



UNIVERSITÀ DEGLI STUDI
DI CAGLIARI

CORSO DI LAUREA IN INFORMATICA

Gianmarco Cherchi – g.cherchi@unica.it

Fondamenti di Programmazione Web

Memorizzare e consultare i dati



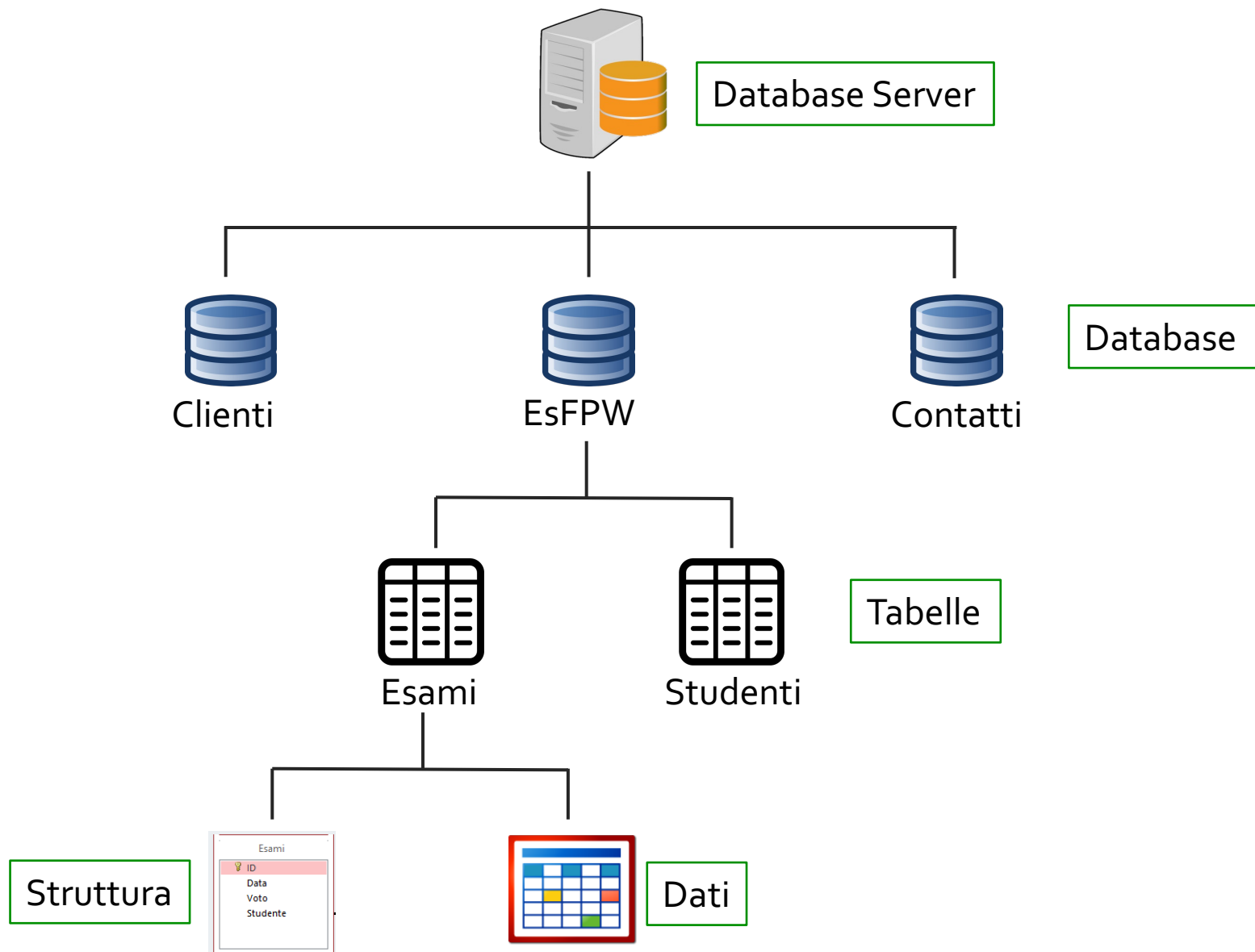
Introduzione ai Database

- **Database:** una collezione strutturata di dati, organizzati in maniera tale da poter essere ricercati efficientemente
- Esistono diversi database engine
 - noi utilizzeremo a lezione MySQL (uno dei più diffusi in ambiente web)
 - mentre in laboratorio PostgreSQL
 - entrambi Open Source
 - ne esistono diversi altri (Oracle, MS SQL Server, ecc)
- I dati contenuti in un database vengono acceduti e modificati tramite un linguaggio standard (SQL)
 - noi vedremo le query principali

