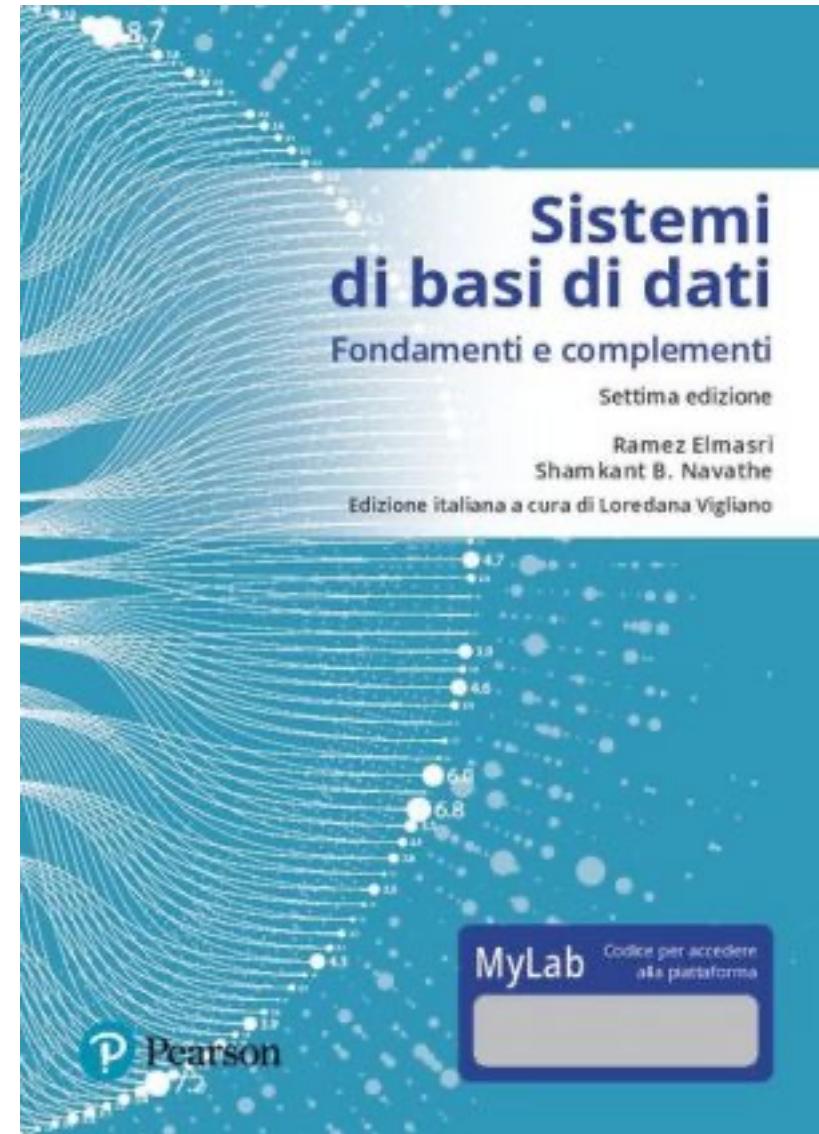
ARCHITETTURA CLIENT/SERVER THREE-TIER

- 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.



INFO SULLA LEZIONE

- Tutto il capitolo 2 tranne il 2.4 e il 2.6
- Il 2.4 spiega l'ambiente di esecuzione dei DBMS
 - Quali sono i moduli che lo compongono
 - Quali tool e utility mettono a disposizione i DBMS attuali
- Il 2.6 effettua una classificazione dei DBMS
 - R-DBMS
 - R-DBMS a oggetti
 - Distributed DBMS (DDBMS)
 - Native XML DBMS (DBMS di tipo sperimentale)

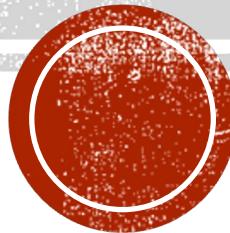




FINE

Per eventuali domande: (in ordine di preferenza personale)

- Ora.
- Chat di Teams
- Mail: silvio.barra@unina.it



Lezione 03: Il modello Entity-Relationship
UML
Nozioni di EER

BASI DI DATI I



A.A. 2022/2023

Prof. Silvio Barra

PROGETTAZIONE DI UNA BASE DI DATI

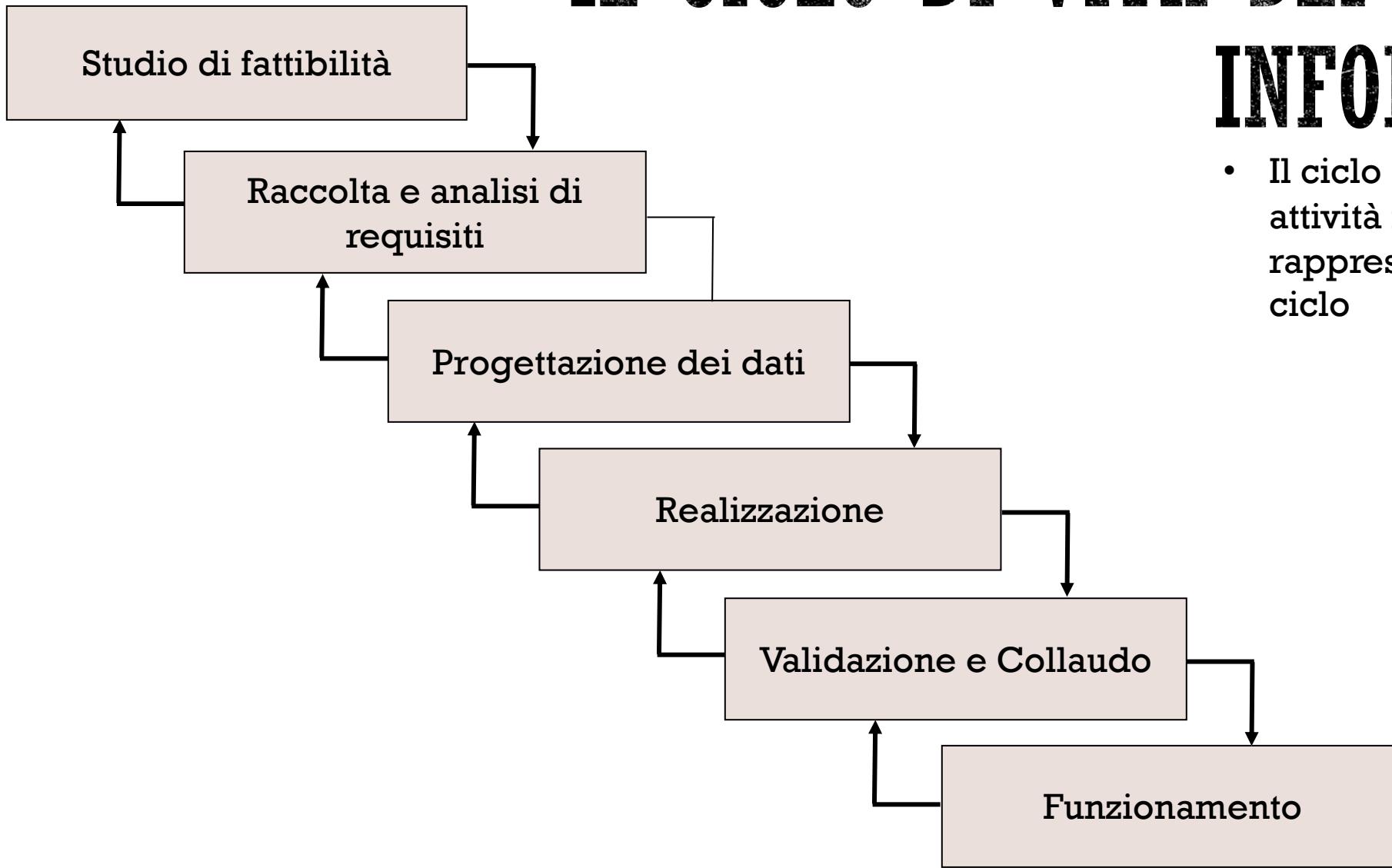


PROGETTAZIONE DI BASI DI DATI

- 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.
- Riprendendo la definizione che abbiamo dato di sistema informativo nella prima lezione,
 - Componente (*sottosistema*)
 - di una organizzazione che gestisce (*acquisisce, elabora, conserva, produce*)
 - le informazioni di interesse (*cioè utilizzate per il perseguimento degli scopi dell'organizzazione*).



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.

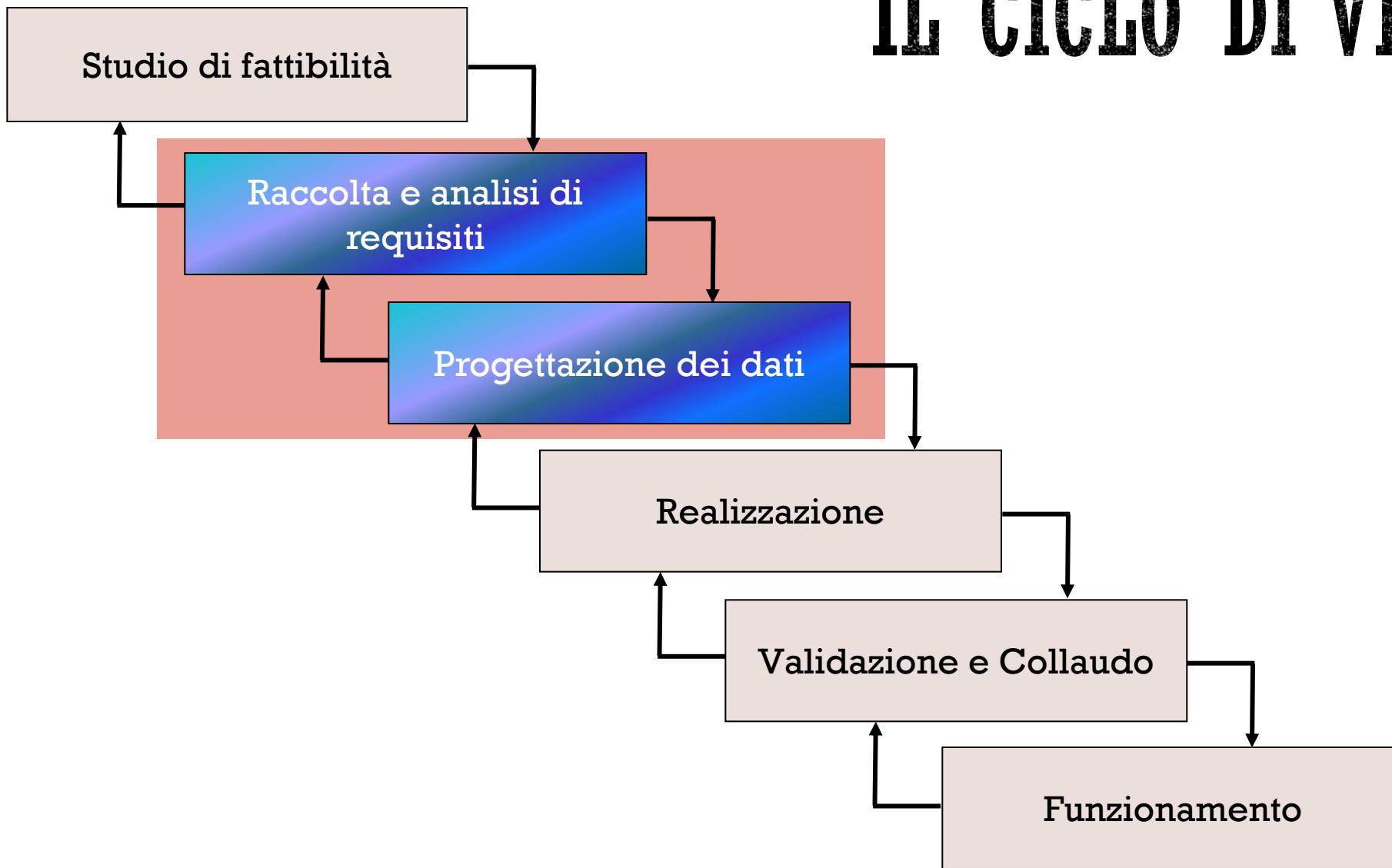
▪ Funzionamento

- 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.



LA MODELLAZIONE

- La modellazione è un'attività fondamentale, per molteplici fasi nello sviluppo di un software
 - Nella fase di raccolta di requisiti serve per comunicare in maniera schematica con gli stakeholder del progetto.
 - Nella fase di progettazione è necessaria per definire gli stati del software e le transizioni da uno stato all'altro.
 - Nella fase di sviluppo è utile per definire i flussi di controllo dell'applicazione.
- Ogni fase utilizza dei diagrammi specifici per l'attività che si porta a termine



LA MODELLAZIONE DEI DATI

- Anche la progettazione dei dati passa attraverso diverse fasi di modellazione.
- Abbiamo due tipi principali di modelli:
 1. **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.
 2. **Modelli logici:** utilizzati nei DBMS esistenti per l'organizzazione dei dati:
 - utilizzati dai programmi,
 - indipendenti dalle strutture fisiche.

Esempi: relazionale, reticolare, gerarchico, a oggetti.



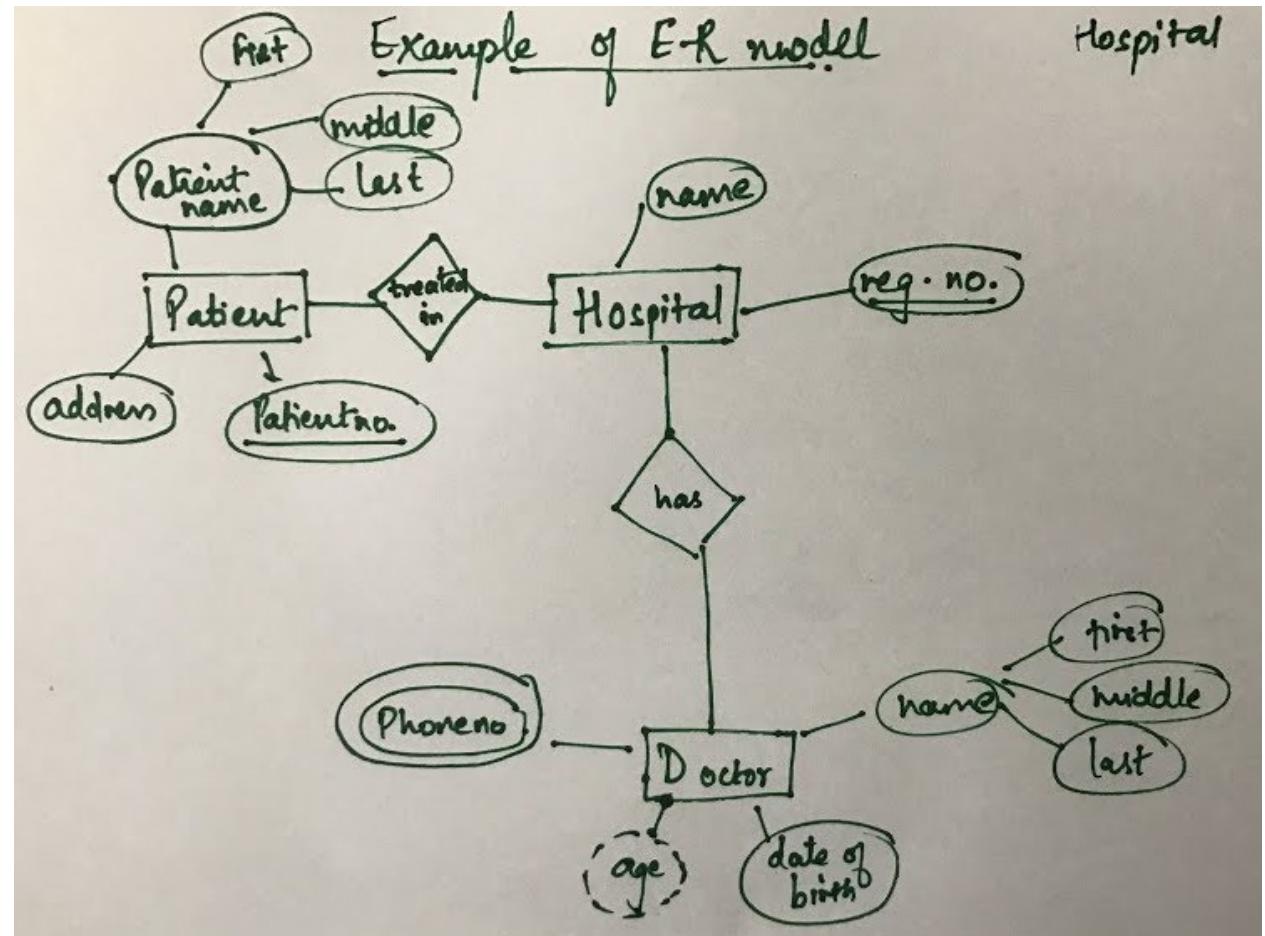
MODELLI CONCETTUALI, *PERCHÉ?*

- La modellazione concettuale fornisce una buona base di partenza per la progettazione di buone basi di dati.
- 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).



PROGETTAZIONE DI UNA BASE DI DATI



IL PROCESSO DI PROGETTAZIONE DEL DATABASE

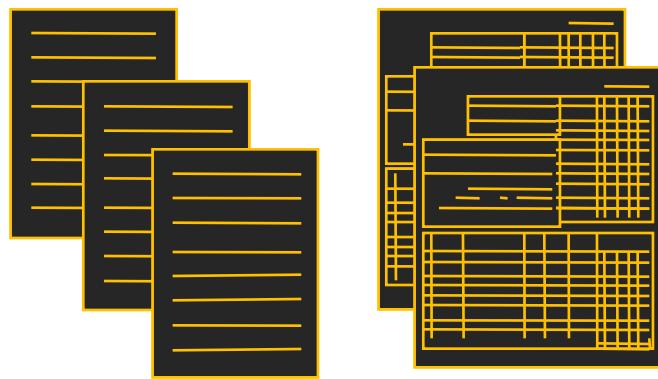
Il processo consta di quattro fasi:

- 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)**

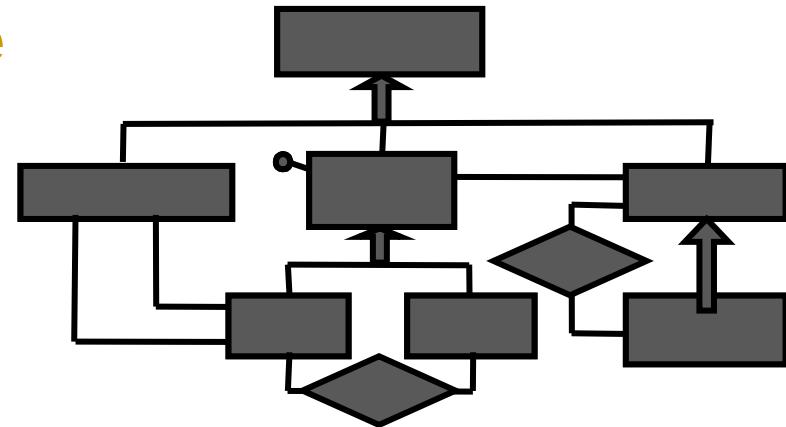




Requisiti



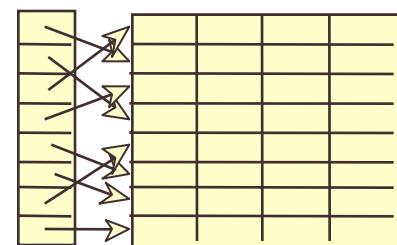
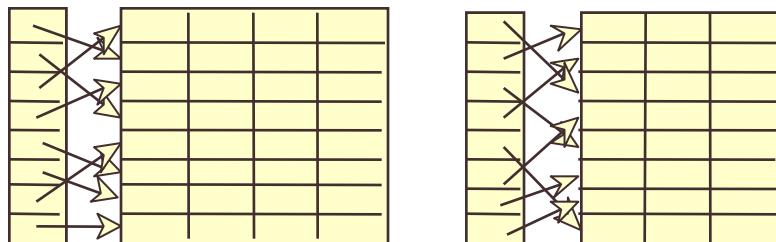
Schema Concettuale



Progettazione concettuale



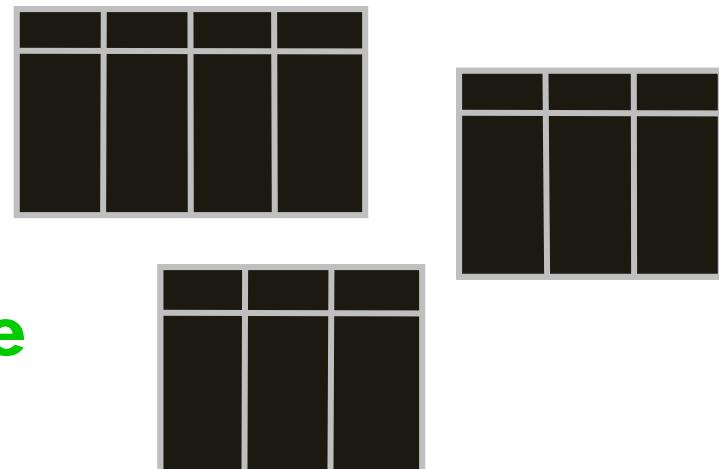
Schema Fisico



Progettazione logica



Schema Logico



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 (Relazioni)
 - I vincoli che i dati memorizzati nel database dovrebbero soddisfare per garantire la qualità e la coerenza dell'informazione (Vincoli)
 - 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
 - Non siamo ancora nella fase in cui questa attività è eseguita



DOCUMENTAZIONE ALLA PROGETTAZIONE CONCETTUALE

- Per la progettazione concettuale si utilizzeranno i diagrammi Entity-Relationship (ER)
 - ...ma vedremo anche i Class Diagram UML
- I diagrammi ER sono lo standard per la descrizione concettuale delle basi di dati
 - Sono comunque facilmente esprimibili mediante Class Diagram UML
- I Class Diagram di UML sono lo standard per la rappresentazione delle strutture dati nella programmazione ad oggetti
 - Sono indipendenti dal modello dei dati relazionale e agevolano le descrizioni per il modello relazionale
- Dizionari delle entità e delle relazioni
 - Descrivono e commentano integrando la presentazione del diagramma ER
- Dizionario dei vincoli
- Facoltativo
 - Dizionario delle interrogazioni e indicazione della loro frequenza

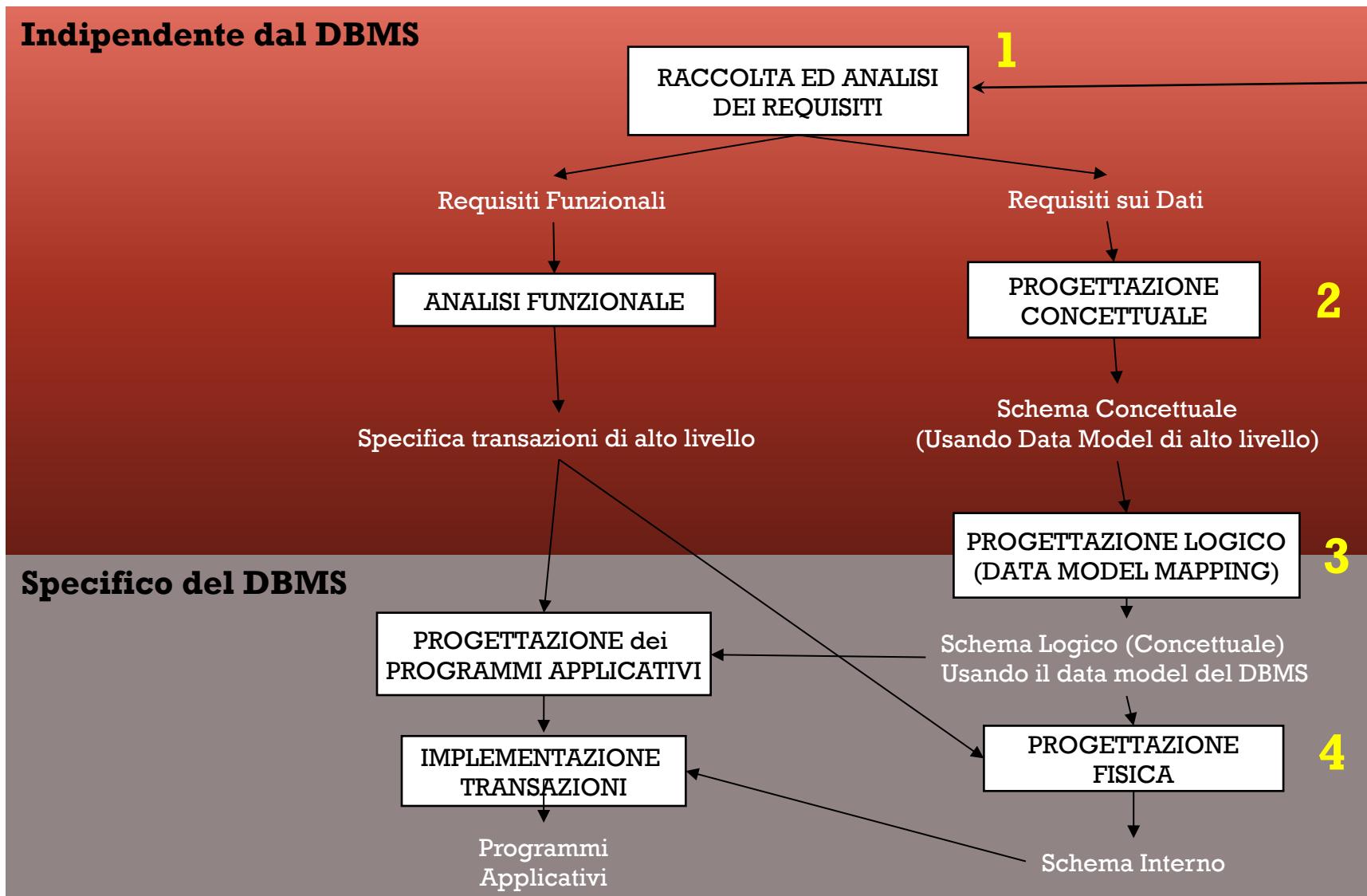


PASSI SUCCESSIVI ALLA PROGETTAZIONE CONCETTUALE

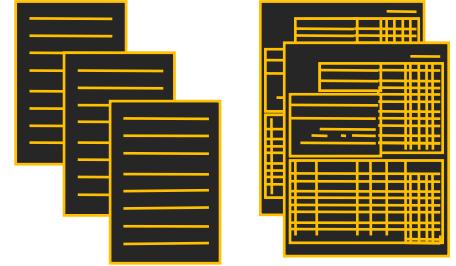
- Progettazione logica
 - Scelta del modello dei dati
 - Codifica delle entità ed associazioni nel modello dei dati
- Progettazione fisica
 - Scelta del DBMS
 - Definizione dei metadati utilizzando il DDL del DBMS
 - Implementazione dei vincoli di consistenza
 - Definizione dei dettagli implementativi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



LE FASI DI PROGETTAZIONE DI UN DATABASE



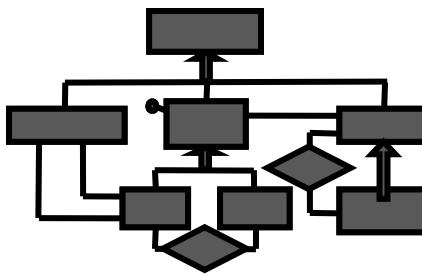
1. RACCOLTA E ANALISI DEI REQUISITI



- Il progettista del DB intervista i potenziali utenti dei DB per capire e documentare i requisiti utente.
- L'output di questa fase è duplice
 - Base di dati
 - Requisiti sui dati
 - Applicativo
 - Requisiti funzionali
 - Operazioni definite dall'utente (transazioni) che saranno applicate al DB (aggiornamenti, ricerche, ...).
 - Specifica dei requisiti funzionali attraverso Diagrammi di flusso, Scenari, Diagrammi di sequenza,



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*).

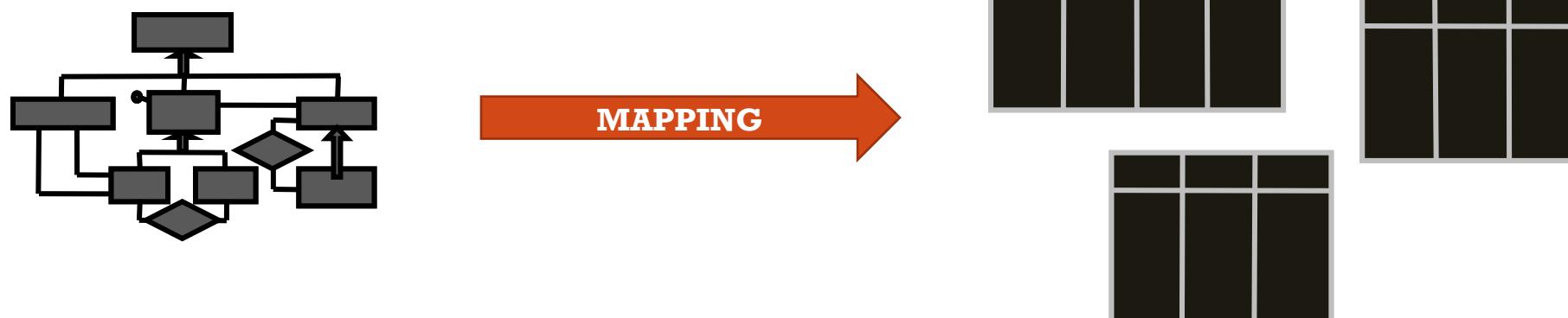


3. DISEGNO DEL DATABASE LOGICO

- Implementazione del database usando un DBMS commerciale.
 - Detto anche **data model mapping** (o **progettazione logica**).
 - 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.

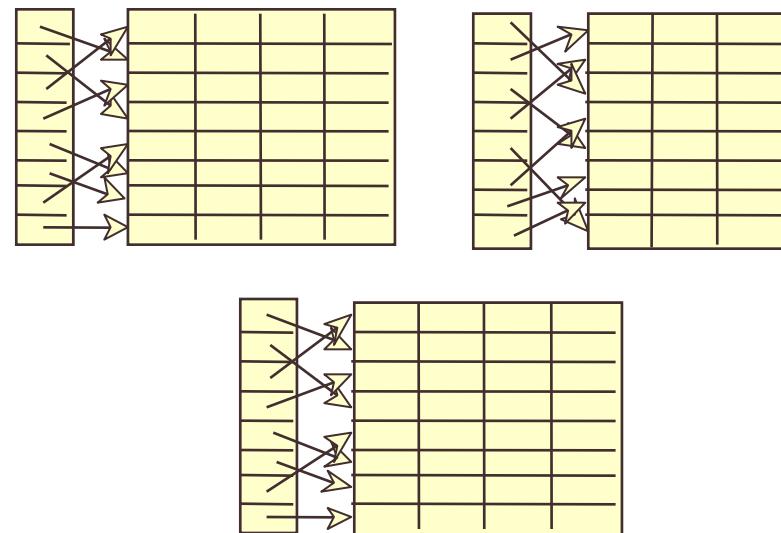
Output:

- 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 CONCETTUALE 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
- Nel corso vedremo entrambi i modelli.



MODELLO ER

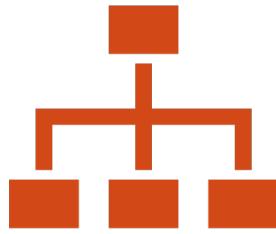
- Il modello Entità-Relazione (ER) è un diffusissimo data model di alto livello, estesamente utilizzato per definire lo schema concettuale di un database.
- È stato concepito per essere più vicino ai concetti “*umani*”, e quindi facilmente comprensibile anche ad utenti non tecnici.
- Il modello ER ha 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 (Class Diagram)* 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

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 una base di dati
costrutti funzionali alla modellazione di
basi di dati

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 della stessa applicazione



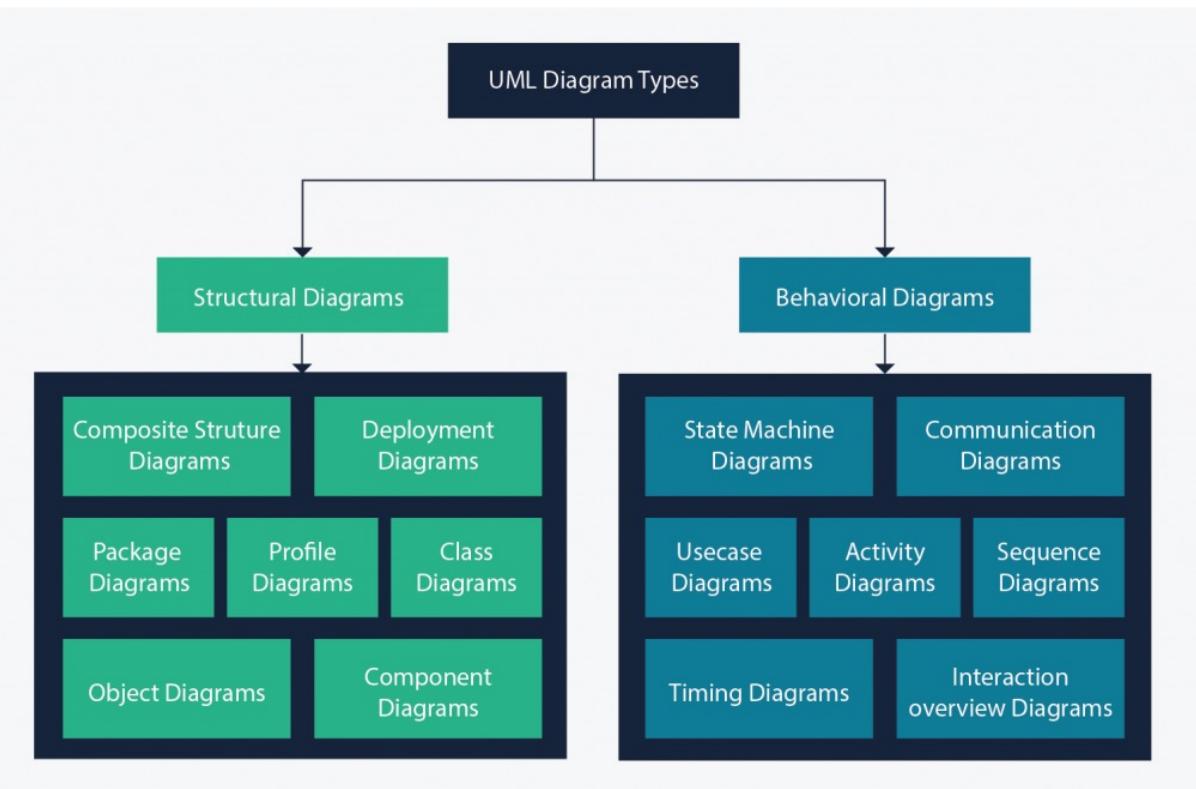
UML: LE ORIGINI

- 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 (14) di **diagrammi fondamentali** che descrivono aspetti strutturali e comportamentali di un'applicazione



UML: I DIAGRAMMI PRINCIPALI

- **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.
- **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.
- **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.



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



ESEMPIO: IL DB 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.



ENTITÀ, ATTRIBUTI E CHIAVI

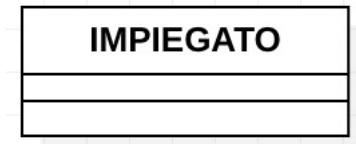


ENTITÀ

- 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 E può essere un oggetto fisico (casa, impiegato, ...) o un oggetto concettuale (un lavoro, una società, ...).



Rappresentazione ER
dell'entità “*Impiegato*”



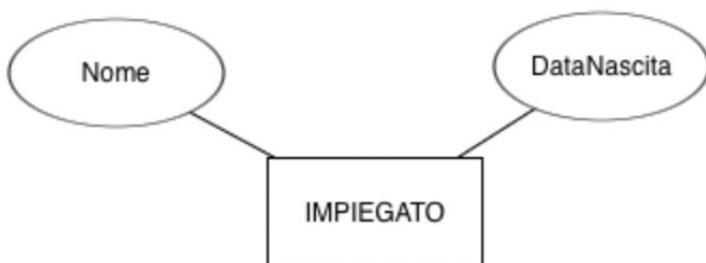
Rappresentazione UML
della classe “*Impiegato*”

In UML, l'entità è
chiamata *Classe*

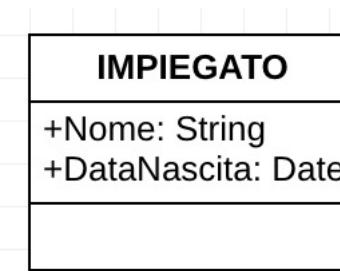


ATTRIBUTI

- Ogni entità E ha delle proprietà dette **attributi $A_1, A_2, A_3, \dots, A_n$** .
 - **E ($A_1, A_2, A_3, \dots, A_n$)**
 - *Es:* l'entità *Impiegato* ha attributi **nome, cognome, età, indirizzo, salario, telefono, ...**
- Ogni entità è caratterizzata da un set di valori per i suoi attributi.
 - Nel diagramma ER, un attributo descrive una proprietà elementare di un'entità o di una relazione.
 - Nel Class Diagram, un attributo rappresenta una proprietà elementare degli oggetti di una classe.



Rappresentazione ER dell'entità “*Impiegato*”
con gli attributi “*Nome*” e “*DataNascita*”

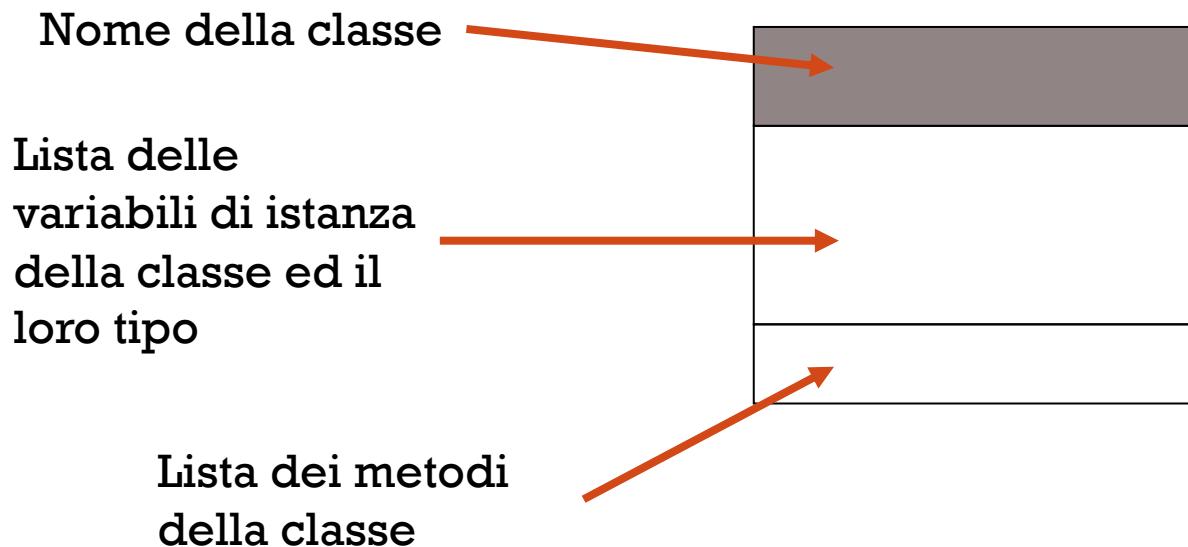


Rappresentazione UML della classe “*Impiegato*”
con gli attributi “*Nome*” e “*DataNascita*”



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.

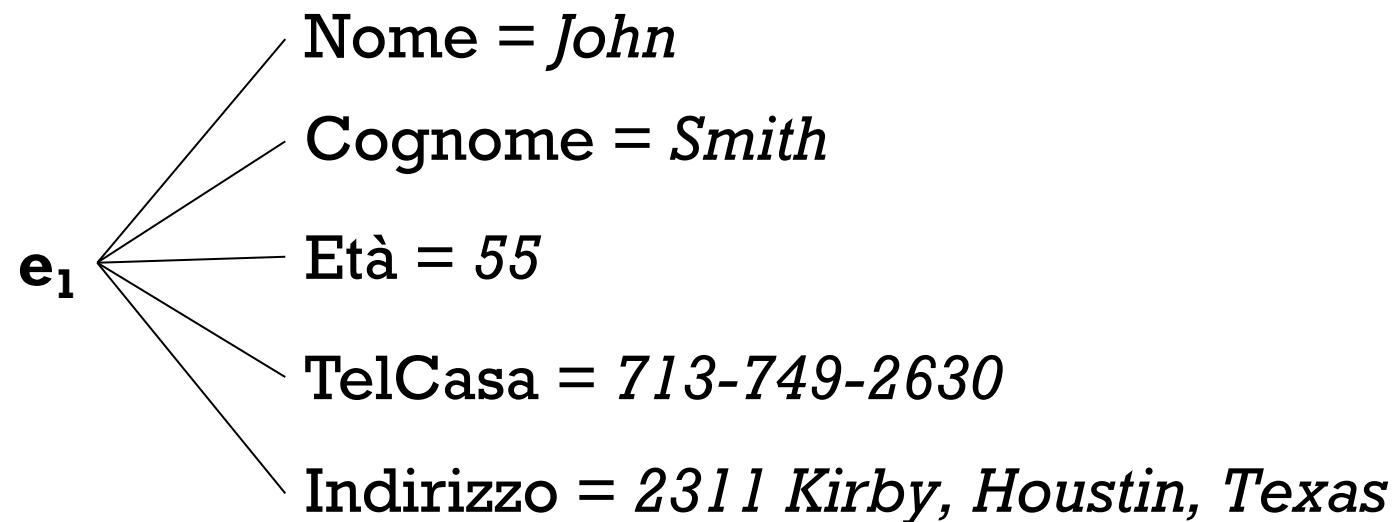


IMPIEGATO
+Nome: String
+Cognome: String
+DataNascita: Date
+CodiceFiscale: String
+Sesso: char
+Stipendio: double
+Indirizzo: String
+SetStipendio()



ENTITÀ E ATTRIBUTI

- **Esempio:** entità *Impiegato*



TIPI DI ATTRIBUTI

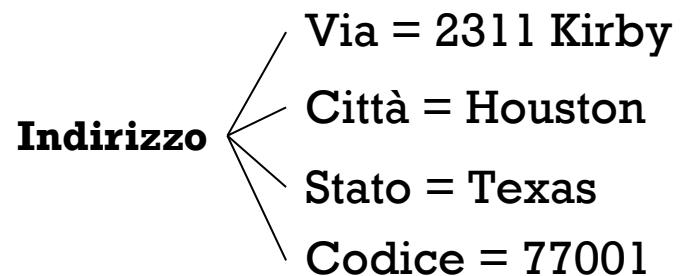
Gli attributi possono essere di più tipi

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



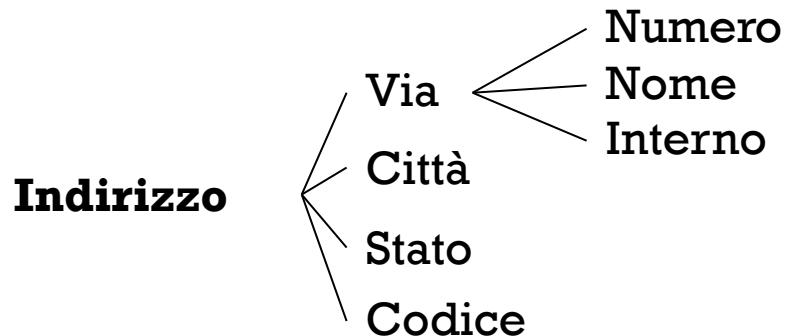
ATTRIBUTI SEMPLICI E COMPOSTI

- **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.
 - *Es.* l'attributo **Indirizzo** dell'entità **IMPIEGATO**:
 - Denotato come *Indirizzo(Via, Città, Stato, Codice)*



ATTRIBUTI SEMPLICI E COMPOSTI (2)

- Gli attributi composti possono formare una gerarchia:

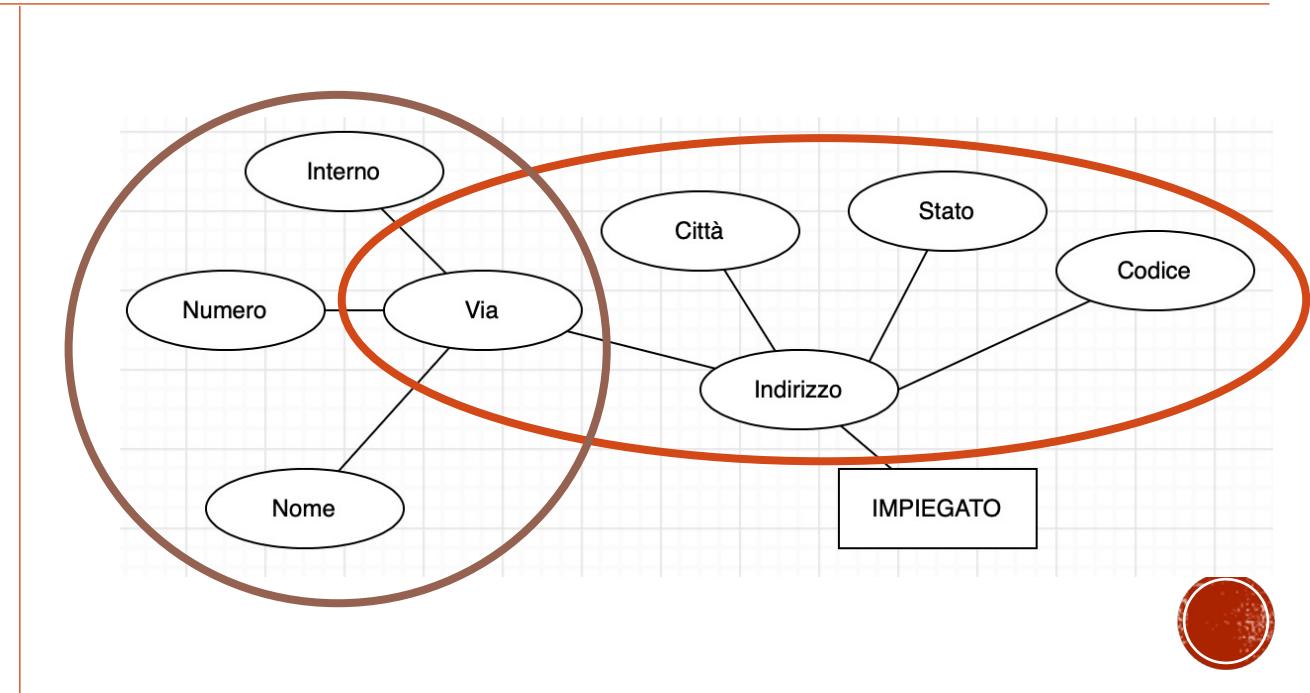
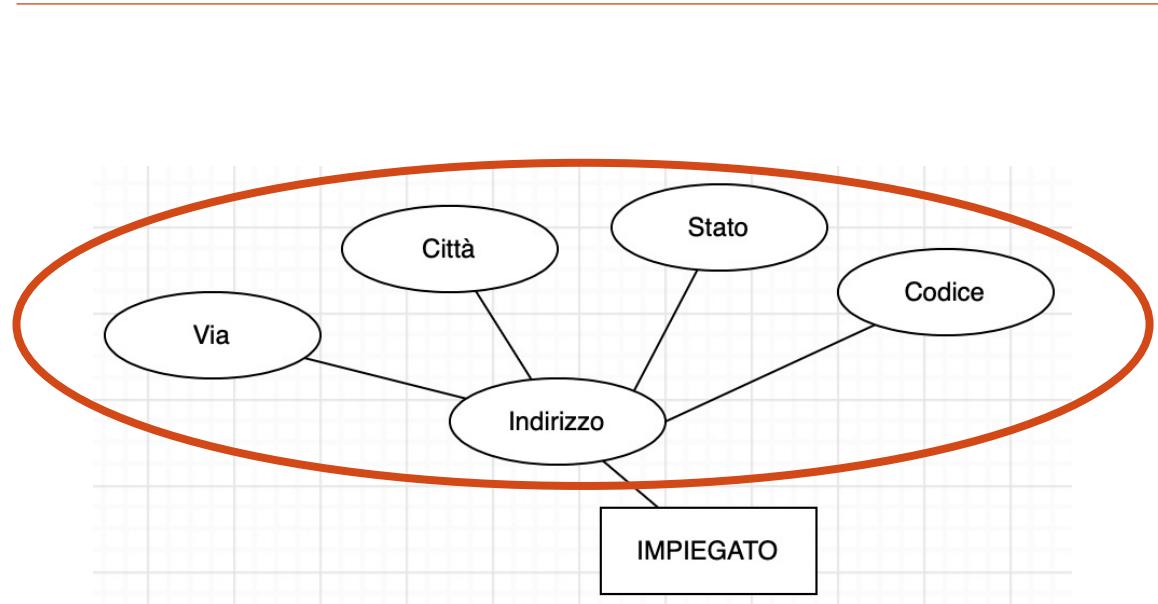
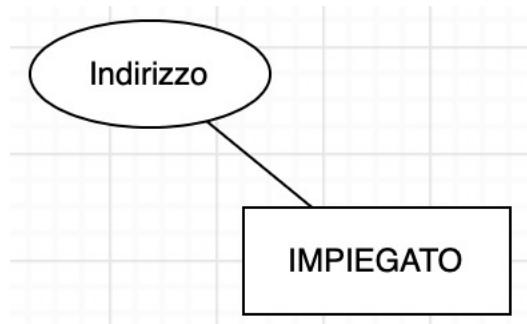


Indirizzo(Via(Numero, Nome, Interno), Città, Stato, Codice)

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

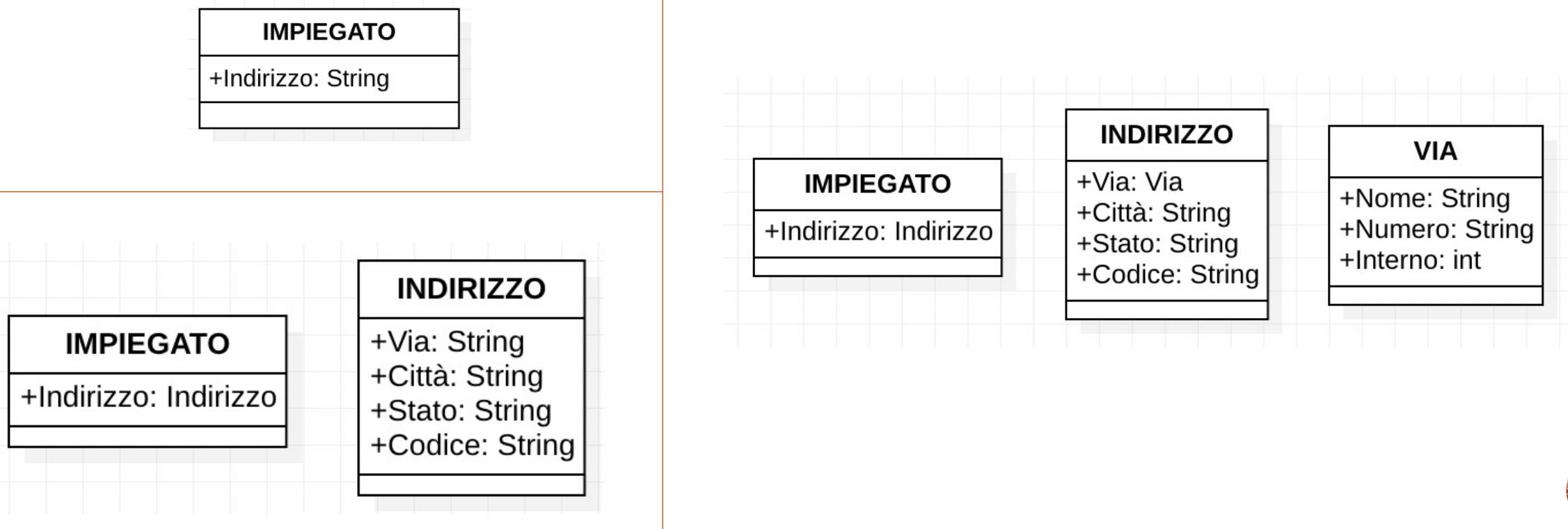


ATTRIBUTI SEMPLICI E COMPOSTI (ER)



ATTRIBUTI SEMPLICI E COMPOSTI (UML)

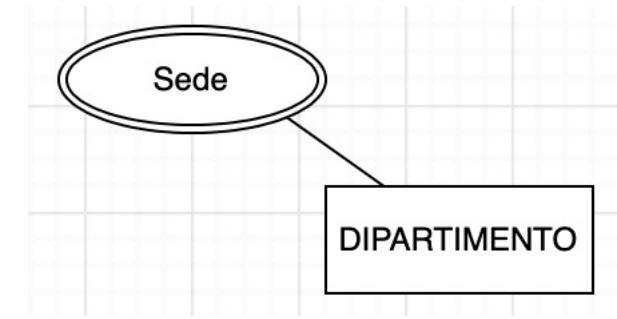
- In UML non c'è una notazione specifica per la rappresentazione di attributi multivalore



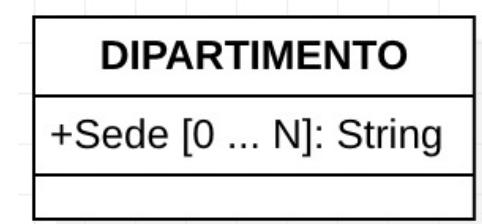
SINGLE-VALUED E MULTIVALUED

- *Attributi single-valued:*
 - hanno un solo valore per ciascuna entità.
 - **Es:** l'età di un impiegato.

- *Attributi multivalued:*
 - può avere un insieme di valori per la stessa entità.
 - **Es:** l'attributo **Sede** per un'entità Dipartimento.
 - Denotato come Dipartimento({Sede})
 - In UML, può essere determinato un limite inferiore e un limite superiore al numero di valori per un'entità.



ER

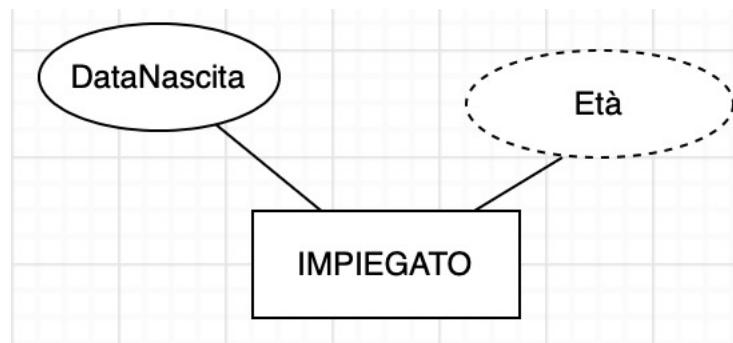


UML



ARCHIVIATI E DERIVATI DA ALTRI ATTRIBUTI

- *Attributi derivati:* alcuni attributi possono essere in relazione tra loro.
 - **Es:** l'età e la data di nascita di un impiegato:
 - DataNascita **attributo memorizzato**
 - Età **attributo derivato**
- Alcuni valori di attributi possono essere derivati da entità correlate.
 - **Es:** NumeroImpiegati di una entità “*dipartimento*” può essere derivato contando il numero di impiegati relati al (che lavorano per) dipartimento

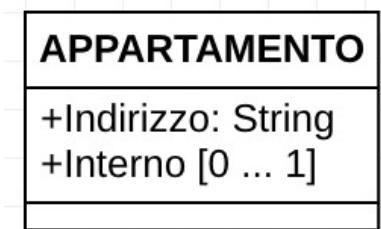
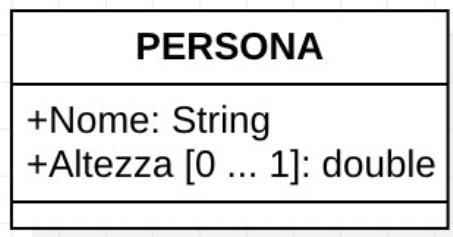


IMPIEGATO
+DataNascita: Date «derive»+Eta: int



VALORI NULLI

- Alcuni attributi possono avere valore **null** col significato di “*non noto*” o “*mancante*” o “*non applicabile*”.
 - L’attributo può esistere ma non essere noto: **Es.** l’altezza di una persona
 - L’attributo può non esistere: **Es.** l’interno di un appartamento per una casa unifamiliare
- Nel diagramma ER non c’è una vera e propria notazione per rappresentare i valori **NULL**.
- Nel diagramma UML si può utilizzare la stessa notazione per gli attributi multi-valore.



ATTRIBUTI COMPLESSI

- In alcuni casi, gli attributi composti e gli attributi multi-valore possono essere annidati in modo arbitrario
 - **Es.** Una persona può avere più di un indirizzo ed ogni indirizzo può avere più di un telefono, e sia telefono che indirizzo sono attributi composti

Persona (Nome, Cognome, DataN, { IndirizzoTelefono ({ Telefono (Prefisso, NumTelefono) } , Indirizzo (IndirizzoVia (Numero, Via, NumeroAppartamento), Città, Stato, CAP)) })

Per la modellazione si utilizzano le convenzioni mostrate prima per gli attributi composti e quelli multi-valore.