Il mondo delle tabelle

- I dati di un database sono organizzati utilizzando delle **tabelle**
- Un **database server** contiene un insieme di database, ognuno dei quali è identificato da un nome
- Ogni **database** è costituito da un insieme di tabelle
- Ogni **tabella** ha una sua **struttura**
 - un insieme di colonne
 - ogni colonna ha un nome (campo)
 - ogni colonna rappresenta un dato con un determinato **tipo** (intero, stringa, float, data, ecc)
- Ogni **tabella** contiene un insieme di dati
 - una collezione di righe
 - ogni riga contiene una serie di campi, associati alle varie colonne della tabella

Struttura di un database server



Creazione delle tabelle

- Per creare delle tabelle in un database dobbiamo specificare
 - un nome per la tabella
 - un nome ed un tipo di dato per ogni colonna
- È molto importante studiare la struttura del database a tavolino, prima di iniziare l'inserimento dei dati
 - in modo da non doverla modificare durante lo sviluppo (magari con dati già presenti)
 - in modo da non dimenticare di mantenere dei dati utili
- Esistono semplici tecniche di design
 - noi accenneremo qualcosa, si vedono in maniera approfondita nel corso di Basi di Dati (3° anno)

Il tabellone (da non fare!)

Presidente	Docente1	Docente2	Insegnamento	Matricola	Studente	Voto
Gianmarco Cherchi	Riccardo Scateni	Marco Livesu	FPW	12345	Marco Attene	24
Gianmarco Cherchi	Riccardo Scateni	Marco Livesu	FPW	54321	Nico Pietroni	30
Gianmarco Cherchi	Riccardo Scateni	Marco Livesu	FPW	23456	Daniela Cabiddu	27
Riccardo Scateni	Caterina Fenu	Cecilia Di Ruberto	PR1	12345	Marco Attene	30

- Una tabella unica per tutti i dati
- Tende ad avere tantissime colonne
- Non ha assolutamente struttura
- I dati sono duplicati
- Ci sono spesso celle vuote
- In poche parole **non si deve usare!**

Passo 1: identificare le entità

- Le entità sono ciò che dovete salvare all'interno del database
- Tecnicamente, tutto ciò che può essere riconosciuto in modo indipendente è un'entità
- Tutto ciò che può essere identificato da un identificatore unico è un'entità
- Le entità possiedono degli attributi che le caratterizzano
 - l'identificatore unico e almeno un altro attributo
- Gli attributi possono avere diversi tipi (intero, stringa, float, data, ecc)
- Ovviamente per la stessa applicazione è possibile identificare diverse entità

Entità: esempio

Studente

- *Id*
- Nome
- Cognome
- Matricola

Docente

- *Id*
- Nome
- Cognome
- Ricevimento

Insegnamento

- *Id*
- Nome
- Codice

Esame

- *Id*
- Voto

Le chiavi primarie

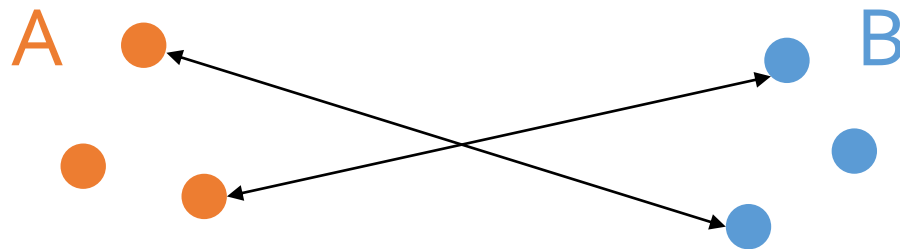
- Una **chiave** è un campo (o un insieme di campi) di una tabella che identifica in modo univoco una riga
- Di solito si utilizza un campo apposito che funziona come chiave, a cui si dà un valore intero progressivo (ID)
- Questo valore può essere incrementato in automatico dal database (esistono dei tipi appositi, li vedremo)
- In alcune tabelle però è necessario utilizzare più di un campo per identificare una riga
- Una **chiave primaria** è una chiave con il numero minimo di campi
- Una tabella può avere più di una chiave, ma ne selezioniamo soltanto una come chiave primaria