TIPI E INSIEMI DI ENTITÀ E CHIAVI



INTENSIONE ED ESTENSIONE

- 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

Un tipo di entità descrive lo schema
(o intenzione) per un insieme di entità
che possiedono gli stessi attributi

Insieme di Entità
(Estensione)

Nome, Cognome, Età, Stipendio

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

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

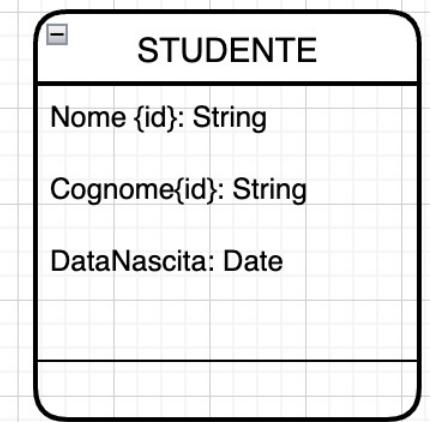
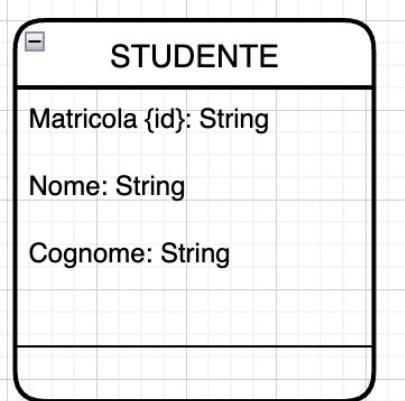


ATTRIBUTI CHIAVE DI UN TIPO DI ENTITÀ

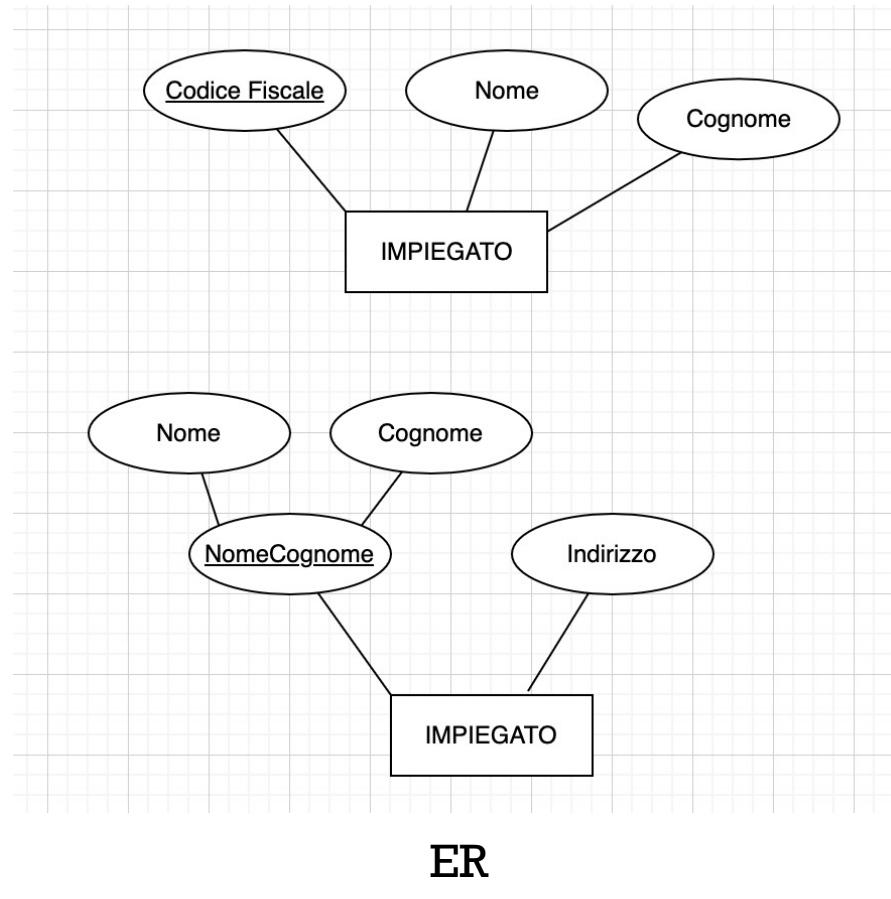
- Un importante vincolo sulle entità di un tipo di entità è la chiave o vincolo di unicità.
 - L'attributo chiave di un tipo di entità è un attributo che deve avere un valore univoco per cui ogni entità.
 - Serve per identificare univocamente una singola entità
 - **Es.:** Il codice fiscale di una persona.
- 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.
- Il vincolo di chiave per un'entità proibisce a qualsiasi coppia di entità di avere contemporaneamente lo stesso valore per l'attributo chiave.
- **Il vincolo di chiave deriva dai vincoli del mini-mondo**
- Vi possono essere entità che non hanno attributi chiave
 - In tal caso questo tipo di entità è chiamato *entità debole* (che vedremo nel seguito)



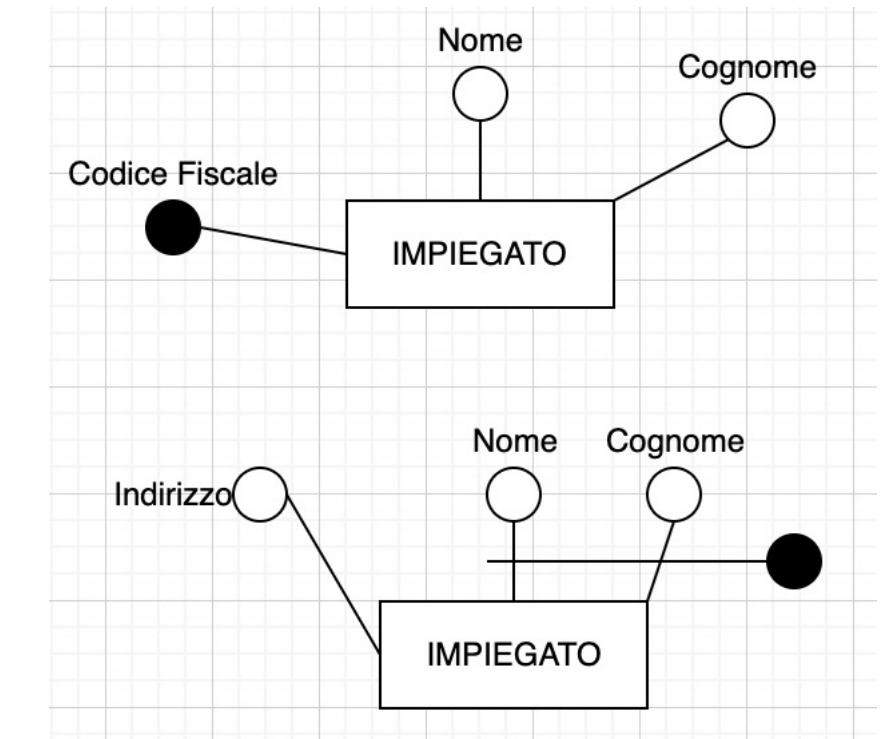
ATTRIBUTI CHIAVE IN UML E ER



UML



ER
(Elmasri, Navathe)



ER
(Atzeni, Ceri, Paraboschi, ...)



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 .
- Per un attributo singolo
 - $A(e)$ il valore dell'attributo A per l'entità e. $A(e)$ viene detto **singleton**
- 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,



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)**

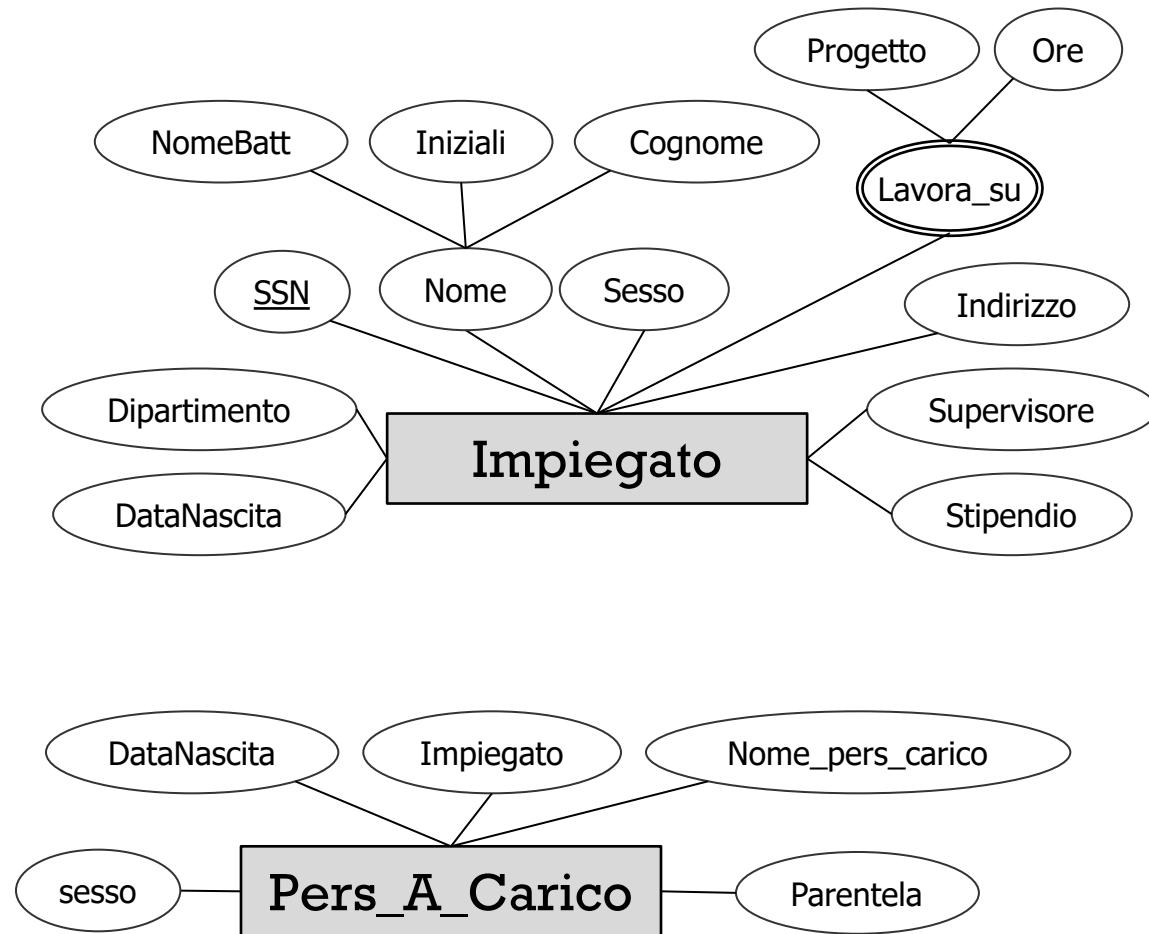
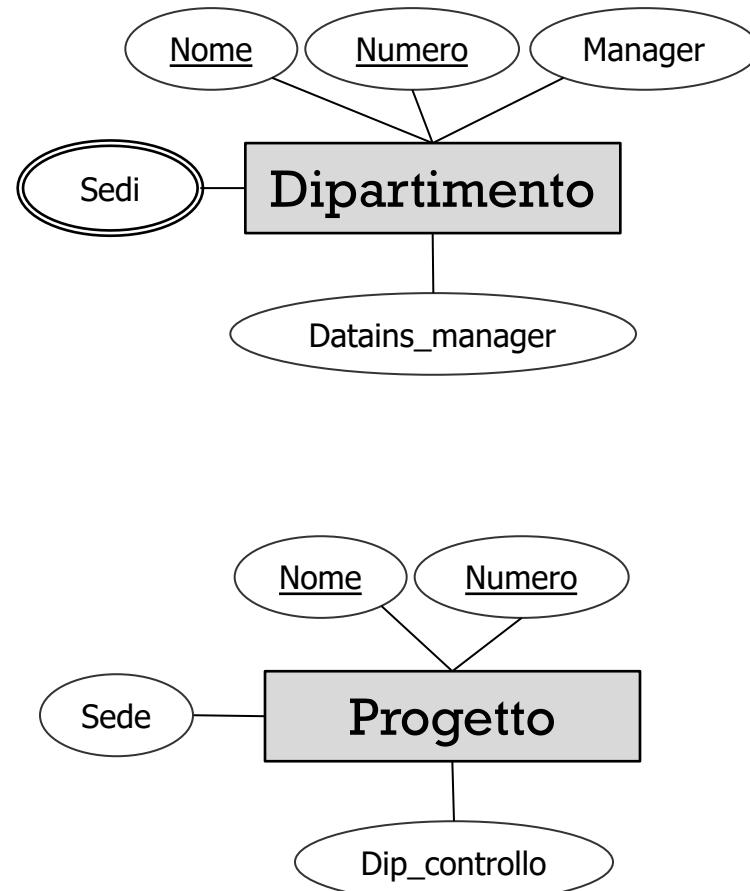


DISEGNO CONCETTUALE DEL DATABASE COMPANY (4)

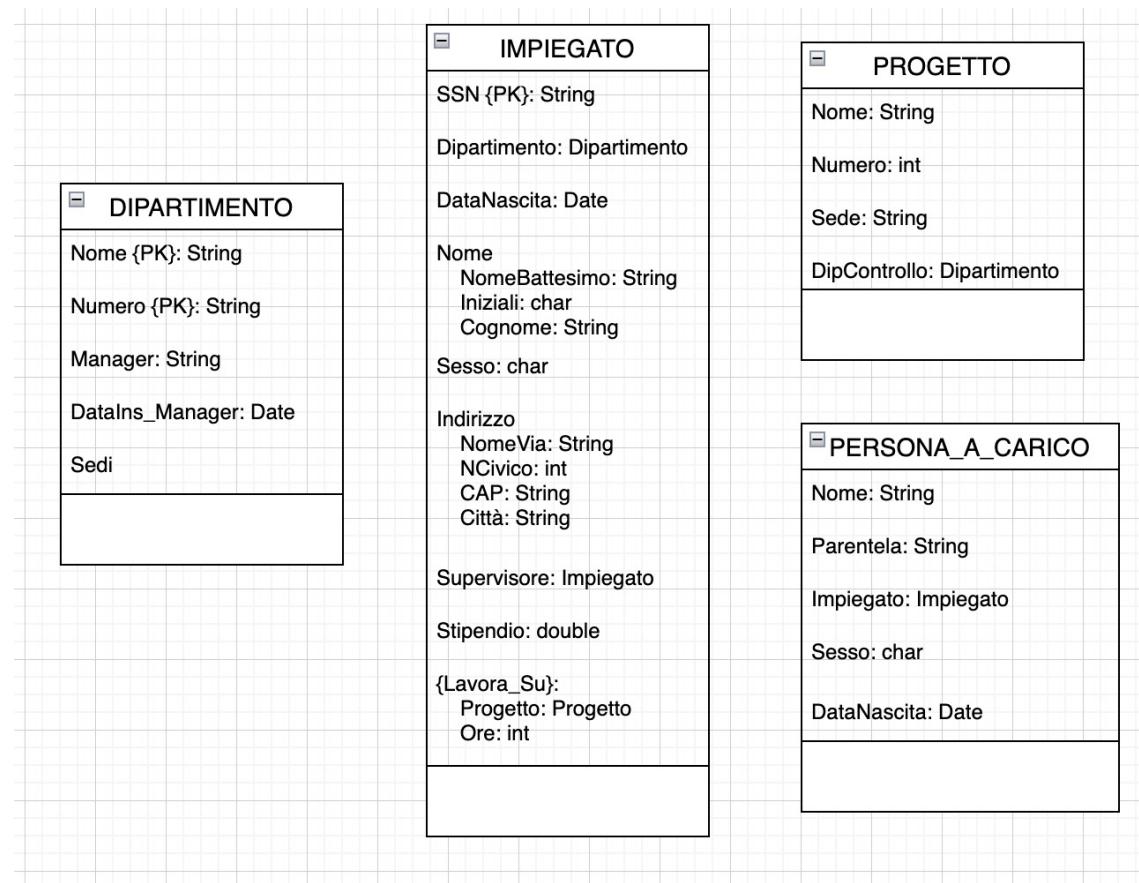
- Esistono varie relazioni implicite:
 - L'attributo Manager di DIPARTIMENTO si riferisce a un impiegato che gestisce il dipartimento;
 - L'attributo Dip_controllo di PROGETTO si riferisce al dipartimento che controlla il progetto;
 - L'attributo Dipartimento di IMPIEGATO si riferisce al dipartimento 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



RELAZIONI, RUOLI E VINCOLI (ER)



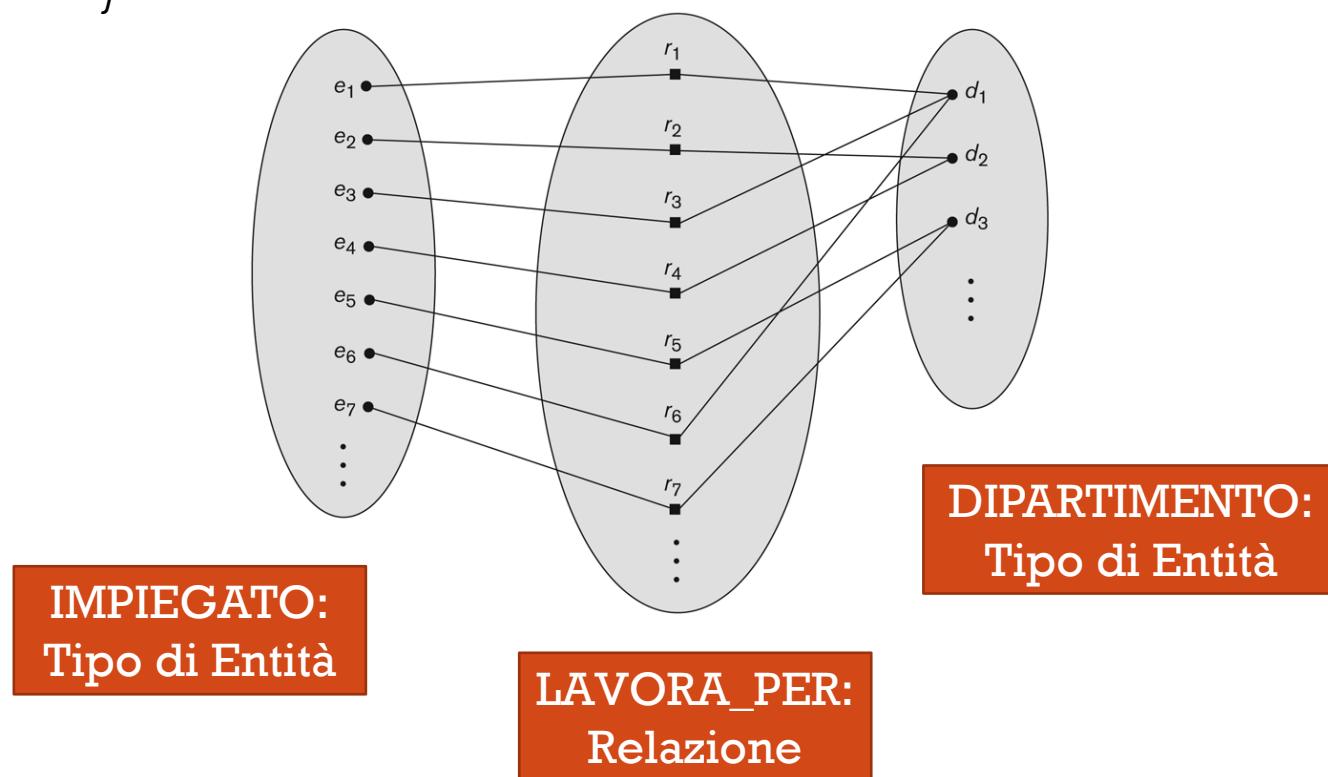
TIPI E Istanze di RELAZIONI

- Le relazioni corrispondono a legami logici tra entità, significativi ai fini dell'applicazione di interesse.
- Un **tipo di relazione R** 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.



TIPI E Istanze DI RELAZIONI (2)

- **Es.:** 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 .



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 *partecipante* al tipo di relazione **R**.
- Ogni entità individuale e_j è detta partecipante all'istanza di relazione

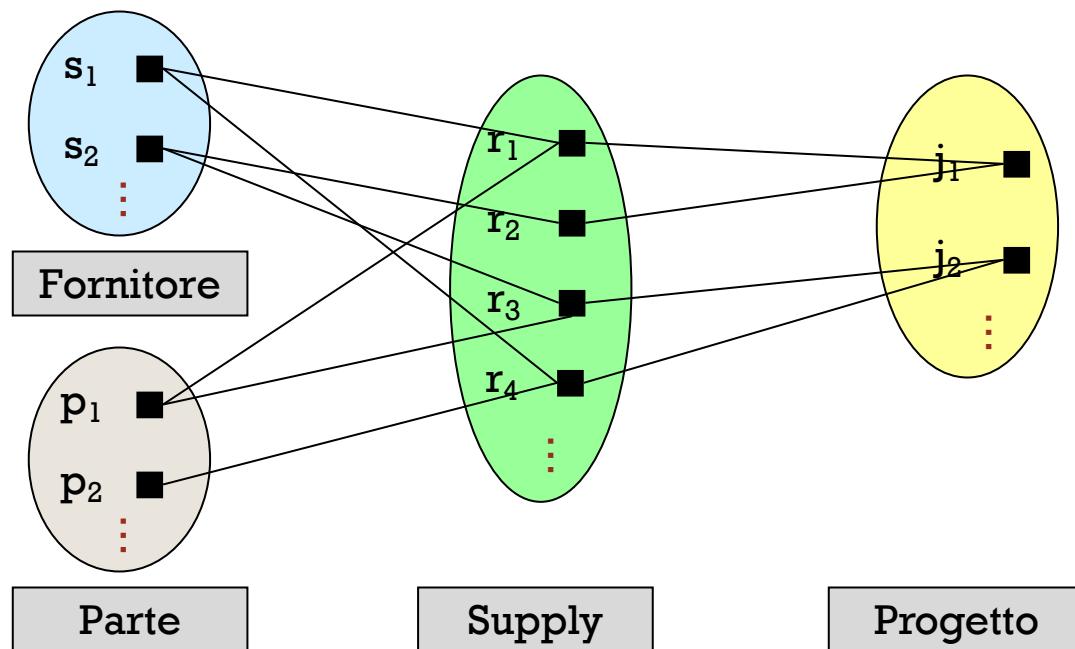
$$r_i = (e_1, e_2, \dots, e_n).$$

- 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.



GRADO DI UN TIPO DI RELAZIONE

- Il grado di un tipo di relazione è il numero di tipi di entità partecipanti.
 - Es.: LAVORA_PER* è di grado 2 (binaria).
 - Es.: 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 .

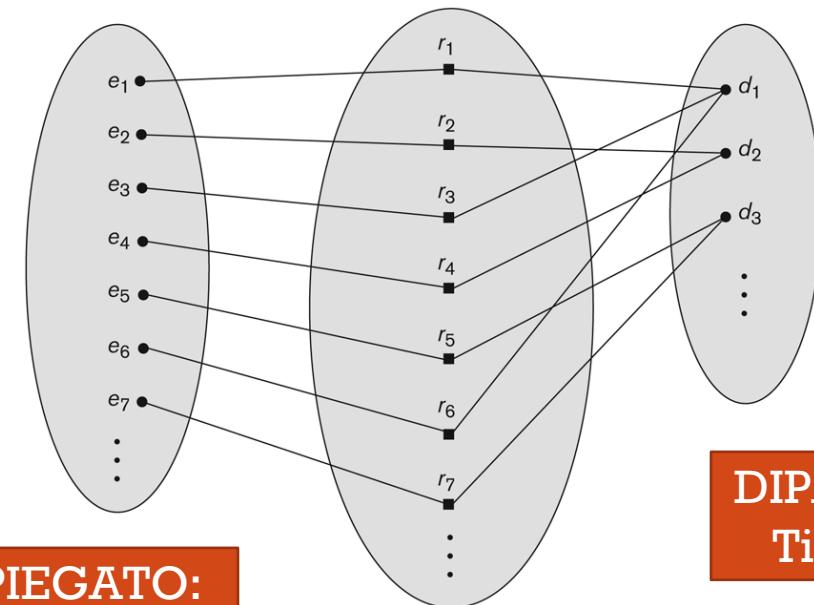


Le relazioni possono essere di qualsiasi grado ma le più ricorrenti sono quelle binarie.



RELAZIONI COME ATTRIBUTI

- Può venire da pensare ad una relazione come ad un attributo di una determinata entità.
- Un IMPIEGATO può avere un attributo *Dipartimento*, il cui valore è un *riferimento* all'entità DIPARTIMENTO per cui lavora l'impiegato
 - L'insieme dei valori per l'attributo *Dipartimento* è l'insieme entità DIPARTIMENTO.
 - Analogamente possiamo inserire la lista di IMPIEGATI come attributo multivalore per ogni entità DIPARTIMENTO!
 - Se facciamo entrambi, siamo vincolati a mantenere gli attributi consistenti tra loro.
 - **Soluzione: RELAZIONI!**



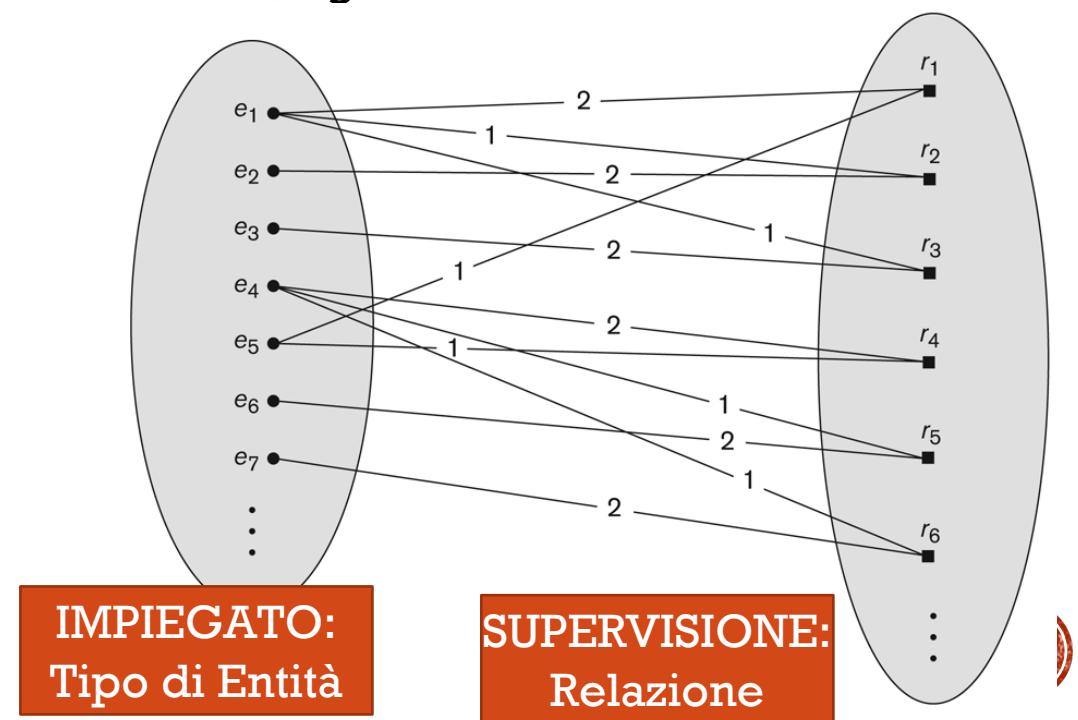
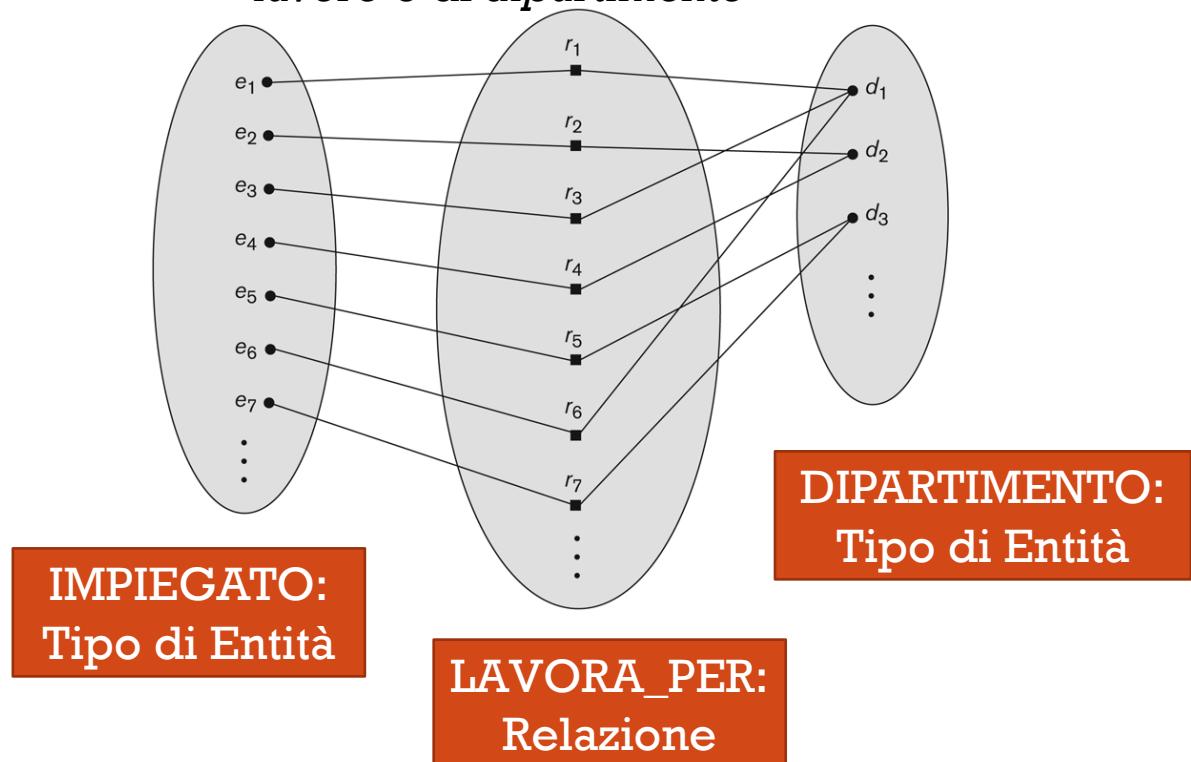
IMPIEGATO:
Tipo di Entità

LAVORA_PER:
Relazione

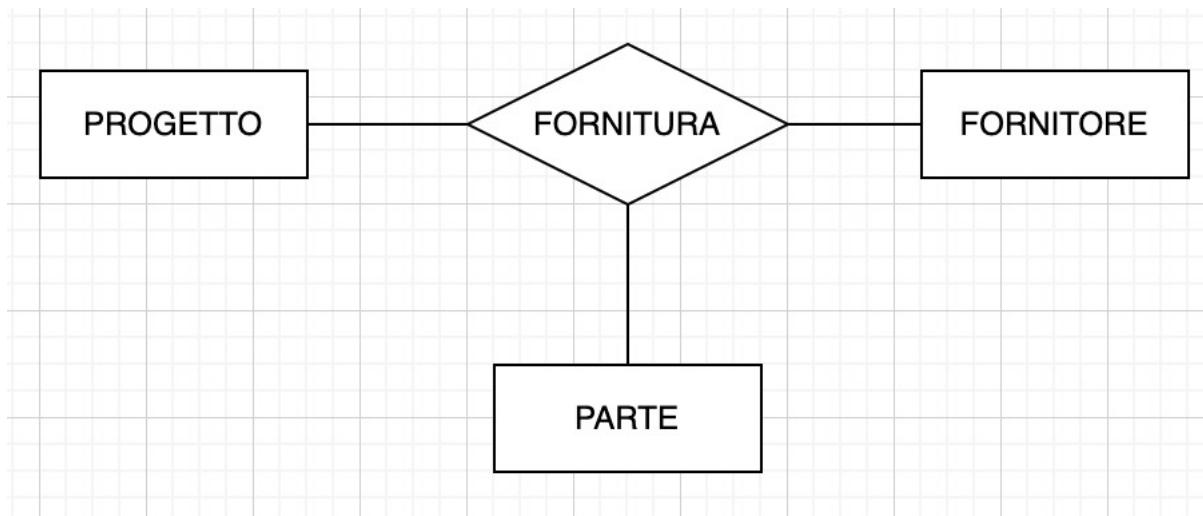
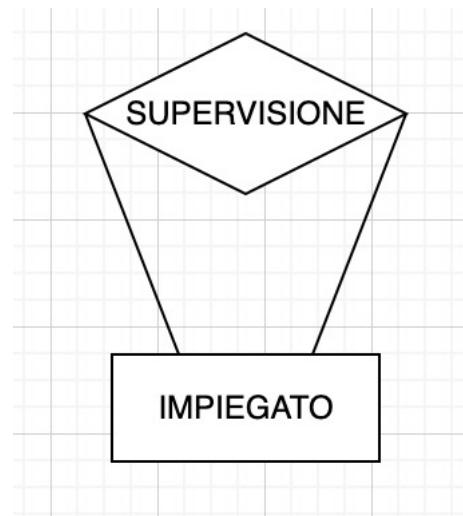
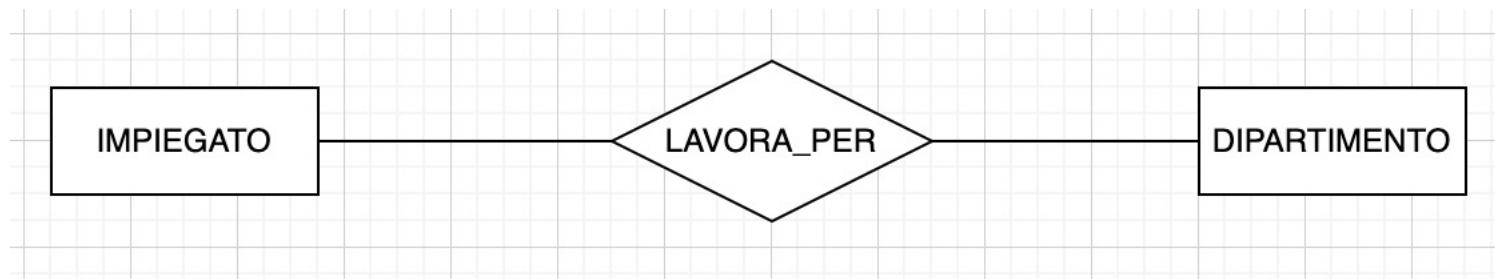
DIPARTIMENTO:
Tipo di Entità

RUOLI E RELAZIONI RICORSIVE

- Ogni entità partecipa ad una relazione recitando un **ruolo** per quella relazione.
- Nella relazione **LAVORA_PER**:
 - l'**IMPIEGATO** ricopre il ruolo di *lavoratore* o di *impiegato*
 - Il **DIPARTIMENTO** ha il ruolo di *datore di lavoro* o di *dipartimento*
- Il nome di ruolo è importante per specificare il ruolo che ha un entità in una relazione.
- In alcuni casi un'entità partecipa più volte ad una relazione, ogni volta con ruoli diversi



RAPPRESENTAZIONE DI RELAZIONI (ER)



RELAZIONI E VINCOLI

- Le relazioni hanno dei vincoli che limitano le possibilità di combinazione tra le entità nella formazione di relazioni
- I vincoli sono determinati dal **MINIMONDO** che andiamo a rappresentare
- **Es.:** Un'azienda può avere i seguenti vincoli:
 - Un impiegato può lavorare per uno ed un solo dipartimento
 - Un impiegato può avere solo un supervisore
 - Un dipartimento deve avere almeno una sede
 - Un dipartimento deve avere solo un dirigente
- È possibile distinguere due tipi di vincoli di relazione:
 - Vincolo di cardinalità
 - Vincolo di partecipazione



VINCOLI DI...

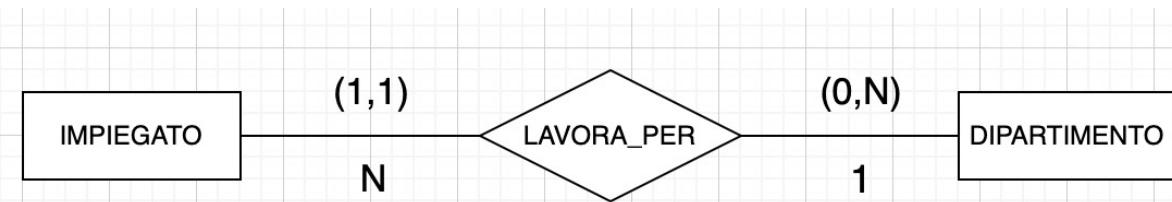
CARDINALITÀ

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



PARTECIPAZIONE

- Serve per specificare se l'esistenza di un'entità dipende dalla sua partecipazione ad una relazione. Permette di specificare il **numero minimo** di istanze di relazione a cui le entità coinvolte possono partecipare



"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: Ogni impiegato deve afferire ad almeno un dipartimento, ma un dipartimento può anche non avere impiegati

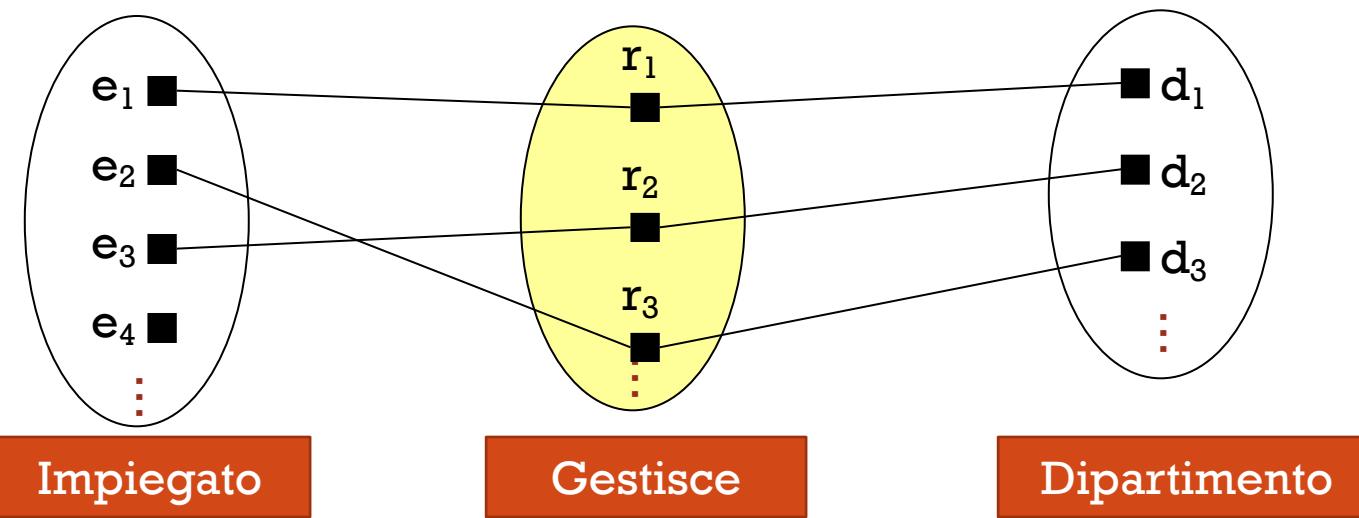
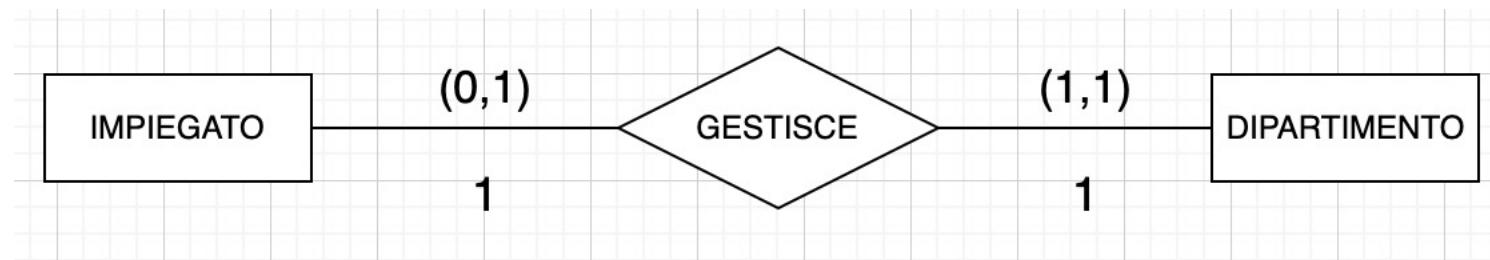


VINCOLI DI CARDINALITÀ E PARTECIPAZIONE

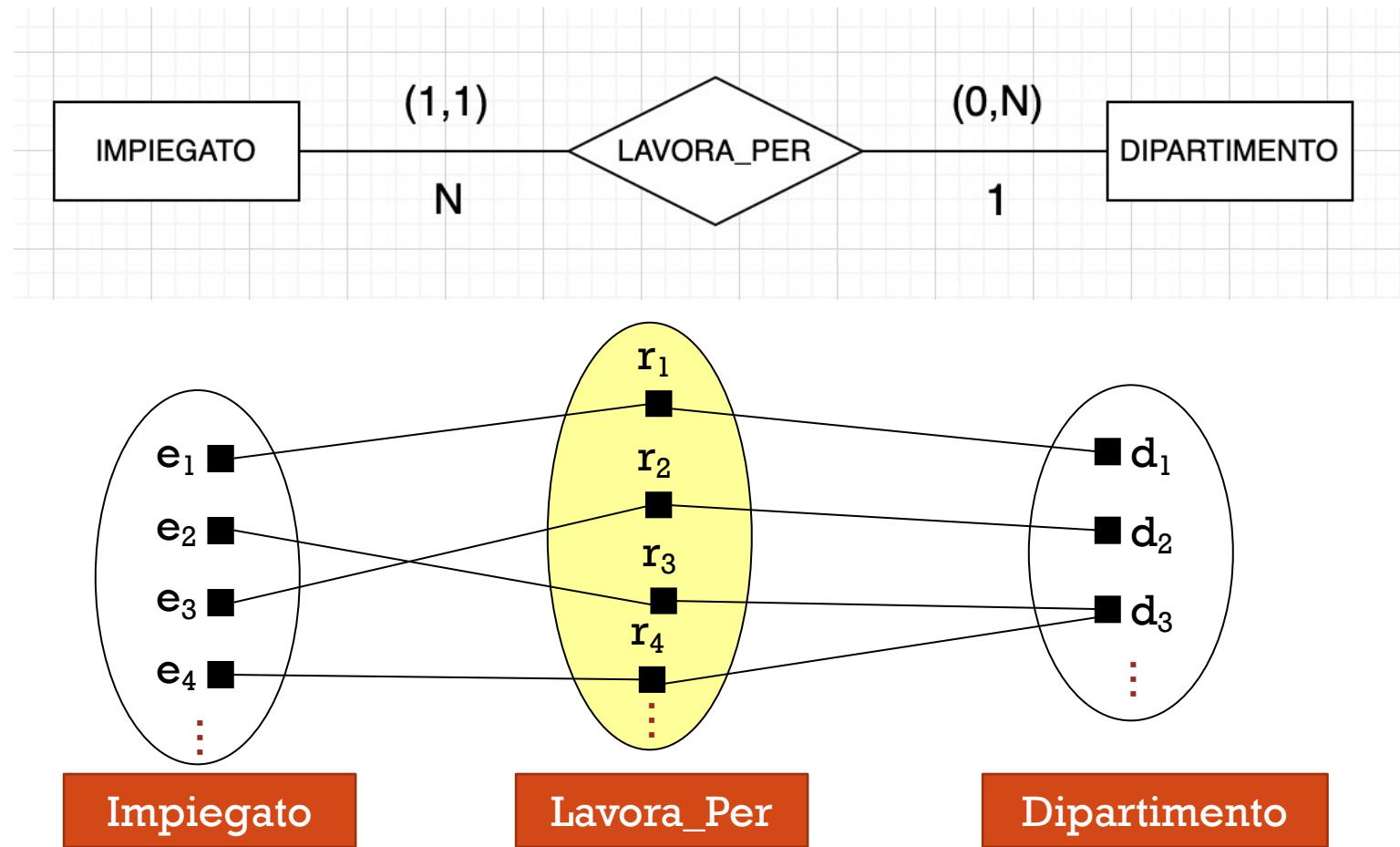
- È possibile assegnare un qualunque intero non negativo a un rapporto di cardinalità, con l'ovvio vincolo che la cardinalità minima deve essere minore o uguale alla cardinalità massima.
- Nella maggior parte dei casi si utilizzano solo tre valori: 0, 1 e N:
 - Il valore 0 per la cardinalità minima indica 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.



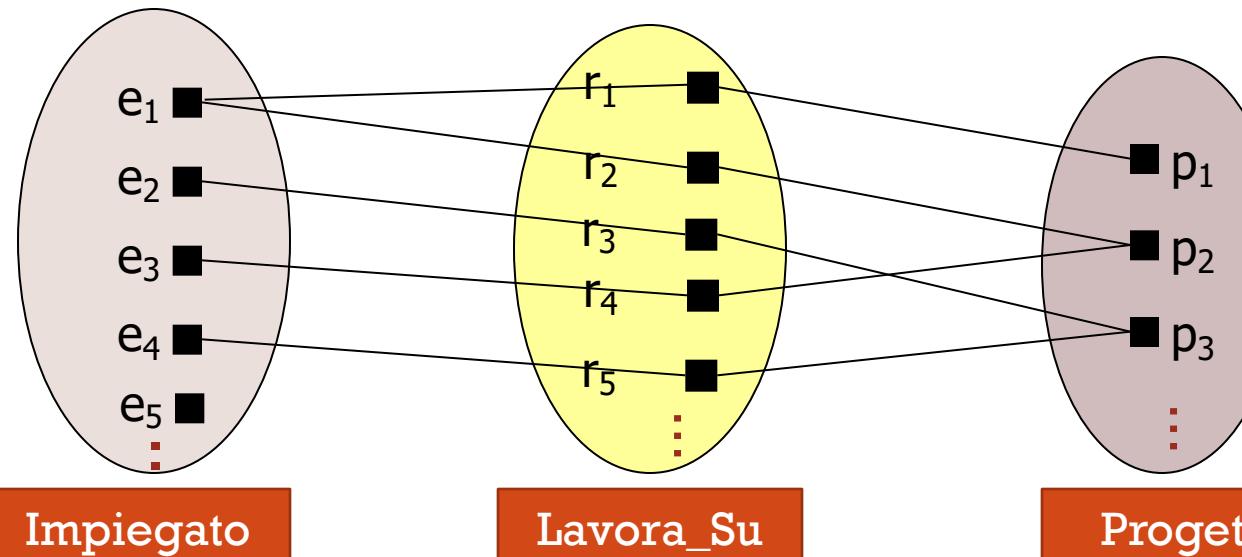
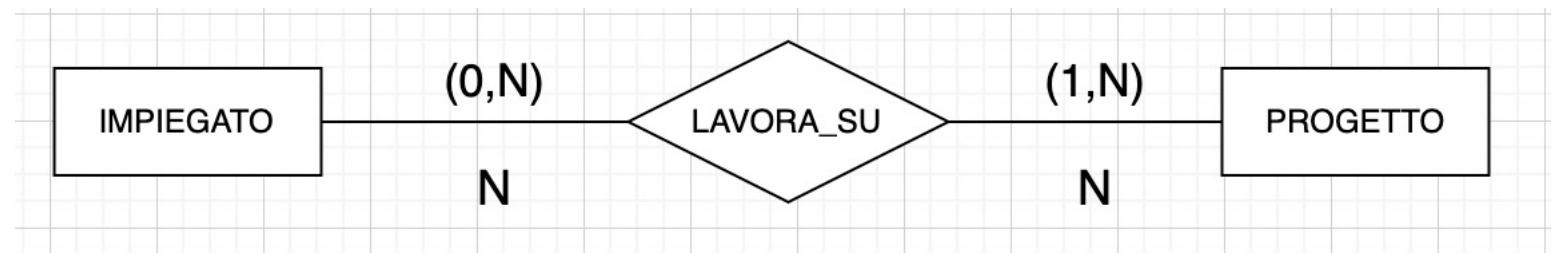
RELAZIONI 1:1



RELAZIONI 1:N (N:1)

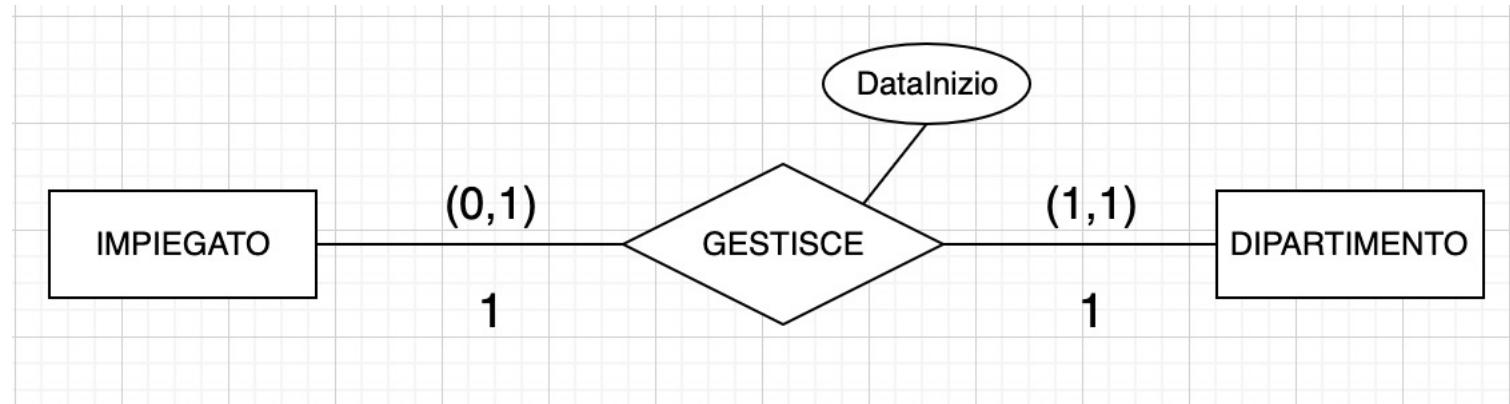
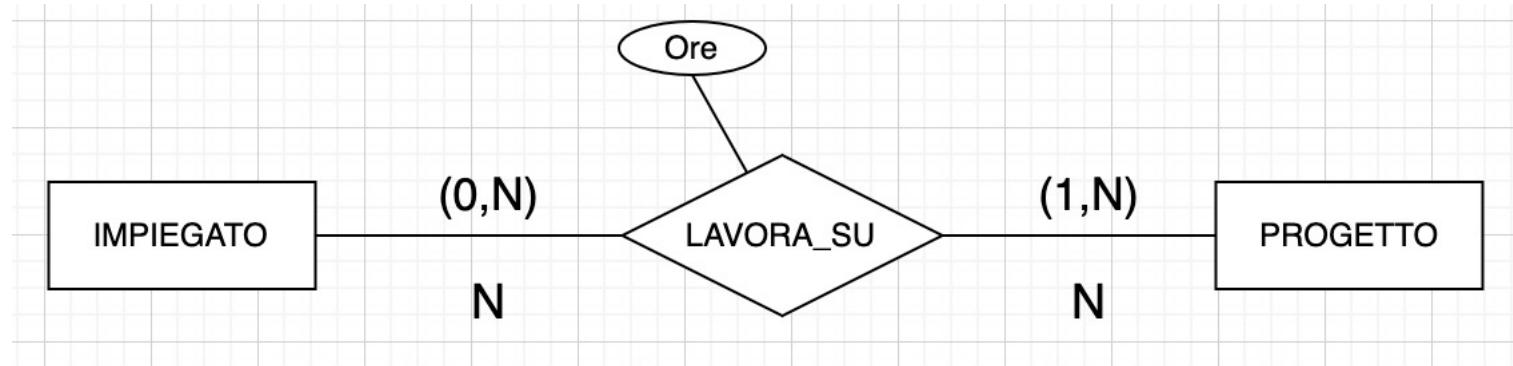


RELAZIONI N:N



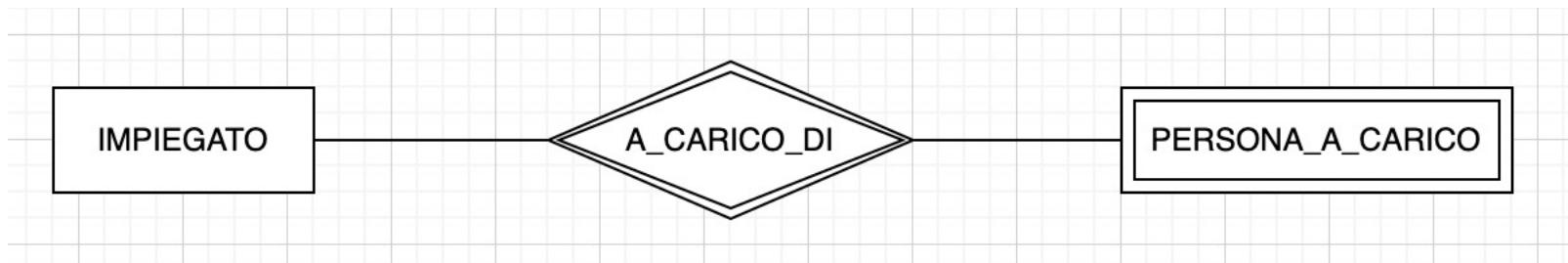
ATTRIBUTI DI RELAZIONI

- Anche le relazioni, come le entità, possono avere degli attributi che le descrivono



ENTITÀ DEBOLI

- Le **entità deboli** sono entità che non possiedono veri e propri attributi chiave
 - Quelle viste fino ad ora sono appunto dette **entità forti**
- Tali entità sono identificate tramite il loro collegamento con entità di un altro tipo
 - **Entità identificante o proprietario**
- La relazione che correla l'entità debole all'entità forte è detta **relazione identificante**
- Un tipo di entità debole ha solitamente una **chiave parziale**, ossia l'insieme degli attributi che identificano le entità deboli collegate alla stessa entità proprietaria.
 - Se non esiste, nel caso peggiore si avrà che tutti gli attributi dell'entità debole faranno parte della chiave parziale.

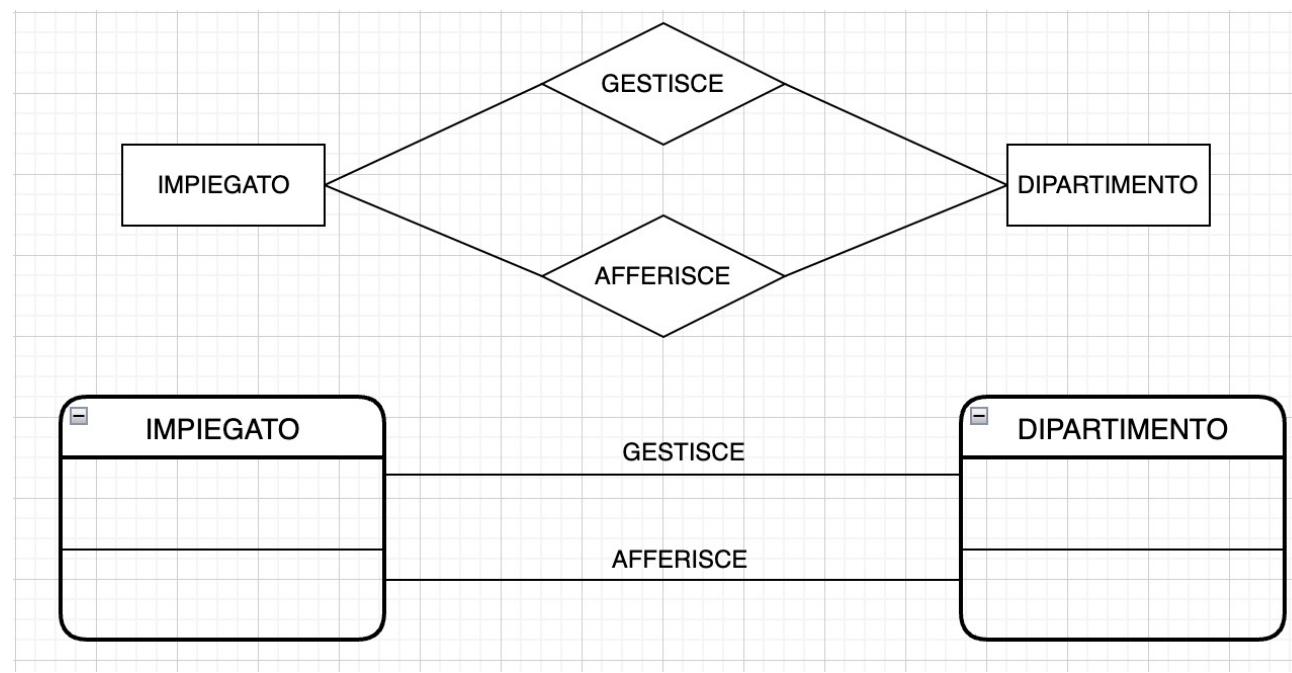


LE ASSOCIAZIONI (UML)



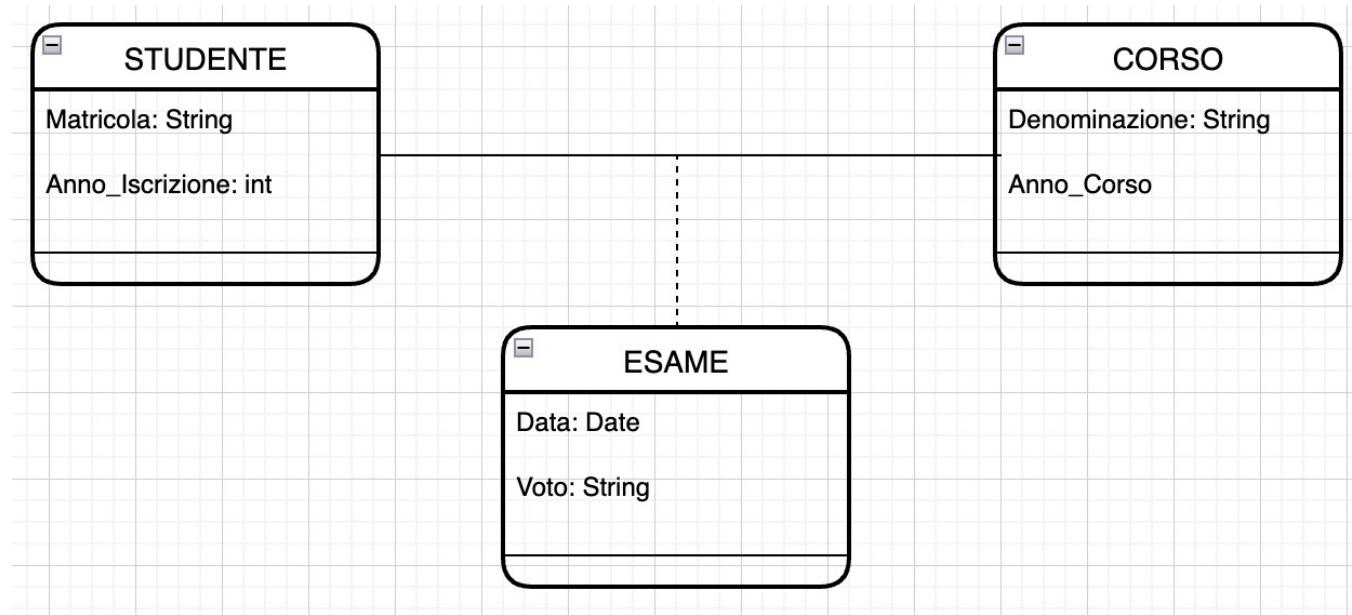
LE ASSOCIAZIONI

- **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.



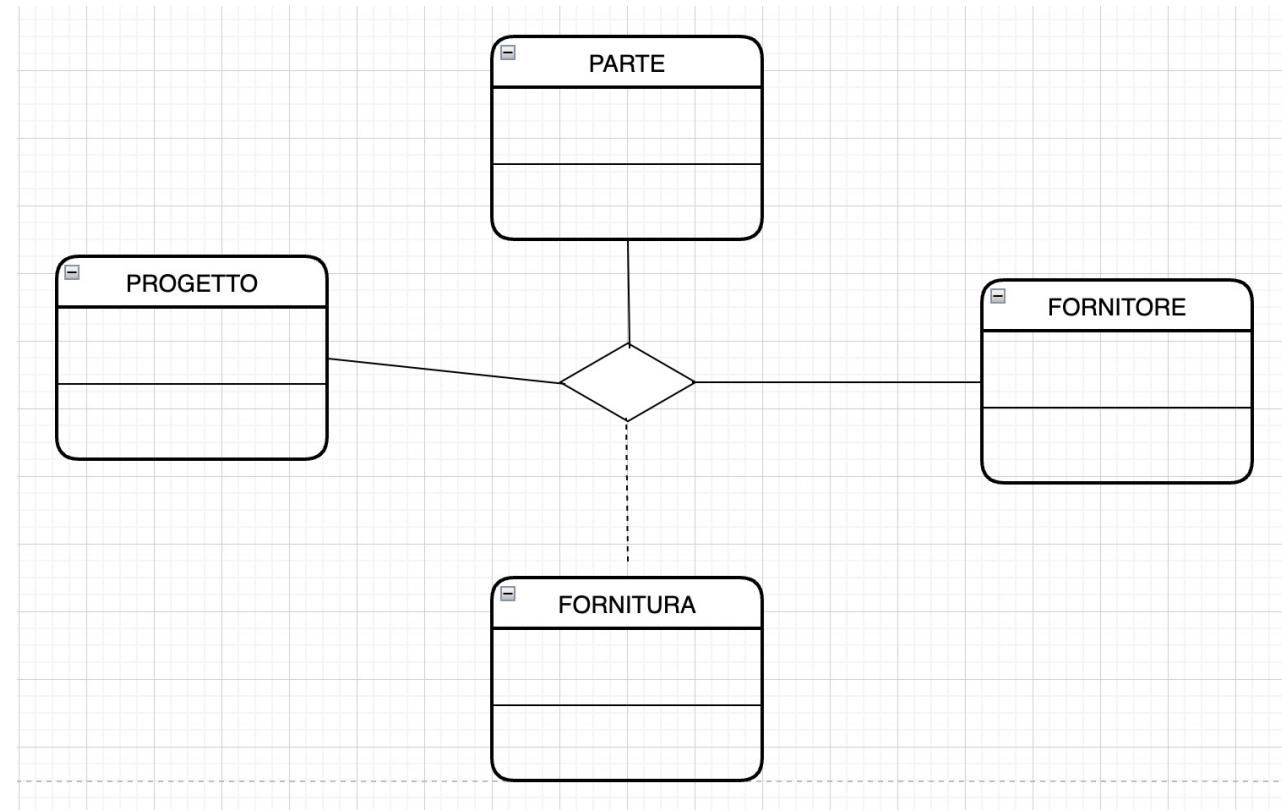
CLASSI DI ASSOCIAZIONE

- 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.



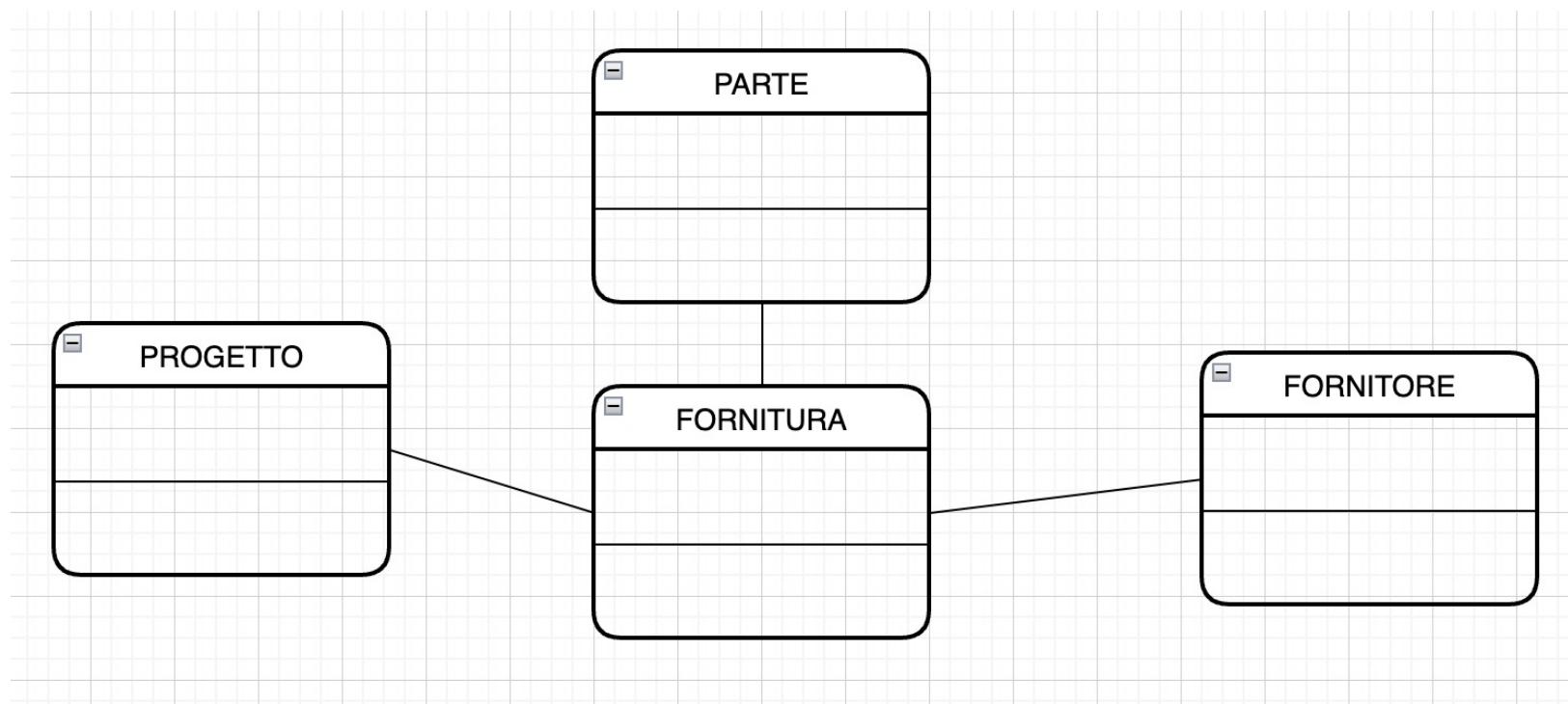
ASSOCIAZIONI N-ARIE

- 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.



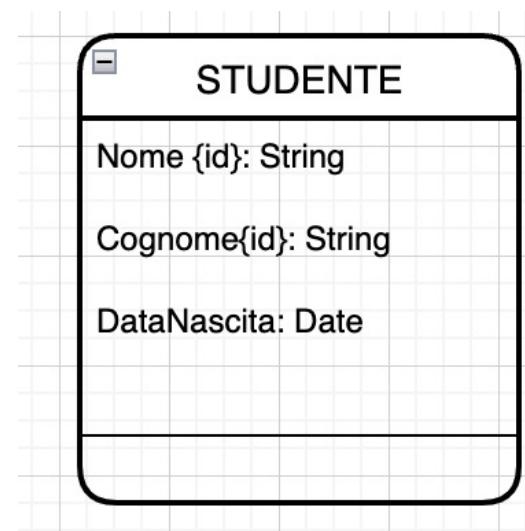
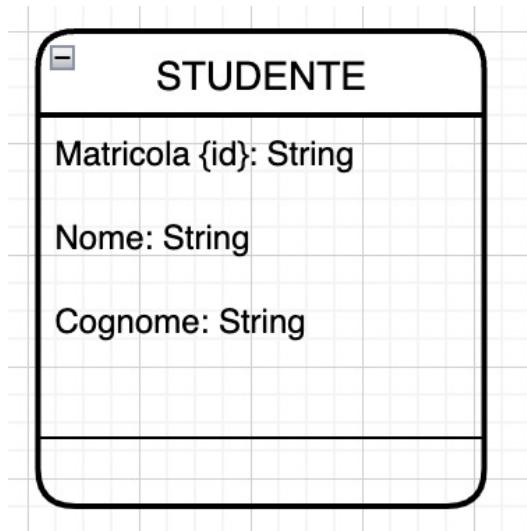
ASSOCIAZIONI N-ARIE (2)

- 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.



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.

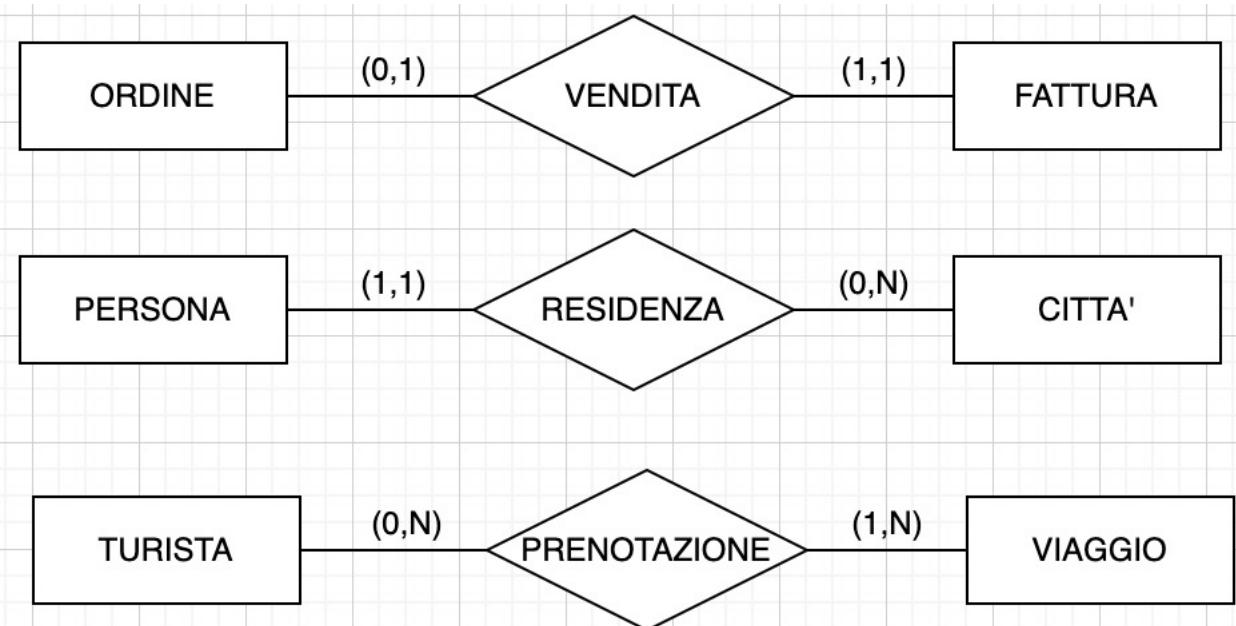


ASSOCIAZIONI CON MOLTEPLICITÀ

- 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).

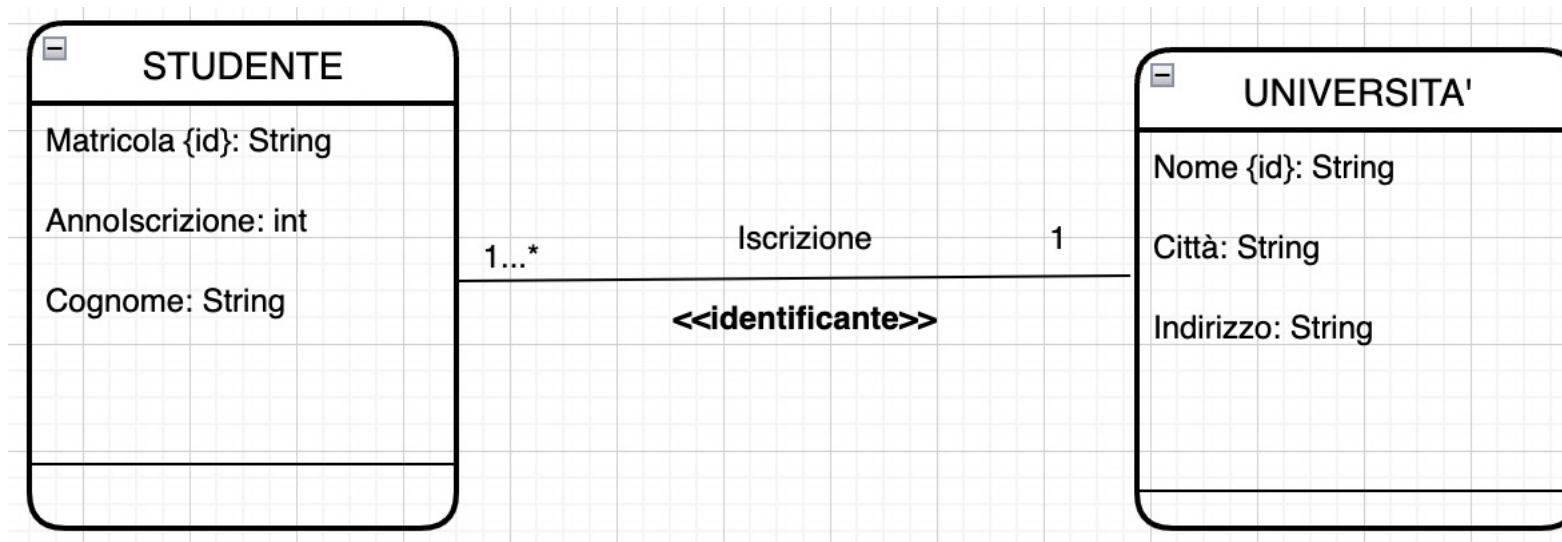


Corrispondenza (*invertita*) con le molteplicità dell'ER:



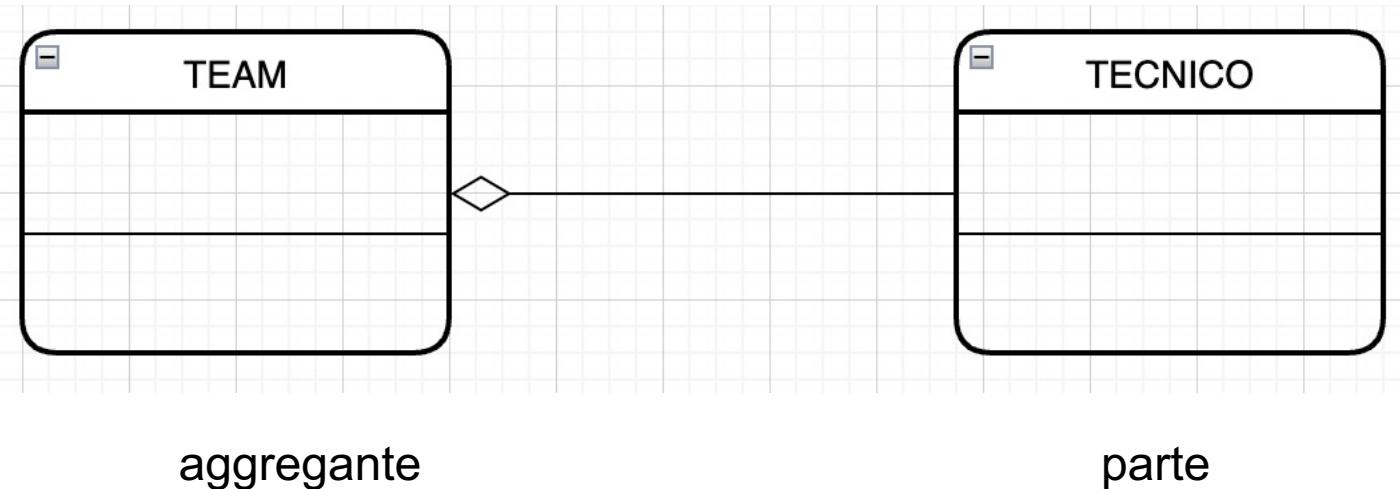
IDENTIFICATORE ESTERNO

- 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*.
 - In tal caso *Studente* è un'entità debole e *Matricola* è la sua chiave parziale



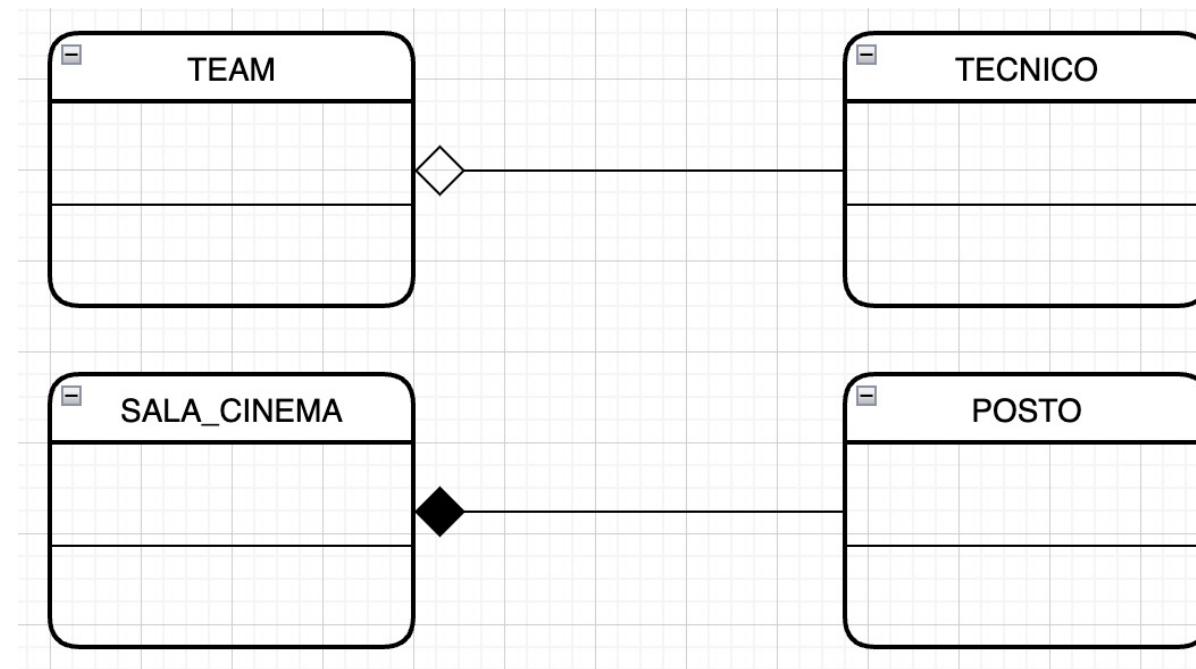
PROPRIETÀ DELLE ASSOCIAZIONI

- 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”. **(Composizione)**

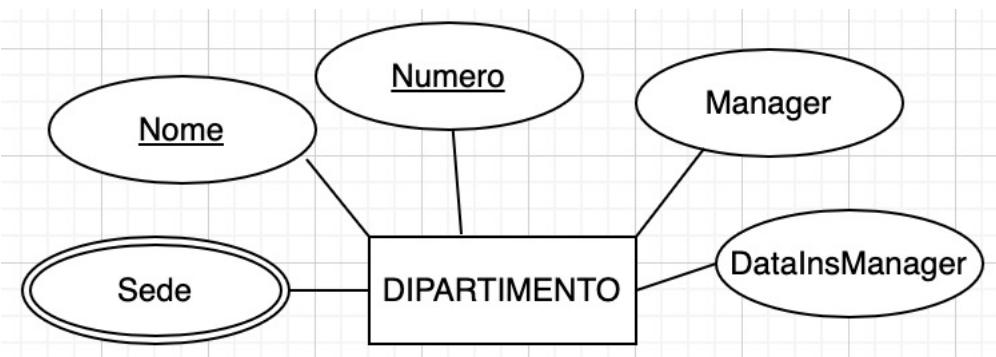


RAFFINAMENTO DEL DB COMPANY



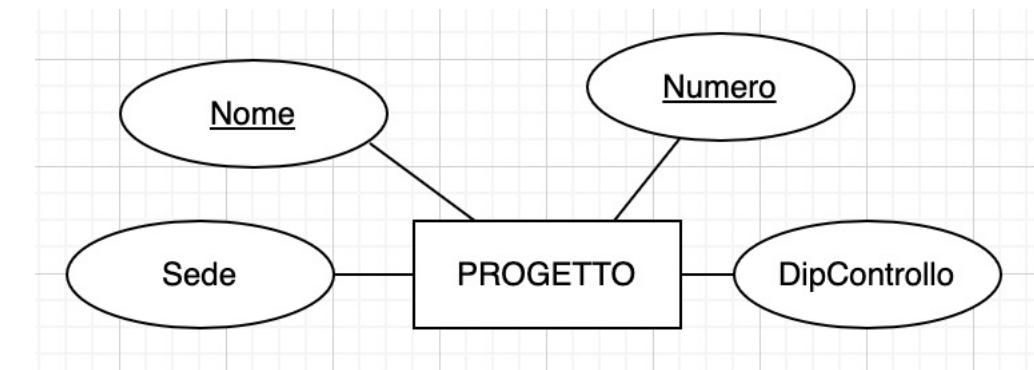
DIPARTIMENTI

- 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.



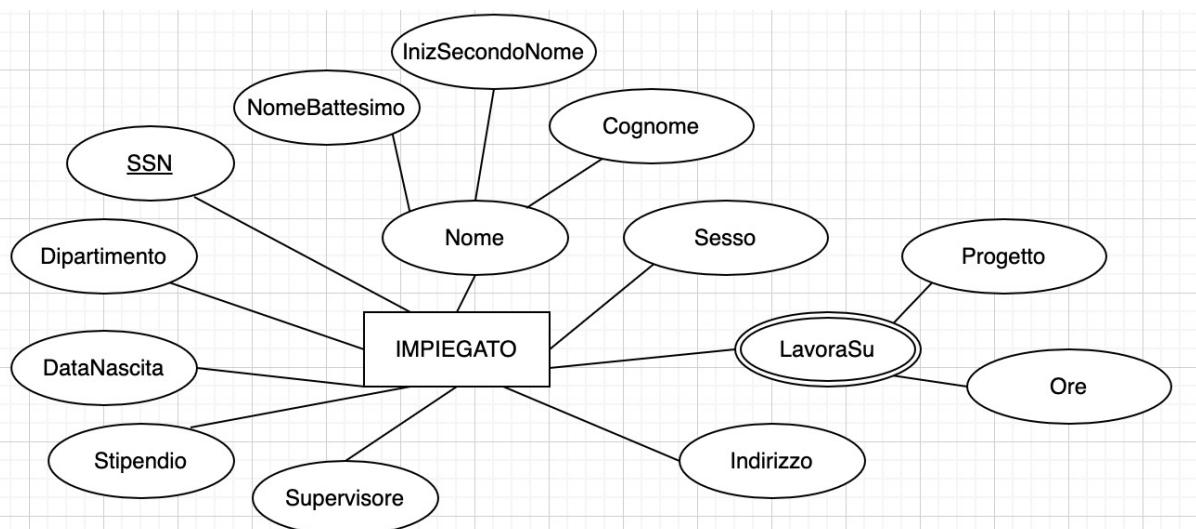
PROGETTI

- Ogni dipartimento controlla una serie di PROGETTI. Ogni progetto ha un nome, un numero ed una singola locazione.



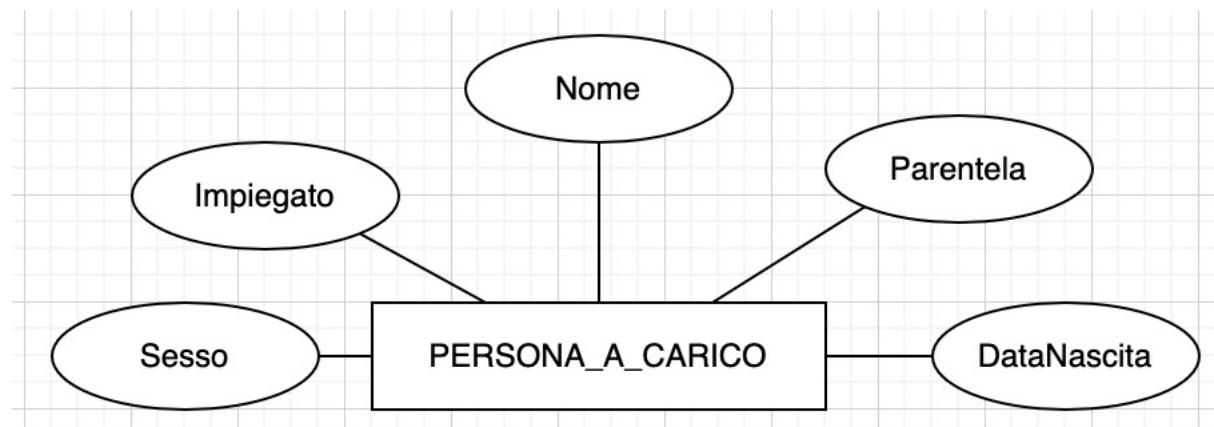
IMPIEGATI

- 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.



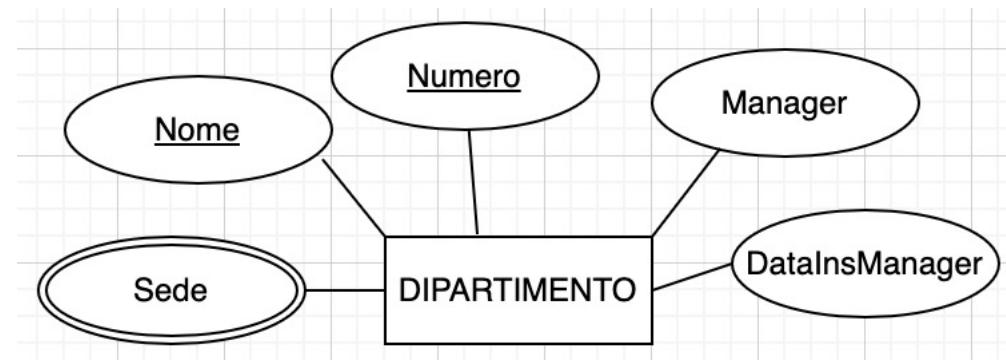
PERSONE A CARICO

- 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.



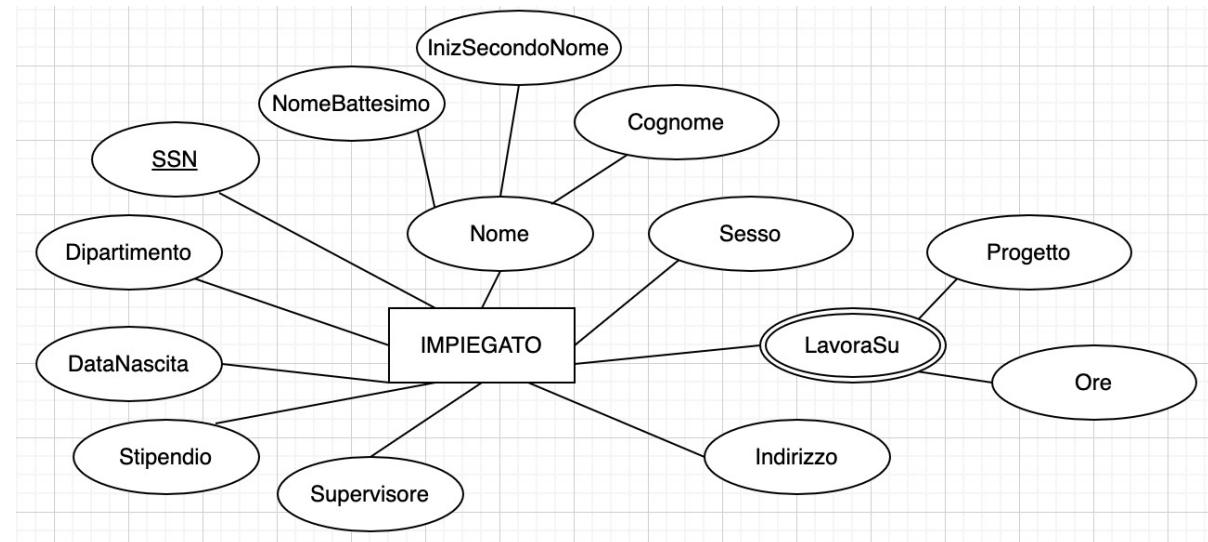
RELAZIONI IDENTIFICATE - DIPARTIMENTO

- L'attributo Manager è un riferimento ad un altro IMPIEGATO.
- L'attributo DataInsManager si riferisce alla data in cui un IMPIEGATO diventa manager di un DIPARTIMENTO
 - Può essere definito come attributo della relazione tra IMPIEGATO e DIPARTIMENTO



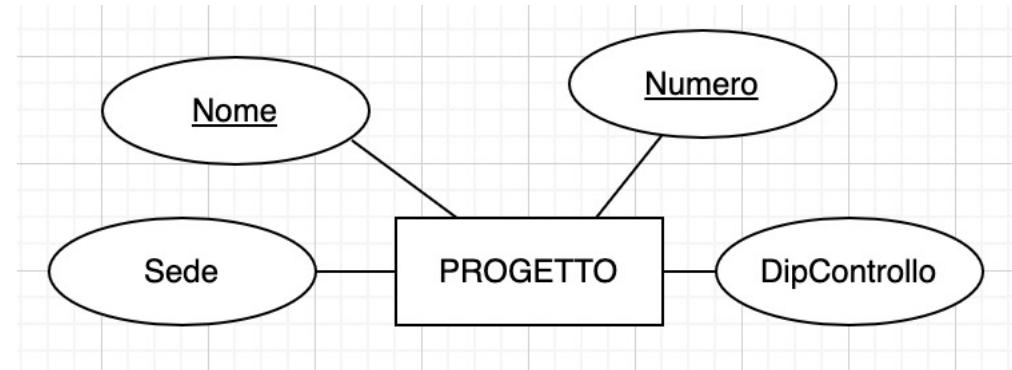
RELAZIONI IDENTIFICATE - IMPIEGATO

- L'attributo Supervisore è un riferimento ad un altro IMPIEGATO.
- L'attributo complesso LavoraSu consiste di
 - Un attributo Progetto che si riferisce ad un altro PROGETTO
 - Un attributo Ore che si riferisce alla relazione tra IMPIEGATO e PROGETTO
- L'attributo Dipartimento si riferisce al DIPARTIMENTO di afferenza dell'IMPIEGATO



RELAZIONI IDENTIFICATE - PROGETTO

- L'attributo DipControllo è un riferimento ad un DIPARTIMENTO.



RELAZIONI IDENTIFICATE – PERSONA_A_CARICO

- L'attributo Impiegato è un riferimento ad un IMPIEGATO.

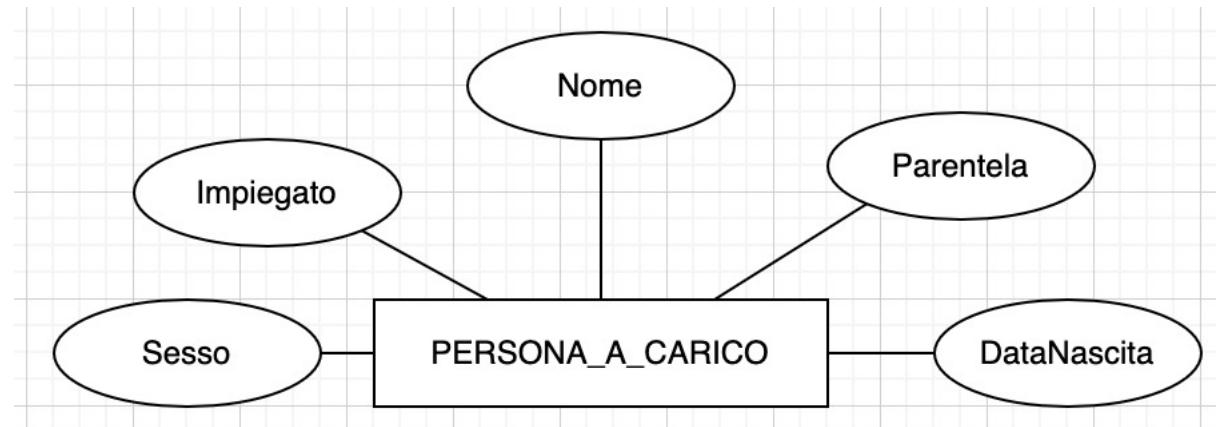
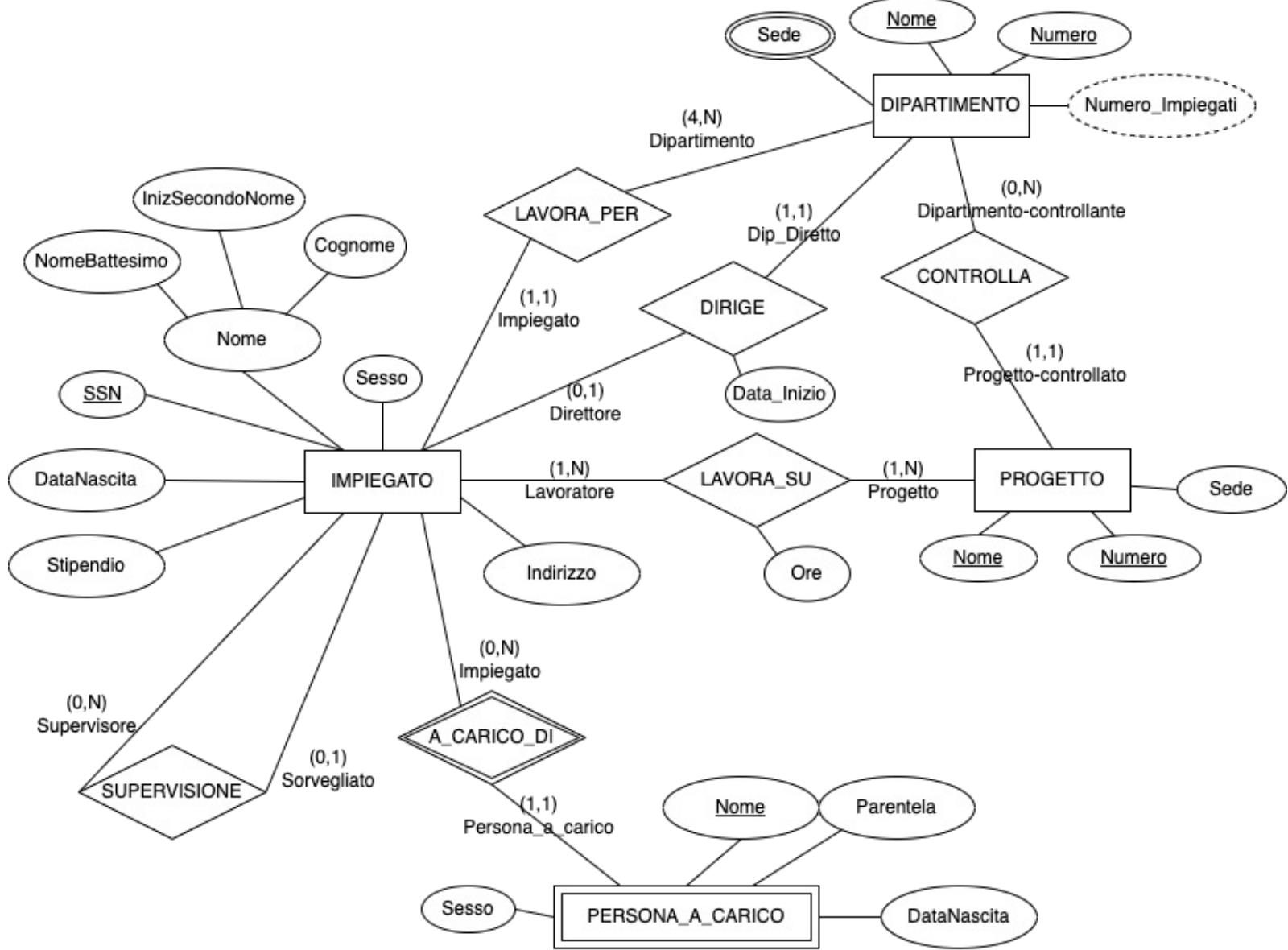
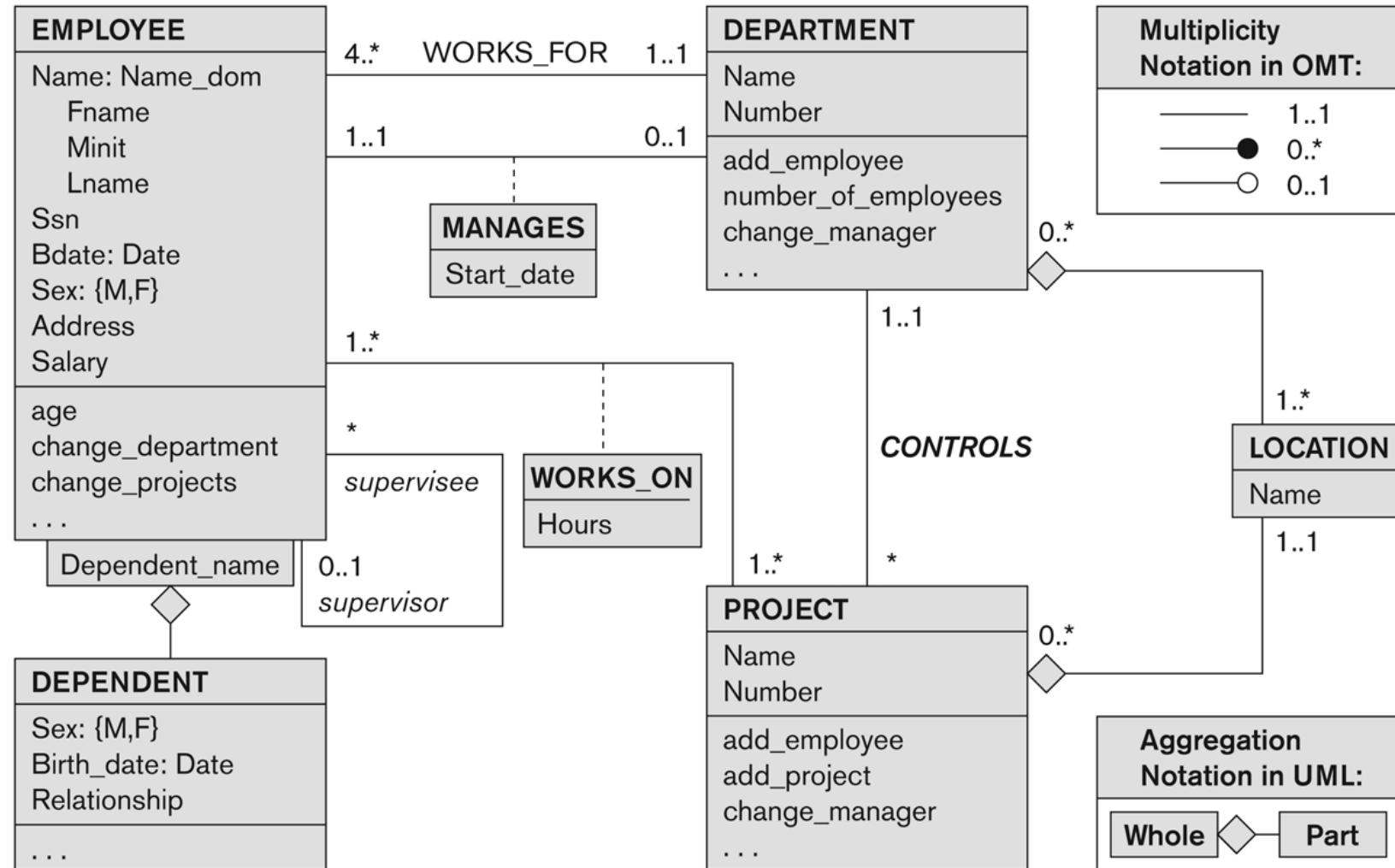


DIAGRAMMA ER COMPLETO



L0 SCHEMA CONCETTUALE IN UML



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.



ESERCIZIO 1

- *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.

MODELLAZIONE EER



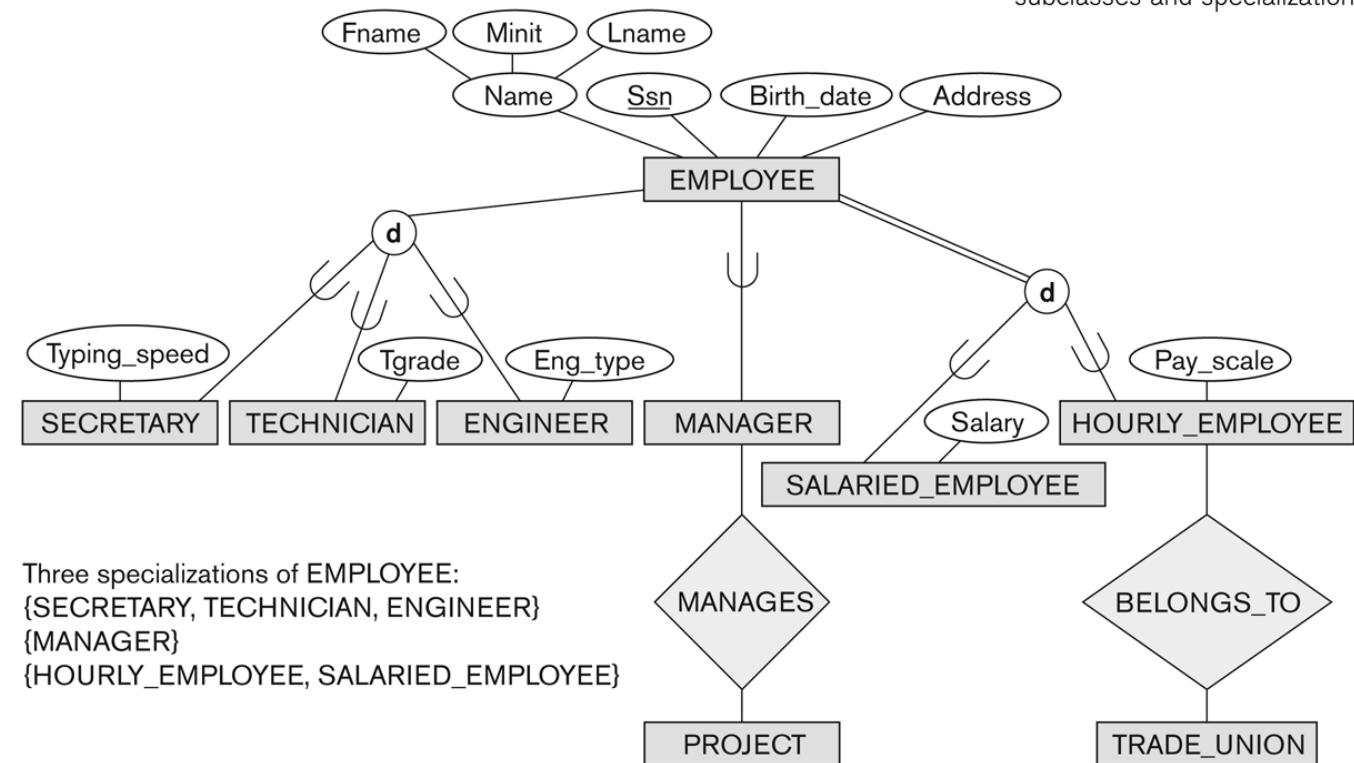
MODELLO ENHANCED ER

- Il modello EER offre dei costrutti per ampliare i concetti base dell'ER
 - Sottoclassi e Superclassi
 - Specializzazioni e Generalizzazioni
 - Categorie
 - ...
- Questi nuovi strumenti servono per modellare un miniworld in maniera ancora più dettagliata
- Le notazioni utilizzate possono essere comprese da un set di utenti più specializzato
 - Alcuni concetti dell'EER provengono dai concetti di programmazione ad oggetti
 - Appunto le superclassi, le sottoclassi ed i relativi concetti di ereditarietà



SOTTOCLASSI, SUPERCLASSI ED EREDITARIETÀ

- Un'entità può avere ulteriori estensioni per massimizzare la sua significatività
- Ad esempio un **IMPIEGATO** può essere ulteriormente categorizzato in
 - **SEGRETARIO, INGEGNERE, TECNICO** (tipo di IMPIEGATO)
 - **MANAGER** (ruolo dell'IMPIEGATO)
 - **STIPENDIATO, PAGATO_A_ORE** (in base al tipo di contratto)



SUPERCLASSI E SOTTOCLASSI (1)

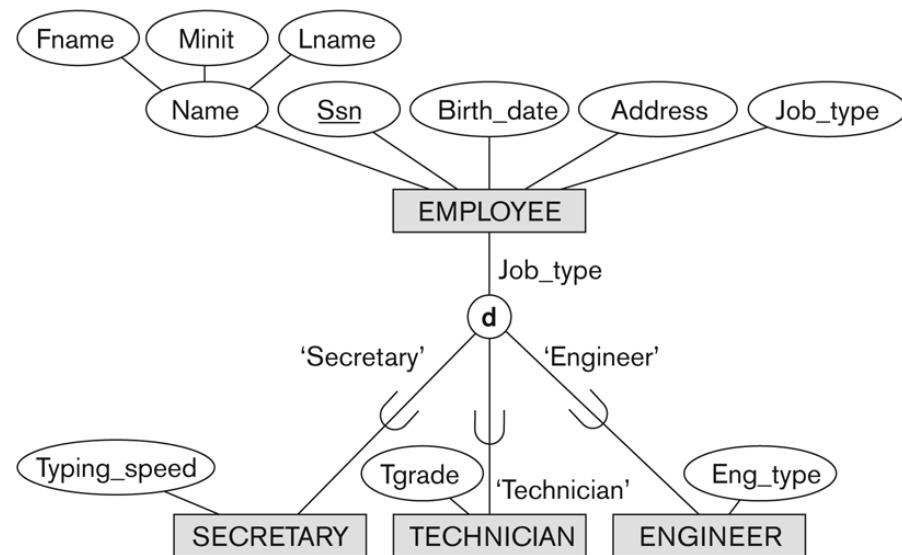
- Ognuna delle entità che appartiene a uno dei sottoinsiemi della classe IMPIEGATO è chiamata SOTTOCLASSE
- IMPIEGATO è la SUPERCLASSE per ognuna di queste entità
- Si avrà quindi una relazione SUPERCLASSE/SOTTOCLASSE
 - IMPIEGATO/TECNICO
 - IMPIEGATO/SEGRETARIO
 - IMPIEGATO/MANAGER
 - ...
- Sono chiamate anche relazioni IS-A poiché:
 - Un TECNICO è un (IS-A) IMPIEGATO
 - Un MANAGER è un (IS-A) IMPIEGATO
 - ...



SUPERCLASSI E SOTTOCLASSI (2)

- Ogni istanza di ognuna delle entità membro di una SOTTOCLASSE rappresenta la stessa istanza dell'entità SUPERCLASSE
- Non può esistere un'istanza di una SOTTOCLASSE che non sia anche istanza della SUPERCLASSE.
- Un'istanza della SUPERCLASSE può OPZIONALMENTE essere inclusa come istanza della SOTTOCLASSE
- Ogni istanza membro di una sottoclassificazione EREDITA tutti gli attributi e tutte le relazioni delle istanze della superclasse.

Figure 4.4
EER diagram notation
for an attribute-defined specialization
on Job_type.



SPECIALIZZAZIONE E GENERALIZZAZIONE

- Il processo di SPECIALIZZAZIONE permette di definire un insieme di sottoclassi per una determinata entità/classe
- La GENERALIZZAZIONE rappresenta il processo inverso alla SPECIALIZZAZIONE

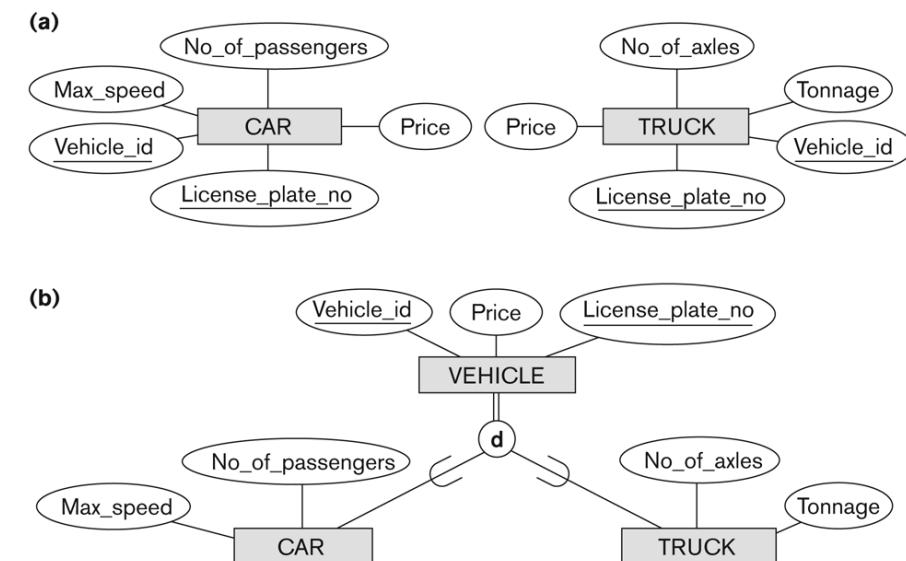
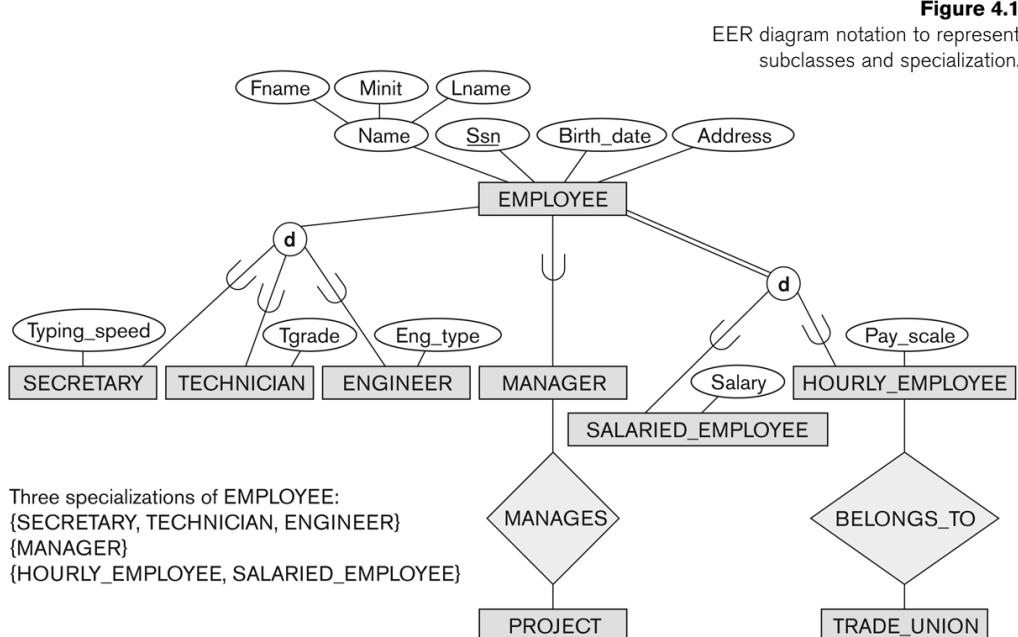


Figure 4.3
Generalization. (a) Two entity types, CAR and TRUCK.
(b) Generalizing CAR and TRUCK into the superclass VEHICLE.

VINCOLI A SPECIALIZZAZIONI/GENERALIZZAZIONI

- Ci sono due vincoli base alle specializzazioni/generalizzazioni
 - Vincolo di Disgiunzione
 - Vincolo di Completezza



VINCOLO DI DISCIUNZIONE

- Specifica che le sottoclassi della specializzazione devono essere **DISGIUNTE**
- Un'istanza può essere membro di al più una sottoclasse
 - Specificata da una **d** nel diagramma EER
- Se non sono disgiunte allora la specializzazione è **OVERLAPPING**
- Un'istanza può essere membro di più di una sottoclasse
 - Specificata da una **o** nel diagramma EER



VINCOLO DI COMPLETEZZA

- Se è **TOTALE** specifica che ogni istanza nella superclasse deve essere membro di qualche sottoclasse nella specializzazione
 - Indicata con una **doppia linea** nel diagramma EER
-
- Se è **PARZIALE** specifica che possono esserci istanze della superclasse che non appartengono ad alcuna sottoclasse nella specializzazione
 - Indicata con una **linea singola** nel diagramma EER



POSSIBILI VINCOLI

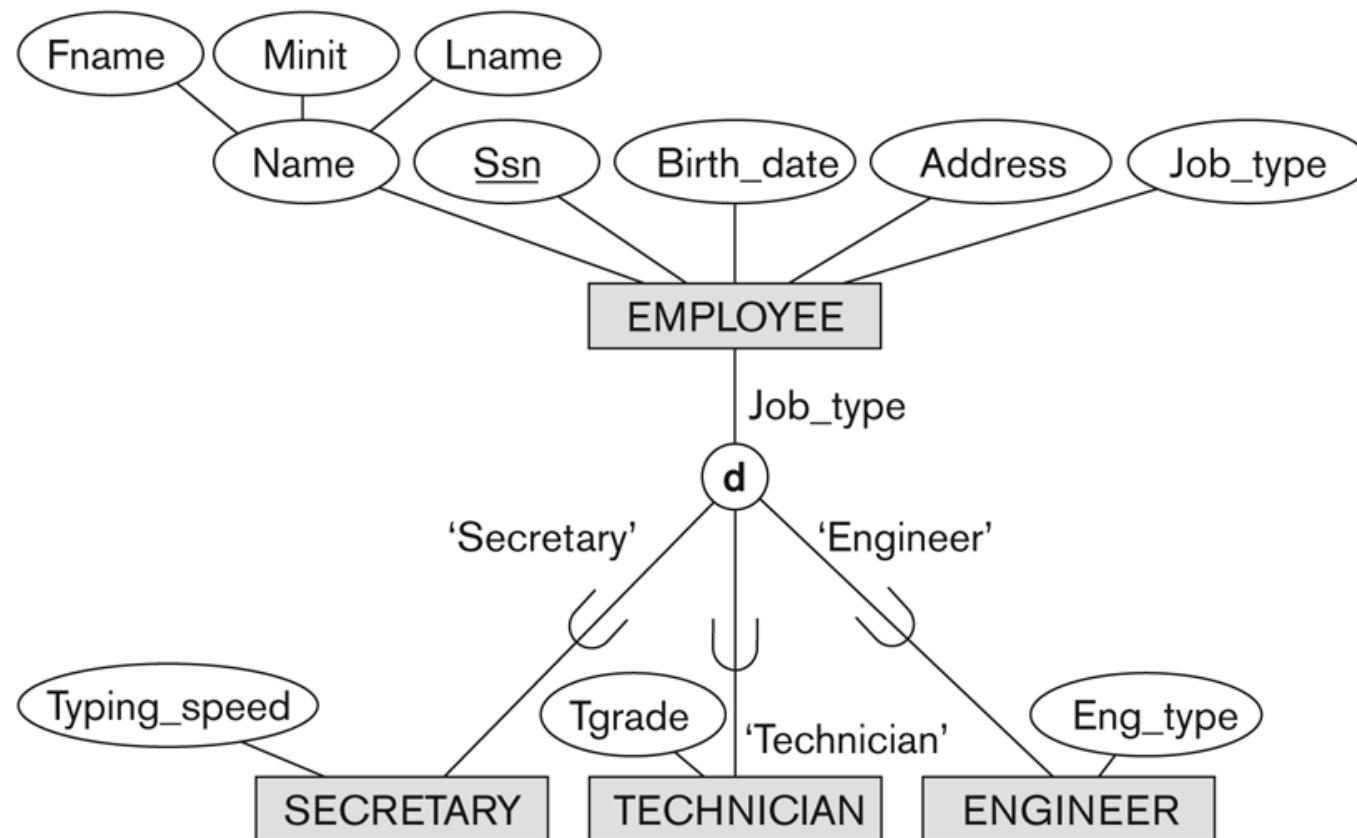
- Possiamo avere quindi quattro tipi di specializzazione/generalizzazione
 - **Disgiunta, Totale**
 - **Disgiunta, Parziale**
 - **Overlapping, Totale**
 - **Overlapping, Parziale**
- Spesso una generalizzazione è totale in quanto la superclasse è derivata dalle sottoclassi.



DISGIUNTA, PARZIALE

Figure 4.4

EER diagram notation for an attribute-defined specialization on Job_type.