Passo 2: identificare le relazioni

- Oltre agli attributi, le entità possono essere collegate ad altre entità
- Quando un'entità è collegata ad un'altra, si dice che tra loro esiste una "**relazione**"
- Una relazione si classifica in base alla loro cardinalità
 - Uno ad Uno
 - Uno a Molti
 - Molti a Molti
- Vediamole nel dettaglio...

Relazione uno ad uno

- Una relazione dove ad un'entità di un tipo ne corrisponde al più una dell'altro

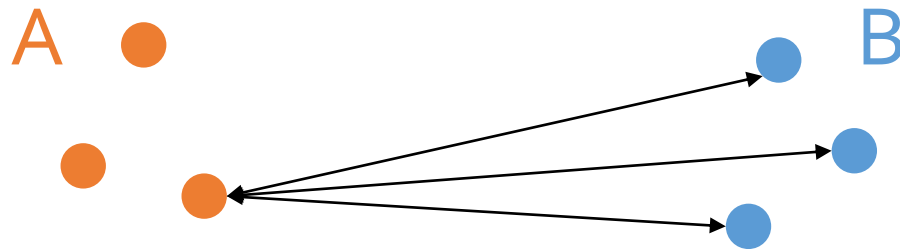


Notazione: 1:1

- Alcune relazioni possono non essere specificate per alcuni elementi
- Per altre invece è necessario che lo siano (esattamente uno)
- Esempi:
 - il conducente di un'auto
 - il coniuge (quello attuale, in monogamia)

Relazione uno a molti

- È una relazione che si ha quando un'entità di un certo tipo è collegata a "molte" entità del secondo tipo

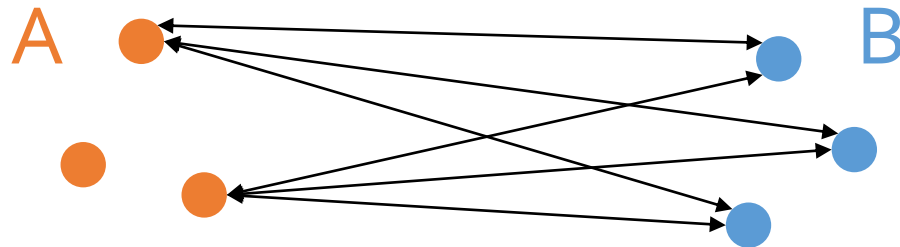


Notazione: 1:N

- Anche in questo caso, alcune relazioni sono specificate per zero, per uno o più valori
- Esempi:
 - editore di un libro
 - acquisto di un oggetto su Amazon

Relazione molti a molti

- È una relazione che collega più entità di un certo tipo a più di un'entità di un altro tipo



Notazione: N:N

- Vuol dire che uno stesso elemento del tipo arancione può essere associato più di un elemento celeste
- E che uno stesso elemento del tipo celeste può essere associato a più di un elemento arancione
- Anche in questo caso si possono avere delle relazioni che per alcuni elementi non sono specificate
- Esempio: autori e canzoni (a Sanremo)

Stabilire il tipo di relazione

- C'è una domanda magica che ci aiuta a stabilire la cardinalità della relazione fra due entità A e B
- **Per ogni A, quanti sono i possibili B?**
- Le risposte possibili sono due: **uno (1)** o **molti (N)**
- Inoltre, bisogna stabilire, in ogni caso, se la relazione debba sempre esistere, oppure se sia opzionale (può essere zero?)
 - **uno**
 - zero o uno ...
 - ... oppure esattamente uno?
 - **molti**
 - zero o più ...
 - ... oppure uno o più?

Stabilire il tipo di relazione

- La risposta alla domanda precedente è la cardinalità della relazione da A a B
- Ma dobbiamo stabilire anche la cardinalità da B a A
- Si opera nello stesso modo rifacciamo la domanda al contrario
- **Per ogni B, quanti sono i possibili A?**
- Da qui otteniamo quattro casi:
 - $(A \rightarrow B = 1) \wedge (B \rightarrow A = 1) \Rightarrow$ relazione **uno a uno**
 - $(A \rightarrow B = N) \wedge (B \rightarrow A = 1) \Rightarrow$ relazione **molti a uno**
 - $(A \rightarrow B = 1) \wedge (B \rightarrow A = N) \Rightarrow$ relazione **uno a molti**
 - $(A \rightarrow B = N) \wedge (B \rightarrow A = N) \Rightarrow$ relazione **molti a molti**

Attributi nelle relazioni

- È possibile che ci sia la necessità di specificare degli attributi anche per le relazioni
- Un esempio tipico è una data o una quantità
 - una data associata alla relazione "matrimonio"
 - una data associata all'ingresso in un team
 - un numero di istanze di un'associazione
- In base al tipo di relazione che vogliamo rappresentare, gli attributi della relazione vengono aggiunti in una delle tabelle che rappresentano le entità o in tabelle create ad-hoc
 - come per le chiavi esterne
 - lo vedremo più avanti...

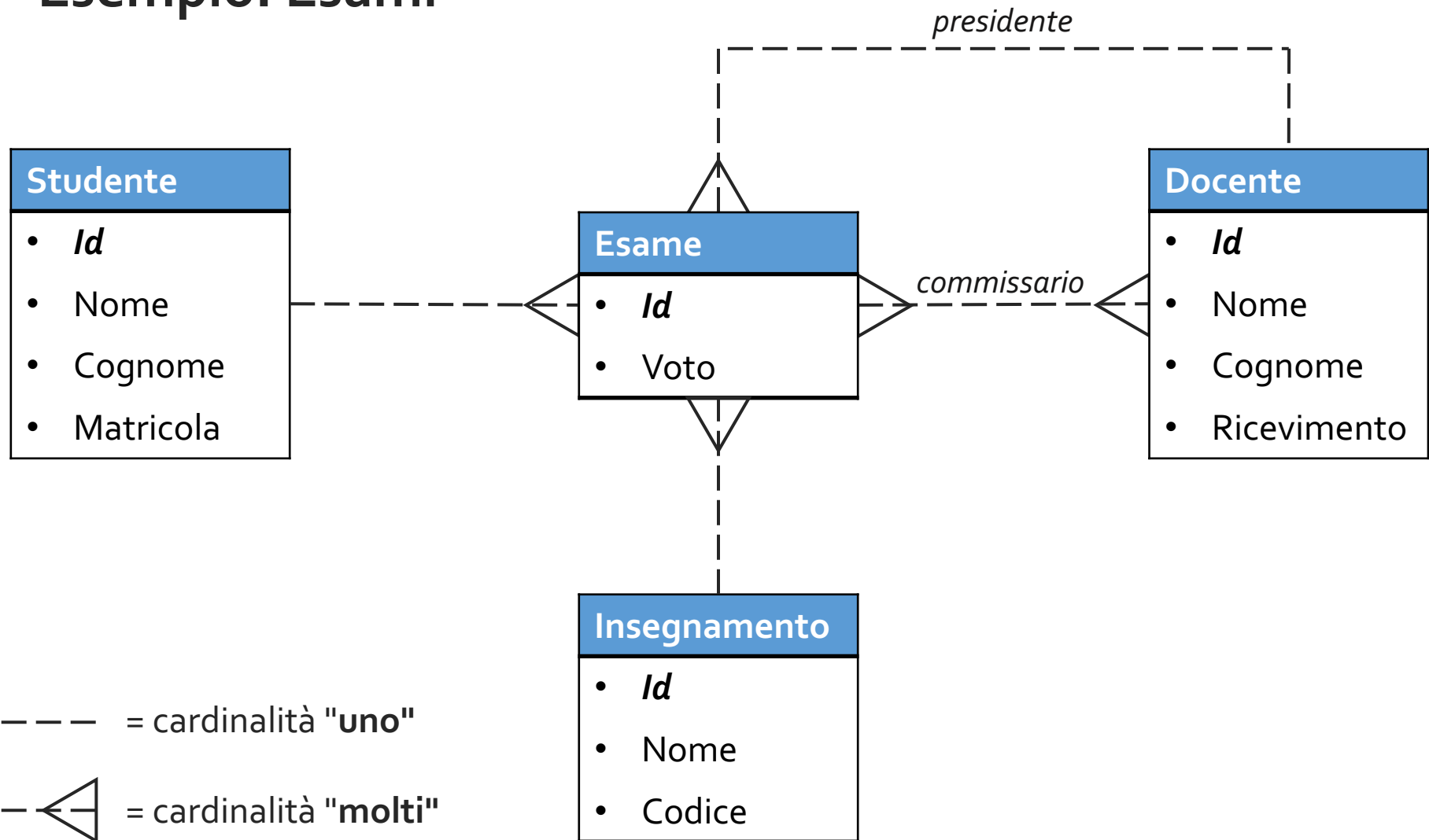
Esempio: il mondo degli esami universitari