OVERLAPPING, TOTALE

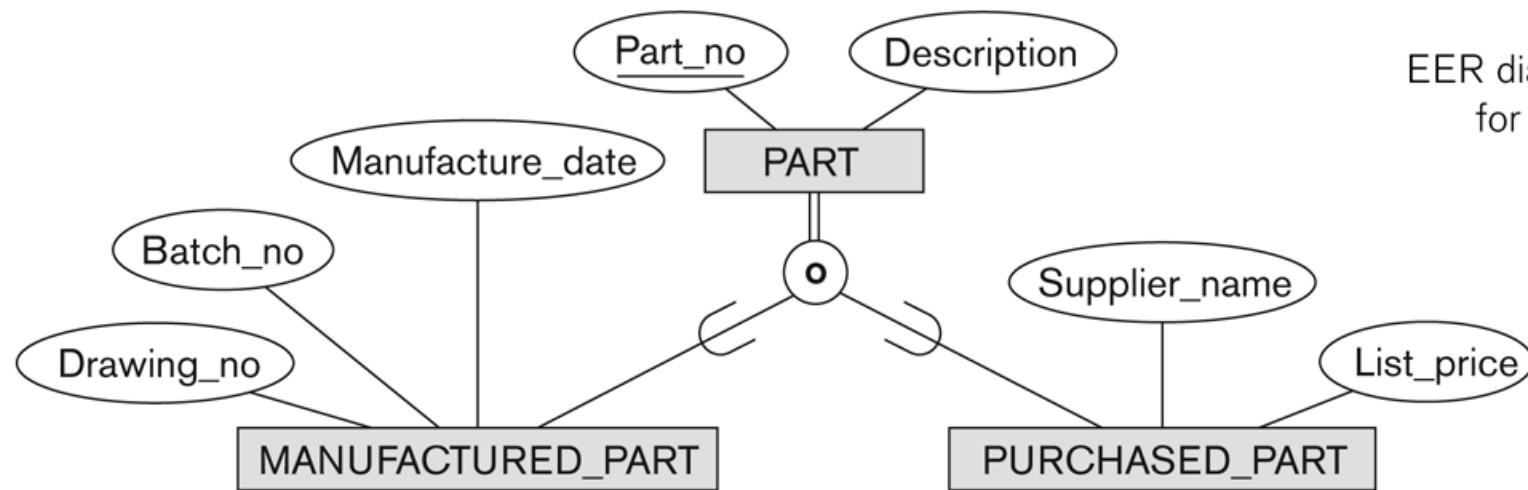
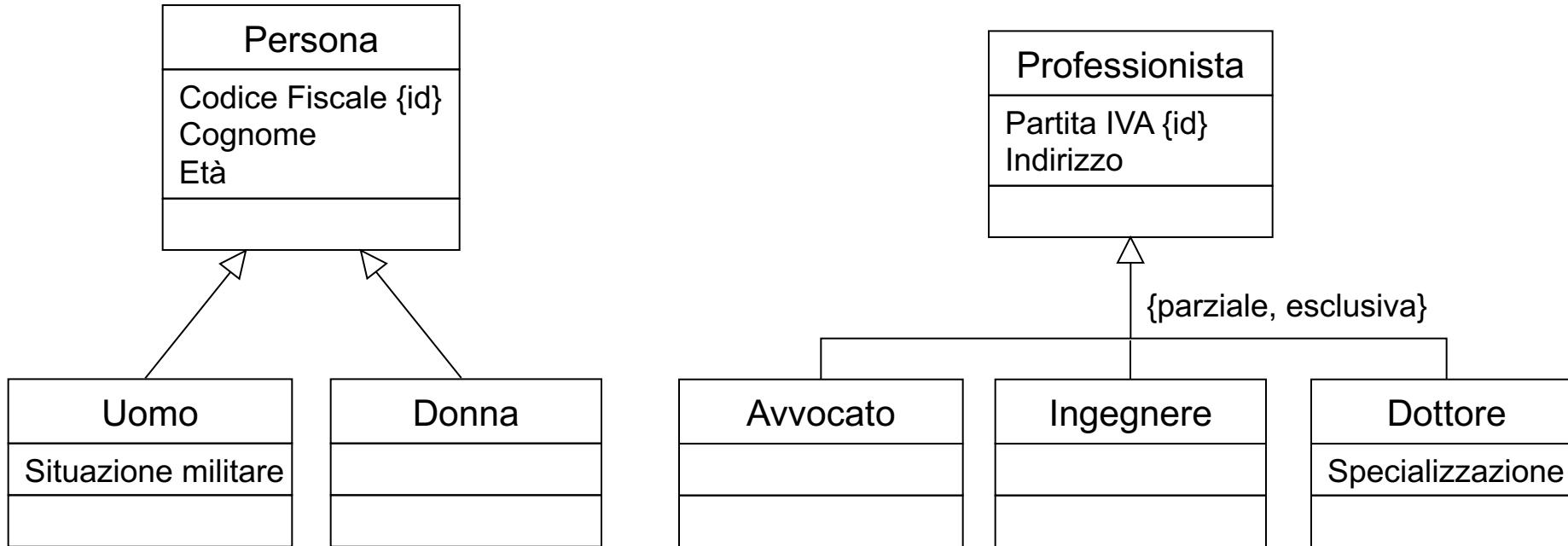


Figure 4.5
EER diagram notation
for an overlapping
(nondisjoint)
specialization.

GENERALIZZAZIONI E SPECIALIZZAZIONI IN UML

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



INFO SULLA LEZIONE

- Tutto il capitolo 3
- Capitolo 4
 - 4.1, 4.2, 4.3
- Il resto del capitolo 4 parla di altri costrutti dell'EER, come l'Unione, che non utilizzeremo particolarmente, ma che può essere comunque interessante imparare.





FINE

Per eventuali domande: (in ordine di preferenza personale)

- Ora.
- Chat di Teams
- Mail: silvio.barra@unina.it

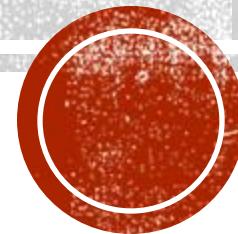


BASI DI DATI I





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



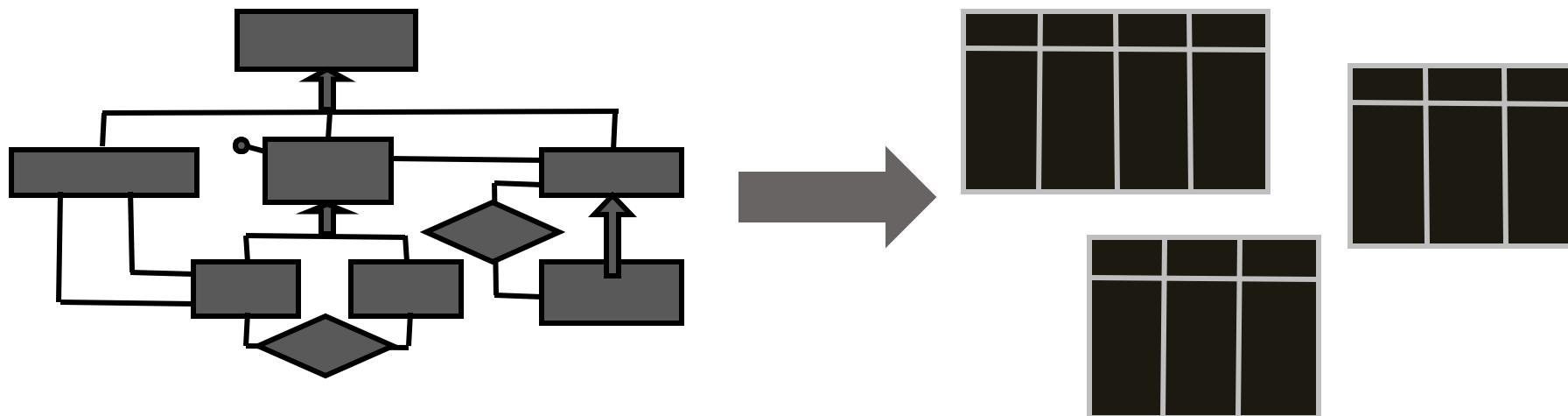
RECAP #3 – PROGETTAZIONE CONCETTUALE

- 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



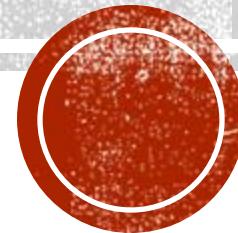
DA SCHEMA CONCETTUALE A LOGICO

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	

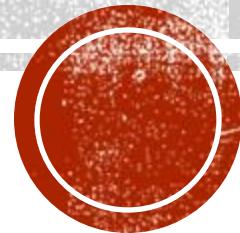
DEPARTMENT	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE	DEPT_LOCATIONS	DNUMBER	DLOCATION
Research		5	333445555	22-MAY-78		1	Houston
Administration		4	987654321	01-JAN-85		4	Stafford
Headquarters		1	888665555	19-JUN-71		5	Bellaire
						5	Sugarland
						5	Houston

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



CONCETTI DEL MODELLO RELAZIONALE

- DOMINIO -



DOMINIO

- 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...).



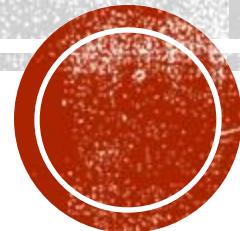
DOMINI

- 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**.



CONCETTI DEL MODELLO RELAZIONALE

- RELAZIONE -



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**:

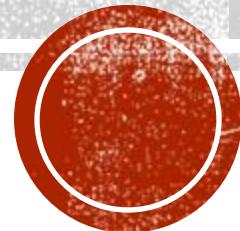
STUDENT	Name	SSN	HomePhone	Address	OfficePhone	Age	GPA
	Benjamin Bayer	305-61-2435	373-1616	2918 BlueBonnet Lane	null	19	3.21
	Katherine Ashly	381-62-1245	375-4409	125 Kirby Road	null	18	2.89
	Dick Davidson	422-11-2420	null	3452 Elgin Road	749-1253	25	3.53
	Charles Cooper	489-22-1100	376-9821	265 Lark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

Esempio di Relazione



CONCETTI DEL MODELLO RELAZIONALE

- SCHEMI E Istanze di RELAZIONE -



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 $t = <v_1, v_2, \dots, v_n>$ 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

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



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

DNUMBER	DLOCATION
---------	-----------

PROJECT

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

WORKS_ON

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

DEPENDENT

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



ISTANZA DI DATABASE RELAZIONALE

Un'istanza del database “Company”

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	999687777	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	
999687777	30	30.0	
999687777	10	10.0	
987987987	10	35.0	
987987987	30	5.0	
987654321	30	20.0	
987654321	20	15.0	
888665555	20	null	

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



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

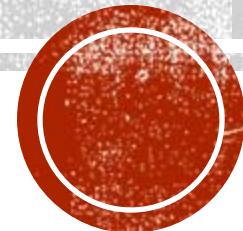
$t = \langle v_1, v_2, \dots, v_n \rangle$ dove v_i è il valore per l'attributo A_i

- $t[A_i]$ 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 $\{carattere, \text{stringa a lunghezza fissa e variabile}, \text{data}, \text{ora}, \text{valuta}, \text{ecc...}\}$ 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

- $\{\text{SSN}\}$ 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)

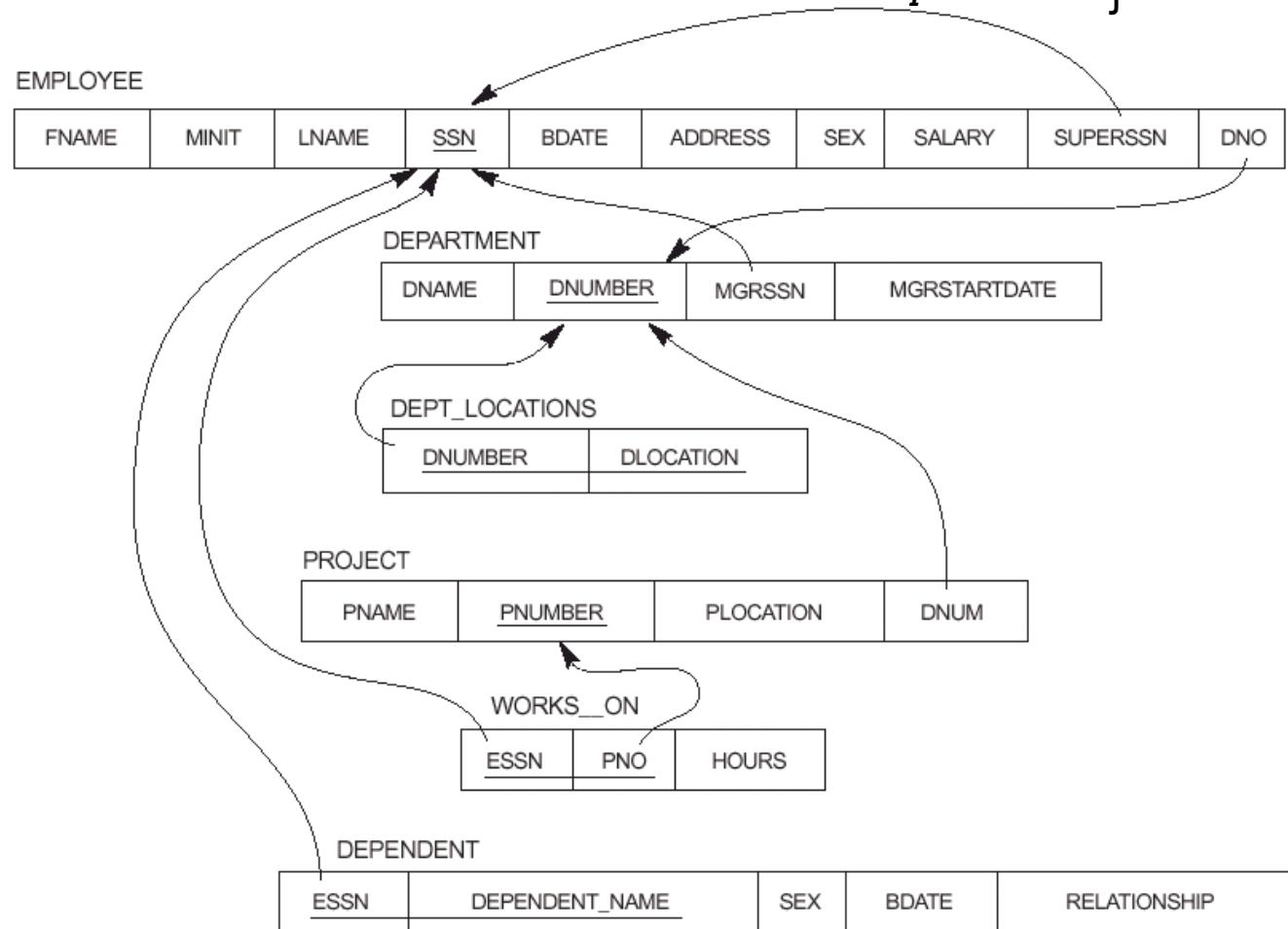
- Le chiavi esterne sono simili al concetto dei “*puntatori*” in C, che o riferiscono ad una variabile allocata o sono *null*.
- L’attributo *DNO* di *Employee* è una chiave esterna, poiché rispetta le condizioni appena elencate.
- Una tupla t_i di una relazione R_i è detta **referenziare** una tupla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

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



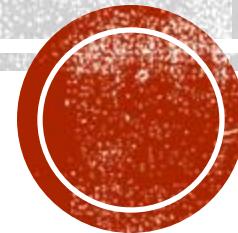
VINCOLI DI INTEGRITÀ REFERENZIALE - ESEMPIO

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





ALTRI VINCOLI



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.



INFO SULLA LEZIONE

- ## ■ Tutto il capitolo 5

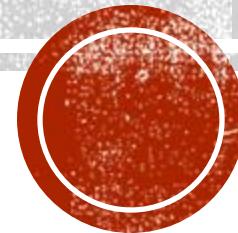




FINE

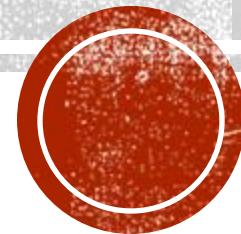
Per eventuali domande: (in ordine di preferenza personale)

- Ora.
- Chat di Teams
- Mail: silvio.barra@unina.it



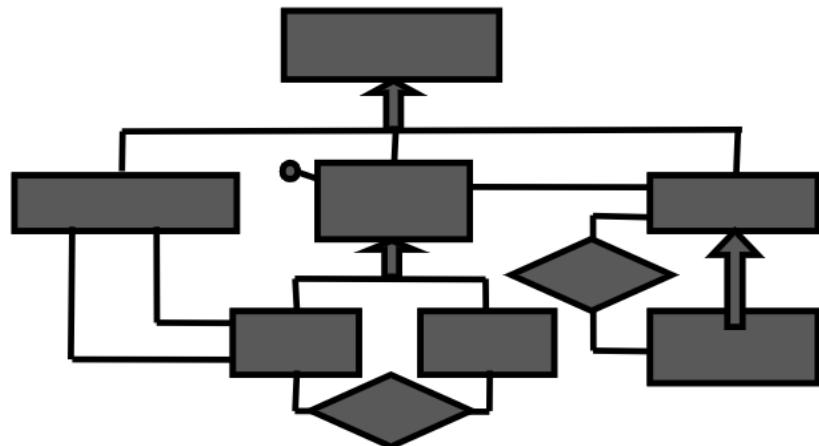
BASI DI DATI I

RISTRUTTURAZIONE DEL MODELLO CONCETTUALE

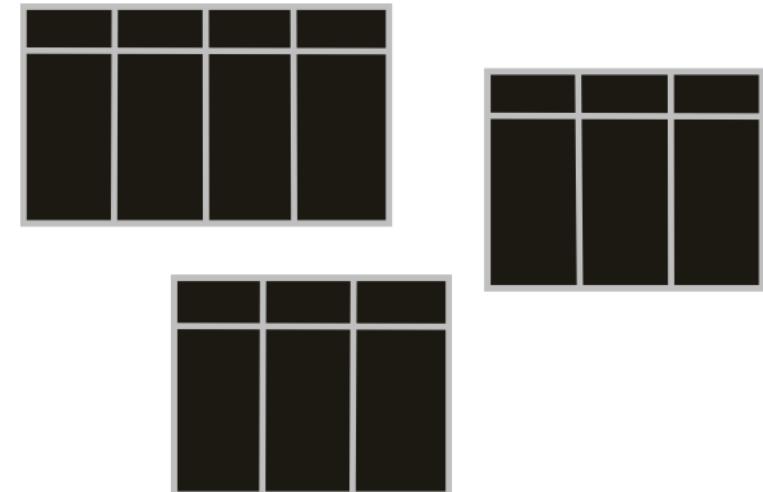


PASSAGGIO DAL MODELLO CONCETTUALE A QUELLO LOGICO

Abbiamo visto come costruire un modello concettuale del nostro minimondo

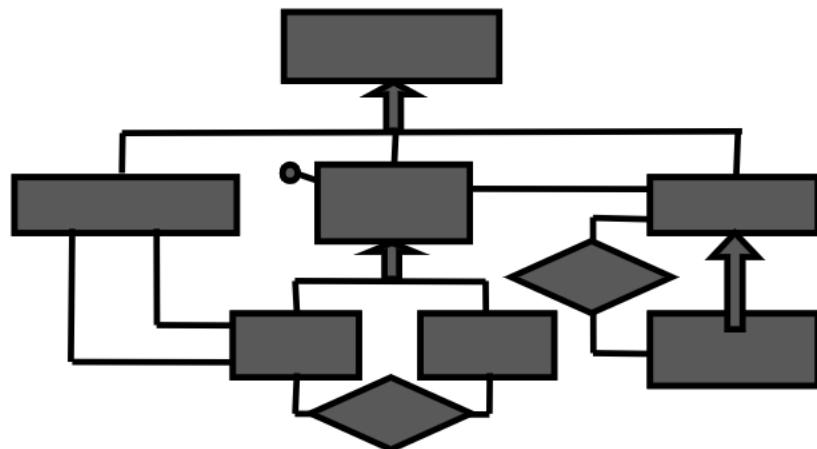


Abbiamo visto in cosa consiste il modello logico, che è la successiva fase di progettazione



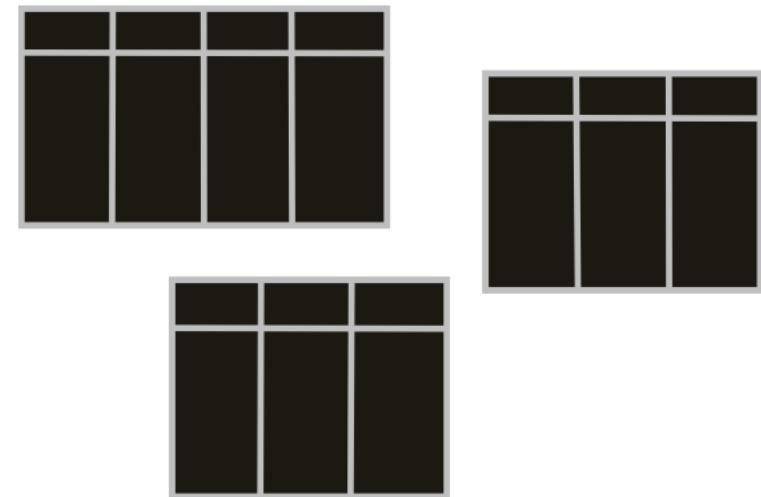
PASSAGGIO DAL MODELLO CONCETTUALE A QUELLO LOGICO

Abbiamo visto come costruire un modello concettuale del nostro minimondo

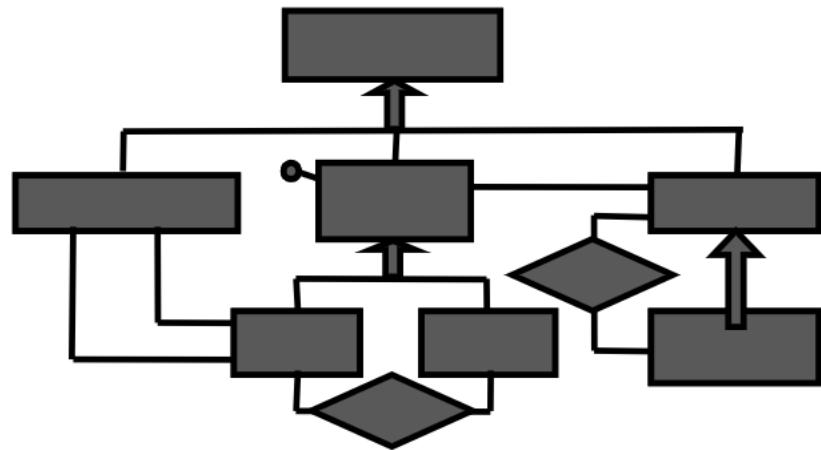


Processo di traduzione
→

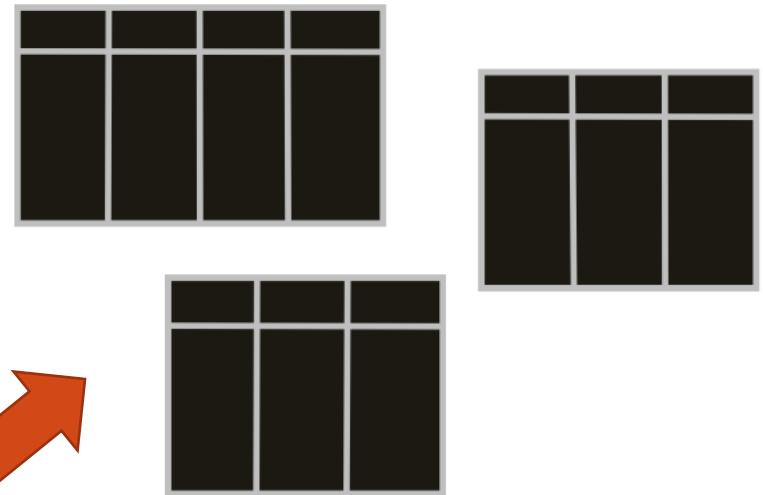
Abbiamo visto in cosa consiste il modello logico, che è la successiva fase di progettazione



STEP INTERMEDI



Ottimizzazione
del diagramma



Applicazione regole
di traduzione



RISTRUTTURAZIONE

- Prima di passare allo schema logico è necessario ristrutturare il Class Diagram (o l'ER 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.

L'obiettivo è creare uno schema logico in grado di descrivere tutte le informazioni contenute negli schema concettuali prodotti nella fase di progettazione concettuale.



ATTIVITÀ DI RIORGANIZZAZIONE E TRADUZIONE

- La fase di ristrutturazione *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).

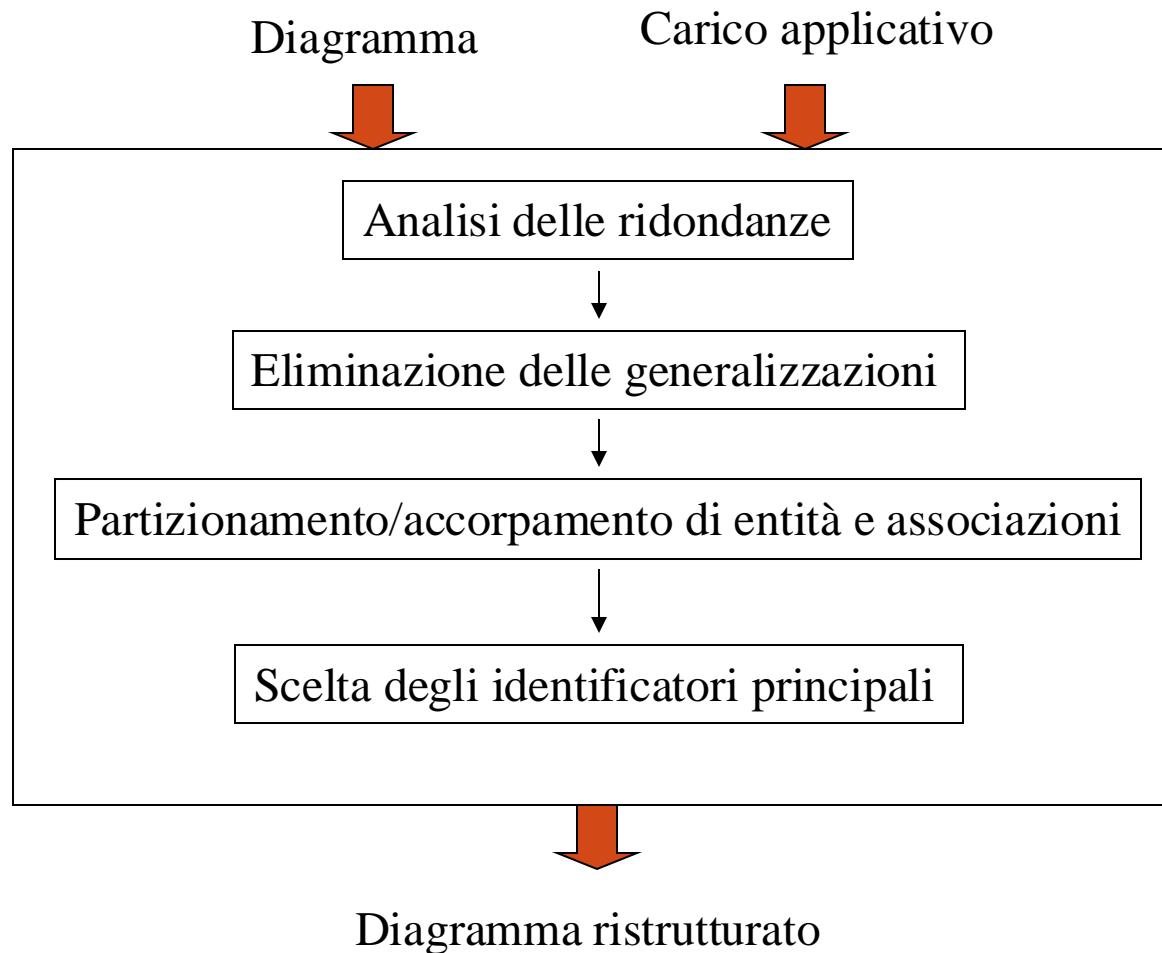


PREAMBOLO

- Le regole di ristrutturazione si applicano sia ai Class Diagram che ai diagrammi ER
- Questo perché
 - Ogni concetto che estende la rappresentazione concettuale del miniworld ha sia un formalismo in ER/EER sia può essere rappresentato tramite una notazione UML
 - L'obiettivo finale è quello di tradurre il modello concettuale nel modello logico, che è unico. Il punto di arrivo è lo stesso



RISTRUTTURAZIONE DI SCHEMI CONCETTUALI

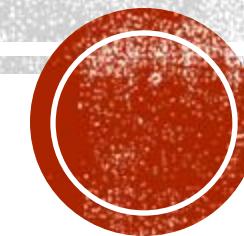


PREAMBOLO

- Le regole di ristrutturazione si applicano sia ai Class Diagram che ai diagrammi ER
- Questo perché
 - Ogni concetto che estende la rappresentazione concettuale del miniworld ha sia un formalismo in ER/EER sia può essere rappresentato tramite una notazione UML
 - L'obiettivo finale è quello di tradurre il modello concettuale nel modello logico, che è unico. Il punto di arrivo è lo stesso
- *Per semplicità, per mostrare le regole di ristrutturazione, utilizzeremo la notazione UML, ma è chiaro che la stessa regola può essere applicata analogamente ai diagrammi ER.*



TAVOLE DI ANALISI



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

- 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

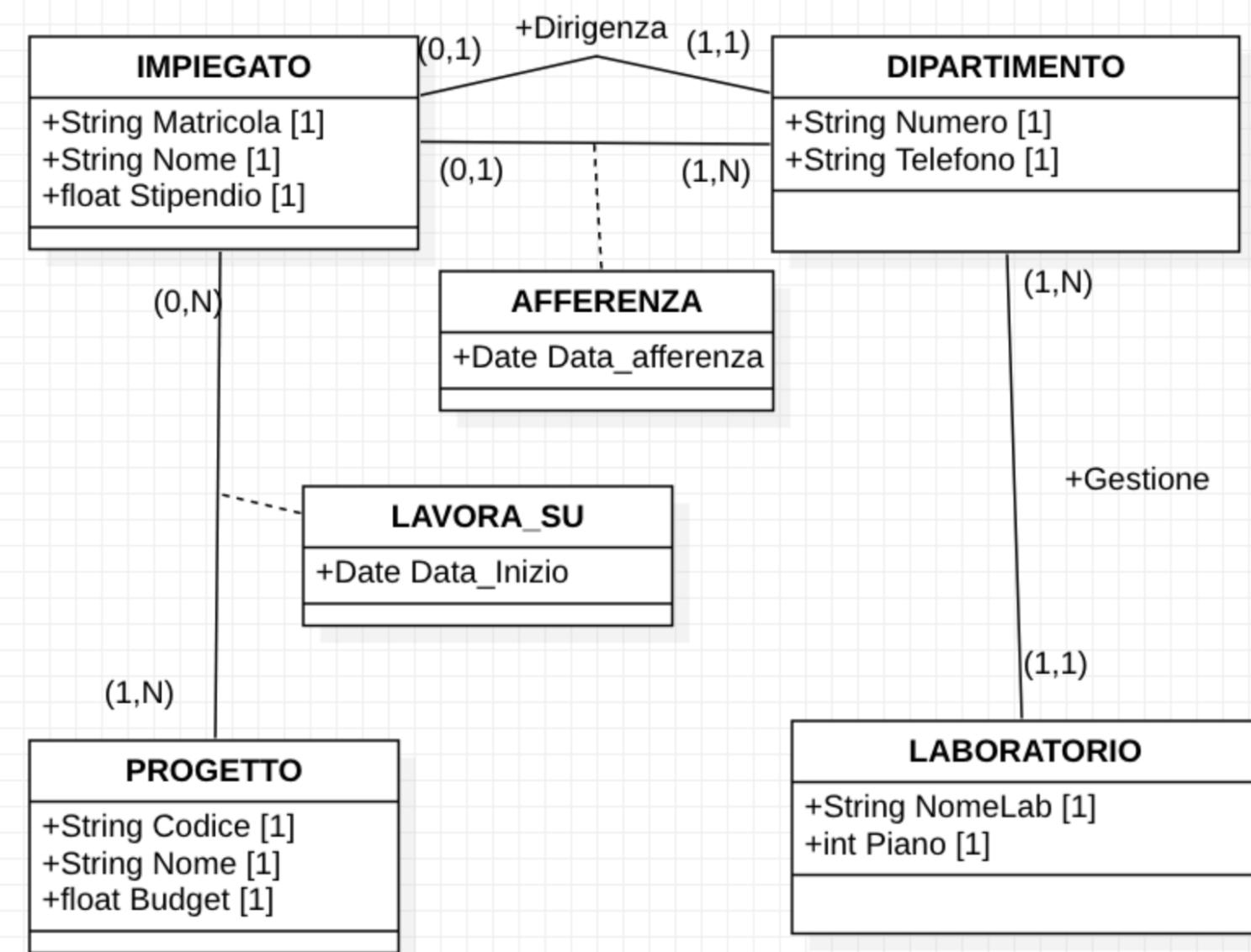


TAVOLA DEI VOLUMI

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

- L'analisi della tavola dei volumi dipende da due parametri:
 - Numero delle occorrenze delle entità coinvolte nelle relazioni.
 - Numero medio di partecipazioni di un'occorrenza di entità alle occorrenze di relazioni.

Es: supponiamo che un impiegato lavori in media a tre progetti.



TAVOLA DELLE OPERAZIONI

- Per l'analisi delle operazioni vado a valutare
 1. Quali sono le operazioni che vengono fatte sulla base di dati
 2. Sulla base di quelle che sono le operazioni più frequenti, vado a vedere il costo per effettuarla



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

I: interattiva. Sono gli utenti della base di dati che effettuano queste operazioni
B: batch. Operazioni pianificate in un determinato arco di tempo

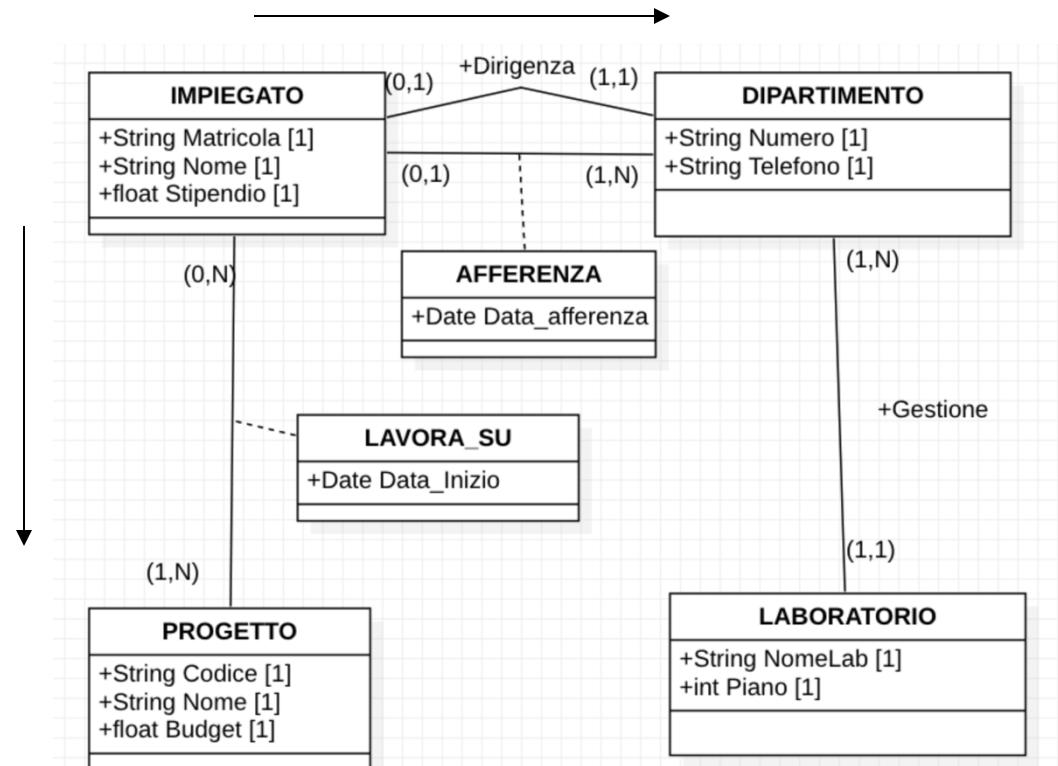


SCHEMA DI OPERAZIONE

- 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

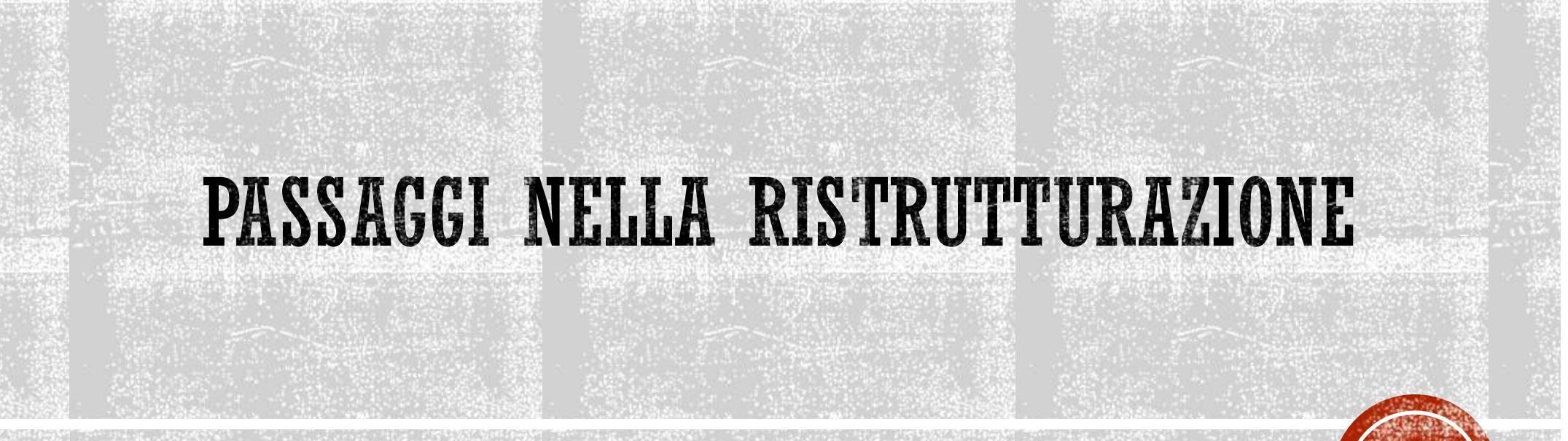
- Si stima il costo di un'operazione contando il numero degli accessi alle occorrenze di entità e di relazioni.
- Ipotizziamo che il costo di una lettura è x , mentre il costo di una scrittura è y .



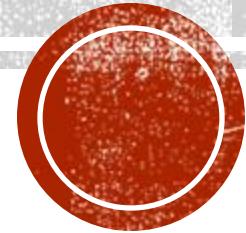
TAVOLA DEGLI ACCESSI (RIE. OP. 2)

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

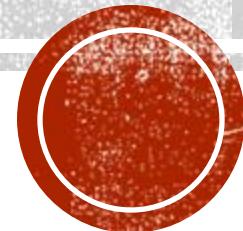


RISTRUTTURAZIONE DEL CLASS DIAGRAM

- 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.

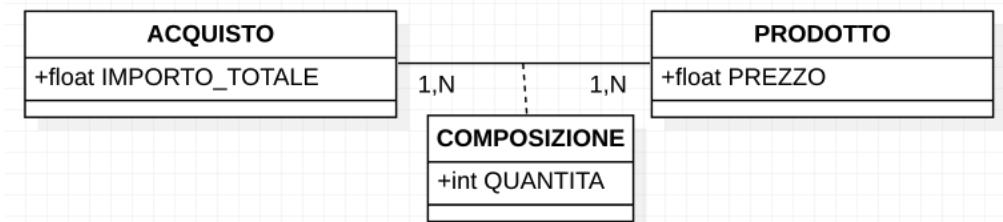


ATTRIBUTI DERIVABILI

- Da attributi della stessa entità /associazione:



-
- Da attributi di altre entità/associazioni

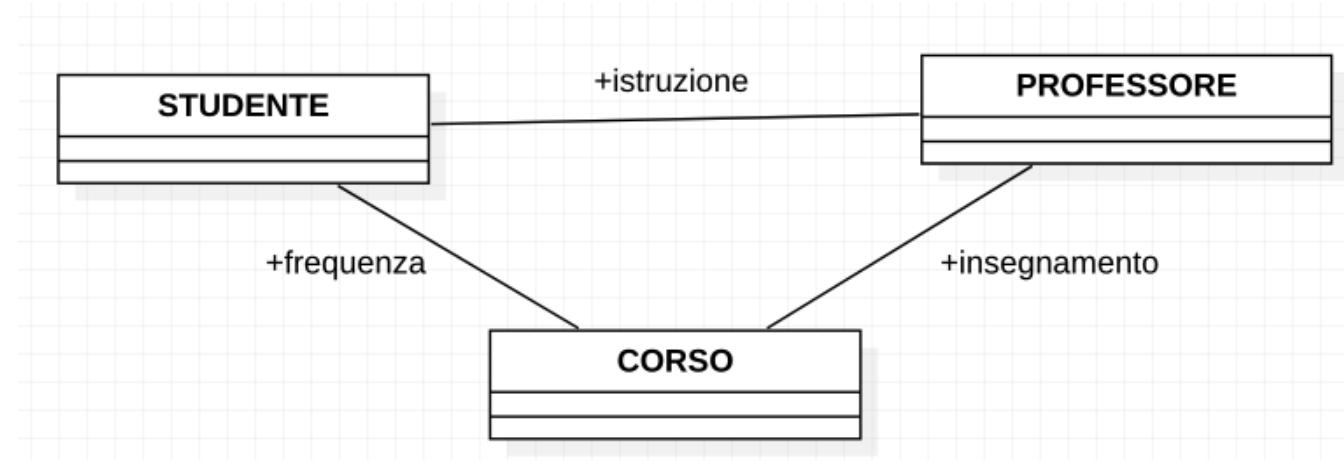


-
- Dal conteggio di occorrenze



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:

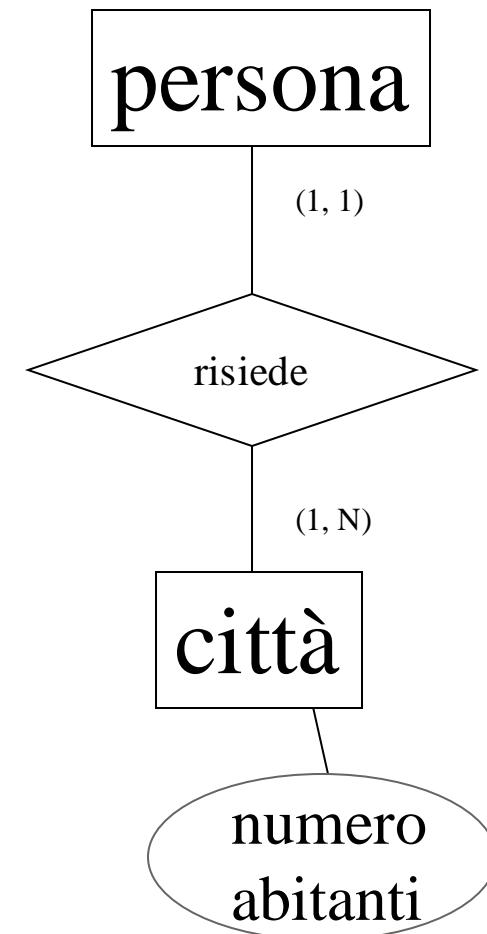
▪ Città	E	200
▪ Persona	E	1000000
▪ Risiede	R	1000000

- *Op₁: memorizza una nuova persona con la relativa città di residenza.*

- *Op₂: stampa tutti i dati di una città (incluso il numero di abitanti).*

- Tavola delle operazioni:

▪ Op ₁	I	500/giorno
▪ Op ₂	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 5000¹ L

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



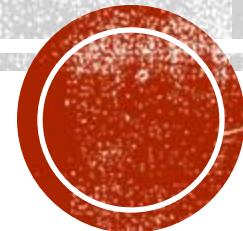
- 10000 accessi/giorno

¹ (#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 della modellazione concettuale (classi/entità, associazioni/relazioni)

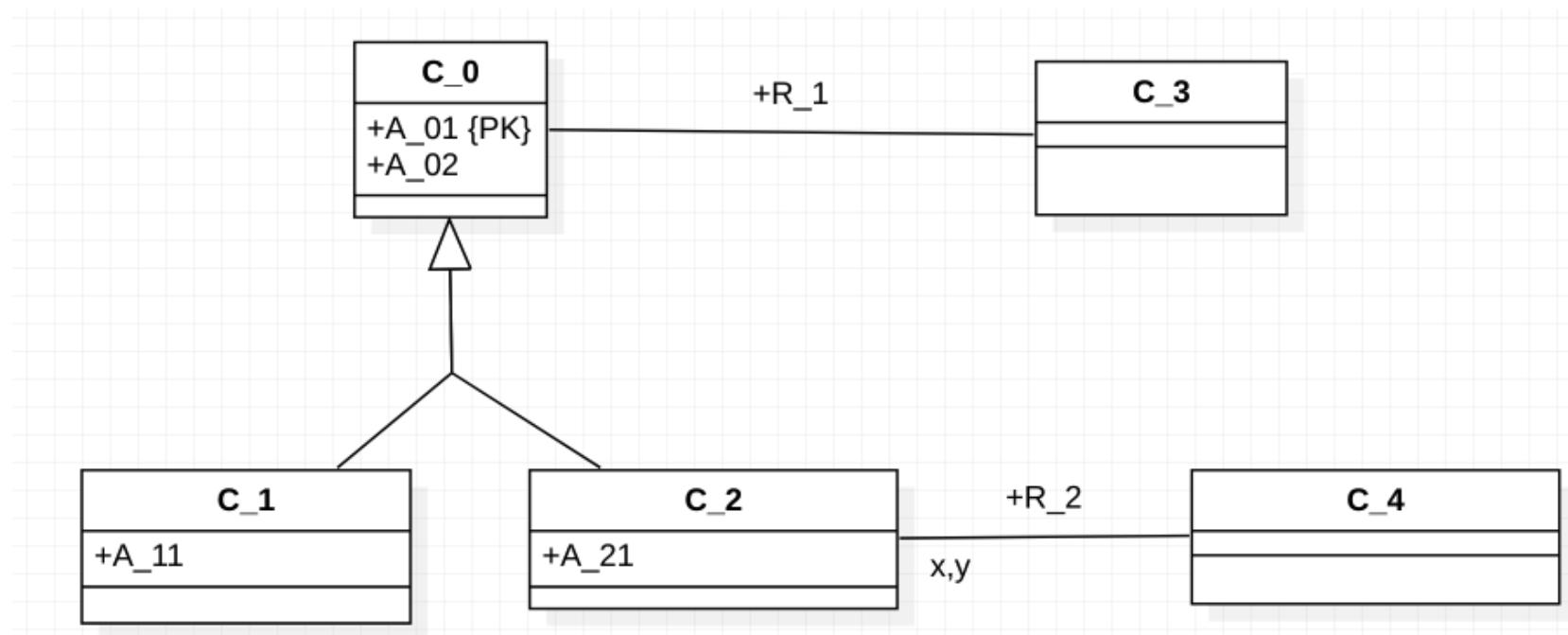


ELIMINAZIONE DELLE GENERALIZZAZIONI

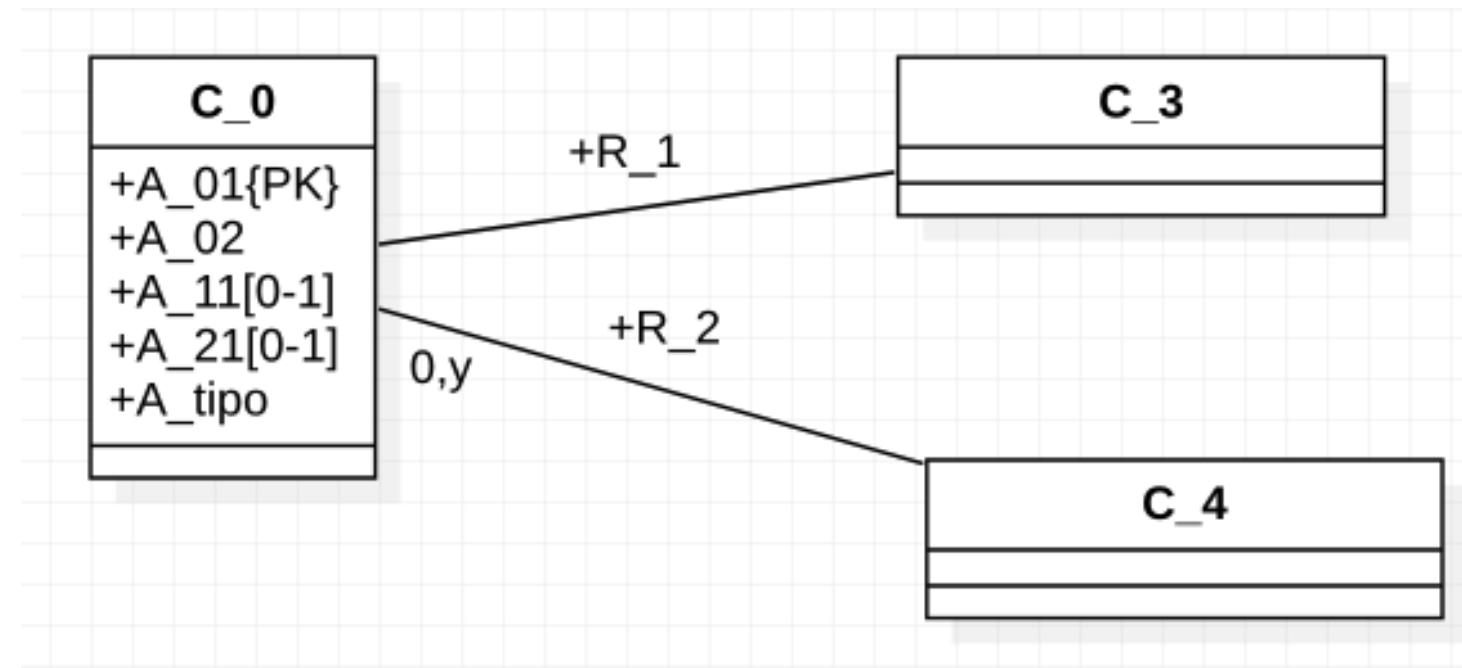
- a) Accorpamento delle figlie della generalizzazione nel padre.**
- b) Accorpamento del padre della generalizzazione nelle figlie.**
- c) Sostituzione della generalizzazione con associazioni.**



SCHEMA INIZIALE



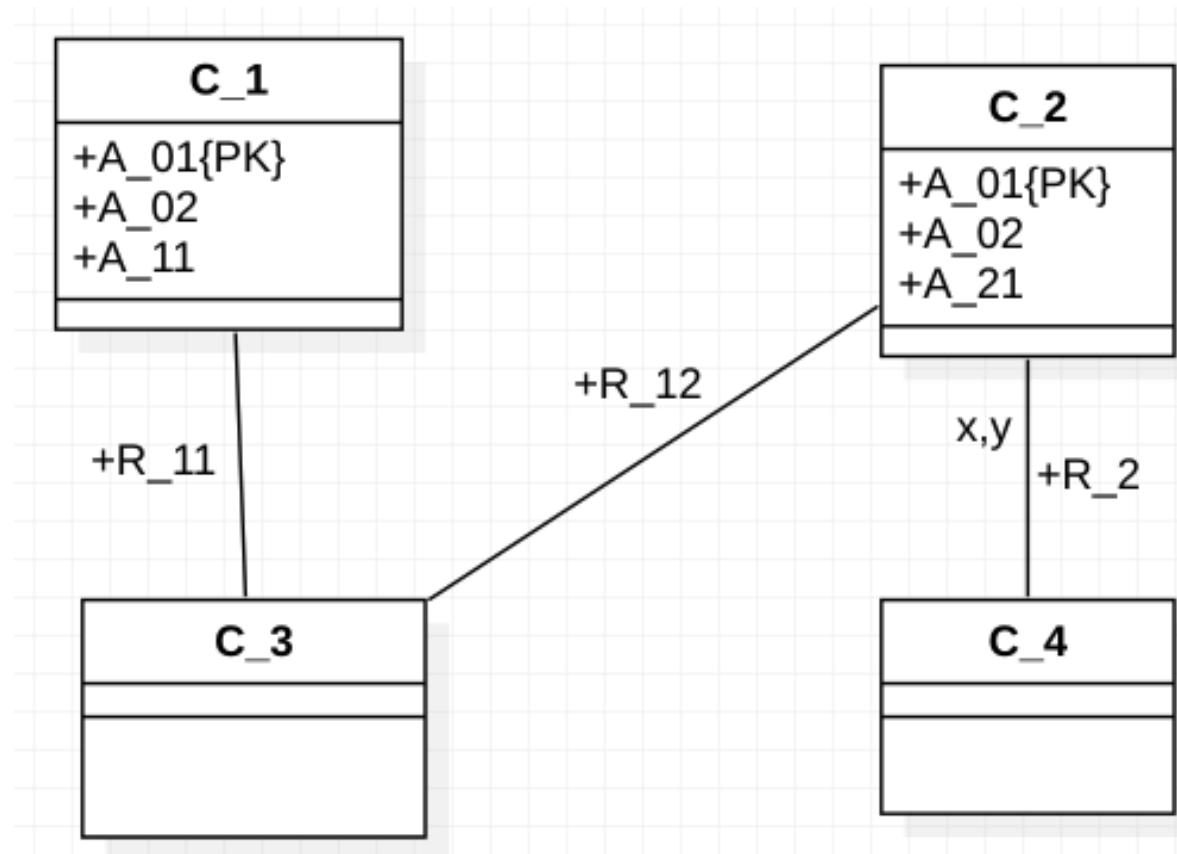
(A) ACCORPAMENTO DELLE FIGLIE DELLA GENERALIZZAZIONE NEL PADRE



Le entità C₁ e C₂ vengono eliminate e le loro proprietà vengono aggiunte al padre C₀.

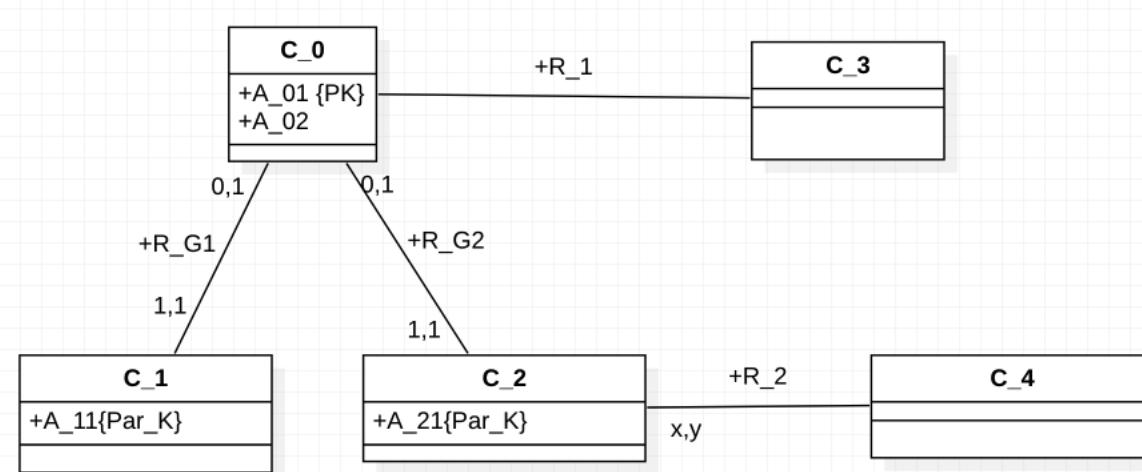


(B) ACCORPAMENTO DEL PADRE DELLA GENERALIZZAZIONE NELLE FIGLIE



(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.

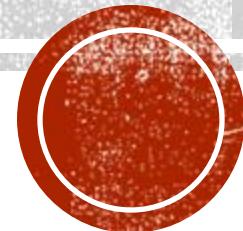


GENERALIZZAZIONI A PIÙ LIVELLI

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



ELIMINAZIONE ATTRIBUTI MULTIVALORE

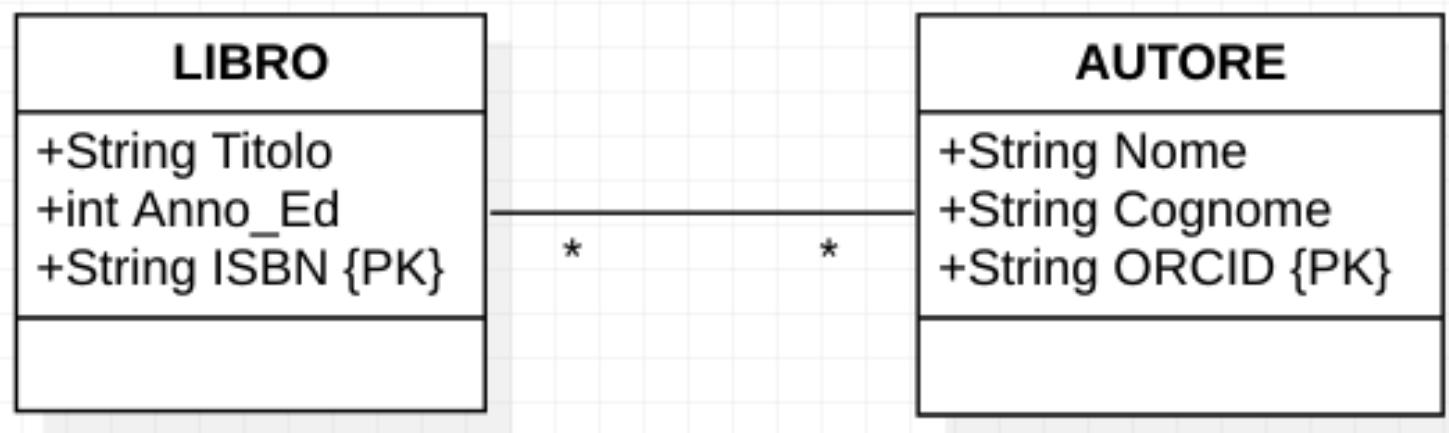


ELIMINAZIONE ATTRIBUTI MULTIVALORE

- 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»



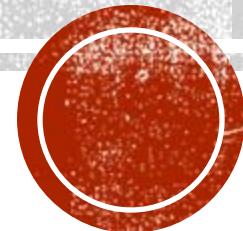
3) REPLICARE L'ATTRIBUTO NELLA CLASSE (QUANTE VOLTE??)