- Vediamo un esempio pratico, con entità e relazioni
- Un **esame** ha una **commissione** composta da **un docente presidente** (il titolare del corso) e **due docenti commissari**
- E ovviamente **uno studente** che lo sostiene
- Le entità le abbiamo già viste
- Dobbiamo modellare quattro relazioni:
 - esame e docente presidente
 - esame e docente commissario
 - esame e insegnamento
 - esame e studente
- Poniamoci le domande che abbiamo imparato...

Esempio: Esami

- Docente presidente
 - per ogni esame, quanti docenti presidenti? **Uno**
 - per ogni docente, quanti esami da presidente? **Uno o più di uno**
 - relazione **Uno a Molti (1:N)**
- Docente commissario
 - per ogni esame, quanti docenti commissari? **Due (più di uno)**
 - per ogni docente, quanti esami da commissario? **Uno o più di uno**
 - relazione **Molti a Molti (N:N)**
- Studente
 - per ogni esame (inteso come compito), quanti studenti? **Uno**
 - per ogni studente, quanti esami? **Molti**
 - relazione **Uno a Molti (1:N)**
- In questo problema non abbiamo relazioni uno ad uno

Esempio: Esami




Dalle entità-relazioni alle tabelle

- Abbiamo creato la struttura logica del nostro database
- Ma il database non era costituito da tabelle?
- E le relazioni come le rappresentiamo?
- C'è un modo praticamente automatico per tradurre il nostro modello relazionale in uno fisico (creato da tabelle)

Trasformare le entità in tabelle

- Per ogni entità identificata, si crea una tabella
- La tabella ha una colonna per ogni attributo identificato
- Ogni attributo deve avere il tipo giusto
- Rappresentiamo le colonne come righe per semplicità

Docente			
id	nome	cognome	ricevimento
INT	VARCHAR(50)	VARCHAR(50)	VARCHAR(250)
...



Studente
<ul style="list-style-type: none">• <i>id</i> INT• nome VARCHAR(50)• cognome VARCHAR(50)• matricola INT

Docente
<ul style="list-style-type: none">• <i>id</i> INT• nome VARCHAR(50)• cognome VARCHAR(50)• ricevimento VARCHAR(250)

Insegnamento
<ul style="list-style-type: none">• <i>id</i> INT• nome VARCHAR(50)• codice VARCHAR(7)