- 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

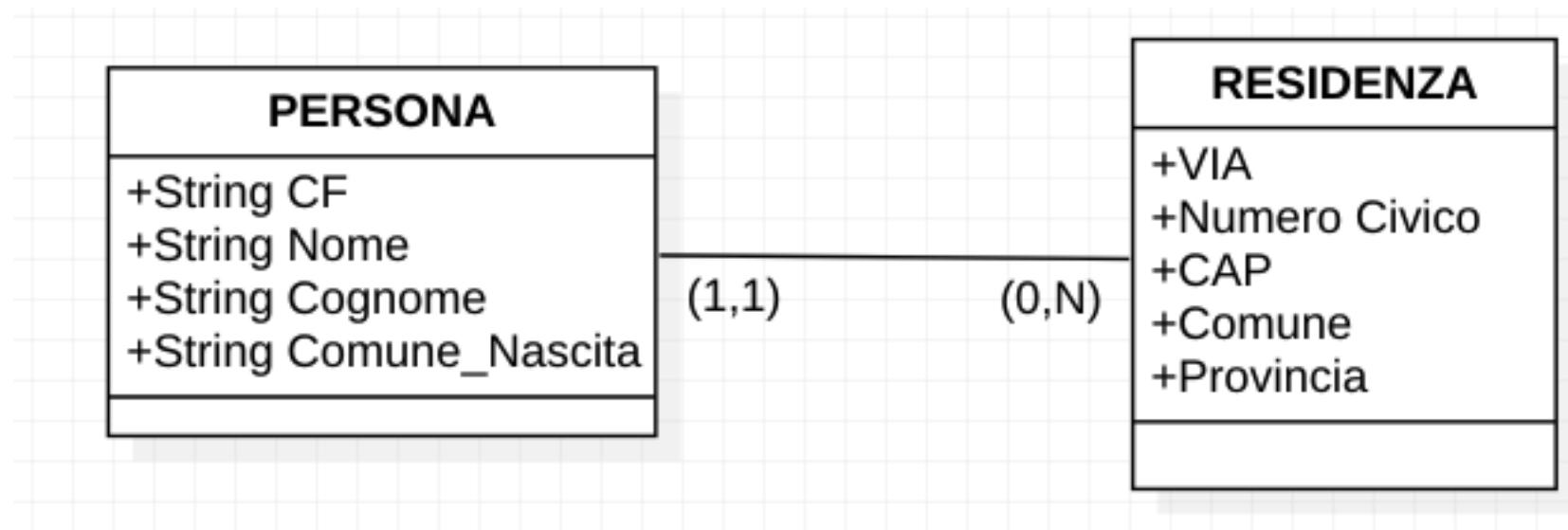
- +String CF
- +String Nome
- +String Cognome
- +Residenza Residenza
- +String Comune_Nascita

RESIDENZA

- +VIA
- +Numero Civico
- +CAP
- +Comune
- +Provincia



1) INTRODUZIONE DI UNA CLASSE PER L'ATTRIBUTO STRUTTURATO

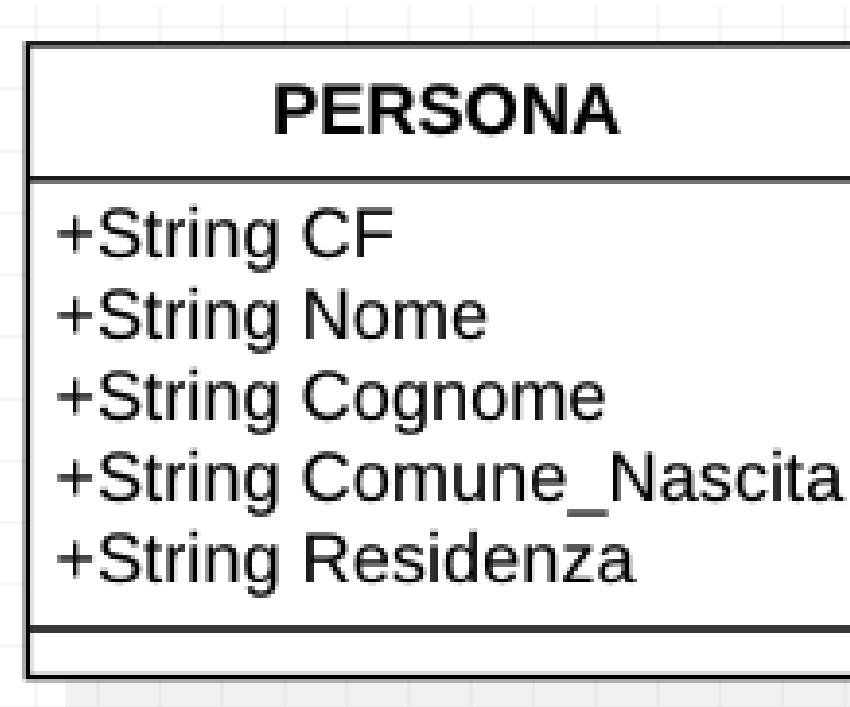


2) ESTRAZIONE DEGLI ATTRIBUTI DELLA CLASSE

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

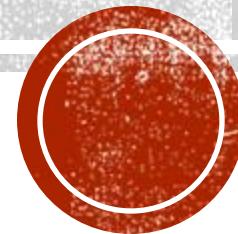


3) TRASCURARE LA STRUTTURA DELL'ATTRIBUTO





**PARTIZIONAMENTO/ACCORPAMENTO DI
ENTITÀ/ASSOCIAZIONI**

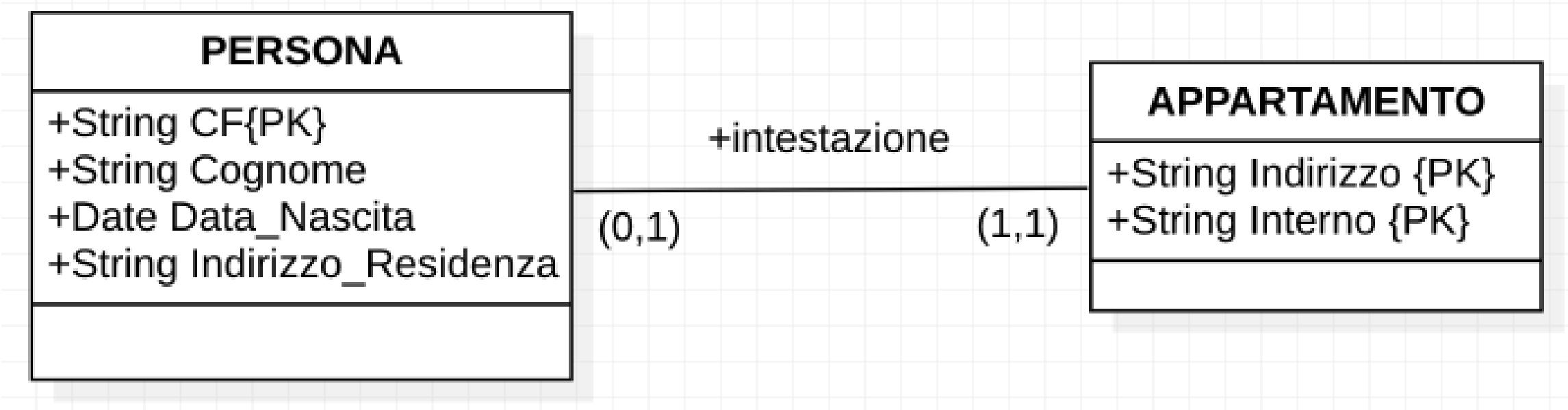


ACCORPAMENTI DI ENTITÀ

- 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



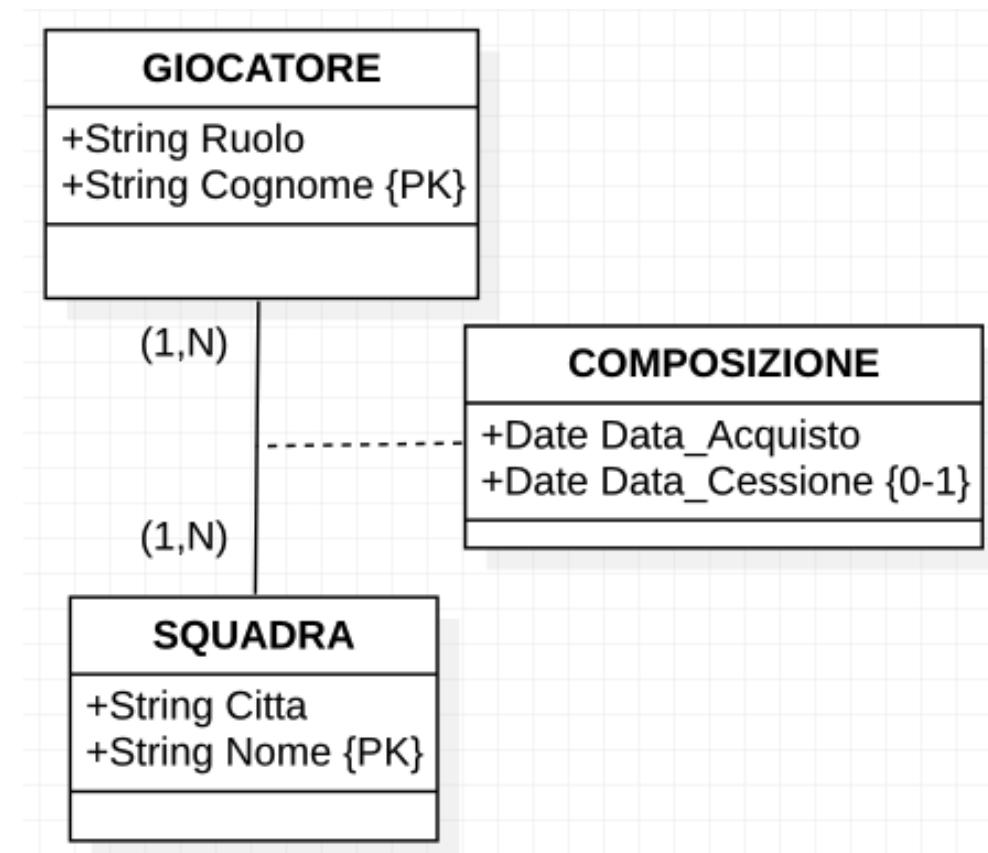
ACCORPAMENTI DI ENTITÀ – ESEMPIO (2)

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

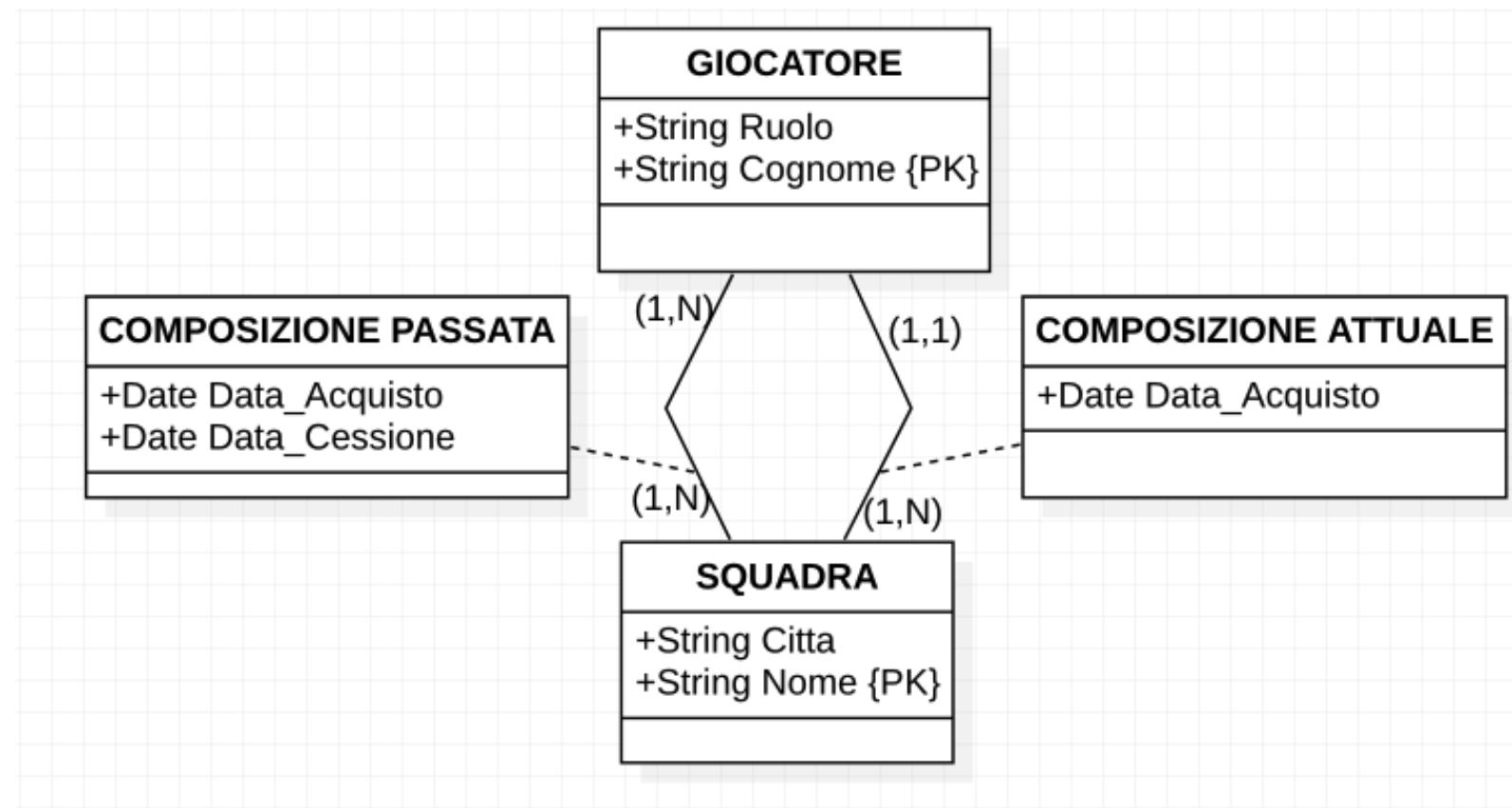
PERSONA
+String CF{PK}
+String Cognome
+String Data_Nascita
+String Indirizzo_Residenza
+String Indirizzo_Appartamento{0-1}
+String Interno_Appartamento{0-1}



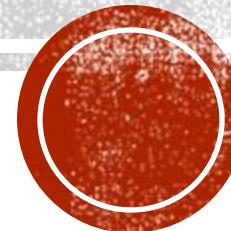
DECOMPOSIZIONE DI ASSOCIAZIONI - ESEMPIO



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*

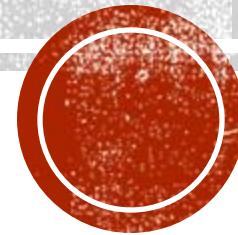




FINE

Per eventuali domande: (in ordine di preferenza personale)

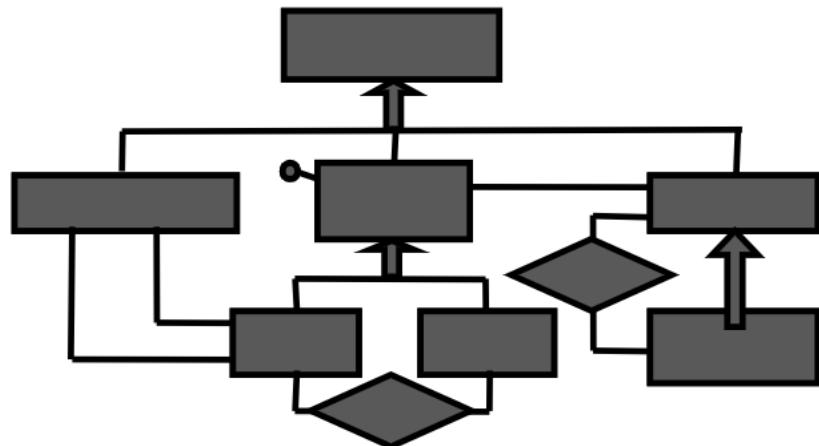
- Ora.
- Chat di Teams
- Mail: silvio.barra@unina.it



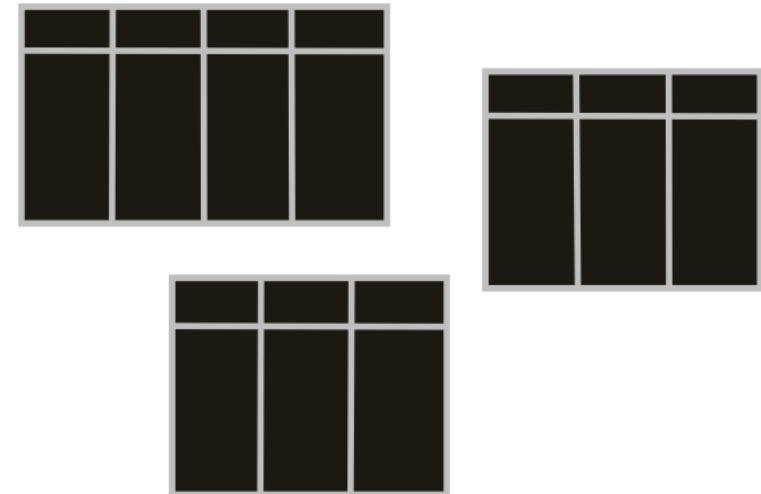
BASI DI DATI I

PASSAGGIO DAL MODELLO CONCETTUALE A QUELLO LOGICO

Abbiamo visto come costruire un modello concettuale del nostro minimondo

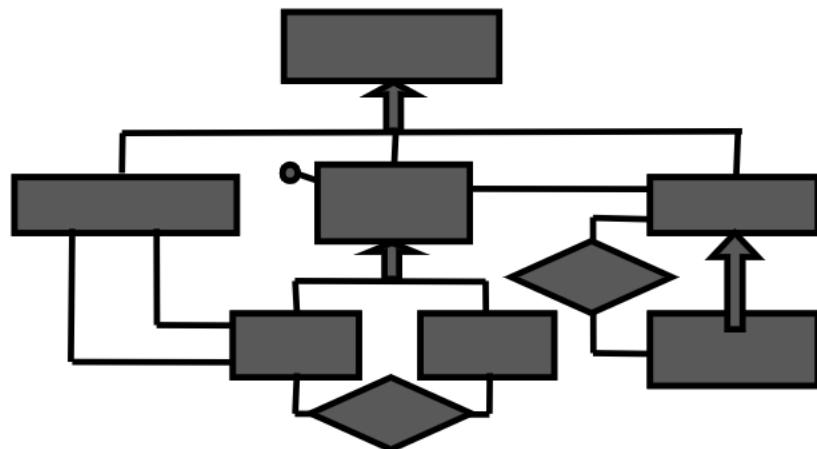


Abbiamo visto in cosa consiste il modello logico, che è la successiva fase di progettazione



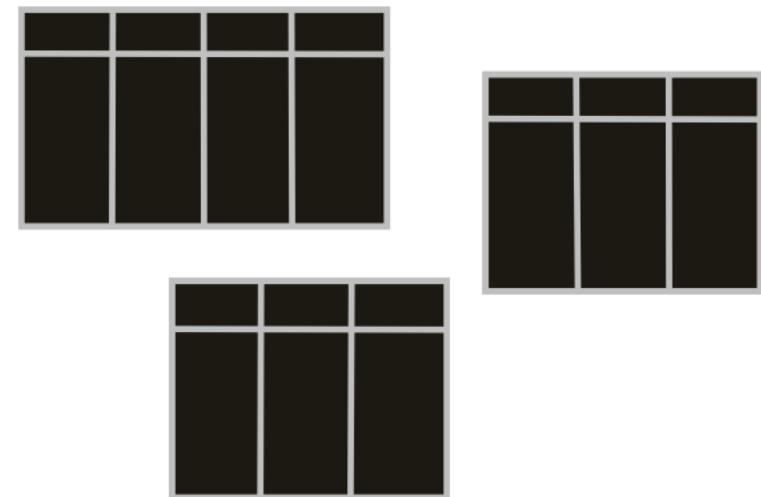
PASSAGGIO DAL MODELLO CONCETTUALE A QUELLO LOGICO

Abbiamo visto come costruire un modello concettuale del nostro minimondo

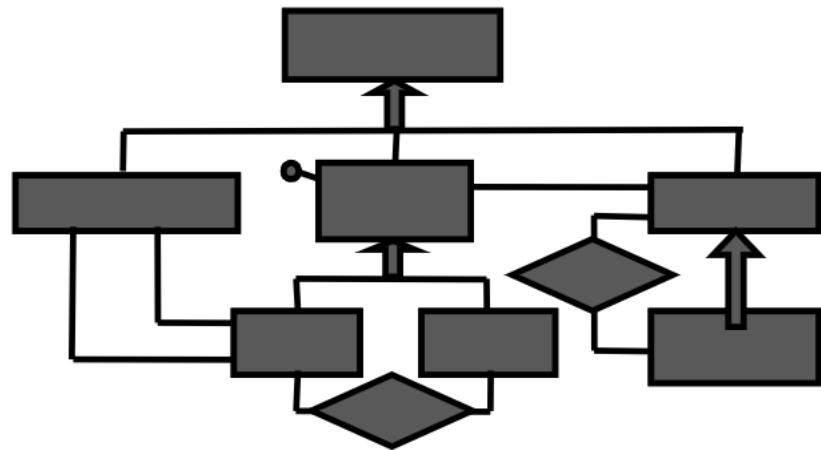


Processo di traduzione
→

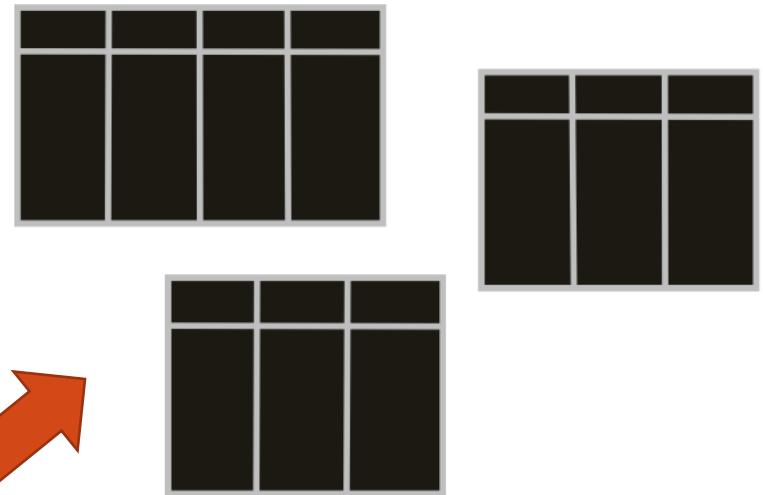
Abbiamo visto in cosa consiste il modello logico, che è la successiva fase di progettazione



STEP INTERMEDI



Ottimizzazione
del diagramma



Applicazione regole
di traduzione



TRADUZIONE VERSO UN MODELLO LOGICO

- **Modello relazionale:**

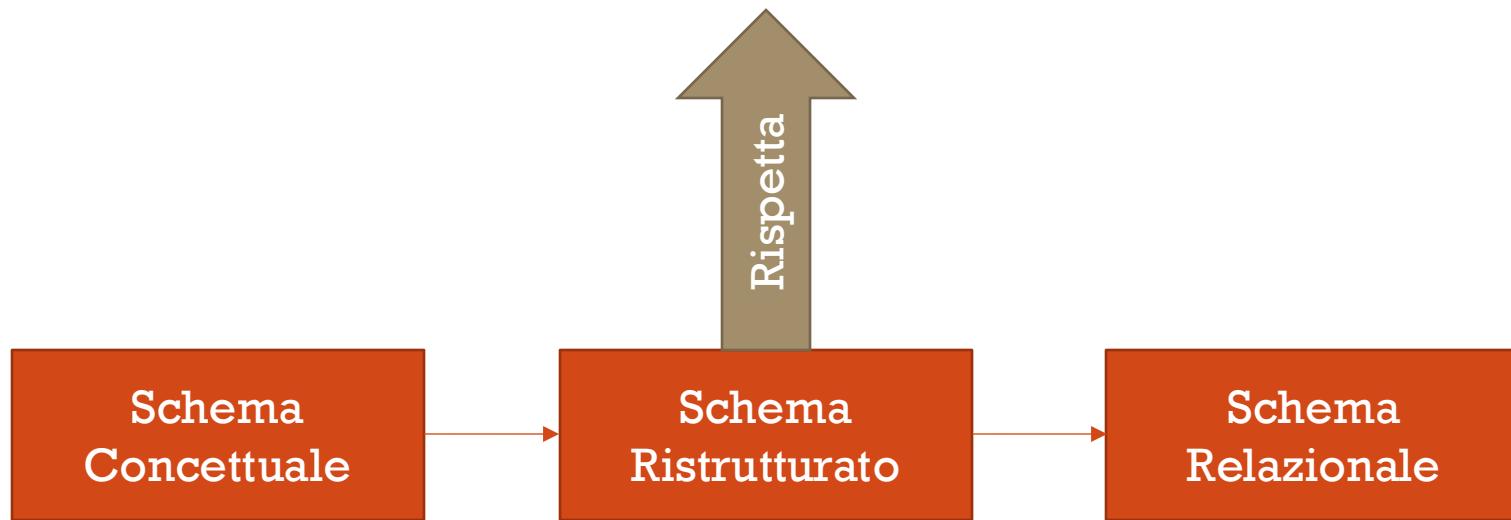
- Per ogni entità, uno schema di relazione con lo stesso nome avente i medesimi attributi dell'entità e per chiave il suo identificatore.
- Per ogni relazione/associazione nel modello concettuale, uno schema di relazione con lo stesso nome avente per attributi gli attributi della relazione e per chiave 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

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

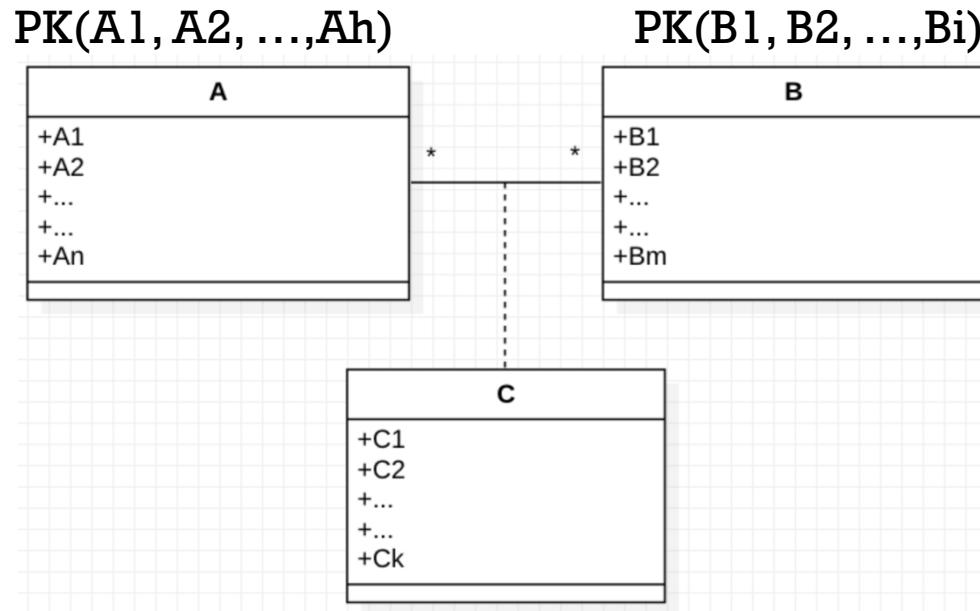


MAPPING ASSOCIAZIONI

- 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(A₁, A₂, ..., A_h, ..., A_n)

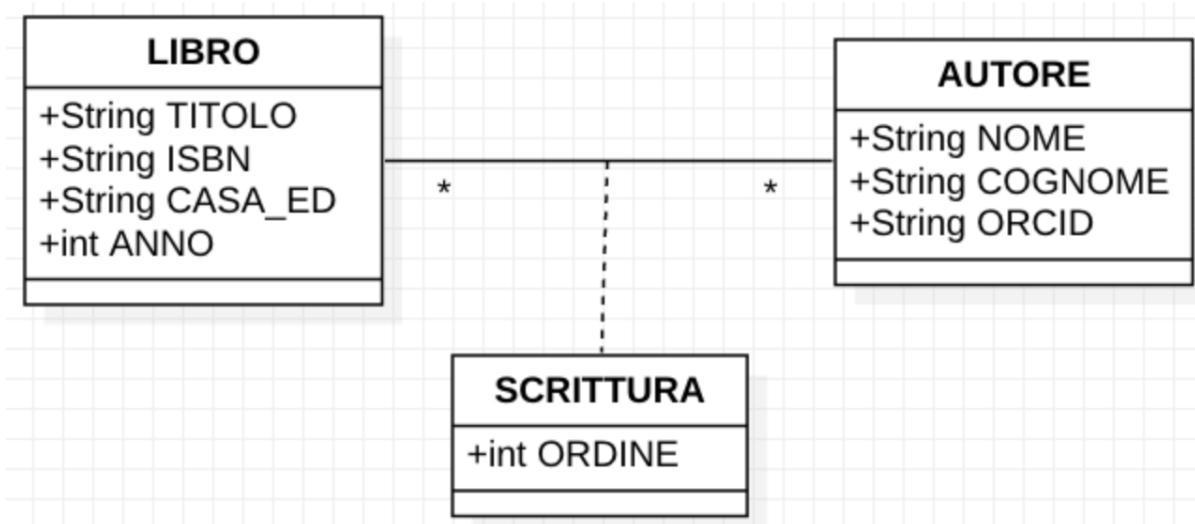
B(B₁, B₂, ..., B_i, ..., B_m)

C(A₁, A₂, ..., A_h, B₁, B₂, ..., B_i, C₁, C₂, ..., C_k)

PK



ESEMPIO



LIBRO(Titolo, ISBN, Casa_Ed, Anno)

AUTORE(Nome, Cognome, ORCID)

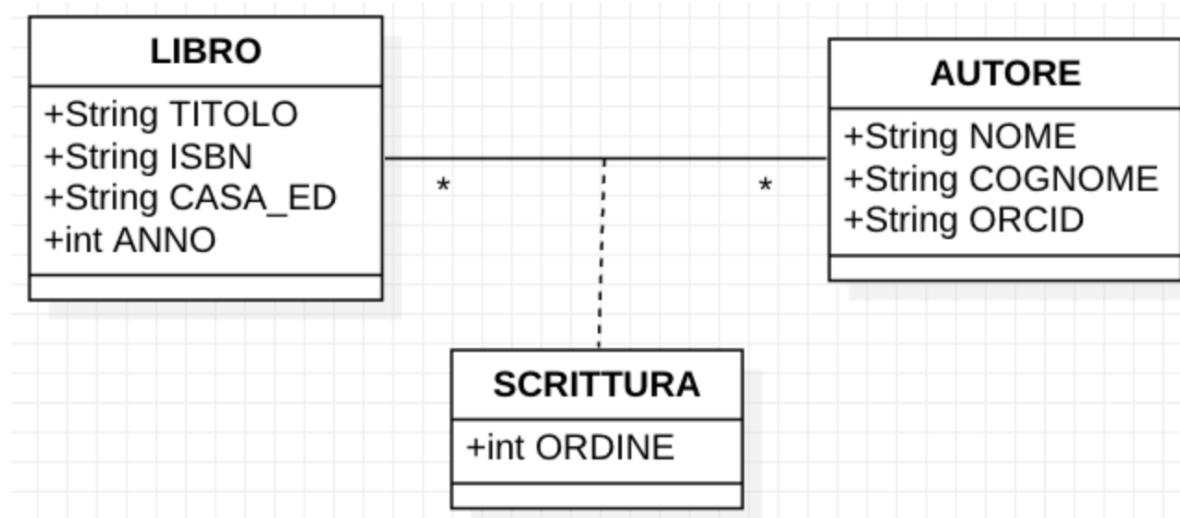
SCRITTURA(ISBN, ORCID, Ordine)

SCRITTURA.ISBN → LIBRO.ISBN

SCRITTURA.ORCID → AUTORE.ORCID



RINOMINARE



LIBRO(Titolo, ISBN, Casa_Ed, Anno)

AUTORE(Nome, Cognome, ORCID)

SCRITTURA(ISBN , ORCID, Ordine)

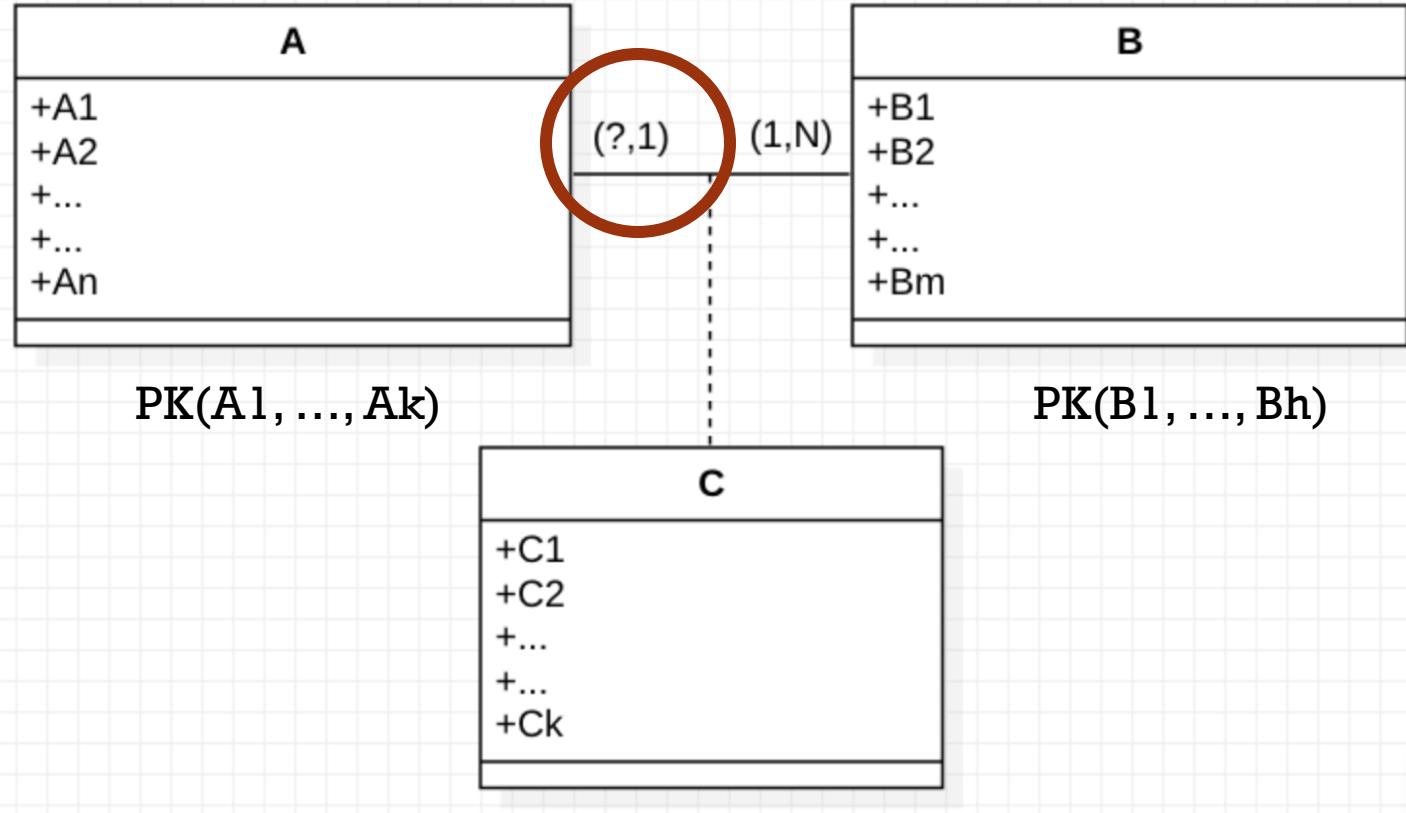
$\text{dom}(\text{SCRITTURA.ISBN}) = \text{dom}(\text{LIBRO.ISBN})$

$\text{dom}(\text{SCRITTURA.ORCID}) = \text{dom}(\text{AUTORE.ORCID})$

sono gli stessi domini delle relative chiavi primarie



ASSOCIAZIONI 1:N (CASO GENERALE)



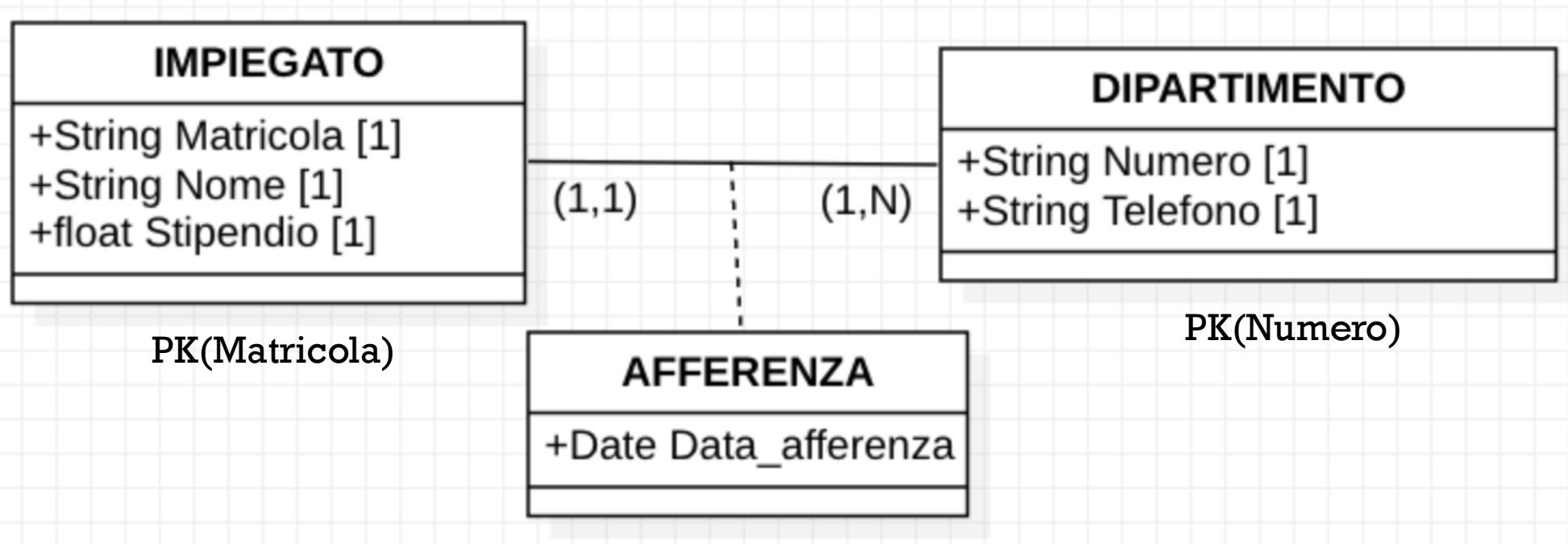
È necessario distinguere 3 situazioni

- 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)

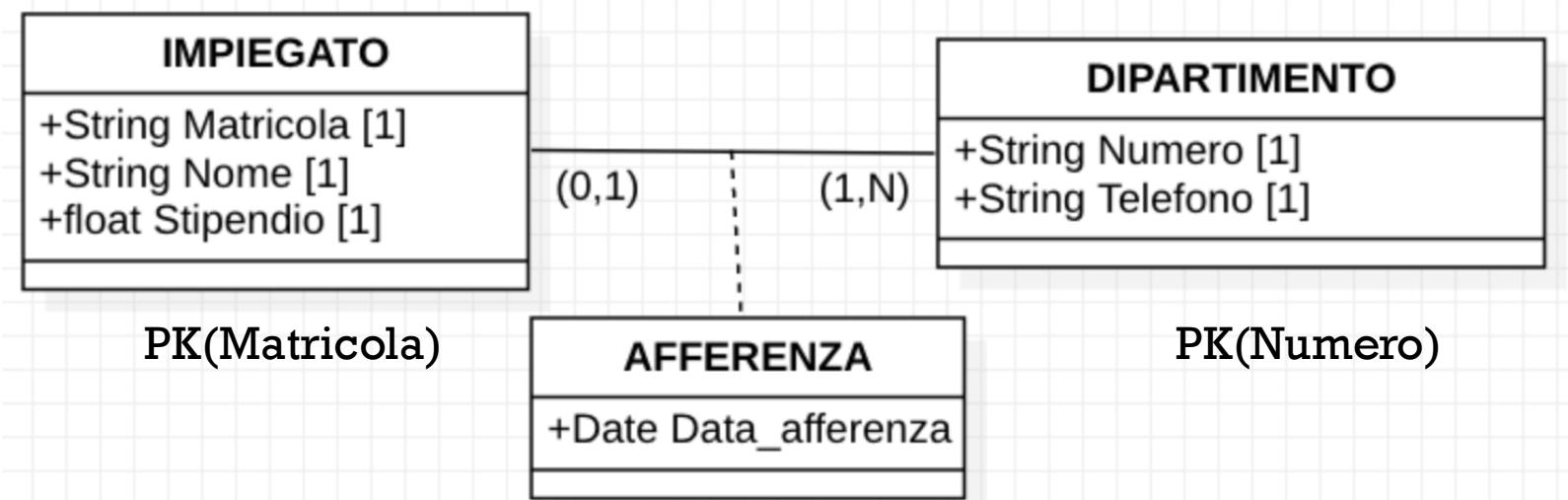
IMPIEGATO(Matricola, Nome, Stipendio, Numero_Dipartimento, Data_Afferenza)

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)

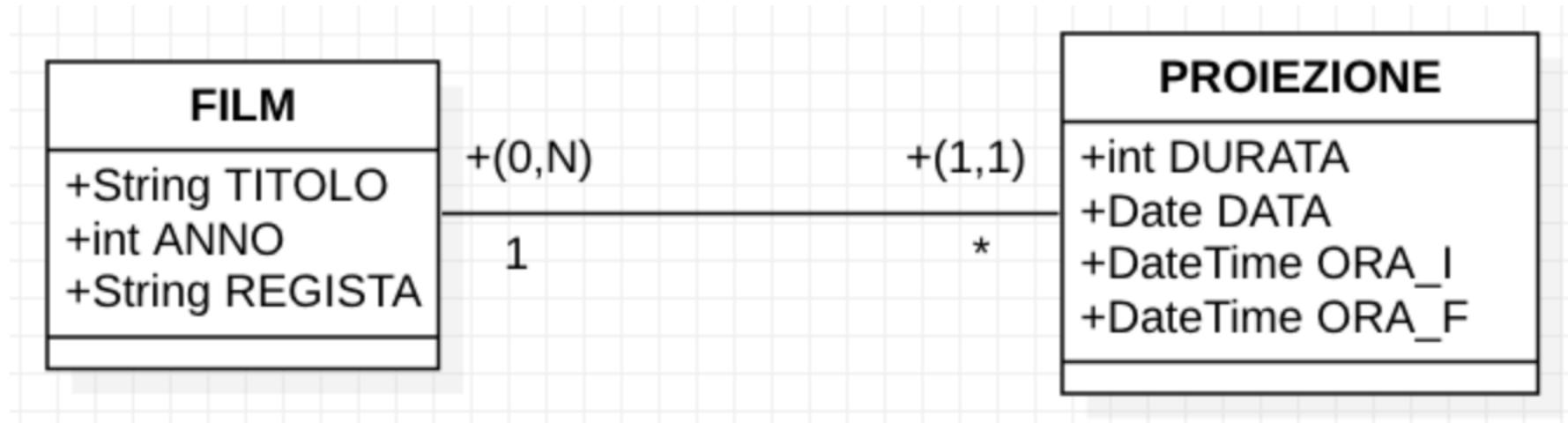
Matricola_I → IMPIEGATO.Matricola

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)

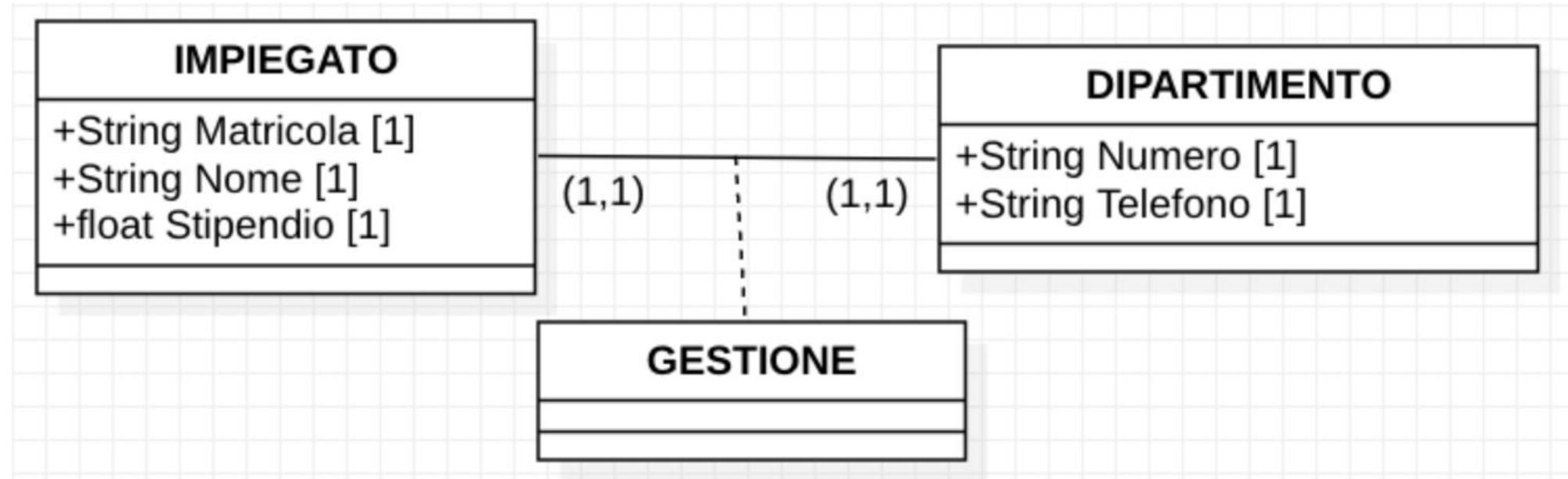
PK (Titolo, Anno, Data, Ora_I)

Chiave Parziale

Foreign Key



ASSOCIAZIONI 1:1 (PARTECIPAZIONE TOTALE)

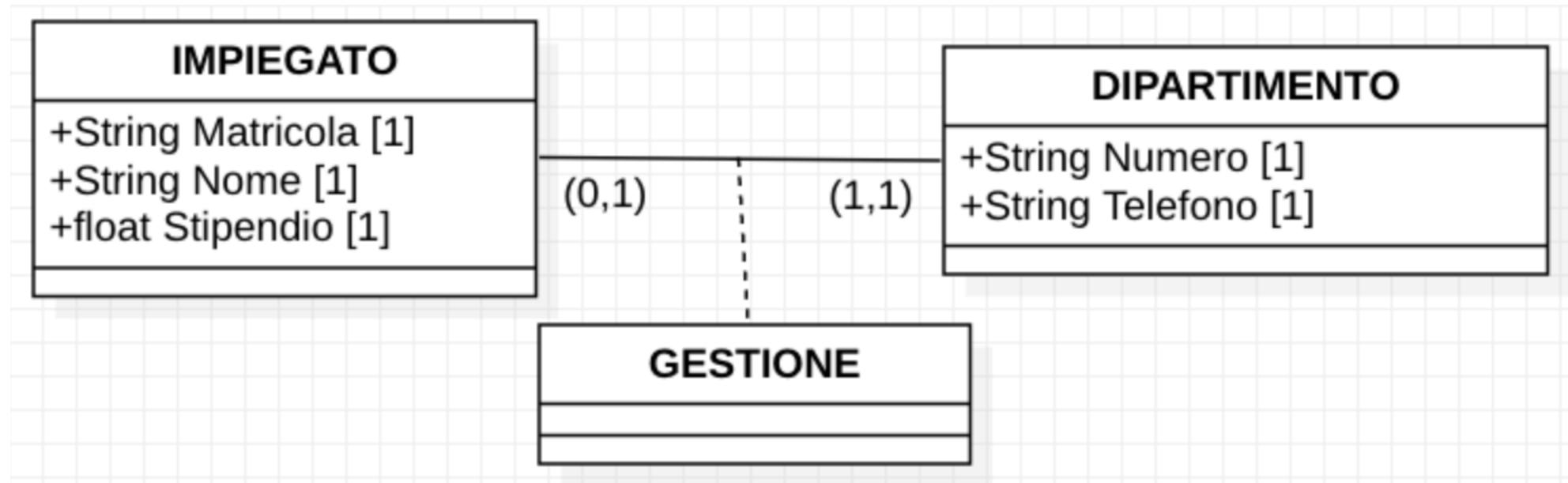


DIPARTIMENTO(Numero, Telefono, Matricola_I)
 $\text{Matricola_I} \rightarrow \text{IMPIEGATO}.\text{Matricola}$
IMPIEGATO(Matricola, Nome, Stipendio)

O DIPARTIMENTO(Numero, Telefono)
IMPIEGATO(Matricola, Nome, Stipendio, Numero_D)
 $\text{Numero_D} \rightarrow \text{DIPARTIMENTO}.\text{Numero}$



ASSOCIAZIONI 1:1 (PARTECIPAZIONE PARZIALE)

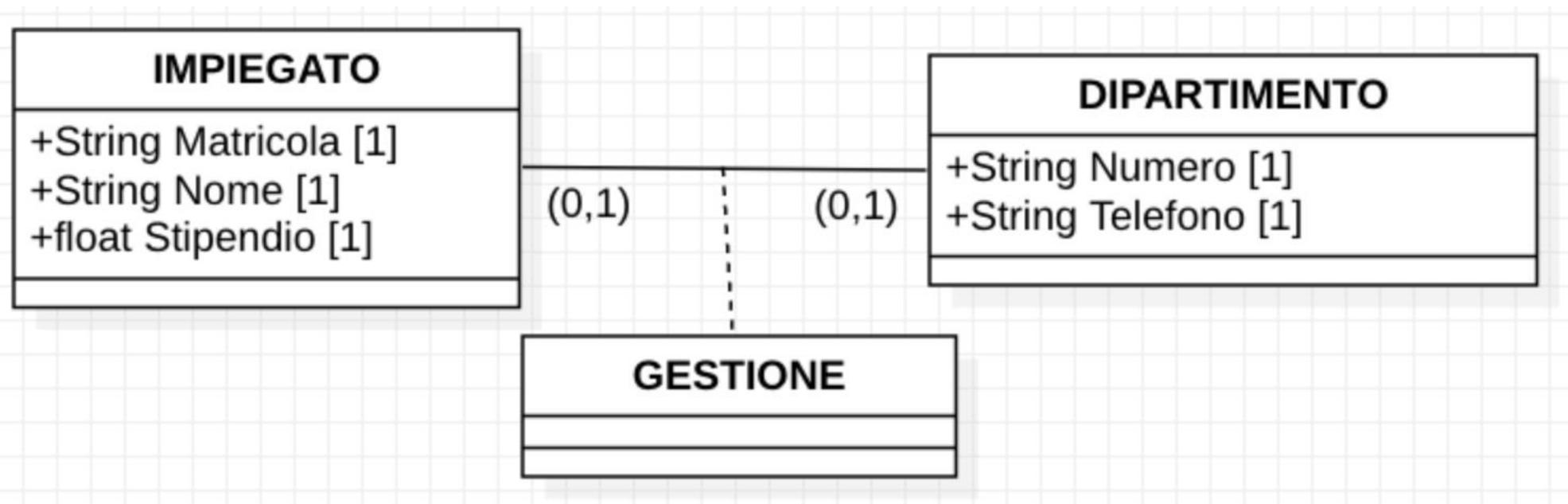


DIPARTIMENTO(Numero, Telefono, Matricola_I)
Matricola_I → IMPIEGATO.Matricola
IMPIEGATO(Matricola, Nome, Stipendio)

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)

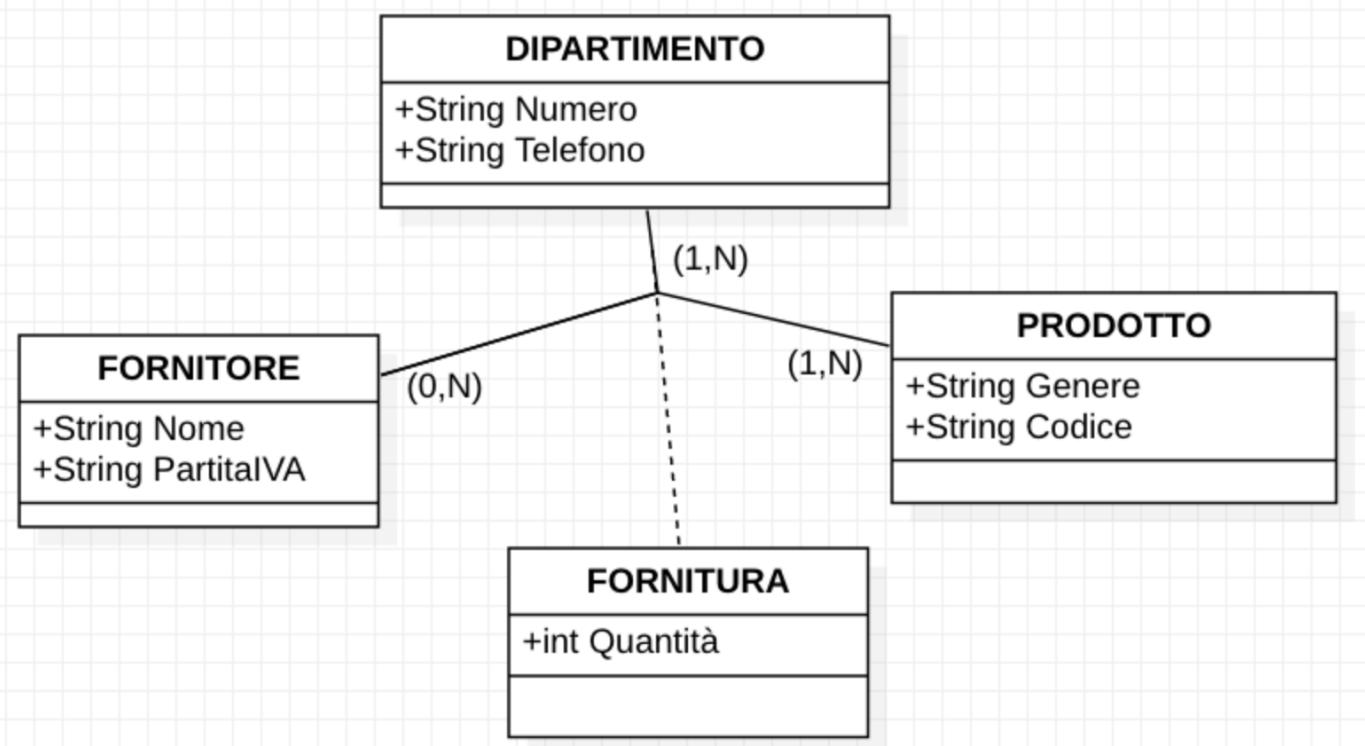
Matricola_I → IMPIEGATO.Matricola

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)

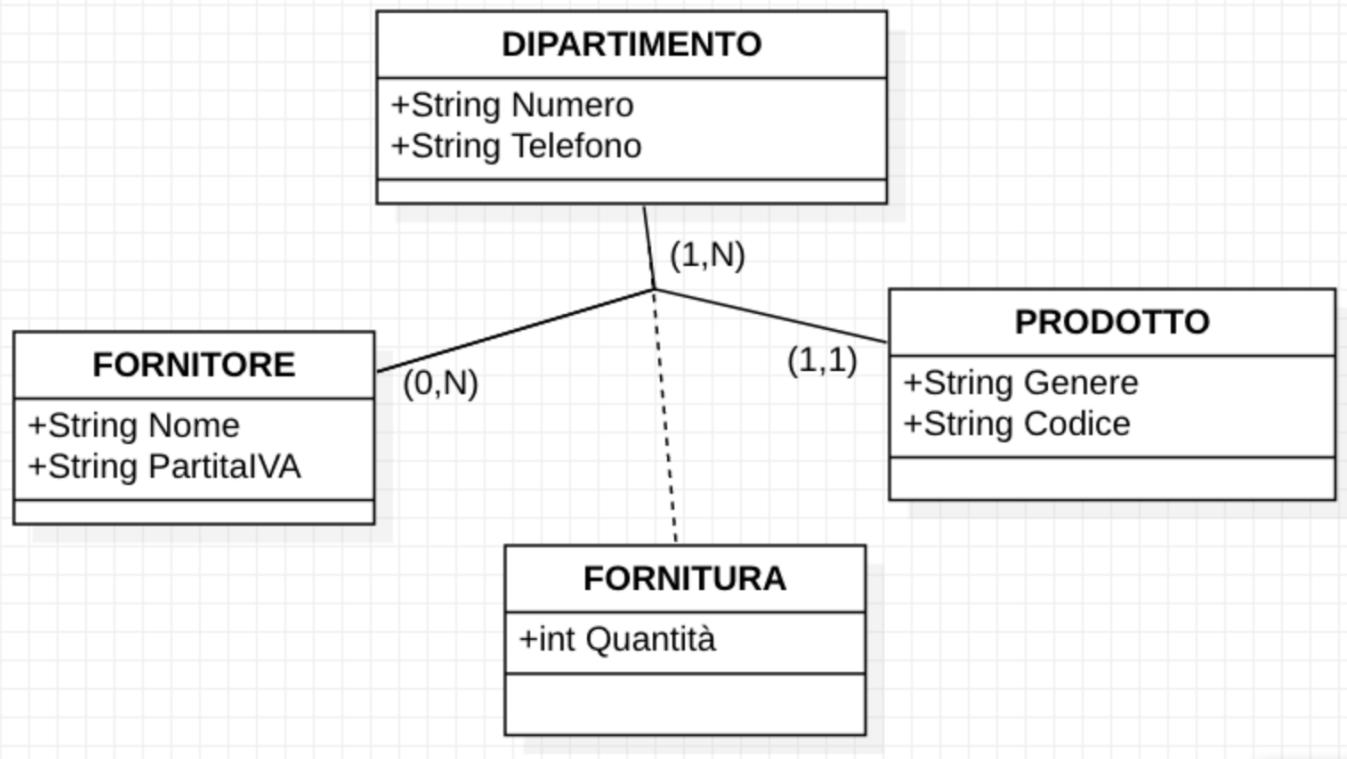
Fornitura(PartitaIVAF, NumeroD, CodiceP)

PartitaIVAF → Fornitore.PartitaIVA

NumeroD → Dipartimento.Numero

CodiceP → Prodotto.Codice

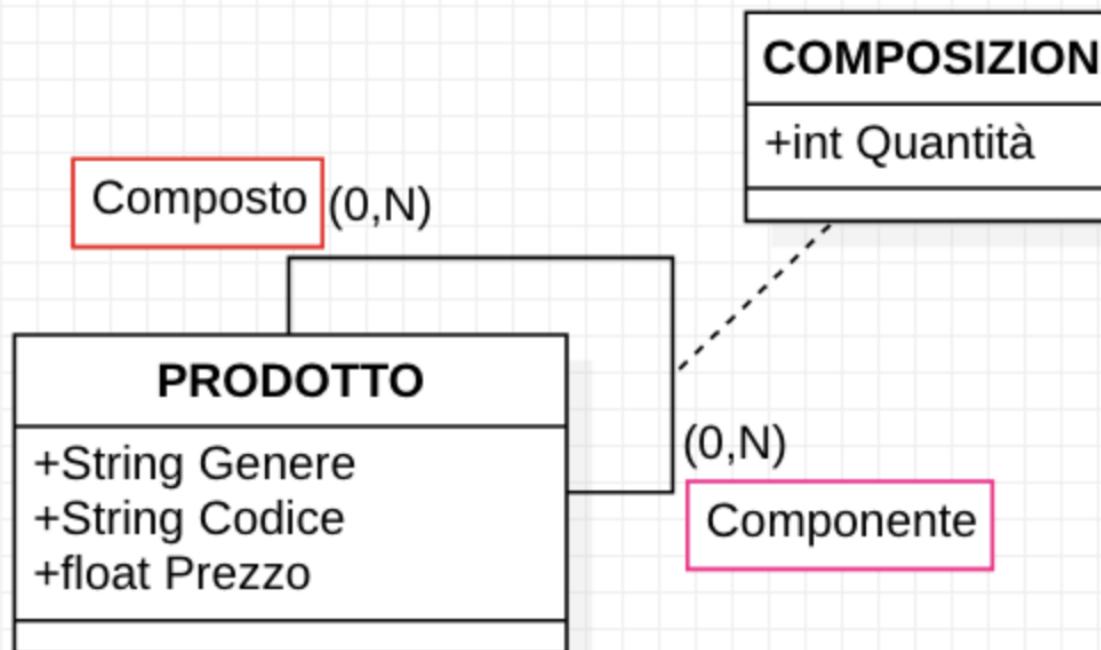
ASSOCIAZIONI N-ARIE



- 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)



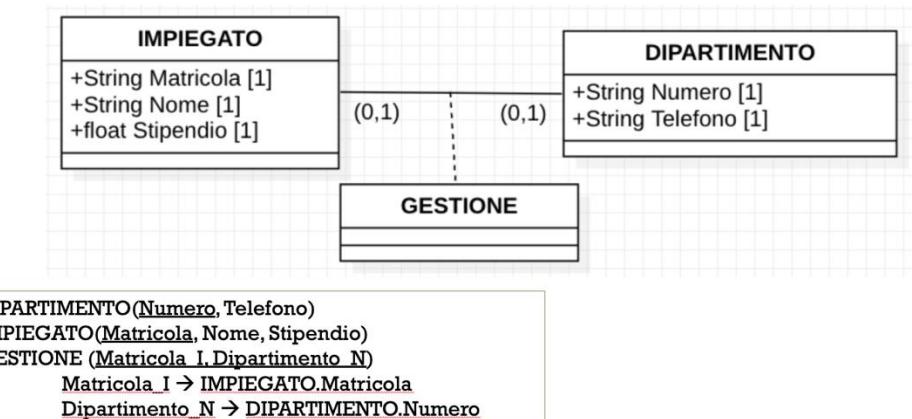
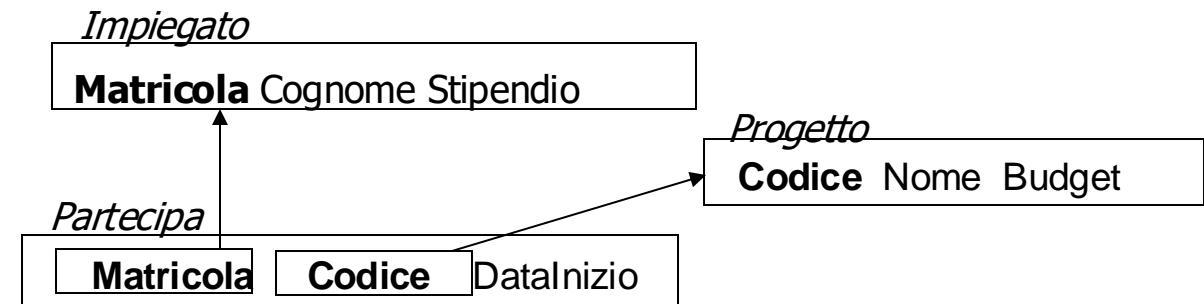
DOCUMENTAZIONE DI SCHEMI LOGICI

- 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.

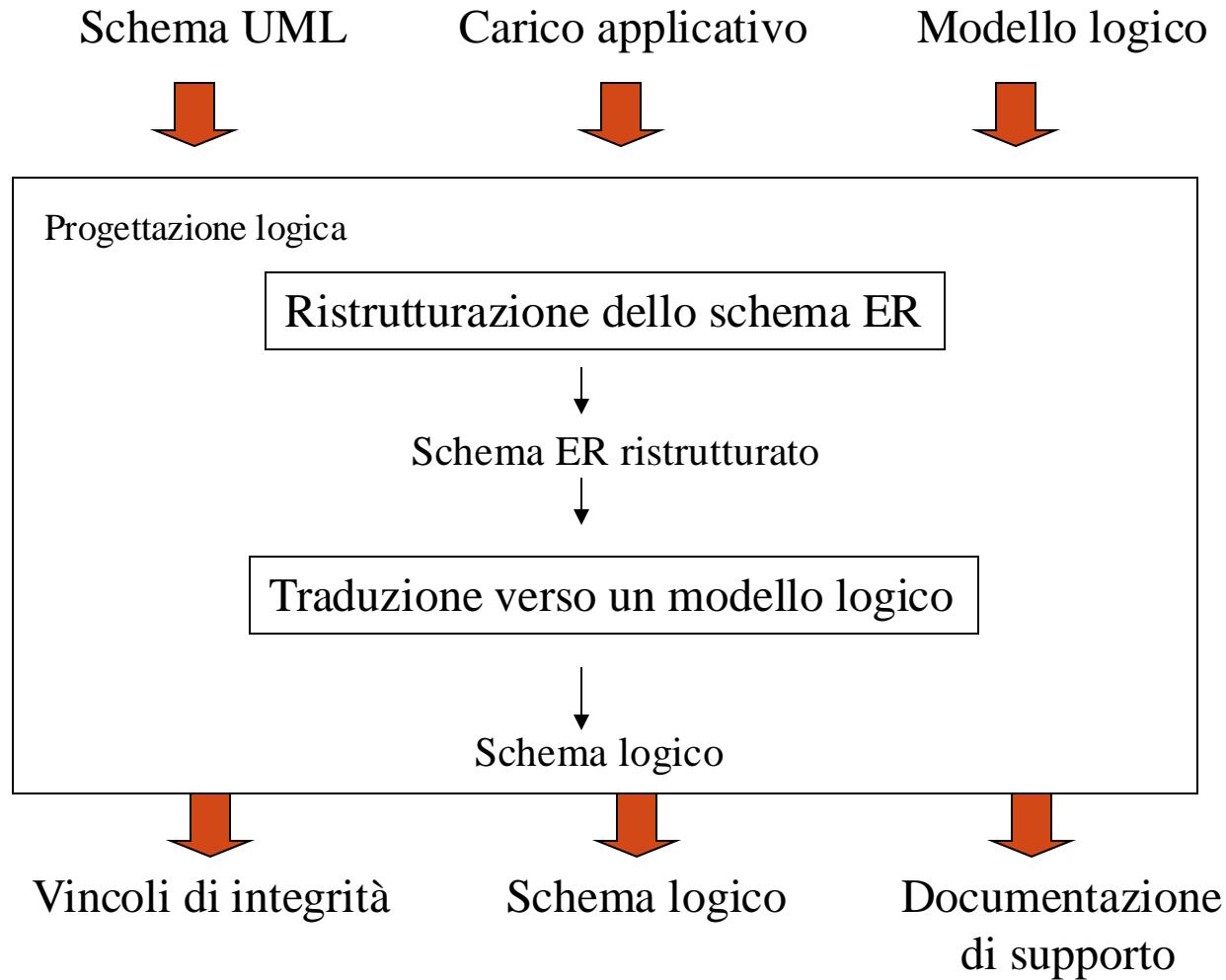


FORMALISMO GRAFICO

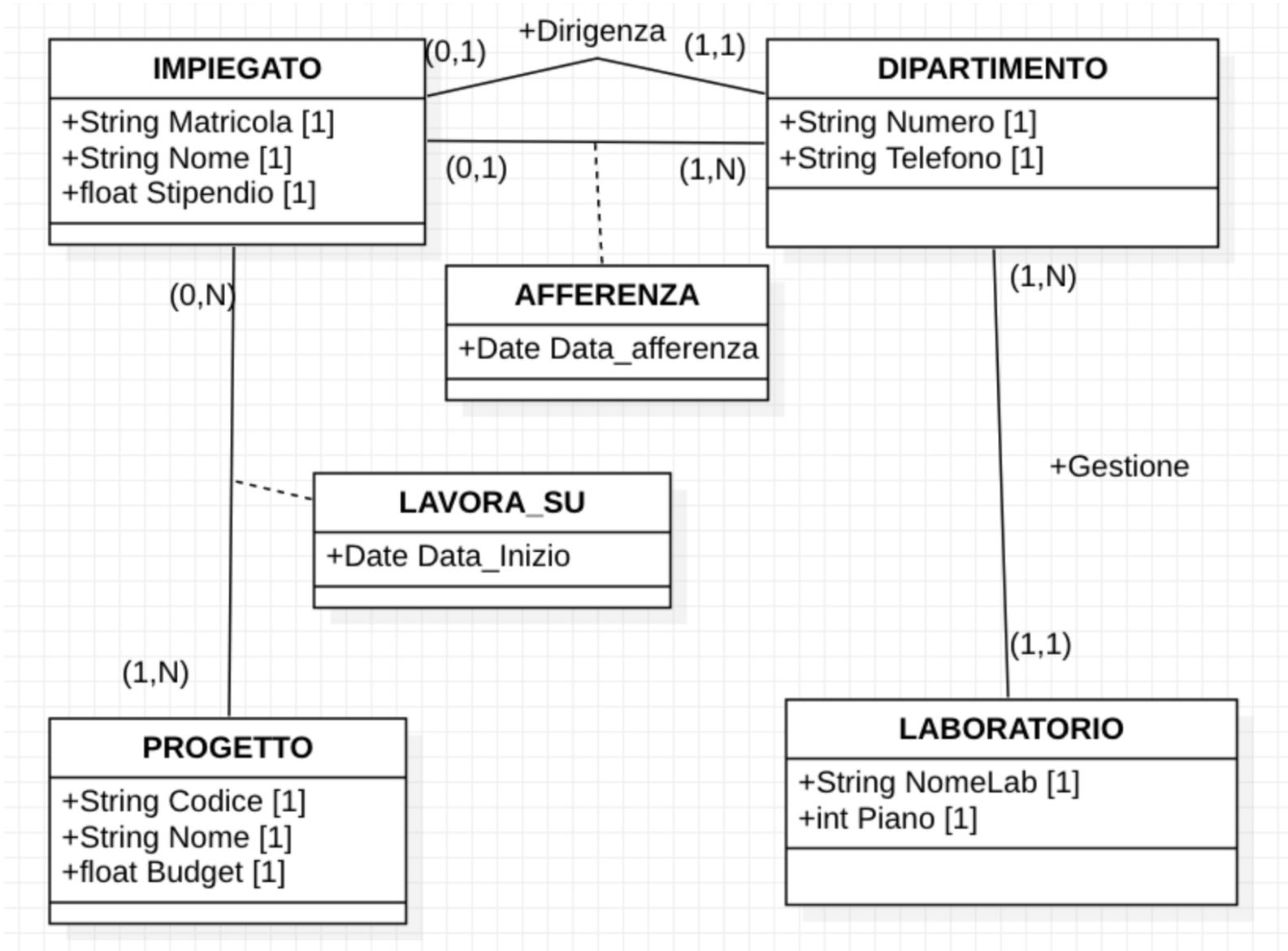
- Le frecce indicano i vincoli di integrità, in grassetto sono indicate le chiavi.
- In alternativa, le chiavi sono indicate con gli attributi sottolineati una volta, e le chiavi esterne con una doppia sottolineatura



PROGETTAZIONE LOGICA DI BASI DI DATI



ESEMPIO



INFO SULLA LEZIONE

- Tutto il capitolo 9
 - Contiene sia la parte di mapping che la parte di ristrutturazione.
 - Il processo è diviso in 9 fasi
 - L'ultima riguarda anche la traduzione delle categorie che non abbiamo visto, ma che potrebbe essere interessante tenere a mente

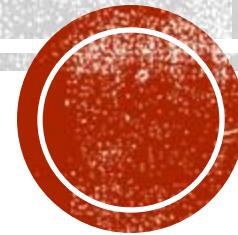




FINE

Per eventuali domande: (in ordine di preferenza personale)

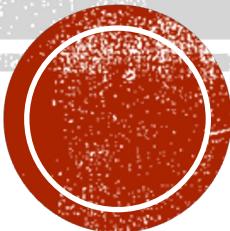
- Ora.
- Chat di Teams
- Mail: silvio.barra@unina.it



BASI DI DATI I

- Structured Query Language
- Data Definition Language – SQL
- Comandi DDL
- Qualche Esempio/Esercizio

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.



STORIA DI SQL (2)

- **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** come linguaggio per interfacciarsi con una DB relazionale sperimentale (SYSTEM R)
 - Facile da imparare e utilizzare;
- **1976:** viene definita una versione rivista, il **SEQUEL/2**, ridenominata **SQL** (*Structured Query Language*)
- **1979:** Il primo prodotto basato su SQL viene chiamato **Oracle**, lanciato dalla Relational Software, Inc.
- **1981:** IBM annuncia un prodotto SQL denominato **SQL/Data System**;
- **1983:** viene rilasciato il DBMS relazionale **DB2** compatibile con SQL/DS.



STORIA DI SQL (2)

- **1986:** grazie all'ANSI (American National Standard Institute) e alla ISO (International Standard Organization) nasce **SQL-1** (anche detto SQL-86)
- **1992:** revisioni ed estensioni fanno nascere **SQL2** (anche detto SQL-92)
- **1999:** nasce **SQL3** meglio conosciuta come SQL
- **2003-2006:** due aggiornamenti successivi (SQL:2003 e SQL:2006) includono caratteristiche per l'XML
- **2008:** nasce SQL:2008 che introduce caratteristiche per le basi di dati ad oggetti, prequel dell'aggiornamento SQL:2011
- *Oggi SQL è implementato da tutti i principali fornitori di DBMS, ed è il linguaggio per database più usato al mondo.*
- 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.

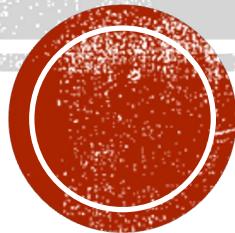


TERMINOLOGIA SQL

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



DEFINIZIONE DI DATI, SCHEMI E VINCOLI IN SQL



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

Un'istanza del database “Company”

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	999687777	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	26000	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	
999687777	30	30.0	
999687777	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



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"

DOMINI E TIPI DI DATI PER GLI ATTRIBUTI IN SQL



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



SPECIFICA DEI VINCOLI DI BASE PER SQL



VINCOLI DI BASE

- I vincoli di base per SQL sono dei seguenti tipi:
 - Vincoli sui valori null e valori predefiniti
 - Vincoli sulle tuple (CHECK)
 - Vincoli sui domini degli attributi
 - Vincoli di chiave
 - Vincoli di integrità referenziale



VINCOLI SUI VALORI NULL E PREDEFINITI

- Poiché SQL consente che un attributo abbia valore **null**, se si vuole impedire ciò si usa il vincolo
NOT NULL.
 - Tale vincolo è sempre specificato implicitamente 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.



VINCOLI SULLE TUPLE (CHECK)

- 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
```



VINCOLI DI CHIAVE

- I vincoli di chiave sono importanti all'interno dell'istruzione **CREATE TABLE**
- Per specificare una chiave primaria, dopo la specifica dell'attributo è necessario inserire il vincolo **PRIMARY KEY**.

Es: **CREATE TABLE IMPIEGATO**

(SSN char(9) PRIMARY KEY, ...)

- Se più attributi compongono la chiave primaria, il vincolo si inserisce a valle della definizione della tabella

CONSTRAINT IMPIEGATO_PK PRIMARY KEY(nome, cognome)



VINCOLI DI UNICITÀ

- È anche possibile definire dei vincoli di chiave alternativa (vincolo di unicità) per specificare che un determinato attributo, o sequenza di attributo, deve avere valori unici all'interno della tabella.
- Il vincolo **UNIQUE** specifica una chiave secondaria

Es: **CREATE TABLE DIPARTIMENTO**

**(Numero char(3) PRIMARY KEY,
Nome varchar(20) UNIQUE...)**

Oppure

CONSTRAINT IMPIEGATO_UNIQUE UNIQUE(Nome)



VINCOLI DI INTEGRITÀ REFERENZIALE

- Si specifica tramite la keyword **FOREIGN KEY** seguita dall'attributo che indica la chiave esterna, la parola **REFERENCES** e l'attributo esterno cui si riferisce

- Es: **CREATE TABLE** impiegato(....

...

...

**CONSTRAINT SUPERIMP_FK FOREIGN KEY(SUPERSSN) REFERENCES
impiegato(SSN)**



UTILIZZO DELLA KEYWORD CONSTRAINT

- L'utilizzo di **CONSTRAINT**, quando viene definito un vincolo è facoltativo
- Ma diventa particolarmente utile, in quanto permette di dare un nome ad un vincolo
 - Questo ci permette di manipolare il vincolo in maniera esplicita in seguito alla sua creazione



UTILIZZO DELLA KEYWORD CONSTRAINT

- L'utilizzo di **CONSTRAINT**, quando viene definito un vincolo è facoltativo
- Ma diventa particolarmente utile, in quanto permette di dare un nome ad un vincolo
 - Questo ci permette di manipolare il vincolo in maniera esplicita in seguito alla sua creazione
- **FORTEMENTE CONSIGLIATO!**



VINCOLI DI INTEGRITÀ REFERENZIALE

- 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**.



ESEMPIO

Specifica di valori di default e azioni referenziali triggered.

```
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 );
```



ESEMPIO (CONT.)

- 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**.



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



```
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)
)
```



DA DDL A SQL

METADATI

DATI

Definizione

CREATE

INSERT

Modifica

ALTER

UPDATE

Cancellazione

DROP

DELETE

SELECT

Interrogazione



POSTGRES

 Home About Download Documentation Community Developers Support Donate Your account

Search for... 

30th September 2021: PostgreSQL 14 Released!

Quick Links — **Downloads** 

- o Downloads
 - o Packages
 - o Source
- o Software Catalogue
- o File Browser

PostgreSQL Downloads

PostgreSQL is available for download as ready-to-use packages or installers for various platforms, as well as a source code archive if you want to build it yourself.

Packages and Installers

Select your operating system family:

[Linux](#) [macOS](#) [Windows](#) [BSD](#) 

[Solaris](#) 



ESEMPIO 1

- Definire un dominio che permetta di rappresentare stringhe di lunghezza massima pari a 256 caratteri, su cui non sono ammessi valori nulli e con valore di default “sconosciuto”.



SOLUZIONE 1

- CREATE DOMAIN String AS CHARACTER VARYING (256) DEFAULT 'sconosciuto'
NOT NULL;



ESEMPIO 2

- Dare le definizioni DDL delle tre entità:

FONDISTA(Nome, Nazione, Età*)

GAREGGIA(NomeFondista↑, NomeGara ↑, Piazzamento)

GARA(Nome, Luogo, Nazione, Lunghezza*)

rappresentando in particolare i vincoli di foreign key della tabella GAREGGIA.



SOLUZIONE 2

```
CREATE TABLE FONDISTA
(
Nome varchar(20) not null,
Nazione varchar(30) not null,
Età smallint,
primary key (Nome)
);
```

```
CREATE TABLE GARA
(
Nome varchar(20) not null,
Luogo varchar(20) not null,
Nazione varchar(30) not null,
Lunghezza integer,
primary key (Nome)
);
```

```
CREATE TABLE GAREGGIA
(
NomeFondista varchar(20) not null,
NomeGara varchar(20) not null,
Piazzamento smallint,
primary key (NomeFondista, NomeGara),
foreign key (NomeFondista)
references FONDISTA(Nome)
foreign key (NomeGara) references GARA(Nome)
);
```



ESEMPIO 3

- Dare le definizioni SQL delle entità:

AUTORE (Nome, Cognome, DataNascita*, Nazionalità*)

LIBRO (TitoloLibro, NomeAutore[↑], CognomeAutore[↑], Lingua*)

- Per il vincolo foreign key specificare una politica di cascade sulla cancellazione e di set null sulle modifiche.



SOLUZIONE 3

```
CREATE TABLE AUTORE
```

```
(  
    Nome varchar(20) not null,  
    Cognome varchar(20) not null,  
    DataNascita date,  
    Nazionalità varchar(30),  
    primary key(Nome, Cognome)  
);
```



```
CREATE TABLE LIBRO
```

```
(  
    TitoloLibro varchar(30) not null,  
    NomeAutore varchar(20) ,  
    CognomeAutore varchar(20),  
    Lingua varchar(20),  
    primary key (TitoloLibro),  
    foreign key (NomeAutore, CognomeAutore)  
        references AUTORE(Nome, Cognome)  
        on delete cascade  
        on update set NULL  
);
```



INFO SULLA LEZIONE

- Capitolo 7
 - Fino al 7.2.4



COME MANIPOLARE UN DB RELAZIONALE

- L'algebra relazionale è il mezzo per interrogare basi di dati relazionali, ma non è sfruttabile in ambito commerciale, principalmente perché le query sono scritte **specificando l'ordine** con cui devono essere eseguite le operazioni.
- SQL è un linguaggio **dichiarativo**, basato in parte sull'algebra relazionale ed in parte sul calcolo relazionale.
 - L'utente specifica **quale risultato** deve essere raggiunto.
 - Il DBMS decide l'ordine delle operazioni.



RELAZIONI BASE E VIRTUALI

- Le relazioni create con **CREATE TABLE** sono dette **tabelle base** o **relazioni base** in SQL, e significa che sono create e memorizzate come file dal DBMS.
- Le relazioni base sono distinte dalle **relazioni virtuali**, create mediante **CREATE VIEW**, cui può o meno corrispondere un file fisico.
- In SQL gli attributi sono considerati ordinati nella sequenza in cui sono stati specificati.
Le righe non sono considerate ordinate.



IL COMANDO DROP SCHEMA

- Se uno schema non è più necessario, si usa il comando **DROP SCHEMA**, con due possibili opzioni (*drop behaviour*): **CASCADE** e **RESTRICT**.
- **Esempi:**
 - **DROP SCHEMA COMPANY CASCADE;**
 - Lo schema del db COMPANY viene rimosso, con tutte le tabelle, domini ed altri elementi.
 - **DROP SCHEMA COMPANY RESTRICT;**
 - Lo schema è eliminato solo se non contiene elementi.



IL COMANDO DROP TABLE

- Permette di eliminare una tabella:
 - **DROP TABLE DEPENDENT CASCADE;**
 - Se non si vuole tenere più traccia delle persone a carico nel db COMPANY.
 - **DROP TABLE DEPENDENT RESTRICT;**
 - La tabella è eliminata solo se non è referenziata in alcun vincolo o vista.
 - Con l'opzione CASCADE sarebbero automaticamente eliminati insieme alla tabella stessa.



IL COMANDO ALTER TABLE

- La definizione di una tabella base può essere cambiata usando il comando **ALTER TABLE**.
- Possibili azioni di modifica di una tabella sono:
 - Aggiunta o rimozione di attributi;
 - Cambio di definizione di una colonna;
 - Aggiunta o rimozione di un vincolo.



ALTER TABLE: ESEMPIO

- Vogliamo aggiungere il lavoro svolto da un impiegato nella tabella Employee:
 - **ALTER TABLE COMPANY.EMPLOYEE ADD JOB VARCHAR(12);**
- Il valore di JOB o si specifica di **default** o sarà **null**.
- *Con la ALTER TABLE non è permessa la clausola NOT NULL.*



ALTER TABLE: ESEMPIO (2)

- Vogliamo eliminare una colonna: occorre scegliere l'opzione CASCADE o RESTRICT:
 - Con CASCADE tutti i vincoli e le viste che referenziano la colonna sono eliminati automaticamente dallo schema.
 - Con RESTRICT il comando ha successo solo se nessun vincolo o vista referenzia la colonna.
- **Esempio:** rimuovere ADDRESS dalla tabella EMPLOYEE:
 - **ALTER TABLE COMPANY.EMPLOYEE DROP ADDRESS CASCADE;**



ALTER TABLE: *ESEMPIO* (3)

- Modifica di una colonna eliminando una clausola di default o definendone una nuova.
- ***Esempio:***
 - **ALTER TABLE COMPANY.DEPARTMENT ALTER MGRSSN DROP DEFAULT;**
 - **ALTER TABLE COMPANY.DEPARTMENT ALTER MGRSSN SET DEFAULT "333444555";**
- Cambio di vincoli:
 - È possibile eliminare un vincolo solo se ha un nome:
 - **ALTER TABLE COMPANY.EMPLOYEE DROP CONSTRAINT EMPSUPERFK CASCADE;**



ESEMPIO 4

- Dato lo schema dell'esercizio precedente, spiegare cosa può capitare con l'esecuzione dei seguenti comandi di aggiornamento:
 1. **DELETE FROM AUTORE WHERE Cognome = 'Rossi';**
 2. **UPDATE LIBRO SET NomeAutore= 'Umberto' WHERE CognomeAutore = 'Eco';**
 3. **INSERT INTO AUTORE(Nome,Cognome) VALUES('Ugo', 'Bianchi');**
 4. **UPDATE AUTORE SET Nome = 'Italo' WHERE Cognome = 'Calvino';**



SOLUZIONE 4

1. Il comando cancella dalla tabella AUTORE tutte le tuple con Cognome = '**Rossi**'. A causa della politica cascade anche le tuple di LIBRO con CognomeAutore = '**Rossi**' verranno eliminate.
2. Il comando modifica la tabella LIBRO per tutti i cognomi di autori '**Eco**' viene settato il nome.
3. Il comando aggiunge una nuova tupla alla tabella AUTORE. *Non ha alcun effetto sulla tabella LIBRO*.
4. Le tuple di AUTORE con Cognome = '**Calvino**' vengono aggiornate a Nome = '**Italo**'. A causa della politica set null gli attributi NomeAutore e CognomeAutore delle tuple di LIBRO con CognomeAutore = '**Calvino**' vengono posti a NULL.



ESEMPIO 5

- Date le definizioni:

```
CREATE DOMAIN Dominio AS INTEGER DEFAULT 10
```

```
CREATE TABLE Tabella (Attributo Dominio DEFAULT 5);
```

- indicare cosa avviene in seguito ai comandi:

1. ALTER TABLE Tabella ALTER COLUMN Attributo DROP DEFAULT;
2. ALTER DOMAIN Dominio DROP DEFAULT;
3. DROP DOMAIN Dominio;



SOLUZIONE 5

1. Il comando cancella il valore di default di Attributo. Il valore di default dopo il comando sarà quello impostato in Dominio, ossia 10.
2. Il comando cancella il valore di default di Dominio. Dopo l'operazione il valore di default sarà NULL.
3. Il comando cancella l'intero dominio Domain. In Tabella il dominio di Attributo diventerà INTEGER.



ESERCIZIO

- Dare le definizioni SQL delle relazioni

Rappresentanti(CodRappr, Cognome, Nome, Via, Citta, Provincia,
TotProvvigioni,PercentProv)

Clienti (CodCliente, Cognome, Nome, Via, Citta, Provincia, Saldo,
Fido, CodRapp ↑)

Ordini (NroOrdine, Data, CodCliente ↑)

DettOrdini (NroOrdine↑, NroArt↑, Qta, Prezzo)

Articoli(NroArt, Descrizione, Giacenza, Categoria, PrezzoUnitario)

dimensionando opportunamente gli attributi ed indicando i
vincoli sia intrarelazionali che interrelazionali



INFO SULLA LEZIONE

- Capitolo 8
 - Solo l' 8.4

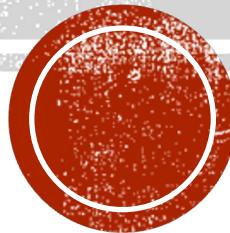




FINE

Per eventuali domande: (in ordine di preferenza personale)

- Ora.
- Chat di Teams
- Mail: silvio.barra@unina.it



BASI DI DATI I

- Algebra Relazionale
- Operatore di Selezione
- Operazione di Proiezione
- Operazione di Rename

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"

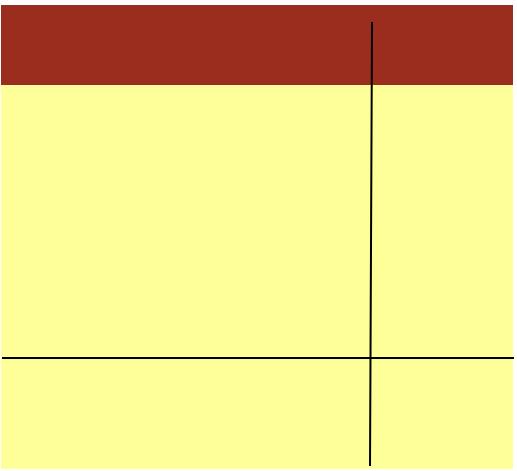
- Applichiamo la definizione di algebra al modello relazionale:
 - Dominio: le relazioni;
 - Operatori:
 - Operazioni su insiemi, ereditate dalla teoria matematica degli insiemi: *Unione*, *Intersezione*, *Differenza* e *Prodotto Cartesiano*.
 - Operazioni specificatamente disegnate per database relazionali: *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

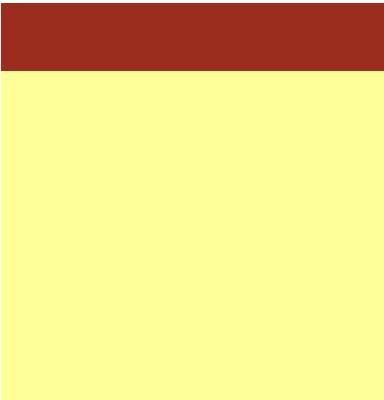




selezione

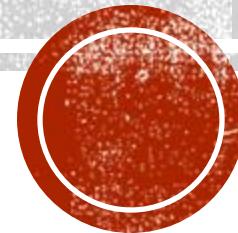


proiezione





OPERAZIONE DI SELEZIONE



L'OPERATORE SELECT: σ

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

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



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.



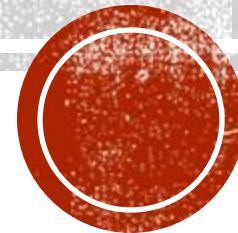
PROPRIETÀ DELLA SELECT

- L'operatore Select è **unario**.
- L'operatore di Select è commutativo:
$$\sigma_{< \text{cond}_1 >}(\sigma_{< \text{cond}_2 >}(R)) = \sigma_{< \text{cond}_2 >}(\sigma_{< \text{cond}_1 >}(R))$$
- Nell'SQL, la SELECT è tipicamente specificata nella clausola WHERE.





OPERATORE DI PROIEZIONE



L'OPERATORE PROJECT: π

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

$$\pi_{<\text{attribute_list}>} (<\text{relazione}>)$$

- 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 $t_r=t_p$.
- $\pi_{<\text{list}_1>}(\pi_{<\text{list}_2>}(R)) = \pi_{<\text{list}_1>}(R)$ 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:

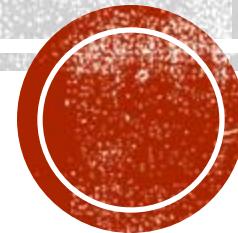
$$\pi_{<\text{FNAME}, \text{LNAME}, \text{SALARY}>}(\sigma_{\text{dno}=5}(\text{IMPIEGATO}))$$

- Alternativamente si possono creare risultati intermedi:

- $\text{DEPS_EMPS} = \sigma_{\text{dno}=5}(\text{IMPIEGATO})$
 - $\text{RESULT} = \pi_{<\text{FNAME}, \text{LNAME}, \text{SALARY}>}(\text{DEPS_EMPS})$



OPERATORE DI RENAMING



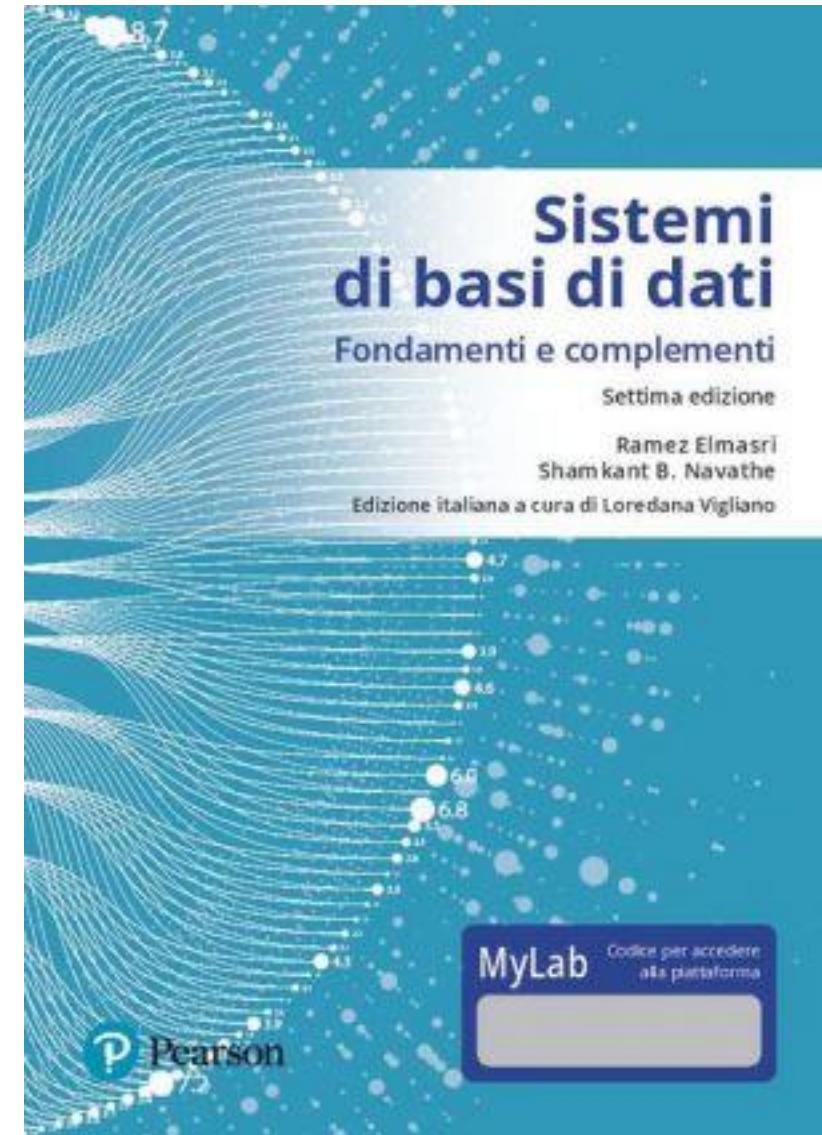
L'OPERAZIONE RENAME

- 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.



INFO SULLA LEZIONE

- Capitolo 6
 - Solo il 6.1

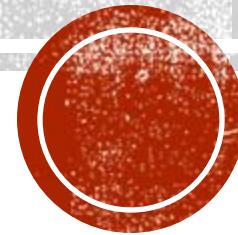




FINE

Per eventuali domande: (in ordine di preferenza personale)

- Ora.
- Chat di Teams
- Mail: silvio.barra@unina.it



BASI DI DATI I

- Algebra Relazionale
- Operazioni Insiemistiche
- Operazioni di Join (prima parte)
- SQL
 - Query di base (SELECT-FROM-WHERE)
 - Prodotto cartesiano & JOIN
 - Renaming, Distinct e Operazioni Insiemistiche
 - Wildcards e Caratteri JOLLY

RECAP



L'OPERATORE SELECT

- 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 PROJECT: π

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

$$\pi_{<\text{attribute_list}>} (<\text{relazione}>)$$

- 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'OPERAZIONE RENAME

- Per rinominare gli attributi in una relazione che risulta dall'algebra relazionale, semplicemente listiamo i nuovi nomi di attributi in parametri:
 - $\text{TEMP} = \sigma_{\text{dno}=5}(\text{EMPLOYEE})$
 - $R(\text{FirstName}, \text{LastName}, \text{Salary}) = \pi_{<\text{FNAME}, \text{LNAME}, \text{SALARY}>}(\text{TEMP})$
- $\rangle_{S(B_1, B_2, \dots, B_n)}(R)$ è una relazione ridevoluta S , basata su $R(A_1, A_2, \dots, A_n)$, con gli attributi ridevoluti in B_1, B_2, \dots, B_n .
- $\rangle_S(R)$ è una relazione ridevoluta S , basata su R , che non specifica i nomi degli attributi.
- $\rangle_{(B_1, B_2, \dots, B_n)}(R)$ è una relazione con attributi ridevoluti B_1, B_2, \dots, B_n , che non specifica il nuovo nome della relazione.



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.



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 $R - S \neq S - R$.



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

$I - S$

Quali sarebbero i risultati delle seguenti operazioni?

- $S \cup I;$
- $S \cap I;$
- $S - I;$
- $I - S;$



$S \cup I$

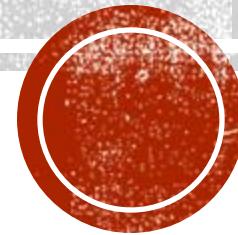


FINE RECAP





OPERAZIONI INSIEMISTICHE



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

- Dato il nostro schema relazionale AZIENDA, vogliamo trovare per ogni impiegato di sesso femminile, una lista dei suoi familiari a carico:
 - FEMALE_EMPS = $\sigma_{Sex='F'}(EMPLOYEE)$
 - EMPNAMES = $\pi_{<FNAME, LNAME, SSN>}(\text{FEMALE_EMPS})$
 - EMP_DEPENDENTS = EMPNAMES x DEPENDENTS
 - ACTUAL_DEPENDENTS = $\sigma_{SSN=ESSN}(\text{EMP_DEPENDENTS})$
 - RESULT = $\pi_{<FNAME, LNAME, DEPENDENT_NAME>} (\text{ACTUAL_DEPENDENTS})$



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

EMPNAMES	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

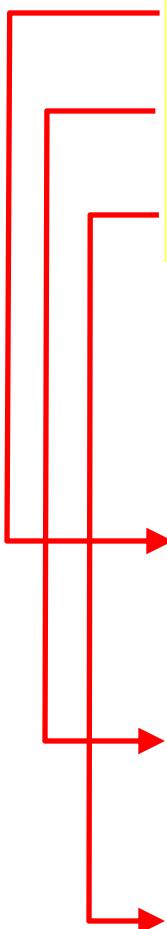
Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparti

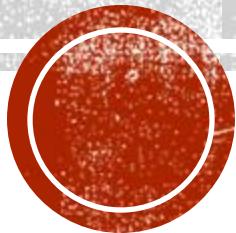
Codice	Capo
A	Mori
B	Bruni

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:

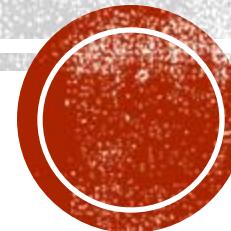
- $\text{EMP_DEPENDENTS} = \text{EMPNAME} \times \text{DEPENDENTS}$
- $\text{ACTUAL_DEPENDENTS} = \sigma_{\text{SSN}=\text{ESSN}}(\text{EMP_DEPENDENTS})$

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>
```

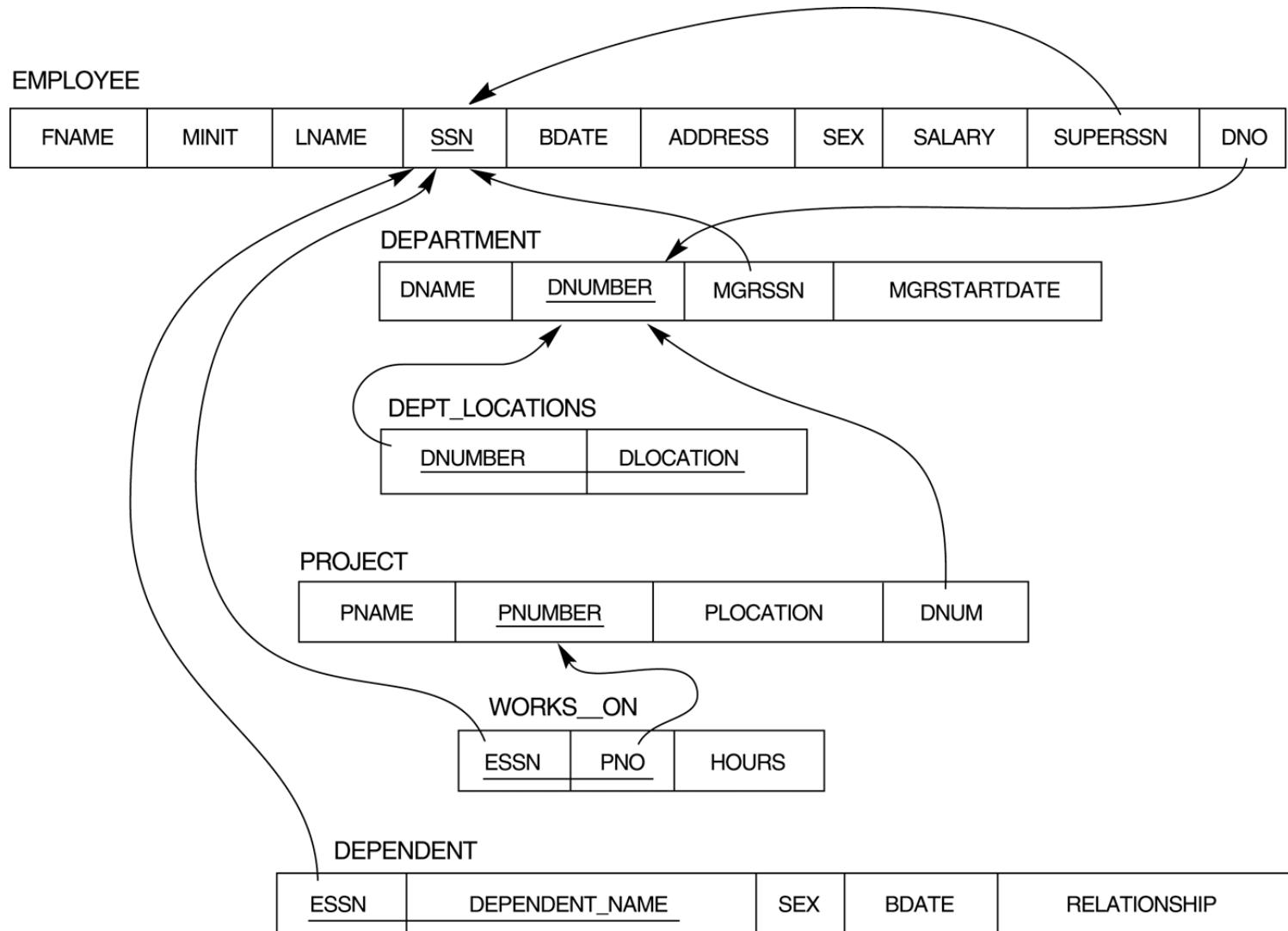


IL COMANDO SELECT (2)

- **<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



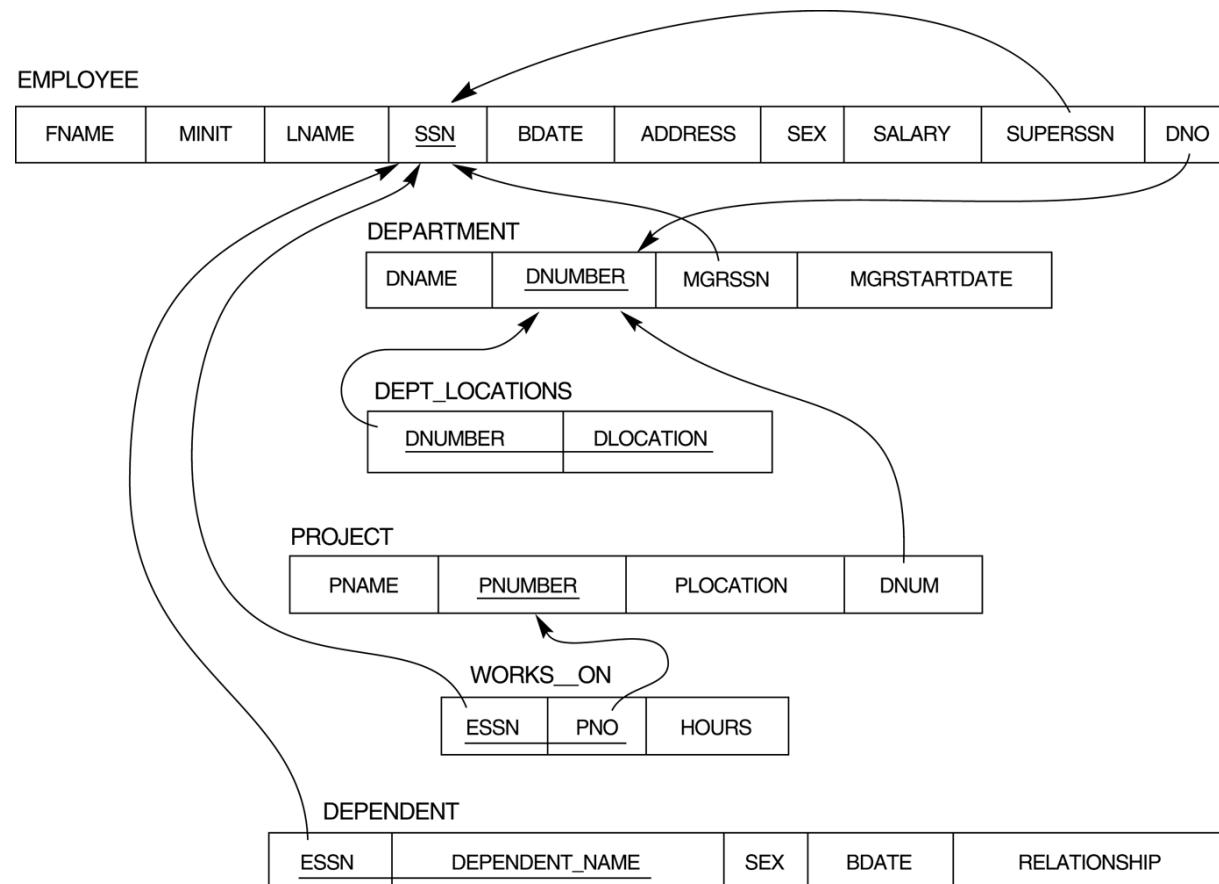
IL COMANDO SELECT: ESEMPIO



- 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:

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


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 età minore o uguale a trenta:
- Algebra relazionale ed SQL

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

```
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
```



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



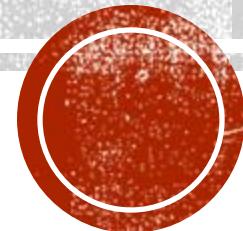
IL COMANDO SELECT: ESEMPIO



- 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:
 - l'utente può essere interessato alle duplicazioni;
 - è un'operazione costosa (l'implementazione richiederebbe l'ordinamento e poi l'eliminazione);



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 SOTTOSTRINGHE

- 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.

σ 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.

σ Età > 40 OR Età IS NULL (Impiegati)

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



INFO SULLA LEZIONE

- Capitolo 7
 - Fino al 7.3.5 compreso

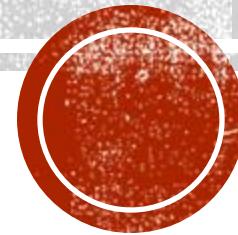




FINE

Per eventuali domande: (in ordine di preferenza personale)

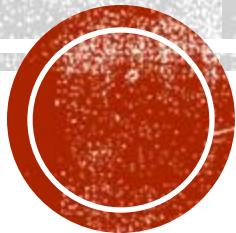
- Ora.
- Chat di Teams
- Mail: silvio.barra@unina.it



BASI DI DATI I

- Operazioni di Join (prima parte)
- Operazioni di Join (seconda parte)
- Divisione

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= $\pi_{DNAME, LNAME, FNAME}(DEPT_MGR)$

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	♦ ♦ ♦



ESEMPIO

- Scrivere un'espressione di AR che, se valutata, restituisca coppie di impiegati (nomi e i cognomi) che hanno lo stesso stipendio.



JOIN: ESEMPIO SU PRODOTTO CARTESIANO

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

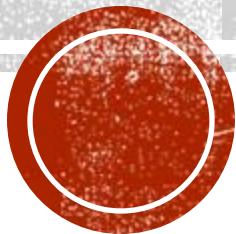
- $\text{EMP_DEPENDENTS} = \text{EMPNAME} \times \text{DEPENDENTS}$
- $\text{ACTUAL_DEPENDENTS} = \sigma_{\text{SSN}=\text{ESSN}}(\text{EMP_DEPENDENTS})$

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 θ generica è detta **THETA JOIN**
- Il join più comune ha come θ 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

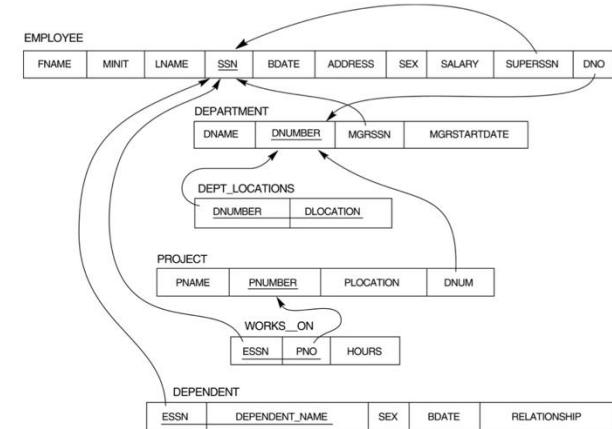
Reparti

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

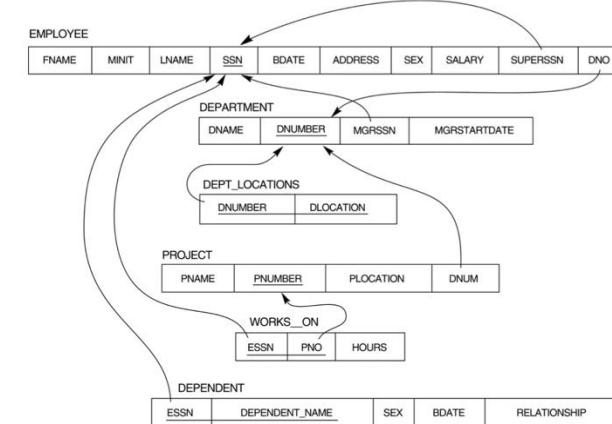


- La definizione standard di Natural join richiede che i due attributi di join (per ogni coppia) abbiano lo stesso nome. Se ciò non vale, occorre prima un'operazione di renaming.
 - $\text{DEPT} = \rho_{(\text{DNAME}, \text{DNUM}, \text{MGRSSN}, \text{MGRSTARTDATE})}(\text{DEPARTMENT})$
 - $\text{PROJ_DEPT} = \text{PROJECT} * \text{DEPT}$
 - 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:**
 - $\text{DEPT_LOCS} = \text{DEPARTMENT} * \text{DEPT_LOCATIONS}$

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
Rossi	A
Neri	B
Bianchi	B

Reparto	Capo
B	Mori
C	Bruni

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

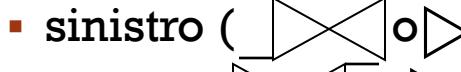
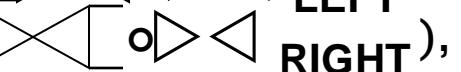
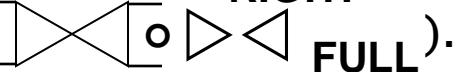
Impiegato	Reparto	Reparto	Capo
Rossi	A	B	Mori
Neri	B	C	Bruni
Bianchi	B		

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_{LEFT} Reparti

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



Impiegati	
Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparti	
Reparto	Capo
B	Mori
C	Bruni

Impiegati \bowtie_{RIGHT} Reparti

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



Impiegati

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Reparti

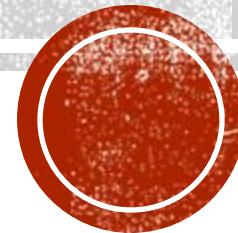
Reparto	Capo
B	Mori
C	Bruni

Impiegati \bowtie_{FULL} Reparti

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



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

- $T(N_D, N_IMPAGATI, STIPENDIO_MEDIO) \leftarrow_{N_D} \mathcal{F}_{COUNT\ SSN, AVERAGE\ STIPENDIO}(IMPIEGATO)$
- $R \leftarrow \rho_{N_D, N_IMP, STIP_MEDIO}(T)$

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

- $S \leftarrow \mathcal{F}_{COUNT\ SSN, AVERAGE\ STIPENDIO}(IMPIEGATO)$

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.