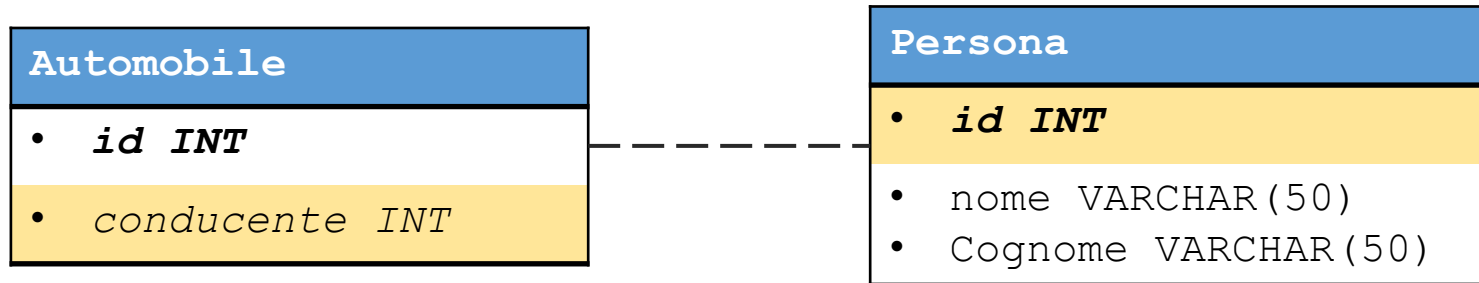
Esame
<ul style="list-style-type: none">• <i>id</i> INT• voto INT

Trasformare le relazioni in tabelle

- Le relazioni si traducono in campi da inserire nelle tabelle ...
- ... oppure in nuove tabelle
- La modalità di traduzione dipende dal tipo di relazione che dobbiamo modellare
- Vediamole una per una

Relazione uno ad uno

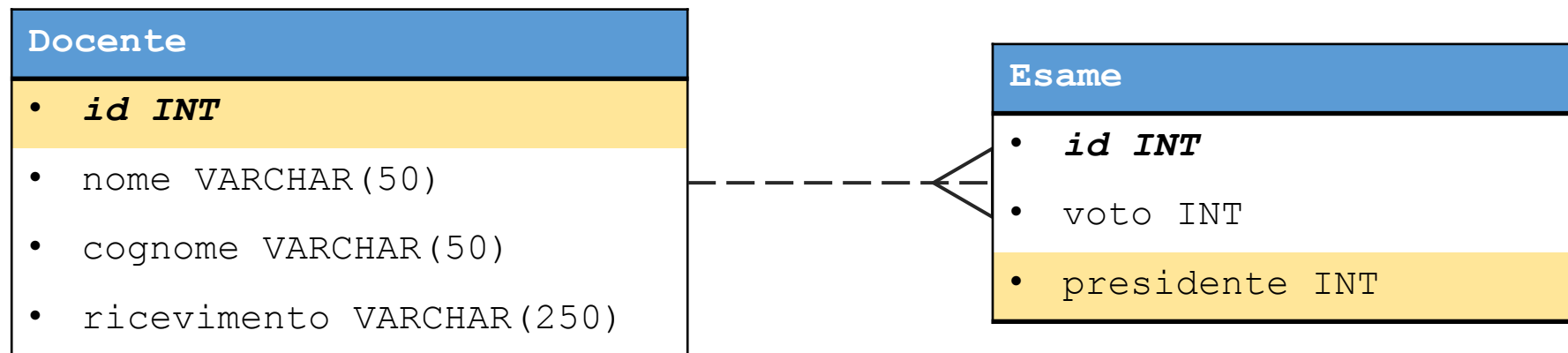
- Si crea una colonna nella tabella di una delle due entità che fa riferimento alla chiave primaria dell'altra entità (vincolo di **chiave esterna**)



- Scegliere la prima o la seconda tabella per ospitare la colonna per la relazione è indifferente
- Di solito si usa come nome della colonna quello della relazione
- Si permette che la colonna possa rimanere vuota nel caso in cui la relazione possa essere non specificata