INFO SULLA LEZIONE

- Capitolo 6
 - Fino al 6.5 compreso

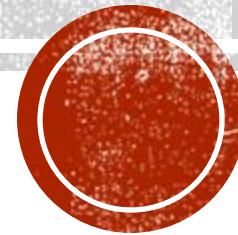




FINE

Per eventuali domande: (in ordine di preferenza personale)

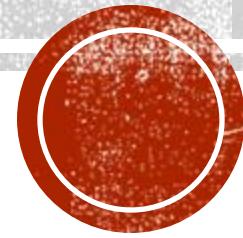
- Ora.
- Chat di Teams
- Mail: silvio.barra@unina.it



BASI DI DATI I

- Query complesse in SQL
- Funzioni di raggruppamento
- Aggiornamenti in SQL

QUERY COMPLESSE IN SQL



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')
```



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.FNAME, 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));
```



EXISTS E NOT EXISTS

- **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, pname;
```



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>]

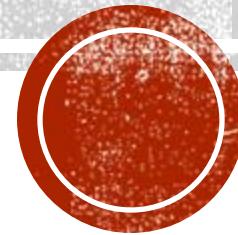
[**GROUP BY** <attributo o attributi di raggruppamento>]

[**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 aggiornarla, 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 VARCHAR(30) NOT NULL,
    PRIMARY KEY (id)
) AUTO_INCREMENT=5;
```

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

```
SELECT * FROM animals;
```

id	name
5	dog
6	cat
7	penguin
8	wolf
9	whale
10	ostrick



SERIAL

- **AUTO_INCREMENT** non è più supportato da Postgres, il quale invece utilizza **SERIAL** come tipo di dati per identificare un attributo che si auto incrementa per le nuove righe.

```
CREATE TABLE animals (
    id SERIAL,
    name VARCHAR(30) NOT NULL,
    PRIMARY KEY (id)
);
```



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 tavelle ‘*virtuali*’ derivate da tavelle esistenti nel db.
- Possono essere definite per nascondere dei dati da alcune tavelle (es. per questioni di privacy), per combinare più tavelle, 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;
```

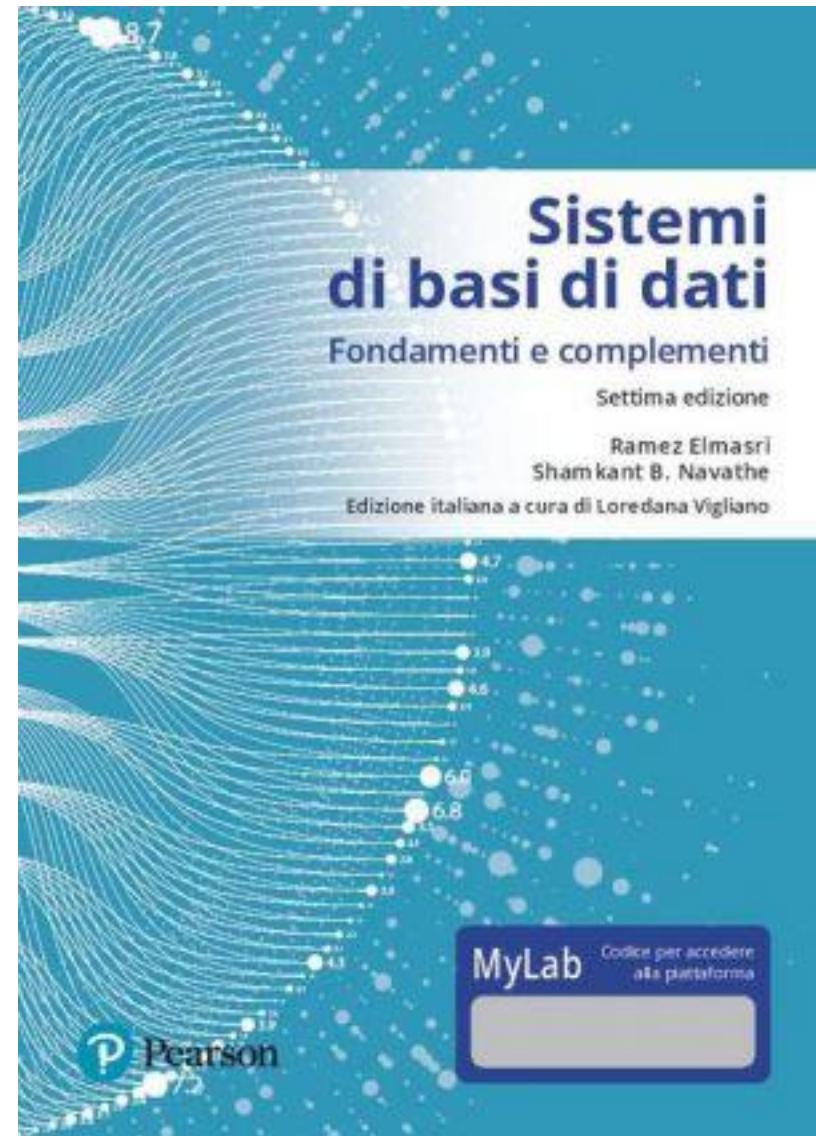
DEPT_INFO

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



INFO SULLA LEZIONE

- Capitolo 7
 - Fino alla fine
- Capitolo 8
 - Fino all'8.1.6 compreso
 - Dall'8.3 alla fine

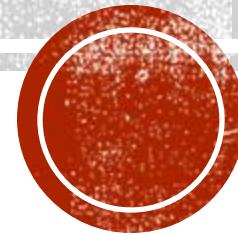




FINE

Per eventuali domande: (in ordine di preferenza personale)

- Ora.
- Chat di Teams
- Mail: silvio.barra@unina.it





DIE UNIVERSITÀ DEGLI STUDI DI
TI. NAPOLÌ FEDERICO II
DIPARTIMENTO DI INGEGNERIA ELETTRICA
E DELLE TECNOLOGIE DELL'INFORMAZIONE

BASI DI DATI I

- ESERCITAZIONE SQL

DOCENTE (Codice_Fiscale, Nome, Cognome, Eta, Qualifica)

2.Calcolare l'età media dei docenti di ogni qualifica nei seguenti 2 casi

CASO 1: per i docenti la cui età non è nota, è utilizzato il valore NULL

```
select Qualifica, avg(Eta) as EtaMedia  
from Docente  
group by Qualifica
```



DOCENTE (Codice_Fiscale, Nome, Cognome, Eta, Qualifica)

1. Calcolare l'età media dei docenti di ogni qualifica nei seguenti 2 casi

CASO 2: per i docenti la cui età non è nota, si usa 0

```
select Qualifica, avg(Eta) as EtaMedia  
from Docente  
where Eta <> 0  
group by Qualifica
```



AEROPORTO(Città, Nazione, NumPiste)

VOLO(IdVolo, GiornoSett, CittàPart, OraPart, CittàArr, OraArr, TipoAereo)

AEREO(TipoAereo, NumPasseggeri, QtaMerci)

1. Le città con un aeroporto di cui non è noto il numero di piste;

```
select Città  
from Aeroporto  
where NumPiste is NULL
```



AEROPORTO(Città, Nazione, NumPiste)

VOLO(IdVolo, GiornoSett, CittàPart, OraPart, CittàArr, OraArr, TipoAereo)

AEREO(TipoAereo, NumPasseggeri, QtaMerci)

2. Le nazioni da cui parte e arriva il volo con codice AZ274;

select A1.Nazione, A2.Nazione

from Aeroporto A1 JOIN Volo V ON A1.Città=V.CittaPart JOIN
Aeroporto A2 ON V.CittaArr=A2.Città

where V.IdVolo='AZ274'

AEROPORTO(Città, Nazione, NumPiste)

VOLO(IdVolo, GiornoSett, CittàPart, OraPart, CittàArr, OraArr, TipoAereo)

AEREO(TipoAereo, NumPasseggeri, QtaMerci)

3. I tipi di aereo usati nei voli che partono da Torino;

select V.TipoAereo

from Volo V

where CittàPart='Torino'



AEROPORTO(Città, Nazione, NumPiste)

VOLO(IdVolo, GiornoSett, CittàPart, OraPart, CittàArr, OraArr, TipoAereo)

AEREO(TipoAereo, NumPasseggeri, QtaMerci)

4. I tipi di aereo e il corrispondente numero di passeggeri per i tipi di aereo usati nei voli che partono da Torino. Se la descrizione dell'aereo non è disponibile, visualizzare solamente il tipo dell'aereo

```
select V.TipoAereo, A.NumPasseggeri  
from Volo V LEFT JOIN Aereo A on V.TipoAereo=A.TipoAereo  
where V.CittàPart='Torino'
```



AEROPORTO(Città, Nazione, NumPiste)

VOLO(IdVolo, GiornoSett, CittàPart, OraPart, CittàArr, OraArr, TipoAereo)

AEREO(TipoAereo, NumPasseggeri, QtaMerci)

5.Le città da cui partono voli internazionali;

select A1.Città

from Aereoporto A1 JOIN on A1.Città=V.CittàPart JOIN Aereoporto
A2 ON V.CittàArr=A2.Volo

where A1.Nazione <> A2.Nazione

AEROPORTO(Città, Nazione, NumPiste)

VOLO(IdVolo, GiornoSett, CittàPart, OraPart, CittàArr, OraArr, TipoAereo)

AEREO(TipoAereo, NumPasseggeri, QtaMerci)

6. Le città da cui partono voli diretti a Bologna, ordinate alfabeticamente;

select V.CittàPart

from Volo V

where V.CittàArr='Bologna'

order by V.CittàPart



AEROPORTO(Città, Nazione, NumPiste)

VOLO(IdVolo, GiornoSett, CittàPart, OraPart, CittàArr, OraArr, TipoAereo)

AEREO(TipoAereo, NumPasseggeri, QtaMerci)

7.Il numero di voli internazionali che partono il giovedì da Napoli;

```
select count(*)  
from Volo V JOIN Aeroporto A ON V.CittàArr=A.Città  
where V.CittàPart='Napoli' AND V.GiornoSett='Giovedì' AND  
A.Nazione <> 'Italia'
```

AEROPORTO(Città, Nazione, NumPiste)

VOLO(IdVolo, GiornoSett, CittàPart, OraPart, CittàArr, OraArr, TipoAereo)

AEREO(TipoAereo, NumPasseggeri, QtaMerci)

8. Il numero di voli internazionali che partono da città italiane.

```
select V.CittàPart, count(*)  
from Aeroporto A1 JOIN Volo V on A1.Città=V.CittàPart JOIN  
Aeroporto A2 on CittàArr=A2.Città  
where A1.Nazione='Italia' and A2.Nazione<>'Italia'  
group by V.CittàPart
```



AEROPORTO(Città, Nazione, NumPiste)

VOLO(IdVolo, GiornoSett, CittàPart, OraPart, CittàArr, OraArr, TipoAereo)

AEREO(TipoAereo, NumPasseggeri, QtaMerci)

9. Le città francesi da cui partono più di venti voli alla settimana diretti in Italia;

select A1.CittàPart

from Aeroporto A1 JOIN Volo V on A1.CittàPart=V.Città JOIN
Aeroporto A2 on V.CittàArr=A2.Città

where A1.Nazione='Francia' and A2.Nazione='Italia'

group by A1.CittàPart

having count(*)>20



AEROPORTO(Città, Nazione, NumPiste)

VOLO(IdVolo, GiornoSett, CittàPart, OraPart, CittàArr, OraArr, TipoAereo)

AEREO(TipoAereo, NumPasseggeri, QtaMerci)

10. Gli aeroporti italiani che hanno solo voli interni. Rappresentare questa interrogazione in tre modi:

a) con operatori insiemistici;

```
select V.CittàPart  
from Volo V JOIN Aeroporto A on V.CittàPart=A.Città  
where Nazione='Italia'  
except  
select V.CittàPart  
from Aeroporto A1 join Volo V on A1.Città=V.CittàPart JOIN  
Aeroporto A2 on V.CittàArr=A2.Città  
where A1.Nazione='Italia' AND A2.Nazione<>'Italia'
```



AEROPORTO(Città, Nazione, NumPiste)

VOLO(IdVolo, GiornoSett, CittàPart, OraPart, CittàArr, OraArr, TipoAereo)

AEREO(TipoAereo, NumPasseggeri, QtaMerci)

10. Gli aeroporti italiani che hanno solo voli interni. Rappresentare questa interrogazione in tre modi:

b)Query annidate + operatore NOT IN;

```
select V.CittàPart  
  
from Volo V JOIN Aeroporto A on V.CittàPart=A.Città  
  
where Nazione='Italia' AND V.CittàPart NOT IN  
  
(select V2.CittàPart  
  
from Aeroporto A2 join Volo V2 on A2.Città=V.CittàPart JOIN  
Aeroporto A3 on V2.CittàArr=A3.Città  
  
where A2.Nazione='Italia' AND A3.Nazione<>'Italia')
```



AEROPORTO(Città, Nazione, NumPiste)

VOLO(IdVolo, GiornoSett, CittàPart, OraPart, CittàArr, OraArr, TipoAereo)

AEREO(TipoAereo, NumPasseggeri, QtaMerci)

10. Gli aeroporti italiani che hanno solo voli interni. Rappresentare questa interrogazione in tre modi:

c) Query annidate + operatore EXISTS/NOT EXISTS;

```
select V.CittàPart  
from Volo V JOIN Aeroporto A on V.CittàPart=A.Città  
where Nazione='Italia' AND  
NOT EXISTS(select *  
from Volo V2 JOIN Aeroporto A2 on V2.CittàArr=V2.Città  
where A1.Città=V.CittàPart AND A2.Nazione<>'Italia')
```



AEROPORTO(Città, Nazione, NumPiste)

VOLO(IdVolo, GiornoSett, CittàPart, OraPart, CittàArr, OraArr, TipoAereo)

AEREO(TipoAereo, NumPasseggeri, QtaMerci)

11. Le città che sono servite dall'aereo caratterizzato dal massimo numero di passeggeri;

select V1.CittàPart

from Volo V1 NATURAL JOIN Aereo A1

where A1.NumPasseggeri = (**select** max(A2.NumPasseggeri)

from Aereo A2)

UNION

select V2.CittàArr

from Volo V2 NATURAL JOIN Aereo A3

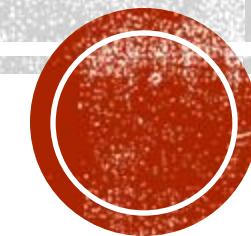
where A3.NumPasseggeri = (**select** max(A4.NumPasseggeri)

from Aereo A4)





ESERCITAZIONE



```
create table Studenti (
matricola numeric not null primary
key,
cognome char(20) not null,
nome char(20) not null,
eta numeric not null
);
create table Esami (
codiceCorso numeric not null,
studente numeric not null
    references Studenti(matricola),
data date not null,
voto numeric not null,
primary key (codiceCorso,
studente, data)
)
```

Si supponga che vengano registrati anche gli esami non superati, con voti inferiori al 18.

- Formulare le seguenti interrogazioni in SQL:
 1. L'interrogazione che trova gli studenti che non hanno superato esami;
 2. L'interrogazione che trova gli studenti che hanno riportato in almeno un esame un voto più alto di Archimede Pitagorico;
 3. L'interrogazione che trova i nomi degli studenti che hanno superato almeno due esami
 4. L'interrogazione che trova, per ogni studente, il numero di esami superati e la relativa media.



```
create table Studenti (
matricola numeric not null primary
key,
cognome char(20) not null,
nome char(20) not null,
eta numeric not null
);
create table Esami (
codiceCorso numeric not null,
studente numeric not null
    references Studenti(matricola),
data date not null,
voto numeric not null,
primary key (codiceCorso,
studente, data)
)
```

Si supponga che vengano registrati anche gli esami non superati, con voti inferiori al 18.

- **Formulare in SQL l'interrogazione che trova gli studenti che non hanno superato esami;**

```
select *
from Studenti
where matricola not in
( select distinct studente
from Esami );
```



```
create table Studenti (
matricola numeric not null primary
key,
cognome char(20) not null,
nome char(20) not null,
eta numeric not null
);
create table Esami (
codiceCorso numeric not null,
studente numeric not null
    references Studenti(matricola),
data date not null,
voto numeric not null,
primary key (codiceCorso,
studente, data)
)
```

Si supponga che vengano registrati anche gli esami non superati, con voti inferiori al 18.

- Fornire l'interrogazione che trova gli studenti che hanno riportato in almeno un esame un voto più alto di Archimede Pitagorico;

```
select *
from Studenti join Esami on
matricola=studente
where voto > any
( select voto
from Esami join Studenti on studente =
matricola
where cognome = 'pitagorico'
and nome = 'archimede' );
```



```
create table Studenti (
matricola numeric not null primary
key,
cognome char(20) not null,
nome char(20) not null,
eta numeric not null
);

create table Esami (
codiceCorso numeric not null,
studente numeric not null
    references Studenti(matricola),
data date not null,
voto numeric not null,
primary key (codiceCorso,
studente, data)
)
```

Si supponga che vengano registrati anche gli esami non superati, con voti inferiori al 18.

- Fornire l'interrogazione che trova i nomi degli studenti che hanno superato almeno due esami;

```
select distinct s.nome, s.cognome
from Studenti s, Esami e1 join Esami e2 on
(e1.studente = e2.studente)
where
e1.codiceCorso <> e2.codiceCorso
and e1.voto >= 18 and
e2.voto >= 18 and s.matricola = e1.studente;
```



```
create table Studenti (
matricola numeric not null primary
key,
cognome char(20) not null,
nome char(20) not null,
eta numeric not null
);
create table Esami (
codiceCorso numeric not null,
studente numeric not null
references Studenti(matricola),
data date not null,
voto numeric not null,
primary key (codiceCorso,
studente, data)
)
```

Si supponga che vengano registrati anche gli esami non superati, con voti inferiori al 18.

- Fornire l'interrogazione che trova, per ogni studente, il numero di esami superati e la relativa media.

```
select s.nome, s.cognome, count(*),
avg(voto)
from Studenti s, Esami e
where s.matricola = e.studente
group by s.nome, s.cognome
```



```
create table Tariffa (
    tipoAuto char(20) not null primary key,
    costoAlKm numeric not null
);

create table Automobile (
    targa numeric not null primary key,
    tipologia char(20) not null
        references Tariffa(TipoAuto),
    lunghezza char(20) not null
);

create table Transito (
    codice numeric not null primary key,
    auto numeric not null references Automobile(targa),
    orarioIngresso numeric not null,
    orarioUscita numeric not null,
    KmPercorsi numeric not null)
```

- Formulare le seguenti interrogazioni in SQL:
 1. L'interrogazione che restituisce, per ogni transito, i dati del veicolo, del transito e il costo del pedaggio (ottenuto moltiplicando il costo al Km per i Km percorsi).
 2. L'interrogazione che restituisce tutti i dati delle automobili che sono transitate più di una volta sull'autostrada
 3. L'interrogazione che trova le auto che in ogni transito hanno percorso sempre lo stesso numero di Km
 4. L'interrogazione che restituisce i dati dei transiti per cui la velocità media è superiore ai 140Km/h (si assuma che una differenza tra i due orari produca il risultato espresso in ore).



```
create table Tariffa (
    tipoAuto char(20) not null primary key,
    costoAlKm numeric not null
);

create table Automobile (
    targa numeric not null primary key,
    tipologia char(20) not null
        references Tariffa(TipoAuto),
    lunghezza char(20) not null
);

create table Transito (
    codice numeric not null primary key,
    auto numeric not null references
        Automobile(targa),
    orarioIngresso numeric not null,
    orarioUscita numeric not null,
    KmPercorsi numeric not null)
```

- Fornire l'interrogazione che restituisce, per ogni transito, i dati del veicolo, del transito e il costo del pedaggio (ottenuto moltiplicando il costo al Km per i Km percorsi).

```
select A.targa, A.tipologia, A.lunghezza,
    T.kmPercorsi * TA.costoAlKm
from Transito T join Automobile A on (T.auto
    = A.targa)
    join Tariffa TA on (TA.tipoAuto =
        A.tipologia);
```



```
create table Tariffa (
    tipoAuto char(20) not null primary key,
    costoAlKm numeric not null
);

create table Automobile (
    targa numeric not null primary key,
    tipologia char(20) not null
        references Tariffa(TipoAuto),
    lunghezza char(20) not null
);

create table Transito (
    codice numeric not null primary key,
    auto numeric not null references
        Automobile(targa),
    orarioIngresso numeric not null,
    orarioUscita numeric not null,
    KmPercorsi numeric not null)
```

- Fornire l'interrogazione che restituisce tutti i dati delle automobili che sono transitate più di una volta sull'autostrada

```
select A.targa, A.tipologia, A.lunghezza
from Automobile A join Transito T1 on
    (T1.auto = A.targa)
join transito T2 on (T1.auto = T2.auto)
where T1.codice <> T2.codice;
```



```

create table Tariffa (
    tipoAuto char(20) not null primary key,
    costoAlKm numeric not null
);

create table Automobile (
    targa numeric not null primary key,
    tipologia char(20) not null
        references Tariffa(TipoAuto),
    lunghezza char(20) not null
);

create table Transito (
    codice numeric not null primary key,
    auto numeric not null references Automobile(targa),
    orarioIngresso numeric not null,
    orarioUscita numeric not null,
    KmPercorsi numeric not null)

```

- Fornire l'interrogazione che restituisce le auto che in ogni transito hanno percorso sempre lo stesso numero di Km

```

select A.targa
from Automobile A join Transito T1 on
    (A.targa = T1.auto)
left join Transito T2 on (T1.auto = T2.auto)
where T1.codice <> T2.codice

except

select A.targa
from Automobile A join Transito T1 on
    (A.targa = T1.auto)
join Transito T2 on (T1.auto = T2.auto)
where T1.codice <> T2.codice and (T1.km <>
    T2.Km);

```



```
create table Tariffa (
    tipoAuto char(20) not null primary key,
    costoAlKm numeric not null
);

create table Automobile (
    targa numeric not null primary key,
    tipologia char(20) not null
        references Tariffa(TipoAuto),
    lunghezza char(20) not null
);

create table Transito (
    codice numeric not null primary key,
    auto numeric not null references Automobile(targa),
    orarioIngresso numeric not null,
    orarioUscita numeric not null,
    KmPercorsi numeric not null)
```

- Fornire l'interrogazione che restituisce i dati dei transiti per cui la velocità media è superiore ai 140Km/h (si assuma che una differenza tra i due orari produca il risultato espresso in ore).

```
select A.* , T.* , T.KmPercorsi * TA.costoKM
from Transito T join Automobile A on (T.auto
= A.targa)
join Tariffa TA on (TA.tipoAuto =
A.Tipologia)
where (T.KmPercorsi / (T.uscita-T.ingresso))
> 140
```

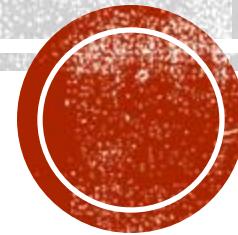




FINE

Per eventuali domande: (in ordine di preferenza personale)

- Ora.
- Chat di Teams
- Mail: 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 (val1, val2, ...)
- WHERE attributo BETWEEN val1 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
```

Insieme dei corsi sostenuti
dallo studente S.Matricola

```
(SELECT E2.Corso
FROM Esami AS E2
WHERE E2.Matricola = 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
```

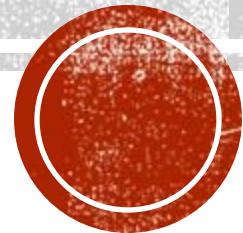
```
(SELECT E.CORSO  
FROM Esami AS E
```

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

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



VINCOLI E ASSERTION



VINCOLI

- I vincoli analizzati fino ad ora sono supportati dal linguaggio di definizione dei dati in quanto si manifestano nella maggior parte dei casi e nella maggior parte delle applicazioni per DB
 - Vincoli sui valori null e valori predefiniti
 - Vincoli sulle tuple (CHECK)
 - Vincoli sui domini degli attributi
 - Vincoli di chiave
 - Vincoli di integrità referenziale
- Esistono altri tipi di vincoli i quali, siccome non comuni a tutte le basi di dati, non fanno parte del DDL e devono essere specificati in maniera differente
 - Sono chiamati **vincoli di integrità semantica**



VINCOLI DI BASE

- 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



VINCOLI DI INTEGRITÀ SEMANTICA

- «Un impiegato non può guadagnare più del suo supervisore»
- «Ad un progetto non possono lavorare più di n impiegati»
- «Un impiegato non può lavorare più di 48 ore settimanali»

- Questa tipologia di vincoli può essere specificata
 - Sia all'interno dei programmi applicativi
 - Sia tramite un linguaggio di specifica dei vincoli

- Si possono utilizzare costrutti quali ASSERTION e TRIGGER



TIPOLOGIA DI VINCOLI

- In linea di massima i vincoli possono essere di 4 tipi
 - Table-level constraints
 - (PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK)
 - Column-level constraints
 - (PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK, NOT NULL)
 - Schema-level constraints
 - Vincoli a livello di schema. Ogni volta che l'intero DB cambia stato, si controlla se un vincolo è violato



VINCOLI INTRARELAZIONALI E INTERRELAZIONALI

- La maggior parte dei vincoli sono intrarelazionali:
 - Per verificarne l'eventuale rispetto si deve andare a controllare esclusivamente la tabella in cui è avvenuta la modifica
 - PRIMARY KEY (vado a verificare che non esistano attributi uguali per la chiave primaria nella tabella dove ho effettuato la modifica)
 - UNIQUE (lo stesso)
 - NOT NULL (vado a verificare che il valore inserito per un attributo non sia null)
 - CHECK (Controllo che i valori dati ad uno o più attributi della tabella in cui è definito il vincolo, rispettino un determinato range/dominio)
- Il vincolo di integrità referenziale (FOREIGN KEY) è un vincolo *interrelazionale*, in quanto va a controllare che il valore definito nell'attributo (o negli attributi) esista nella tabella di riferimento.



VINCOLO DI CHECK

- È un costrutto molto potente
 - TEORICAMENTE Tutti i vincoli di tabella o di colonna (UNIQUE, PRIMARY KEY, FOREIGN KEY, ...) possono essere espressi tramite un vincolo di CHECK.
 - Almeno in Postgres questo non funziona perché non è possibile creare subquery nei vincoli di check.
 - Far riferimento sia agli attributi della tabella in cui è definito, sia ad attributi di altre tabelle.
 - Anche questo in Postgres non è possibile.

```
create table impiegato(  
matricola char(8) PRIMARY KEY,  
dipartimento char(8),  
check (matricola like 'c%' OR dipartimento in  
      (SELECT nomedip  
       | FROM dipartimento)));
```

```
create table impiegato(  
matricola char(8) PRIMARY KEY,  
dipartimento char(8),  
check(1>=(SELECT count(*)  
      |-----  
      | FROM impiegato I  
      | WHERE matricola = I.matricola)  
      )  
);
```

VINCOLI INTERRELAZIONALI

- Quando un vincolo di CHECK coinvolge due o più tabelle ci possono essere comportamenti indesiderati.
- Nell'esempio sotto, se inseriamo nella tabella DIPARTIMENTO più di 100 dipartimenti, il vincolo rimane soddisfatto, dato che il controllo viene fatto **SOLO** quando si inserisce la tupla all'interno della tabella IMPIEGATO

Cosa succede se vado a fare l'update di *nomedip* in DIPARTIMENTO?

```
create table impiegato(  
matricola char(8) PRIMARY KEY,  
dipartimento char(8),  
check (matricola like 'c%' OR dipartimento in  
      (SELECT nomedip  
       | FROM dipartimento)));
```

```
create table impiegato(  
matricola char(8) PRIMARY KEY,  
dipartimento char(8),  
check((SELECT count(DISTINCT dipartimento) FROM impiegato)+  
      (SELECT count(nome) FROM dipartimento)<100));
```

ASSERTION

- Per evitare situazioni spiacevoli e anomalie causate dai vincoli di check su più tabelle, si possono usare le **assertion**
 - Questi sono vincoli non collegati ad alcun attributo o tabella in particolare, ma appartengono direttamente allo schema di basi di dati (Schema-Level)

CREATE ASSERTION <nome vincolo>
espressione booleana

CREATE ASSERTION <nome_assertion>
check (condizione)

```
1 CREATE ASSERTION ass1
2 check ((SELECT count(distinct nomedip)+10<100));
```

Data output Messages Notifications

ERROR: CREATE ASSERTION is not yet implemented
SQL state: 0A000

ASSERTION

```
CREATE ASSERTION <nome_assertion>
check (condizione)
```

- Il nome del vincolo è spesso seguito dalla parola CHECK che a sua volta è seguita da una condizione **che viene espressa tra parentesi e che deve essere sempre vera in ogni stato della base di dati affinchè l'assertion sia soddisfatta**
 - La maggior parte delle volte si utilizzano le clausole EXISTS e NOT EXISTS
- Come per i vincoli visti fino ad ora, il nome può essere utilizzato per modificare o rimuovere l'assertion in un secondo momento.
 - Il DBMS si occupa di verificare che la condizione non sia mai violata
- Il vincolo risulta violato ogni volta che una o più tuple fanno sì che la condizione della CHECK risulta FALSE
 - Altrimenti, il vincolo è soddisfatto



ASSERTION

```
CREATE ASSERTION <nome_vincolo>
CHECK ([NOT] EXISTS (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 arco sono diversi*



ESEMPIO

ALBERO(CodAlbero, CodRadice)

NODE(CodNodo, Label, CodAlbero)

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

Vincoli

- *Sorgente e target di un arco 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 arco 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 EXISTS (SELECT * FROM ALBERO AS T

WHERE T.CodRadice NOT IN

(SELECT N.CodNodo FROM NODO N

WHERE N.CodNodo= T.CodRadice)));



PERCHÉ I DBMS NON LE SUPPORTANO?

- Le **assertion** nei database (in SQL standard) sono vincoli che permettono di esprimere **condizioni generali** sull'intero stato del database, non limitate a una singola tabella o colonna (come i vincoli CHECK, FOREIGN KEY, ecc.).
- **Quasi nessun DBMS supporta le assertion**, per i seguenti motivi:
 - Costi computazionali elevati
 - Difficoltà di implementazione
 - Ci sono alternative più semplici ed efficienti
 - Scarsa domanda pratica



1. COSTI COMPUTAZIONALI ELEVATI

Le assertion impongono al DBMS di:

- **Verificare vincoli globali** su più tabelle o su l'intero database;
 - Eseguire queste verifiche **dopo ogni modifica** che potrebbe potenzialmente violarle;
 - Determinare **automaticamente** quali modifiche potrebbero impattare l'assertion.
- Questo è **computazionalmente oneroso** e difficile da ottimizzare, specialmente in database di grandi dimensioni e ad alto tasso di modifiche.
- Esempio:
- "Ogni studente deve avere sostenuto almeno 3 esami."



```
CREATE TABLE Studenti (
    id INTEGER PRIMARY KEY,
    nome TEXT);
```

```
CREATE TABLE Esami (
    id INTEGER PRIMARY KEY,
    studente_id INTEGER REFERENCES Studenti(id),
    voto INTEGER);
```

```
CREATE ASSERTION almeno_tre_esami
CHECK ( NOT EXISTS (
    SELECT * FROM Studenti s
    WHERE (
        SELECT COUNT(*) FROM Esami e
        WHERE e.studente_id = s.id
    ) < 3 ));
```

Perché ha un costo computazionale elevato?

- Il vincolo riguarda due tavelle (Studenti e Esami), quindi il DBMS deve considerare ogni cambiamento in entrambe.**
- Qualsiasi inserimento, cancellazione o modifica** in Esami o Studenti potrebbe invalidare l'assertion.
- Per ogni modifica, il DBMS dovrebbe ricontrollare che nessuno studente sia sceso sotto la soglia dei 3 esami,** potenzialmente rieseguendo l'intera subquery annidata.
- Inoltre, non esiste un modo semplice per il DBMS di sapere quali studenti potrebbero essere impattati da un'operazione su Esami.**

Si immagini un sistema con **decine di migliaia di studenti e centinaia di migliaia di esami**: il costo di questa verifica globale è **enorme e non prevedibile in anticipo**.



```
CREATE OR REPLACE FUNCTION check_minimo_esami()
RETURNS trigger AS $$

DECLARE
    esami_count INTEGER;

BEGIN

    SELECT COUNT(*) INTO esami_count FROM Esami WHERE studente_id = NEW.studente_id;

    IF TG_OP = 'DELETE' THEN esami_count := esami_count - 1;
    ELSIF TG_OP = 'INSERT' THEN esami_count := esami_count + 1;
    END IF;

    IF esami_count < 3 THEN
        RAISE EXCEPTION 'Ogni studente deve avere almeno 3 esami';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_check_esami
AFTER INSERT OR DELETE ON Esami
FOR EACH ROW
EXECUTE FUNCTION check_minimo_esami();
```



2. DIFFICOLTÀ DI IMPLEMENTAZIONE

Le assertion possono contenere **query SQL arbitrarie**, con JOIN, GROUP BY, EXISTS, ecc., quindi:

- Il DBMS dovrebbe **analizzare le dipendenze** e triggerare verifiche appropriate.
- In pratica, è **quasi impossibile garantire efficienza** in tutti i casi, o anche solo in molti casi realistici.



```
CREATE TABLE Magazzino (
    prodotto_id INTEGER PRIMARY KEY,
    quantità INTEGER NOT NULL );
```

```
CREATE TABLE Spedizioni (
    prodotto_id INTEGER,
    quantità_spedita INTEGER,
    data_spedizione DATE );
```

"Per ogni prodotto, la somma delle quantità spedite in Spedizioni non deve superare la quantità disponibile in Magazzino."

```
CREATE ASSERTION spedizioni_valide
CHECK ( NOT EXISTS (
    SELECT prodotto_id
    FROM Spedizioni
    GROUP BY prodotto_id
    HAVING SUM(quantità_spedita) >
        (SELECT quantità FROM Magazzino
        WHERE Magazzino.prodotto_id = Spedizioni.prodotto_id)
    ) );
```



PERCHÉ È COMPLESSO!?

1. **Subquery correlate + aggregazione + HAVING**: il DBMS deve valutare, per ogni prodotto in Spedizioni, la quantità totale spedita e confrontarla con una subquery che legge Magazzino.
2. **Il vincolo è dinamico e dipende da join e aggregazioni**: cambia se modifichi **una singola spedizione, un prodotto nel magazzino, oppure aggiungi una nuova spedizione**.
3. Il DBMS deve capire **quali modifiche** impattano il vincolo e quando rieseguire la verifica.
4. Non esiste una tecnica generica per **decomporre questa logica in verifiche locali per INSERT, UPDATE, DELETE** su entrambe le tabelle.

Per supportare questo vincolo, il motore del DBMS dovrebbe:

- Analizzare il contenuto dell'intera Spedizioni per ogni possibile cambiamento;
 - Tenere traccia delle dipendenze tra le righe di due tabelle diverse;
 - Capire se un'operazione su Magazzino può violare un HAVING su Spedizioni.
- In pratica: **non è realistico né efficiente**.



```
CREATE OR REPLACE FUNCTION check_spedizione()
RETURNS trigger AS $$

DECLARE
    total_spedito INTEGER;
    max_disponibile INTEGER;

BEGIN
    SELECT SUM(quantità_spedita) INTO total_spedito FROM Spedizioni
    WHERE prodotto_id = NEW.prodotto_id;

    SELECT quantità INTO max_disponibile FROM Magazzino
    WHERE prodotto_id = NEW.prodotto_id;

    IF total_spedito + NEW.quantità_spedita > max_disponibile THEN
        RAISE EXCEPTION 'Quantità spedita supera quella disponibile in magazzino';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_check_spedizione
BEFORE INSERT ON Spedizioni
FOR EACH ROW EXECUTE FUNCTION check_spedizione();
```



3. ALTERNATIVE PIÙ SEMPLICI ED EFFICIENTI

I DBMS offrono meccanismi più gestibili e performanti, come:

- **Vincoli relazionali locali** (CHECK, UNIQUE, FOREIGN KEY, NOT NULL);
- **Trigger** personalizzati (che possono simulare certe assertion in modo controllato);
- **View con WITH CHECK OPTION**, per limitare le modifiche attraverso viste controllate.

4. SCARSA DOMANDA PRATICA

Nel mondo reale:

- Le constraint globali sono spesso **gestite a livello applicativo** o tramite trigger mirati;
- I DBMS si concentrano su performance, scalabilità e transazionalità, evitando feature difficili da ottimizzare e poco usate.



TECNICHE PER DEFINIRE UN'ASSERTION

- La tecnica base per definire una assertion consiste nello specificare un'interrogazione che selezioni le tuple che violano la condizione desiderata
- Utilizzando quest'interrogazione insieme alla clausola NOT EXISTS, l'assertion specificherà che il risultato della query dovrà necessariamente essere vuoto, per fare in modo che la condizione sia sempre TRUE.



CHE VINCOLI SONO?

ALBERO (CodAlbero, CodRadice)

NODE (CodNodo, Label, CodAlbero)

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

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



UN NODO HA UN SOLO PADRE

- **Vincolo intra-relazionale**
 - Riguarda soltanto la relazione ARCO

UNIQUE (CodAlbero, NodoTarget)

```
ALTER TABLE ARCO
ADD CONSTRAINT un_nodo_un_padre
UNIQUE (CodAlbero, NodoTarget);
```

ALBERO (CodAlbero, CodRadice)

NODE (CodNodo, Label, CodAlbero)

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



UNA RADICE NON HA PADRI

- **Vincolo inter-relazionale**
 - Riguarda le relazioni ARCO e ALBERO

```
CREATE ASSERTION RadiceSenzaPadri
CHECK (
    NOT EXISTS (
        SELECT *
        FROM ALBERO A
        JOIN ARCO R
        ON A.CodAlbero = R.CodAlbero AND
           A.CodRadice = R.NodoTarget
    )
);
```

*ALBERO(CodAlbero, CodRadice)
NODE(CodNodo, Label, CodAlbero)
ARCO(CodArco, NodoSorgente, NodoTarget, Label, CodAlbero)*



TUTTI I NODI SONO CONNESSI

Esiste un cammino (sequenza di archi) dalla radice a ogni altro nodo dell'albero.

- Fatela a casa!

ALBERO (CodAlbero, CodRadice)

NODE (CodNodo, Label, CodAlbero)

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

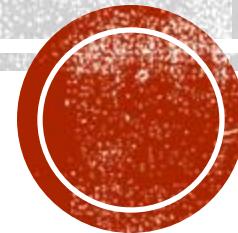




FINE

Per eventuali domande: (in ordine di preferenza personale)

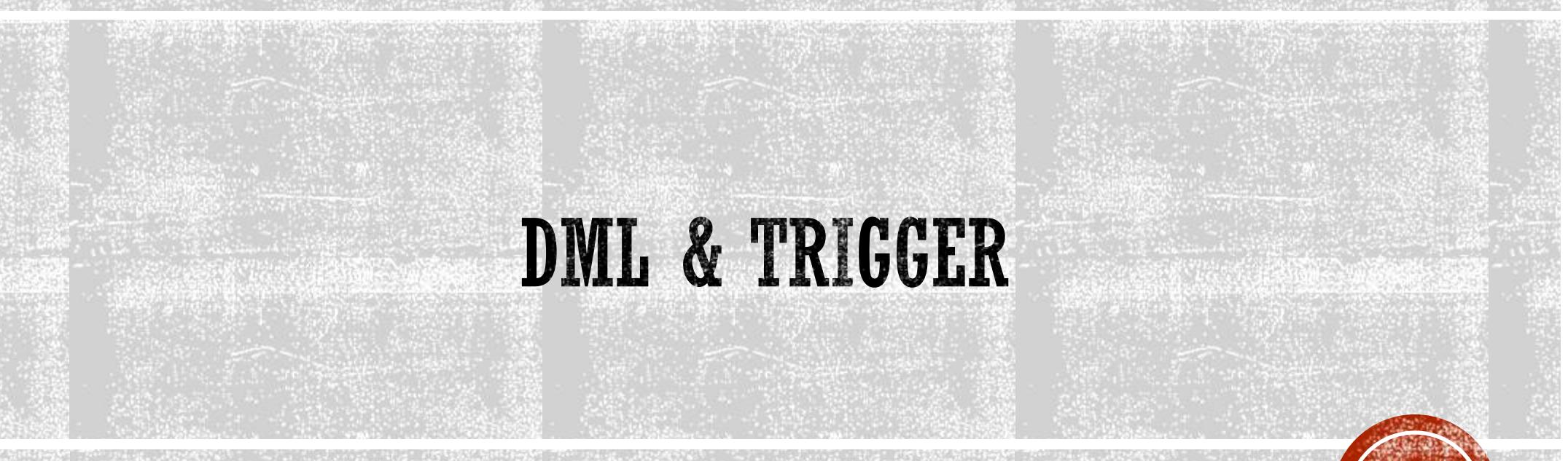
- Ora.
- Chat di Teams
- Mail: silvio.barra@unina.it



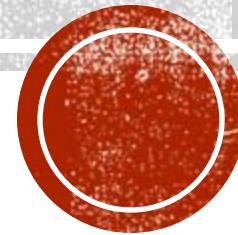
BASI DI DATI I

- 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>]

1. Si lavora su NomeTabella
2. Si modificano le righe che soddisfano <condizione>
3. 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)

- Quando il carrello è completo
 - Creare un nuovo ordine con la struttura del carrello (Lavoriamo con il codice 'XYZ')

Magazzino (CodArticolo, Descrizione, Quantità)

Ordine (CodOrdine, Data, PIVA, DataInvio)

CompOrdine (CodOrdine, CodArticolo*, Quantità, Prezzo)*

Carrello (CodCarrello, Data, PIVA, Completo)

CompCarrello (CodCarrello, CodArticolo*, Quantità, Prezzo)*



ESEMPIO (3)

- Creare un nuovo ordine con la struttura del carrello (Lavoriamo con il carrello con il codice 'XYZ')

1) **INSERT INTO** Ordine (CodOrdine, Data, PIVA, DataInvio)

```
(SELECT C.CodCarrello, C.Data, C.PIVA, NULL  
  FROM Carrello C  
 WHERE C.CodCarrello = 'XYZ')
```

Oppure

INSERT INTO Ordine (CodOrdine, Data, PIVA)

```
(SELECT C.CodCarrello, C.Data, C.PIVA  
  FROM Carrello C  
 WHERE C.CodCarrello = 'XYZ')
```



ESEMPIO (4)

- Quando il carrello è completo
 - Creare un nuovo ordine con la struttura del carrello (Lavoriamo con il codice 'XYZ')

Magazzino (CodArticolo, Descrizione, Quantità)

Ordine (CodOrdine, Data, PIVA, DataInvio)

CompOrdine (CodOrdine, CodArticolo*, Quantità, Prezzo)*

Carrello (CodCarrello, Data, PIVA, Completo)

CompCarrello (CodCarrello, CodArticolo*, Quantità, Prezzo)*



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)

- Quando il carrello è completo
 - Creare un nuovo ordine con la struttura del carrello (Lavoriamo con il codice 'XYZ')

Magazzino (CodArticolo, Descrizione, Quantità)

Ordine (CodOrdine, Data, PIVA, DataInvio)

CompOrdine (CodOrdine, CodArticolo*, Quantità, Prezzo)*

Carrello (CodCarrello, Data, PIVA, Completo)

CompCarrello (CodCarrello, CodArticolo*, Quantità, Prezzo)*



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)

- Quando il carrello è completo
 - Aggiornare le scorte nel magazzino
 - Per ogni articolo nell'ordine 'XYZ' devo rimuoverne la quantità venduta nel magazzino

Magazzino (CodArticolo, Descrizione, Quantità)

Ordine (CodOrdine, Data, PIVA, DataInvio)

CompOrdine (CodOrdine, CodArticolo*, Quantità, Prezzo)*

Carrello (CodCarrello, Data, PIVA, Completo)

CompCarrello (CodCarrello, CodArticolo*, Quantità, Prezzo)*



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



TRIGGER

- Un tipico trigger è definito come una la composizione di tre componenti
 - L'**EVENTO** che lancia il trigger. Solitamente operazioni di aggiornamenti esplicativi alla base di dati.
 - Inserimento di un record
 - Cancellazione di un record
 - Modifica di un record
 - Eventi innescati temporalmente o altri eventi esterni
 - La **CONDIZIONE** che determina se l'azione della regola deve essere eseguita
 - Quando si verifica l'evento si può valutare una condizione (facoltativa – in mancanza l'azione è effettuata ogni volta che si verifica l'evento)
 - Se la condizione è specificata allora l'azione è eseguita ogni volta che questa da valore TRUE
 - L'**AZIONE** da intraprendere
 - Di solito l'azione corrisponde ad una sequenza di istruzioni SQL ma può essere anche l'esecuzione di una transazione, di una procedura o di un programma esterno



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**]

[**WHEN** <condizione>]

BEGIN

<CORPO DELLA PROCEDURA>

END

*INSERT ON <tabella>
DELETE ON <tabella>
UPDATE ON <tabella> OF <ATTRIBUTO>*



SINTASSI

- **CREATE TRIGGER** <*nomeTrigger*>
 - Specifica il nome del trigger
- **[AFTER/BEFORE/INSTEAD OF]** <*operazione*>
 - AFTER indica che la regola deve essere attivata dopo il verificarsi degli eventi
 - BEFORE indica che la regola deve essere attivata prima del verificarsi degli eventi
 - INSTEAD OF esegue il trigger invece di eseguire l'evento (aggiornamenti su viste)
- **[FOR EACH ROW]**
 - Indica che la regola verrà attivata *una volta per ogni riga* influenzata dall'evento
 - In assenza della clausola [FOR EACH ROW] l'azione verrà eseguita una sola volta, anche se più righe sono state influenzate dall'evento
 - L'uso della clausola attiva due variabili
 - **NEW** che permette di riferirsi alla tupla appena inserita/aggiornata (in caso di INSERT o UPDATE)
 - **OLD** che permette di riferirsi alla tupla eliminata o prima dell'aggiornamento (DELETE o UPDATE)
- **[WHEN** <*condizione*>]
 - Specifica la condizione che deve essere controllata dopo che la regola è innescata, MA prima che l'azione è eseguita



ESEMPIO

IMPIEGATO				
Nome	SSN	Stipendio	Num_D	SSN_Super

DIPARTIMENTO			
Nome_D	Num_D	Stip_Totale	SSN_Direttore

- Stip_Totale è un attributo derivato che deve essere aggiornato per mantenere il valore consistente all'interno del dipartimento
- Gli eventi che *possono causare* un cambiamento sono
 1. Inserimento di un nuovo impiegato
 2. Il cambiamento dello stipendio di uno o più impiegati
 3. Il passaggio di un impiegato da un dipartimento ad un altro
 4. L'eliminazione di un impiegato



EVENTO 1

- Bisogna ricalcolare STIP_TOTALE per il dipartimento all'interno del quale è inserito il nuovo impiegato



EVENTO 1

- Bisogna ricalcolare STIP_TOTALE per il dipartimento all'interno del quale è inserito il nuovo impiegato

```
CREATE TRIGGER StipTotale1  
AFTER INSERT ON Impiegato  
FOR EACH ROW  
WHEN(NEW.Num_D IS NOT NULL)  
    UPDATE Dipartimento  
        SET Stip_Totale = Stip_Totale + NEW.Stipendio  
        WHERE Num_D = NEW.Num_D;
```



EVENTO 2

- Lo stipendio deve essere ricalcolato nel caso in cui lo stipendio di uno o più impiegati venisse modificato



EVENTO 2

- Lo stipendio deve essere ricalcolato nel caso in cui lo stipendio di uno o più impiegati venisse modificato

```
CREATE TRIGGER StipTotale2  
AFTER UPDATE OF Stipendio ON Impiegato  
FOR EACH ROW  
WHEN(NEW.Num_D IS NOT NULL)  
    UPDATE Dipartimento  
        SET Stip_Totale = Stip_Totale + NEW.Stipendio - OLD.Stipendio  
        WHERE Num_D = NEW.Num_D;
```



EVENTO 3

- Se un impiegato passa da un dipartimento ad un altro, sono due i dipartimenti che devono essere modificati



EVENTO 3

- Se un impiegato passa da un dipartimento ad un altro, sono due i dipartimenti che devono essere modificati

```
CREATE TRIGGER StipTotale3
AFTER UPDATE OF Num_D ON Impiegato
FOR EACH ROW
BEGIN
    UPDATE Dipartimento
    SET Stip_Totale = Stip_Totale + NEW.Stipendio
    WHERE Num_D = NEW.Num_D;
    UPDATE Dipartimento
    SET Stip_Totale = Stip_Totale - OLD.Stipendio
    WHERE Num_D = OLD.Num_D;
END;
```



EVENTO 4

- Lo stipendio deve essere ricalcolato nel caso in cui un impiegato è licenziato o lascia il dipartimento



EVENTO 4

- Lo stipendio deve essere ricalcolato nel caso in cui un impiegato è licenziato o lascia il dipartimento

```
CREATE TRIGGER StipTotale4  
AFTER DELETE ON Impiegato  
FOR EACH ROW  
WHEN(OLD.Num_D IS NOT NULL)  
    UPDATE Dipartimento  
        SET Stip_Totale = Stip_Totale - OLD.Stipendio  
        WHERE Num_D = OLD.Num_D;
```



FOR EACH ROW

- È importante notare l'effetto della clausola facoltativa FOR EACH ROW
 - La regola viene innescata separatamente per ogni tupla
 - Noto come **trigger a livello di riga**
- Se la clausola venisse tralasciata, il trigger sarebbe noto come **trigger a livello di istruzione** e verrebbe attivato una volta per ogni istruzione
- Esempio: si consideri la seguente operazione di aggiornamento che aumenta del 10% lo stipendio a tutti gli impiegati

UPDATE Impiegato

SET Stipendio = 1.1*Stipendio

WHERE Num_D=5



EVENTO 5

- Vogliamo rilevare tutte le volte che lo stipendio di un impiegato supera lo stipendio del suo supervisore
- Sono molti gli eventi che possono innescare questo evento
 - Inserimento di un nuovo impiegato, modifica di stipendio o modifica di dipartimento per un impiegato
 - Supponiamo che l'azione da eseguire è una procedura SUPER_INFORM che informa il superiore dell'evento



EVENTO 5

- Vogliamo rilevare tutte le volte che lo stipendio di un impiegato supera lo stipendio del suo supervisore
- Sono molti gli eventi che possono innescare questo evento
 - Inserimento di un nuovo impiegato, modifica di stipendio o modifica di dipartimento per un impiegato
 - Supponiamo che l'azione da eseguire è una procedura SUPER_INFORM che informa il superiore dell'evento

CREATE TRIGGER Super_Inform

BEFORE INSERT OR UPDATE OF Stipendio, SSN_Super **ON** Impiegato

FOR EACH ROW

WHEN(NEW.Stipendio > (SELECT Stipendio **FROM Impiegato**

WHERE SSN = NEW.SSN_Super))

SUPER_INFORM(NEW.SSN_Super, NEW.SSN)



TRIGGER SENZA FOR EACH ROW

```
CREATE TABLE IMPIEGATO (
    id_impiegato NUMBER PRIMARY KEY,
    nome VARCHAR2(100),
    id_dipartimento NUMBER,
    FOREIGN KEY (id_dipartimento) REFERENCES DIPARTIMENTO(id_dipartimento)
);
```

```
CREATE TABLE DIPARTIMENTO (
    id_dipartimento NUMBER PRIMARY KEY,
    nome VARCHAR2(100)
);
```

```
CREATE TABLE LOG_DIPARTIMENTO (
    id_log NUMBER GENERATED ALWAYS AS IDENTITY,
    azione VARCHAR2(100),
    data_azione DATE
);
```



- Ogni volta che viene eseguita **una cancellazione** sulla tabella IMPIEGATO, vogliamo registrare nel LOG_DIPARTIMENTO che "qualche impiegato è stato eliminato", **senza preoccuparci di quali o quanti.**

```
CREATE OR REPLACE TRIGGER trg_log_delete_impiegato
AFTER DELETE ON IMPIEGATO
BEGIN
    INSERT INTO LOG_DIPARTIMENTO (azione, data_azione)
    VALUES ('Cancellazione da IMPIEGATO', SYSDATE);
END;
```



TORNANDO ALL'ESEMPIO DI DEL MAGAZZINO

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?**



TORNANDO ALL'ESEMPIO DI DEL MAGAZZINO

CREATE TRIGGER creaOrdine

AFTER UPDATE ON CARRELLO OF Completo

WHEN OLD.Completo = 'Incompleto' AND NEW.Completo = 'Completo'

FOR EACH ROW

BEGIN

INSERT INTO Ordine (CodO, Data, PIVA)

(SELECT C.CodCarrello, C.Data, C.PIVA

FROM Carrello C

WHERE C.CodCarrello = '**NEW.CodCarrello**');


```
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 = 'XYZ');
```

END;



INFO SULLA LEZIONE

- Capitolo 8
 - 8.2
- Capitolo 12
 - Fino al 12.2 escluso

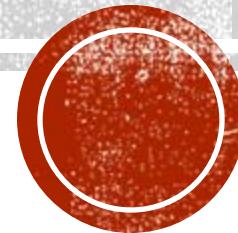




FINE

Per eventuali domande: (in ordine di preferenza personale)

- Ora.
- Chat di Teams
- Mail: silvio.barra@unina.it



Basi di Dati 1, 10 novembre 2023 - Compito B

Mara Sangiovanni, Silvio Barra

DIETI, Corso di Laurea in Informatica, Università di Napoli ‘Federico II’, Italy

Si consideri il seguente schema relazionale che descrive un frammento della base di dati di una applicazione per lo scambio di file peer-to-peer. Ogni utilizzatore (*UTENTE*) del sistema è caratterizzato da un identificativo (*IdUtente*), un Nome (*NomeUtente*), da una banda in ingresso (*Banda_IN*) e da una banda in uscita (*Banda_OUT*). Un *FILE* è caratterizzato da un identificativo (*IdFile*), da un nome (*NomeFile*), dal tipo (*Tipo*) e dalle sue dimensioni (*Dimensioni*). Ogni utente può condividere diversi file come espresso dalla tabella *POSSIEDE*, dove *IdFile* è l'identificativo del file e *IdUtente* è il nome di un utente che lo condivide. Lo schema *TRASFERIMENTO* rappresenta i trasferimenti attivi e conclusi: *IdPropr* è l'utente da cui viene scaricato il file, *IdF* è l'identificativo del file scaricato, *IdRich* è l'utente che lo sta scaricando, *Banda* è la banda impiegata per quel trasferimento. L'attributo parziale *Completo* è diverso da *NULL* se il trasferimento è stato completato con successo, in tal caso anche la data *DataC* è diversa da *NULL*.

UTENTE(IdUtente, NomeUtente, Banda_IN, Banda_OUT)
FILE(IdFile, NomeFile, Tipo, Dimensioni)
POSSIEDE(IdUtente, IdFile)
TRASFERIMENTO(IdPropr, IdRich, IdF, Banda, DataC, Completo)

Esercizio 01 (Punti 8)

Si scriva un'espressione in algebra relazionale che, se valutata, fornisce il nome di tutti gli utenti che non hanno mai scaricato lo stesso file completo.

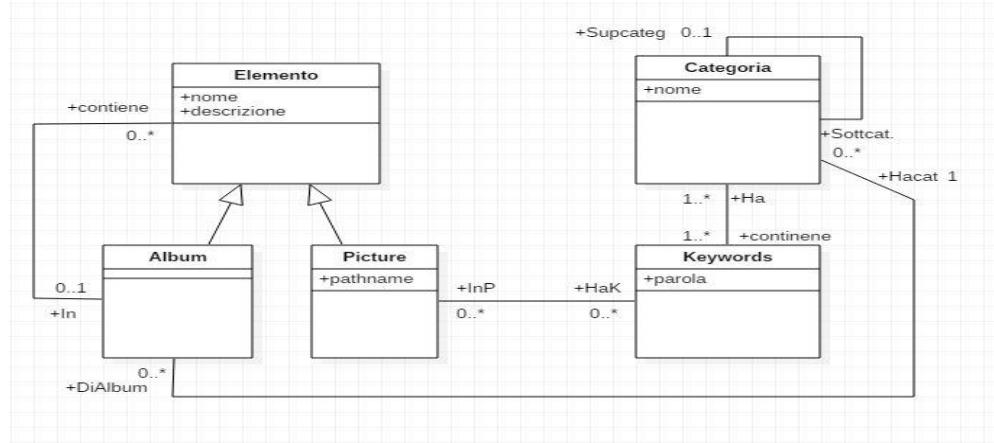
Esercizio 02 (Punti 8)

Si crei una vista che fornisca, per ogni utente, il riepilogo dei download completi raggruppati per mese ed anno. Si vogliono in output: nome utente, il numero totale di file scaricati con successo per ogni anno e mese, e la banda totale utilizzata in quei periodi.

Esercizio 03 (Punti 8) Per ognuno dei seguenti vincoli se ne descriva il tipo, e lo si implementi in SQL nel modo più opportuno:

1. *Un utente non può possedere più volte lo stesso file.*
2. *In un trasferimento completo sia DataC che Completo devono essere diversi da null.*
3. *Un trasferimento non può far riferimento ad un proprietario inesistente.*
4. *Non possono esistere nella base di dati due file con lo stesso ID.*

Esercizio 04 (*Punti 8*)



Si consideri la bozza di Class Diagram in figura che descrive gli elementi componenti un class diagram.

- Effettuare la ristrutturazione e motivarne il processo;
- Definire lo schema logico;

Esercitazione

Silvio Barra

DIETI, Corso di Laurea in Informatica, Università di Napoli ‘Federico II’, Italy

Si consideri il seguente schema relazionale che descrive un frammento della base di dati di una applicazione per lo scambio di file peer-to-peer. Ogni utilizzatore (*UTENTE*) del sistema è caratterizzato da un identificativo (*IdUtente*), un Nome (*NomeUtente*), da una banda in ingresso (*Banda_IN*) e da una banda in uscita (*Banda_OUT*). Un *FILE* è caratterizzato da un identificativo (*IdFile*), da un nome (*NomeFile*), dal tipo (*Tipo*) e dalle sue dimensioni (*Dimensioni*). Ogni utente può condividere diversi file come espresso dalla tabella *POSSIEDE*, dove *IdFile* è l'identificativo del file e *IdUtente* è il nome di un utente che lo condivide. Lo schema *TRASFERIMENTO* rappresenta i trasferimenti attivi e conclusi: *IdPropr* è l'utente da cui viene scaricato il file, *IdF* è l'identificativo del file scaricato, *IdRich* è l'utente che lo sta scaricando, *Banda* è la banda impiegata per quel trasferimento. L'attributo parziale *Completo* è diverso da *NULL* se il trasferimento è stato completato con successo, in tal caso anche la data *DataC* è diversa da *NULL*.

UTENTE(IdUtente, NomeUtente, Banda_IN, Banda_OUT)
FILE(IdFile, NomeFile, Tipo, Dimensioni)
POSSIEDE(IdUtente, IdFile)
TRASFERIMENTO(IdPropr, IdRich, IdF, Banda, DataC, Completo)

Esercizio 01 Algebra Relazionale

- Si scriva un'espressione in algebra relazionale che, se valutata, fornisce il nome utente degli utenti con il numero massimo di file posseduti.
- Si scriva un'espressione in algebra relazionale che, se valutata, fornisce id e nome dei File con il maggiore numero di download completati.
- Si scriva un'espressione in algebra relazionale che, se valutata, fornisce, per ogni mese e anno, il file più scaricato.

Esercizio 02 SQL

- Si scriva una query SQL che fornisca la lista di utenti che hanno scaricato solo file di tipo “PDF”.
- Si scriva una query SQL che fornisca la lista di file che nessuno ha mai scaricato per intero.
- Si scriva una query SQL che fornisca gli utenti che non hanno mai scaricato un file che invece è stato scaricato da tutti gli altri utenti.

NomeUtente | IDFile

Per cui

- 1) L'utente NomeUtente non ha scaricato il file IDFile**
- 2) IDFile è stato scaricato da tutti gli altri utenti**



AR (1)

$$\# \text{filePossedui} = \sum_{\text{id_utente}, \text{esiste}} \text{fcont}(+) \quad (\text{Ressource})$$

$$\# \text{MAX} = \sum_{\text{id_utente}} (\# \text{filePossedui}) \quad (\# \text{FILE})$$

$$\text{USERS_MAX} = \frac{\# \text{filePossedui} \times \# \text{MAX}}{\# \text{FILE}} = \frac{\# \text{filePossedui} \times \# \text{MAX}}{\# \text{MAXFILE}}$$

$$R = \frac{\prod_{\text{id_utente}} (\text{USERS_MAX} * \text{UTENTE})}{\prod_{\text{id_utente}}}$$

AR (2)

$$\text{Conf} = G \quad (\text{transfimento})$$

$$\text{conf_egi} = \text{true}$$

$$\text{CONTENUTO} = \sum_{\text{id_file}} \text{fcont}(+) \rightarrow \# \text{download} \quad (\text{id_file, id_utente} \quad (\text{CONF}))$$

$$\text{MAX_D} = \sum_{\text{id_file}} (\# \text{download}) \quad (\text{CONTENUTO})$$

$$\text{FILEMAX} = \sum_{\text{id_file}} (\text{CONTENUTO} \times \text{MAX_D})$$

$$\text{MAX_D} = \frac{\# \text{download}}{\# \text{FILE}}$$

$$R = \frac{\prod_{\text{id_file}, \text{noncf}} (\text{FILEMAX} * \text{FILE})}{\prod_{\text{id_file}, \text{noncf}}}$$



AR (3)

$$\text{Conf} @ = \Gamma \quad (\text{CLASSIFICAZIONE})$$

CLASSIFICAZIONE
= TRUE

$$\text{DATI} = " \quad (\text{Conf})$$

(year(DATAC), \rightarrow ANNO
month(DATAC) \rightarrow MESE
IDF)

$$\text{CONT} = \text{ANNO}, \text{MESE}, \text{IDF}, \cancel{\text{DATA}}$$

(DATI)
 $\nexists \text{COUNT}(\star) \rightarrow \# \text{DOVVL}$

$$\text{MAX-G} = \text{ANNO}, \text{MESE} \quad (\text{CONT})$$

$\nexists \text{MAX}(\# \text{DOVVL}) \rightarrow \text{MAX-D}$

$$\text{TOP-G} = \Gamma \quad (\text{CONT} \times \text{MAX-G})$$

$\frac{\# \text{DOVVL}}{\text{MAX-D}}$



SQL(1)

```
SELECT u.idUtente  
FROM UTENTE u  
WHERE NOT EXISTS (  
    SELECT *  
    FROM TIPASK T JOIN FILE f  
    ON T.idf = f.idfile  
    WHERE t.idUtente = u.idUtente  
    AND T.completo = TRUE  
    AND f.tipo <> 'PDF' ))
```

SQL(2)

```
SELECT f.idfile, f.nomefile  
FROM file f  
WHERE NOT EXISTS (  
    SELECT *  
    FROM TIPASK T  
    WHERE T.idf = f.idfile AND  
        T.completo = TRUE )
```



SQL (3)

Esiste un file
che

SELECT u.NameUtente
from UTENTE u
WHERE EXISTS (
SELECT *

1) non è file
scambiato con U

2) è stato scambiato

da tutti gli altri

from file f

WHERE NOT EXISTS (

SELECT *

from TRASFERIMENTO T

WHERE T.idRich = u.idUtente

AND T.idFile = f.idFile

AND T.completato = true)

7

And NOT EXISTS (

SELECT *

from UTENTE u2

WHERE u2.idUtente <> u.idUtente

And u2.idUtente NOT IN (

SELECT T2.idRich

from TRASFERIMENTO T2

Where T2.idF = f.idFile

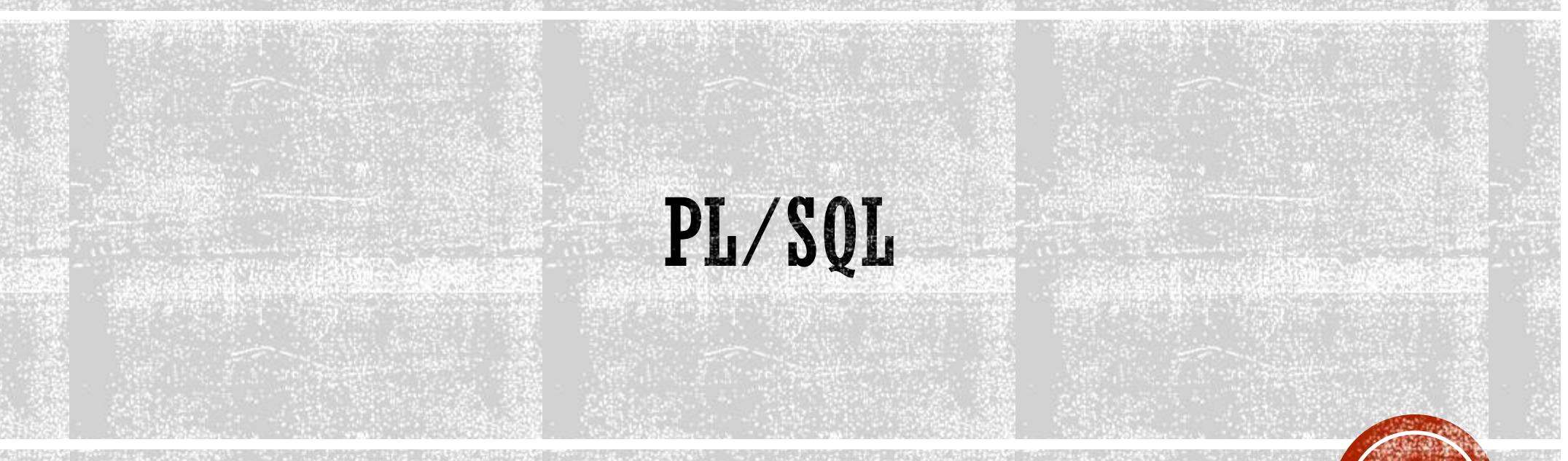
And T2.completato = true)

8

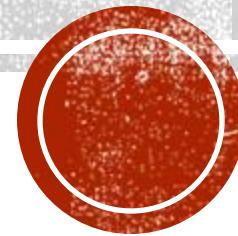
BASI DI DATI I

- PL/SQL
- SQL dinamico





PL/SQL



COSA È PL/SQL

- **PL/SQL (Procedural Language/SQL)** è un'estensione procedurale del linguaggio SQL sviluppata da Oracle.
- **Caratteristiche principali:**
 - Integrazione nativa con SQL
 - Supporto per la **programmazione strutturata** (blocchi, condizioni, cicli)
 - Gestione **robusta degli errori** (eccezioni)
 - Migliora la **modularità** e la **leggibilità** del codice SQL
 - Permette di **automatizzare** operazioni complesse nei database Oracle
- **Contesto d'uso:**
 - Trigger
 - Stored procedures e functions
 - Packages
 - Script di manutenzione e automazione



PERCHÉ USARE PL/SQL

- **Limiti del solo SQL:**

- SQL è dichiarativo, non procedurale
- Non consente condizioni complesse, cicli, o strutture modulari

- **Vantaggi di PL/SQL:**

- **Performance:** riduce il numero di chiamate tra applicazione e database
- **Controllo:** permette logiche più articolate rispetto al solo SQL
- **Riutilizzabilità:** codice organizzato in procedure, funzioni, e package
- **Affidabilità:** gestione strutturata degli errori

- "PL/SQL è al database Oracle quello che un linguaggio di programmazione è a un sistema operativo."



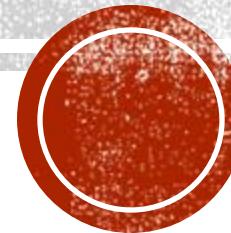
COSA È PL/SQL

- **Procedural Language extension for the Structured Query Language**
- Vengono aggiunti molti costrutti tipici della programmazione procedurale
- PL/SQL è EMBEDDED
 - è un linguaggio proprietario, che può essere eseguito solo all'interno del DBMS
- Permette di spostare parte del carico computazionale sul server DBMS

- È Naturalmente integrato con SQL
- Non c'è bisogno di fare conversione di tipi (***%type, %rowtype***)
- È possibile lavorare sui risultati di una query anche una riga alla volta.



STRUTTURA PL/SQL



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 (/)



BLOCCHI ANONIMI

- È un blocco di codice **non salvato nel database**.
- Viene **eseguito una tantum**, tipicamente in ambienti di sviluppo o test.

```
DECLARE
    -- dichiarazioni (opzionale)
BEGIN
    -- istruzioni esecutive
EXCEPTION
    -- gestione eccezioni (opzionale)
END;
```



BLOCCHI ANONIMI: CARATTERISTICHE

- **Non ha un nome**
 - non può essere richiamato direttamente.
- **Volatile**
 - esiste solo durante l'esecuzione.
- **Non può essere invocato da altri programmi.**
- Ideale per test, script ad hoc, e operazioni non ricorrenti.



BLOCCHI ANONIMI: HELLO, WORLD!

```
DECLARE
    message VARCHAR(100) := 'Hello, World!'
BEGIN
    DBMS_OUTPUT.put_line(message);
EXCEPTION
    WHEN OTHERS
    THEN
        DBMS_OUTPUT.put_line(SQLERRM);
END;
```



BLOCCHI ANONIMI: ALTRO ESEMPIO

```
SET SERVEROUTPUT ON          Attiva il server, in modo da poter vedere  
                            le righe prodotte dalle successive chiamate  
  
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;  
/
```

Dichiaro 3 interi, di cui il secondo e il terzo sono anche inizializzati

La chiamata al DBMS_OUTPUT, propria dei DBMS ORACLE, mostra a video un testo.

Eventuali eccezioni sono gestite nel blocco exception.



BLOCCHI ANONIMI: NESTED HELLO, WORLD!

```
DECLARE
    message VARCHAR2 (100) := 'Hello';
BEGIN
    DECLARE
        message2 VARCHAR2 (100) := message || ' World!';
        BEGIN
            DBMS_OUTPUT.put_line (message2); END;
    EXCEPTION
        WHEN OTHERS
        THEN
            DBMS_OUTPUT.put_line (DBMS_UTILITY.format_error_stack);
END;
```



BLOCCO ANONIMO SU POSTGRES

```
DO $$  
DECLARE  
    saluto TEXT := 'Ciao dal blocco anonimo!';  
BEGIN  
    RAISE NOTICE '%', saluto;  
END;  
$$;
```

NOTICE: Ciao dal blocco anonimo!

- DO \$\$... \$\$; è il contenitore del blocco anonimo.
- La sezione DECLARE serve per dichiarare variabili (opzionale).
- In BEGIN ... END scrivi il codice da eseguire.
- RAISE NOTICE stampa messaggi simili a DBMS_OUTPUT.PUT_LINE in Oracle.



POSTGRES VS ORACLE

- **Blocco Anonimo:**

- **ORACLE:** DECLARE... BEGIN... END;
- **POSTGRES:** DO \$\$ DECLARE... BEGIN... END; \$\$;

- **Output Su Console:**

- **ORACLE:** DBMS_OUTPUT.PUT_LINE;
- **POSTGRES:** RAISE NOTICE

- **Obbligo di Linguaggio:**

- **ORACLE:** Implicito PL/SQL;
- **POSTGRES:** Esplicito (DO LANGUAGE plpgsql) se non predefinito



BLOCCHI NON ANONIMI

- È un blocco di codice **salvato nel database con un nome identificativo**.
- Può essere richiamato più volte e da altri programmi.
- **Tipi principali**
 - **Procedure** – eseguono un'azione.
 - **Function** – restituiscono un valore.
 - **Package** – gruppi di procedure e funzioni.
 - **Trigger** – eseguiti automaticamente su eventi specifici (INSERT, UPDATE, DELETE).

```
CREATE OR REPLACE PROCEDURE saluta_utente(p_nome IN VARCHAR2) IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Ciao ' || p_nome);
END;
```



BLOCCHI NON ANONIMI: CARATTERISTICHE

- **Ha un nome**
 - Può essere richiamato da altri programmi o query.
- **Persistente**
 - Salvato nel database finché non viene rimosso.
- **Favorisce il riuso e la modularizzazione.**
- Può accettare **parametri di input/output**.



BLOCCHI NON ANONIMI: PROCEDURE

```
CREATE OR REPLACE PROCEDURE hello_world
IS
    Message VARCHAR2 (100) := 'Hello World!';
BEGIN
    DBMS_OUTPUT.put_line (message);
END hello_world;
```

```
BEGIN
    hello_world;
END
```



BLOCCHI CON SQL

```
DECLARE
    l_name employees.last_name%TYPE;
BEGIN
    SELECT last_name INTO l_name
    FROM employees
    WHERE employee_id = 138;
    DBMS_OUTPUT.put_line (l_name);
END;
```



DICHIARAZIONE DI VARIABILI

```
DECLARE
part_number      NUMBER(6) ; -- SQL data type
part_name        VARCHAR(20) ; -- SQL data type
in_stock          BOOLEAN; -- PL/SQL Only data type
part_price        NUMBER(6,2); -- SQL data type
part_description  VARCHAR2(50); -- SQL data
type
BEGIN NULL;
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;
```



ESEMPIO

```
IF salary BETWEEN 10000 AND 20000
THEN
    give_bonus(employee_id, 1000);
ELSIF salary > 20000
THEN
    give_bonus(employee_id, 500);
ELSE
    give_bonus(employee_id, 0);
END IF;
```



COSTRUTTI ITERATIVI

- Abbiamo 3 tipi di costrutti iterativi
 1. **FOR LOOP** che cicla per un predeterminato numero di volte
 2. **SIMPLE LOOP** semplici che vengono eseguiti finché non si chiude esplicitamente il ciclo
 3. **WHILE** che cicla finché è verificata una determinata condizione



FOR LOOP

- 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;
```



FOR LOOP (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;
```



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;
```



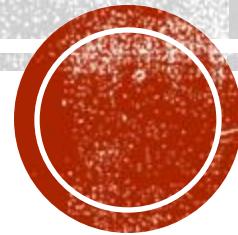
WHILE LOOP

*WHILE condition LOOP
statements
END LOOP;*

```
v_counter := 0;  
WHILE v_counter < 6 LOOP  
    v_counter := v_counter + 1;  
END LOOP;
```



CURSORI



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;
/

```

ORACLE

VS

POSTGRES

```
DECLARE
  CURSOR estrai_cognome IS
    SELECT S.nome, S.matricola
    FROM Studenti S
    WHERE S.cognome = 'Barra';
  riga_corrente estrai_cognome%ROWTYPE
BEGIN
  OPEN estrai_cognome;
  LOOP
    FETCH estrai_cognome INTO riga_corrente;
  END LOOP;
  CLOSE estrai_cognome
END;
```

```
DECLARE
  estrai_cognome CURSOR FOR
    SELECT S.nome, S.matricola
    FROM Studenti S
    WHERE S.cognome = 'Barra';
  riga_corrente estrai_cognome%TYPE;
BEGIN
  OPEN estrai_cognome;
  LOOP
    FETCH estrai_cognome INTO riga_corrente;
  END LOOP;
  CLOSE estrai_cognome
END;
```



ESEMPI BLOCCHI ANONIMI

EMPLOYEE (SSN, fname, minit, lname, bdate, address, sex, salary, superSSN, dno)

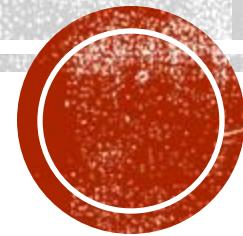
DEPARTMENT (dnumber, dname, mgrssn, mgrstartdate)

DEPENDENT (ESSN, dependent_name, sex, bdate, relationship)

DEPT_LOCATIONS (dnumber, dlocations)

PROJECT (pnumber, pname, plocation, dnum)

WORKS_ON (ESSN, PNO, hours)



ESEMPIO 1

- CALCOLA IL SALARIO MEDIO E LO STAMPA

```
DO $$  
DECLARE  
    avg_salary NUMERIC;  
BEGIN  
    SELECT AVG(salary) INTO avg_salary FROM EMPLOYEE;  
    RAISE NOTICE 'Salario medio: %', avg_salary;  
END;  
$$;
```



ESEMPIO 2

- VERIFICA SE ESISTE UN DIPENDENTE CON SSN SPECIFICO

```
DO $$  
DECLARE  
    exists_flag BOOLEAN;  
BEGIN  
    SELECT EXISTS ( SELECT 1 FROM EMPLOYEE WHERE SSN = '123456789')  
    INTO exists_flag;  
  
    IF exists_flag THEN RAISE NOTICE 'Il dipendente esiste.';  
    ELSE RAISE NOTICE 'Il dipendente NON esiste.';  
    END IF;  
END;  
$$;
```



ESEMPIO 3

- INSERISCE UN NUOVO PROGETTO SE NON ESISTE

```
DO $$  
DECLARE  
    found BOOLEAN;  
BEGIN  
    SELECT EXISTS ( SELECT 1 FROM PROJECT WHERE pname = 'Progetto X' )  
    INTO found;  
  
    IF NOT found THEN  
        INSERT INTO PROJECT(pnumber, pname, plocation, dnum)  
        VALUES (99, 'Progetto X', 'Roma', 1);  
        RAISE NOTICE 'Progetto inserito.';  
    ELSE RAISE NOTICE 'Il progetto esiste già.';  
    END IF;  
END;  
$$;
```



ESEMPIO 4

- ELIMINA TUTTE LE PERSONE A CARICO PER UN IMPIEGATO

```
DO $$  
BEGIN  
    DELETE FROM DEPENDENT WHERE ESSN = '123456789';  
    RAISE NOTICE 'Tutti i dependent del dipendente 123456789  
sono stati rimossi.';  
END;  
$$;
```



ESEMPIO 5

- AGGIORNA LO STIPENDIO DI UN DIPENDENTE

```
DO $$  
BEGIN  
    UPDATE EMPLOYEE  
    SET salary = salary + 2000  
    WHERE SSN = '123456789';  
  
    RAISE NOTICE 'Stipendio aggiornato per il dipendente 123456789.';  
END; $$;
```



ESEMPIO 6

- ELENCA TUTTI I PROGETTI ED IL DIPARTIMENTO ASSOCIATO

```
DO $$  
DECLARE  
    rec RECORD;  
    cur CURSOR FOR  
        SELECT p.pname, d.dname  
        FROM PROJECT p  
        JOIN DEPARTMENT d ON p.dnum =  
d.dnumber;  
    BEGIN  
        OPEN cur;  
        LOOP  
            FETCH cur INTO rec;  
            EXIT WHEN NOT FOUND;  
            RAISE NOTICE 'Progetto: %,  
Dipartimento: %', rec.pname, rec.dname;  
        END LOOP;  
        CLOSE cur;  
    END; $$;
```



ESEMPIO 7

- MOSTRA TUTTI I DIPENDENTI ED IL RELATIVO SALARIO

```
DO $$  
DECLARE  
    emp RECORD;  
    emp_cur CURSOR FOR SELECT fname, lname, salary FROM EMPLOYEE;  
BEGIN  
    OPEN emp_cur;  
    LOOP  
        FETCH emp_cur INTO emp;  
        EXIT WHEN NOT FOUND;  
        RAISE NOTICE 'Dipendente: % %, Salario: %', emp.fname, emp.lname, emp.salary;  
    END LOOP;  
    CLOSE emp_cur;  
END; $$;
```



ESEMPIO 8

- ELENCA I DIPENDENTI E LE ORE LAVORATE PER OGNI PROGETTO

```
DO $$  
DECLARE  
    rec RECORD;  
    cur CURSOR FOR SELECT e.fname, e.lname, p.pname, w.hours  
        FROM EMPLOYEE e JOIN WORKS_ON w ON e.ssn = w.essn JOIN PROJECT p ON p.pnumber = w.pno;  
BEGIN  
    OPEN cur;  
    LOOP  
        FETCH cur INTO rec;  
        EXIT WHEN NOT FOUND;  
        RAISE NOTICE '% % lavora al progetto "%" per % ore', rec.fname, rec.lname, rec.pname, rec.hours;  
    END LOOP;  
    CLOSE cur;  
END; $$;
```



ESEMPIO 9

- MOSTRA I MANAGER E LA LORO DATA DI INIZIO MANDATO

```
DO $$

DECLARE

    mgr RECORD;

    cur CURSOR FOR SELECT e.fname, e.lname, d.mgrstartdate
                  FROM EMPLOYEE e JOIN DEPARTMENT d ON e.ssn = d.mgrssn;

BEGIN

    OPEN cur;

    LOOP

        FETCH cur INTO mgr;
        EXIT WHEN NOT FOUND;
        RAISE NOTICE 'Manager: % %, Gestisce dal: %', mgr.fname, mgr.lname, mgr.mgrstartdate;
    END LOOP;

    CLOSE cur;

END; $$;
```



ESEMPIO 10

- MOSTRA I DIPENDENTI CON IL RELATIVO NUMERO DI PERSONE A CARICO

```
DO $$  
DECLARE  
    rec RECORD;  
  
    cur CURSOR FOR SELECT e.fname, e.lname, COUNT(d.dependent_name) AS num_dependents  
        FROM EMPLOYEE e JOIN DEPENDENT d ON e.ssn = d.essn  
        GROUP BY e.ssn;  
  
BEGIN  
  
    OPEN cur;  
  
    LOOP  
        FETCH cur INTO rec;  
        EXIT WHEN NOT FOUND;  
        RAISE NOTICE '% % ha % dependent(i)', rec.fname, rec.lname, rec.num_dependents;  
    END LOOP;  
  
    CLOSE cur;  
  
END; $$;
```



ESEMPIO 11

- STAMPA I NOMI DEGLI IMPIEGATI, E LE LOCATION DEI PROGETTI SU CUI LAVORA

```
DO $$

DECLARE

    rec RECORD;

    cur CURSOR FOR SELECT e.fname, e.lname, p.plocation
        FROM EMPLOYEE e JOIN WORKS_ON w ON e.ssn = w.essn JOIN PROJECT p ON w.pno = p.pnumber;

BEGIN

    OPEN cur;

    LOOP

        FETCH cur INTO rec;

        EXIT WHEN NOT FOUND;

        RAISE NOTICE '% % lavora su un progetto localizzato a %', rec.fname, rec.lname, rec.plocation;

    END LOOP;

    CLOSE cur;

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	
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.)			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.
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.			



SQLSTATE IN POSTGRES

- <https://www.postgresql.org/docs/current/errcodes-appendix.html>



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 fa match 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



ESEMPI ECCEZIONI IN POSTGRES (1)

- GESTIONE DELL'ERRORE DIVISION_BY_ZERO

```
DO $$  
DECLARE  
    risultato NUMERIC;  
BEGIN  
    risultato := 100 / 0;  
    RAISE NOTICE 'Risultato: %', risultato;  
EXCEPTION  
    WHEN division_by_zero THEN  
        RAISE NOTICE 'Errore: divisione per zero intercettata.';  
END; $$;
```



ESEMPI ECCEZIONI IN POSTGRES (2)

- GESTIONE DELL'ERRORE NO_DATA_FOUND

```
DO $$  
DECLARE  
    nome TEXT;  
BEGIN  
    SELECT fname INTO nome FROM EMPLOYEE  
    WHERE SSN = '000000000';  
  
    RAISE NOTICE 'Nome trovato: %', nome;  
  
EXCEPTION  
    WHEN NO_DATA_FOUND THEN  
        RAISE NOTICE 'Nessun dipendente trovato con quell''SSN.';  
END; $$;
```



ESEMPI ECCEZIONI IN POSTGRES (3)

- GESTIONE DELL'ERRORE UNIQUE_VIOLATION

```
DO $$  
BEGIN  
    INSERT INTO PROJECT (pnumber, pname, plocation, dnum)  
    VALUES (1, 'Duplicato', 'Napoli', 1);
```

EXCEPTION

```
    WHEN uniqueViolation THEN  
        RAISE NOTICE 'Errore: pnumber duplicato nel progetto.';  
END;  
$$;
```



ESEMPI ECCEZIONI IN POSTGRES (4)

- GESTIONE DELL'ERRORE FOREIGN_KEY_VIOLATION

```
DO $$
```

```
BEGIN
```

```
    INSERT INTO WORKS_ON (ESSN, PNO, hours)  
    VALUES ('999999999', 1, 10); -- ESSN non esiste
```

```
EXCEPTION
```

```
    WHEN foreign_keyViolation THEN  
        RAISE NOTICE 'Errore: chiave esterna non valida (ESSN  
non trovato).';  
END; $$;
```



ESEMPI ECCEZIONI IN POSTGRES (5)

- GESTIONE DELL'ERRORE OTHERS

```
DO $$  
DECLARE  
    x INTEGER := 10;  
BEGIN  
    x := x + 'abc';  
  
EXCEPTION  
    WHEN OTHERS THEN  
        RAISE NOTICE 'Errore generico intercettato: %', SQLERRM;  
END;  
$$;
```



ESEMPI ECCEZIONI IN POSTGRES (6)

- GESTIONE DELL'ERRORE UNDEFINED_TABLE

```
DO $$  
BEGIN  
    EXECUTE 'SELECT * FROM empolye';  
  
EXCEPTION  
    WHEN undefined_table THEN  
        RAISE NOTICE 'Errore: la tabella non esiste.';  
END;  
$$;
```



ESEMPI ECCEZIONI IN POSTGRES (7)

- GESTIONE DELL'ERRORE NUMERIC_VALUE_OUT_OF_RANGE

```
DO $$  
DECLARE  
    numero NUMERIC := 1e1000; -- numero troppo grande  
BEGIN  
    INSERT INTO EMPLOYEE (SSN, fname, minit, lname, bdate, address, sex, salary,  
    superSSN, dno)  
        VALUES ('999999999', 'Max', 'Z', 'Overflow', '2000-01-01', 'N/A', 'M', numero,  
    NULL, 1);  
  
EXCEPTION  
    WHEN numeric_value_out_of_range THEN  
        RAISE NOTICE 'Errore: valore numerico troppo grande per il campo salary.';  
END; $$;
```



ESEMPI ECCEZIONI IN POSTGRES (8)

- GESTIONE DELL'ERRORE TOO_MANY_ROWS

```
DO $$  
DECLARE  
    nome TEXT;  
BEGIN  
    SELECT fname INTO nome FROM EMPLOYEE WHERE fname = 'John';  
  
    RAISE NOTICE 'Nome: %', nome;  
  
EXCEPTION  
    WHEN TOO_MANY_ROWS THEN  
        RAISE NOTICE 'Errore: troppi risultati in SELECT INTO.';  
END; $$;
```



ESEMPI ECCEZIONI IN POSTGRES (9)

- GESTIONE DELL'ERRORE UNDEFINED_COLUMNS

```
DO $$
```

```
BEGIN
```

```
    EXECUTE 'SELECT colore_capelli FROM EMPLOYEE';
```

```
EXCEPTION
```

```
    WHEN undefined_column THEN
```

```
        RAISE NOTICE 'Errore: la colonna richiesta non esiste.';
```

```
END;
```

```
$$;
```



ESEMPI ECCEZIONI IN POSTGRES (10)

- USO DI GET STACKED DIAGNOSTICS

```
DO $$  
DECLARE  
    err_msg TEXT;  
BEGIN  
    PERFORM 1 / 0;  
  
EXCEPTION  
    WHEN OTHERS THEN  
        GET STACKED DIAGNOSTICS err_msg = MESSAGE_TEXT;  
        RAISE NOTICE 'Errore intercettato: %', err_msg;  
END;  
$$;
```



PROCEDURE

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
    procedure_body
END procedure_name;
```

- ***IN*** implica che il parametro può essere utilizzato come INPUT nella procedura ma non può essere utilizzato come output (non può essere modificato);
- ***OUT*** implica che ad esso può essere modificato e utilizzato come output, ma non come input;
- ***IN OUT*** implica che il parametro può essere utilizzato sia in lettura che scrittura.



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

ESEMPIO IN

```
CREATE OR REPLACE PROCEDURE saluta_dipendente(IN nome TEXT)
LANGUAGE plpgsql
AS $$

BEGIN

    RAISE NOTICE 'Ciao, %!', nome;

END; $$;
```

Esecuzione: CALL saluta_dipendente('Mario');



ESEMPIO OUT

```
CREATE OR REPLACE PROCEDURE ottieni_salario_medio(OUT media NUMERIC)
LANGUAGE plpgsql
AS $$

BEGIN
    SELECT AVG(salary) INTO media FROM employee;
END;
$$;
```

Esecuzione:

```
CALL ottieni_salario_medio(media := NULL);
```

serve specificare OUT per pgAdmin o psql



ESEMPIO INOUT

```
CREATE OR REPLACE PROCEDURE raddoppia(INOUT valore INTEGER)
LANGUAGE plpgsql
AS $$

BEGIN
    valore := valore * 2;
END;

$$;
```

Esecuzione: CALL raddoppia(valore := 10);



ESEMPIO PROCEDURE (1)

- AUMENTA LO STIPENDIO DI UN DIPENDENTE

```
CREATE OR REPLACE PROCEDURE aumenta_stipendio(
    IN p_ssn TEXT, IN p_incremento NUMERIC )
LANGUAGE plpgsql
AS $$

BEGIN

    UPDATE employee
        SET salary = salary + p_incremento
        WHERE ssn = p_ssn;

    IF NOT FOUND THEN
        RAISE NOTICE 'Dipendente con SSN % non trovato.', p_ssn;
    END IF;

END; $$;
```



ESEMPIO PROCEDURE (2)

- RESTITUISCE IL NUMERO DI DIPENDENTI DI UN DIPARTIMENTO

```
CREATE OR REPLACE PROCEDURE conta_dipendenti(
    IN p_dno INTEGER, OUT tot INTEGER )
LANGUAGE plpgsql
AS $$

BEGIN

    SELECT COUNT(*) INTO tot
    FROM employee
    WHERE dno = p_dno;

END;
$$;
```



ESEMPIO PROCEDURE (3)

- RADDOPPIA LE ORE LAVORATE SU UN PROGETTO

```
CREATE OR REPLACE PROCEDURE conta_dipendenti(
    IN p_dno INTEGER, OUT tot INTEGER )
LANGUAGE plpgsql
AS $$

BEGIN

    SELECT COUNT(*) INTO tot
    FROM employee
    WHERE dno = p_dno;

END;
$$;
```



ESEMPIO PROCEDURE (4)

- AGGIUNGE UN DIPENDENTE SOLO SE NON ESISTE GIÀ'

```
CREATE OR REPLACE PROCEDURE inserisci_dipendente_se_nuovo(
    IN p_ssn TEXT, IN p_nome TEXT,
    IN p_cognome TEXT, IN p_salary NUMERIC, IN p_dno INTEGER )
LANGUAGE plpgsql
AS $$

BEGIN

    IF NOT EXISTS (SELECT 1 FROM employee WHERE ssn = p_ssn) THEN
        INSERT INTO employee (ssn, fname, lname, salary, dno)
        VALUES (p_ssn, p_nome, p_cognome, p_salary, p_dno);
    ELSE
        RAISE NOTICE 'Dipendente con SSN % esiste già.', p_ssn;
    END IF;

END; $$;
```



ESEMPIO PROCEDURE (5)

- ELIMINA LE PERSONE A CARICO DI UN IMPIEGATO

```
CREATE OR REPLACE PROCEDURE rimuovi_dependents (
    IN p_ssn TEXT
)
LANGUAGE plpgsql
AS $$

BEGIN

    DELETE FROM dependent
    WHERE essn = p_ssn;

    RAISE NOTICE 'Dependent(i) rimossi per SSN: %', p_ssn;

END; $$;
```



ESEMPIO PROCEDURE (6)

- CALCOLA IL TOTALE DI ORE LAVORATE DA UN IMPIEGATO

```
CREATE OR REPLACE PROCEDURE totale_ore_lavorate(
    IN p_ssn TEXT,
    OUT tot_ore NUMERIC
)
LANGUAGE plpgsql
AS $$

BEGIN
    SELECT COALESCE(SUM(hours), 0) INTO tot_ore
    FROM works_on
    WHERE essn = p_ssn;

END;
$$;
```



ESEMPIO PROCEDURE (7)

- CAMBIA IL RESPONSABILE DI UN DIPARTIMENTO

```
CREATE OR REPLACE PROCEDURE cambia_manager(
    IN p_dnumber INTEGER, IN p_nuovo_mgrssn TEXT, IN p_data_inizio DATE
)
LANGUAGE plpgsql
AS $$

BEGIN

    UPDATE department
    SET mgrssn = p_nuovo_mgrssn, mgrstartdate = p_data_inizio
    WHERE dnumber = p_dnumber;

    IF NOT FOUND THEN
        RAISE NOTICE 'Dipartimento % non trovato.', p_dnumber;
    END IF;

END; $$;
```



PROCEDURE E TRANSAZIONI

- Le **procedure** e le **transazioni** rappresentano due strumenti fondamentali ma distinti per la gestione dell'integrità e della logica applicativa.
- In PostgreSQL, la loro **coesistenza richiede attenzione**, poiché il sistema adotta una gestione rigorosa e centralizzata delle transazioni.
- Le **procedure PL/pgSQL** consentono di eseguire sequenze complesse di operazioni SQL, con supporto per il controllo di flusso, gestione degli errori ed elaborazione condizionale.



ESEMPIO: AGGIORNA_DIPENDENTE

```
CREATE OR REPLACE PROCEDURE aggiorna_dipendente( IN p_ssn TEXT, IN p_incremento NUMERIC, IN p_nuovo_indirizzo TEXT)
LANGUAGE plpgsql AS $$

BEGIN

    UPDATE employee
    SET salary = salary + p_incremento
    WHERE ssn = p_ssn;

p_nuovo_indirizzo non può essere NULL

    IF p_nuovo_indirizzo IS NULL THEN
        RAISE EXCEPTION 'Indirizzo non valido!';
    END IF;

    UPDATE employee
    SET address = p_nuovo_indirizzo
    WHERE ssn = p_ssn;

END; $$;
```



CASO 1: ESECUZIONE SENZA TRANSAZIONE ESPLICITA

```
CALL aggiorna_dipendente('123456789', 1000, NULL);
```

- Lo **stipendio viene aumentato**.
- Errore per l'indirizzo = **NULL**.
- Ma lo stipendio **rimane aumentato** → operazione **parzialmente riuscita**.



CASO 2: ESECUZIONE IN TRANSAZIONE ESPLICITA

```
BEGIN;  
    CALL aggiorna_dipendente('123456789', 1000, NULL);
```

```
COMMIT;
```

- Alza eccezione → transazione interrotta.
- **Tutte le modifiche annullate** → stipendio resta invariato.

```
BEGIN;  
    CALL aggiorna_dipendente(123456789', 1000, 'Via Roma 10');
```

```
COMMIT;
```

- Tutto va a buon fine, e viene **committato insieme**.



COMMIT E ROLLBACK NELLA PROCEDURA?

- **PostgreSQL non consente l'uso esplicito dei comandi BEGIN, COMMIT o ROLLBACK all'interno di una procedura (o funzione)**
 - Il controllo transazionale è riservato al **blocco esterno** o al **client** che invoca la procedura.
- Nonostante ciò, è possibile **simulare comportamenti transazionali locali** attraverso blocchi annidati con BEGIN ... EXCEPTION, che consentono di isolare errori e controllare il flusso senza interrompere l'intera operazione.
- Questa distinzione evidenzia l'importanza di **progettare le procedure in modo che siano transazionalmente sicure**, mantenendo al contempo la coerenza del database quando vengono integrate in transazioni più ampie.



STRING FUNCTIONS

ASCII	ASCII('A')	65	Returns an ASCII code value of a character.
CHR	CHR('65')	'A'	Converts a numeric value to its corresponding ASCII character.

CONCAT	CONCAT('A','BC')	'ABC'	Concatenate two strings and return the combined string.
--------	------------------	-------	---

CONVERT	CONVERT('Ä È Í', 'US7ASCII', 'WE8ISO8859P1')	'A E I'	Convert a character string from one character set to another.
---------	--	---------	---

DUMP	DUMP('A')	Typ=96 Len=1: 65	Return a string value (VARCHAR2) that includes the datatype code, length measured in bytes, and internal representation of a specified expression.
------	-----------	------------------------	--

INITCAP	INITCAP('hi there')	'Hi There'	Converts the first character in each word in a specified string to uppercase and the rest to lowercase.
---------	---------------------	------------	---

INSTR	INSTR('This is a playlist', 'is')	3	Search for a substring and return the location of the substring in a string
-------	-------------------------------------	---	---

LENGTH	LENGTH('ABC')	3	Return the number of characters (or length) of a specified string
--------	---------------	---	---

LOWER	LOWER('Abc')	'abc'	Return a string with all characters converted to lowercase.
-------	--------------	-------	---

LPAD	LPAD('ABC';5,'*')	***ABC'	Return a string that is left-padded with the specified characters to a certain length.
------	-------------------	---------	--

LTRIM	LTRIM(' ABC ')	'ABC '	Remove spaces or other specified characters in a set from the left end of a string.
-------	----------------	--------	---

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.
--------------	----------------------------------	---	---

REGEXP_LIKE	REGEXP_LIKE('Year of 2017','\d+')	true	Match a string based on a regular expression pattern.
REGEXP_REPLACE	REGEXP_REPLACE('Year of 2017','\d+', 'Dragon')	'Year of Dragon'	Replace substring in a string by a new substring using a regular expression.
REGEXP_SUBSTR	REGEXP_SUBSTR('Number 10', '\d+')	10	Extract substrings from a string using a pattern of a regular expression.
REPLACE	REPLACE('JACK AND JOND','J','BL');	'BLACK AND BLOND'	Replace all occurrences of a substring by another substring in a string.
RPAD	RPAD('ABC';5,'*')	'ABC**'	Return a string that is right-padded with the specified characters to a certain length.
RTRIM	RTRIM(' ABC ')	' ABC '	Remove all spaces or specified character in a set from the right end of a string.
SOUNDEX	SOUNDEX('sea')	'S000'	Return a phonetic representation of a specified string.
SUBSTR	SUBSTR('Oracle Substring', 1, 6)	'Oracle'	Extract a substring from a string.
TRANSLATE	TRANSLATE('12345', '143', 'bx')	'b2x5'	Replace all occurrences of characters by other characters in a string.
TRIM	TRIM(' ABC ')	'ABC'	Remove the space character or other specified characters either from the start or end of a string.
UPPER	UPPER('Abc')	'ABC'	Convert all characters in a specified string to uppercase.



STRING FUNCTIONS (POSTGRESQL)

- <https://www.postgresql.org/docs/9.1/functions-string.html>

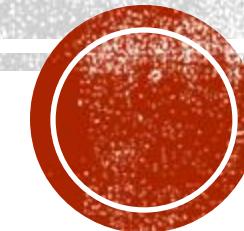


INSTR('THIS IS A PLAYLIST', 'IS') := 3

- S = '32,33,34,35';
- D = Prezzi_prodotti(S);
- recuperare la prima posizione della virgola
 - Pos = INSTR(S, ',')
 - corrente = SUBSTR(S, 1, pos-1)
- Select prezzo into p
FROM prodotti
WHERE id = corrente



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 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;
```



SINTASSI DI BASE

- PREPARE nome_query (tipo_parametri) AS
 UPDATE ... [con \$1, \$2, ...;]
 EXECUTE nome_query(valore1, valore2, ...);
- EXECUTE 'SELECT * FROM EMPLOYEE;';
- EXECUTE 'SELECT * FROM EMPLOYEE WHERE SALARY > \$1' USING valore



COS'È L'SQL DINAMICO

- L'**SQL dinamico** consente di costruire ed eseguire comandi SQL **a runtime**, usando stringhe.
- È utile quando non si conoscono a priori i nomi di tabelle, colonne, o condizioni.
- Esempio:
 1. Query con **nomi di tabelle/colonne variabili**
 2. **DDL** (CREATE, DROP) costruito dinamicamente
 3. Operazioni condizionali su **più entità**
 4. Procedure generiche e riutilizzabili
- **Attenzione: Rischio SQL injection** se non gestito correttamente



1) QUERY CON NOMI DI TABELLE / COLONNE VARIABILI

...

```
BEGIN
```

```
    EXECUTE format('SELECT fname, lname FROM %I WHERE %I > $1', tabella,  
colonna)
```

```
    INTO rec
```

```
    USING soglia;
```

```
    RAISE NOTICE 'Dipendente: % %', rec.fname, rec.lname;
```

```
END; $$;
```

- La tabella e la colonna sono variabili.
- La query viene **costruita dinamicamente**, ma i valori sono passati in modo sicuro con **USING**.



2) DDL DINAMICO ESEMPIO CON DROP TABLE

```
...
BEGIN
IF EXISTS (
    SELECT FROM information_schema.tables
    WHERE table_name = tabella_da_cancellare
) THEN
    EXECUTE format('DROP TABLE IF EXISTS %I CASCADE',
tabella_da_cancellare);
    RAISE NOTICE 'Tabella % cancellata.', tabella_da_cancellare;
ELSE
    RAISE NOTICE 'Tabella % non esiste.', tabella_da_cancellare;
END IF;
END;
$$;
```



3) OPERAZIONI CONDIZIONALI SU PIÙ ENTITÀ (ES. TABELLE PARTIZIONATE)

...

DECLARE

tabella_corrente TEXT;

totale NUMERIC;

BEGIN

tabella_corrente := tabella_base || anno;

EXECUTE format('SELECT SUM(importo) FROM %I WHERE status = %L',
tabella_corrente, 'confermato')

INTO totale;

RAISE NOTICE 'Totale %: %', tabella_corrente, COALESCE(totale, 0);

END;



4) PROCEDURA GENERICA E RIUTILIZZABILE

```
CREATE OR REPLACE PROCEDURE aggiorna_colonna(tab TEXT, col TEXT,  
nuovo_valore TEXT, condizione TEXT)  
LANGUAGE plpgsql  
AS $$  
BEGIN  
    EXECUTE format('UPDATE %I SET %I = %L WHERE ' || condizione, tab, col,  
nuovo_valore);  
    RAISE NOTICE 'Aggiornamento su %: % impostato a %', tab, col,  
nuovo_valore;  
END; $$;  
  
-- Esempio di chiamata:  
CALL aggiorna_colonna('employee', 'address', 'Via Roma 100', 'ssn ='  
'123456789');
```



USO DI FORMAT

- L'uso di `format` permette di far fronte a problemi di SQL Injection, query errate o errori non gestiti.
- Inoltre permette di gestire la creazione della query in una maniera più semplice



SENZA FORMAT()

- Un'errata concatenazione non permette di eseguire il comando in maniera corretta

```
EXECUTE 'SELECT fname FROM ' || tabella || ' WHERE salary > 50000';
```

Se tabella **contiene un nome formattato male, il comando da errori**

- EXECUTE 'SELECT fname FROM ' || tabella;

Cosa accade se tabella = 'employee; DROP TABLE employee;' ?

- Con format

```
EXECUTE format('SELECT * FROM %I WHERE %I > %L', tabella, colonna, 50000);
```

- Senza format

```
EXECUTE 'SELECT * FROM "' || tabella || '"' WHERE '"' || colonna || '"' > quote_literal(50000);
```

PLACEHOLDER (1)

- Format utilizza 3 differenti placeholder per l'inserimento di valori nella query

Segnaposto	Significato
%I	Identifier → nomi di tabelle, colonne
%L	Literal → valori (stringhe, numeri)
%S	Stringa raw (senza quote o escaping)



PLACEHOLDER (2)

```
SELECT format('SELECT %I FROM %I', 'nome', 'studenti');
```

```
SELECT format('INSERT INTO log VALUES (%L)', 'Errore critico');
```

```
SELECT format('SELECT * FROM %S', 'studenti');
```

Nell'ultimo caso, ci può essere un pericolo di SQLInjection



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
 - in un parametro di tipo stringa che contiene il comando
 - in un comando specificato direttamente come parametro della EXECUTE IMMEDIATE



MODALITA' 1: PREPARE & EXECUTE

- Usata per eseguire una **query dinamica con parametri** più volte in modo efficiente.
- La query viene **compilata una volta sola** e poi eseguita con parametri diversi.
- Utilizza le istruzioni SQL standard: PREPARE, EXECUTE, DEALLOCATE .
- Il comando PREPARE analizza l'istruzione SQL e ne prepara una traduzione
PREPARE <nome comando> FROM <comando SQL> --ORACLE
PREPARE <nome comando> AS<comando SQL> --POSTGRES
- Dove **<nome comando>** è il nome associato da **PREPARE** alla traduzione del comando SQL



MODALITA' 1: PREPARE & EXECUTE

- Il comando SQL può contenere dei parametri in ingresso rappresentati da ?

```
PREPARE comandoSQL  FROM
```

```
'SELECT nome FROM studente WHERE matricola = ?'
```

oppure

```
PREPARE comandoSQL  AS
```

```
'SELECT nome FROM studente WHERE matricola = $1'
```

```
EXECUTE comandoSQL('N86001941');
```



ESEMPIO

```
PREPARE aggiorna_stipendio (numeric, text) AS
    UPDATE employee SET salary = $1 WHERE lname = $2;

EXECUTE aggiorna_stipendio(60000, 'Rossi');
EXECUTE aggiorna_stipendio(70000, 'Verdi');

DEALLOCATE aggiorna_stipendio;
```



MODALITA' 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.
- Se il comando lo costruiamo come una stringa, 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

ESEMPIO

comandoSQL := 'SELECT fname FROM employee WHERE ssn = \$1';
EXECUTE comandoSQL INTO nomeStudente USING matr_studente;



MODALITA' 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>]
```

ESEMPIO

```
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;
```



ESEMPIO IN BLOCCO ANONIMO

```
DO $$  
DECLARE  
    tabella TEXT := 'employee';  
    colonna TEXT := 'salary';  
    soglia NUMERIC := 50000;  
    risultato RECORD;  
  
BEGIN  
    EXECUTE format('SELECT fname, lname FROM %I WHERE %I > $1', tabella, colonna)  
    INTO risultato  
    USING soglia;  
  
    RAISE NOTICE 'Nome: % %', risultato.fname, risultato.lname;  
END;  
$$;
```



USO DI FORMAT

- L'uso di `format` permette di far fronte a problemi di SQL Injection, query errate o errori non gestiti.
- Inoltre permette di gestire la creazione della query in una maniera più semplice



SENZA FORMAT()

- Un'errata concatenazione non permette di eseguire il comando in maniera corretta

```
EXECUTE 'SELECT fname FROM ' || tabella || ' WHERE salary > 50000';
```

Se tabella **contiene un nome formattato male, il comando da errori**

- EXECUTE 'SELECT fname FROM ' || tabella;

Cosa accade se tabella = 'employee; DROP TABLE employee;' ?

- Con format

```
EXECUTE format('SELECT * FROM %I WHERE %I > %L', tabella, colonna, 50000);
```

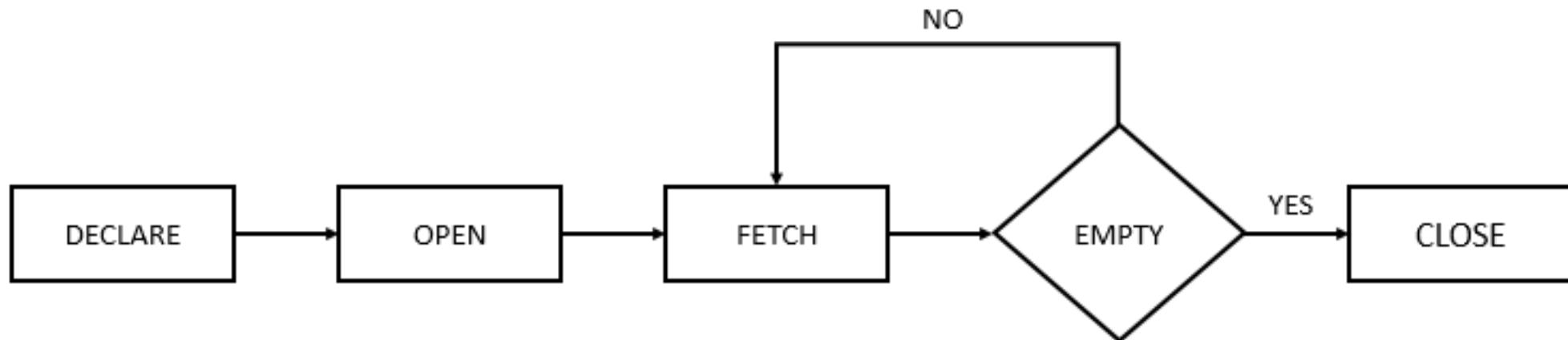
- Senza format

```
EXECUTE 'SELECT * FROM "' || tabella || '"' WHERE '"' || colonna || '"' > ' || quote_literal(50000);
```

CLASSICO CURSOR

Quando il comando SQL è esplicito

```
<nome_cursore> CURSOR FOR <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;
/
```

REFCURSOR IN POSTGRES

```
CREATE OR REPLACE PROCEDURE get_high_salary_employees(
    soglia NUMERIC,
    OUT out_cursor REFCURSOR
)
LANGUAGE plpgsql
AS $$

BEGIN
    OPEN out_cursor FOR
        SELECT ssn, fname, lname, salary
        FROM employee
        WHERE salary > soglia;
END;
$$;

CALL get_high_salary_employees(50000, 'mio_cursore');
```



REFCURSOR IN POSTGRES

```
CREATE OR REPLACE PROCEDURE get_high_salary_employees(
    soglia NUMERIC,
    OUT out_cursor REFCURSOR
)
LANGUAGE plpgsql
AS $$

DECLARE
    comandoSQL TEXT;

BEGIN
    comandoSQL := 'SELECT ssn, fname, lname, salary FROM employee WHERE salary > $1';
    OPEN out_cursor FOR EXECUTE comandoSQL USING soglia;
END;
$$;

CALL get_high_salary_employees(50000, 'mio_cursore');
```



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 (ORACLE)

```
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;
/
```



FUNCTIONS IN POSTGRES

```
CREATE FUNCTION get_salary(emp_id INT)
RETURNS NUMERIC
LANGUAGE plpgsql
AS $$

BEGIN
    RETURN (SELECT salary FROM employee WHERE ssn = emp_id);
END;

$$;

SELECT get_salary(123456789); oppure PERFORM get_salary(123456789);
```



ATTENZIONE

```
DO $$  
DECLARE  
    risultato NUMERIC;  
BEGIN  
    SELECT get_salary(123456789) INTO risultato;  
  
    RAISE NOTICE 'Lo stipendio è: %', risultato;  
END;  
$$;
```



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).



ESEMPIO PRATICO

- **Oracle**

```
PreparedStatement ps = conn.prepareStatement("SELECT get_salary(?)  
FROM dual");
```

```
ps.setInt(1, 123);
```

```
ResultSet rs = ps.executeQuery();
```

- **Postgres**

```
PreparedStatement ps = conn.prepareStatement("SELECT get_salary(?)");
```

```
ps.setInt(1, 123);
```

```
ResultSet rs = ps.executeQuery();
```



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));
    }
}
```



RIFERIMENTI IMPORTANTI

- **Postgres**

- <https://www.postgresql.org/docs/15/plpgsql.html>
- Ed in particolare sulle differenze fra PL/SQL e PL/pgSQL:
 - <https://www.postgresql.org/docs/15/plpgsql-porting.html>

- **Oracle**

- <https://docs.oracle.com/en/database/oracle/oracle-database/21/lnpls/overview.html>
- Ed in particolare una serie di lezioni scritte molto bene da un esperto:
 - <https://blogs.oracle.com/connect/category/omz-pl-sql-101>



ESERCIZIO 1

- ALBERO (CodA, root)
NODO (CodA, CodN, label)
ARCO (CodA, CodArco, padre, figlio)
- Scrivere una funzione somma_nodi che riceve in input un codice di albero ed un codice nodo e restituisce la somma delle label di tutti i nodi che vanno dal nodo input fino alla radice.

```
CREATE OR REPLACE FUNCTION somma_nodi(albero albero.coda%TYPE, s_nodo nodo.codn%type)
RETURN NUMERIC
LANGUAGE plpgsql
AS $$

DECLARE...

BEGIN...

END; $$;
```



COSTRUZIONE QUERY (1)

SOMMA_NODI (ALBERO ALBERO.CODA%TYPE, S_NODO NODO.CODN%TYPE)

ALBERO (CodA, root)

NODO (CodA, CodN, label)

ARCO (CodA, CodArco, padre, figlio)

- Dobbiamo costruire la query per recuperare la label del nodo corrente.

```
SELECT label
```

```
FROM nodo
```

```
WHERE CodA = <albero in input> AND CodN = <nodo in input>
```



COSTRUZIONE QUERY (2)

SOMMA_NODI (ALBERO ALBERO.CODA%TYPE, S_NODO NODO.CODN%TYPE)

ALBERO (CodA, root)

NODO (CodA, CodN, label)

ARCO (CodA, CodArco, padre, figlio)

- Dobbiamo passare dal nodo corrente al nodo padre

```
SELECT padre
```

```
FROM ARCO
```

```
WHERE CodA = <albero in input> AND figlio = <nodo in input>
```



ESERCIZIO 2

- ALBERO (CodA, root)
NODO (CodA, CodN, label)
ARCO (CodA, CodArco, padre, figlio)
- Usando i cursori e la funzione *somma_nodi* sviluppata nell'esercizio 1, scrivere una funzione *somma_max* che, dato un albero, trovi il valore del cammino di somma massima dalle foglie alle radici.

```
CREATE or REPLACE FUNCTION somma_max (albero albero.coda%TYPE)
RETURN NUMERIC
LANGUAGE plpgsql
AS $$

DECLARE . . .

BEGIN...

END; $$;
```

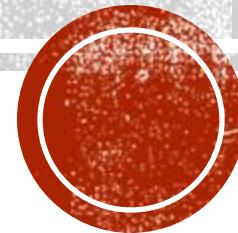




FINE

Per eventuali domande: (in ordine di preferenza personale)

- Ora.
- Chat di Teams
- Mail: silvio.barra@unina.it



BASI DI DATI I

- Trigger Function



TRIGGER FUNCTION

- Una *trigger function* è una funzione speciale eseguita **automaticamente** quando si verifica un determinato evento (INSERT, UPDATE, DELETE) su una tabella.
- È associata a una **trigger** tramite CREATE TRIGGER.
- Può essere eseguita **prima** o **dopo** l'evento.
- Può accedere a **valori vecchi (OLD)** e **nuovi (NEW)**.



SINTASSI BASE

```
CREATE FUNCTION nome_funzione()
RETURNS trigger AS $$

BEGIN
    -- logica
    RETURN NEW; // o RETURN OLD;

END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER nome_trigger
[BEFORE | AFTER] [INSERT | UPDATE | DELETE]
ON nome_tabella
FOR EACH ROW
EXECUTE FUNCTION nome_funzione();
```



ESEMPIO

- BLOCCARE L'INSERIMENTO DI NODI ORFANI

```
CREATE OR REPLACE FUNCTION check_collegamento_nodo()
RETURNS trigger AS $$

DECLARE
    esiste BOOLEAN;
BEGIN
    SELECT EXISTS (
        SELECT 1 FROM arco
        WHERE coda = NEW.coda AND figlio = NEW.codn
    ) INTO esiste;

    IF NOT esiste THEN
        RAISE EXCEPTION 'Nodo % non collegato ad alcun arco', NEW.codn;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

- Impedire che venga inserito un nodo in una tabella NODO se non è collegato tramite un ARCO (cioè nessun arco in cui sia figlio).

```
CREATE TRIGGER trg_check_collegamento
AFTER INSERT ON nodo
FOR EACH ROW
EXECUTE FUNCTION check_collegamento_nodo();
```



ESEMPIO

-AGGIORNARE IN AUTOMATICO IL LABEL DEI NODI FOGLIA

```
CREATE OR REPLACE FUNCTION aggiorna_nodi_foglia()
RETURNS trigger AS $$

DECLARE
    count_figli INTEGER;
BEGIN
    -- Verifica se il padre ha ancora figli
    SELECT COUNT(*) INTO count_figli
    FROM arco
    WHERE coda = OLD.coda AND padre = OLD.padre;

    IF count_figli = 0 THEN
        UPDATE nodo
        SET label = -1
        WHERE coda = OLD.coda AND codn = OLD.padre;
    END IF;

    RETURN OLD;
END;
$$ LANGUAGE plpgsql;
```

- Ogni volta che si elimina un arco, se un nodo rimane senza figli, aggiornare il suo label

```
CREATE TRIGGER trg_aggiorna_foglia
AFTER DELETE ON arco
FOR EACH ROW
EXECUTE FUNCTION aggiorna_nodi_foglia();
```



RETURN NEW

- Si utilizza in:
 - Trigger **BEFORE INSERT** → per **modificare o validare** i dati prima che vengano inseriti.
 - Trigger **BEFORE UPDATE** → per **modificare il nuovo valore** prima della scrittura.

RETURN OLD

- Si utilizza in:
 - Trigger **BEFORE DELETE** o **AFTER DELETE** → per riferimento alla **riga eliminata**
 - Quando **non vuoi che la riga venga modificata**



ESEMPI

**Trigger BEFORE INSERT che forza
il label in maiuscolo**

```
CREATE FUNCTION forza_label_maiuscolo()
RETURNS trigger AS $$  
BEGIN
    NEW.label := UPPER(NEW.label);
    RETURN NEW;
END;  
$$ LANGUAGE plpgsql;
```

Trigger AFTER DELETE che logga un nodo

```
CREATE FUNCTION log_nodo_eliminato()
RETURNS trigger AS $$  
BEGIN
    INSERT INTO log_nodi(codn, messaggio)
    VALUES (OLD.codn, 'Nodo eliminato');
    RETURN OLD;
END;  
$$ LANGUAGE plpgsql;
```



ESERCIZI

ALBERO (CodA, root)
NODO (CodA, CodN, label)
ARCO (CodA, CodArco, padre, figlio)

1. Scrivere un trigger che evita di creare cicli all'interno di un albero

- Ipotizzare di avere una funzione `CONTAINS(elemento, array)` che prende in input un elemento ed un array e restituisce true se l'array contiene l'elemento
- Un array si dichiara
 - `nome_array TEXT[] := ARRAY[elemento]`
 - Inizializza un array di elementi TEXT e che contiene elemento
 - `nome_array := array_append(nome_array, elemento_nuovo)`

1. Scrivere un trigger che imposta come label di un nodo il numero dei figli

2. Scrivere un trigger (con l'SQL dinamico) che all'inserimento di un arco verifica l'esistenza dei nodi che lo compongono.





FINE

Per eventuali domande: (in ordine di preferenza personale)

- Ora.
- Chat di Teams
- Mail: silvio.barra@unina.it