Relazione uno a molti

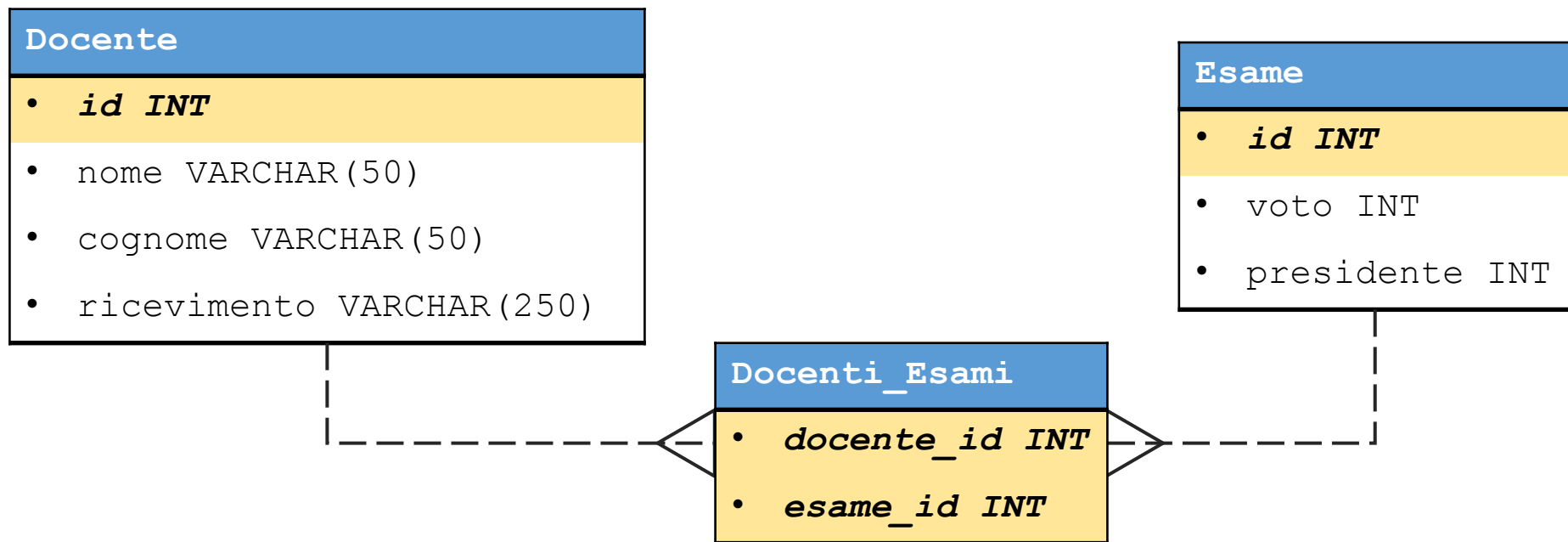
- Si seleziona la tabella che rappresenta l'entità con cardinalità "molti"
- Si aggiunge una colonna che fa riferimento alla tabella dell'entità che ha cardinalità "uno" (vincolo di **chiave esterna**)



- Si permette che la colonna possa rimanere vuota nel caso in cui la relazione possa essere non specificata

Relazione molti a molti

- Si aggiunge una tabella apposita che contiene un riferimento alla prima tabella ed un riferimento alla seconda tabella



- Questa tabella non rappresenta un'entità, serve solo a modellare la relazione
- La tabella contiene una riga per ogni coppia di entità associate tra loro
- La chiave primaria di questa tabella è la coppia di chiavi esterne verso le altre due (**esame_id** e **docente_id** nell'esempio sopra)



Creazione e gestione di un Database

Accesso ad un database

- Il database server mantiene una **lista di utenti** che possono accedere ai vari database in esso contenuti
- Esiste la possibilità di **specificare diversi diritti** di accesso (lettura/scrittura) sui diversi database
- Per questo è necessario specificare le **credenziali** per connettersi
- Sia quando si accede manualmente (es: terminale) che quando si accede via codice (es: Express.js)
- Ci si può connettere ad un database server in tre modi:
 - tramite command line (es: terminale e comandi sql)
 - tramite interfaccia grafica (es: pgAdmin)
 - tramite API di linguaggi di programmazione (es: Javascript)

Inviare comandi ad un database

- Una volta connessi, si possono inviare dei comandi al database
- Si specificano tramite un linguaggio standard chiamato **SQL**
 - **Structured Query Language**
 - viene utilizzato da tutti i database relazionali più diffusi, con eventualmente qualche piccola variante (noterete piccolissime differenze con quello che vedrete in laboratorio)
- Tipi di comando:
 - **Data Definition**: definizione della struttura dei dati (entità e relazioni)
 - **Data Manipulation**: inserimento, cancellazione e modifica dei dati
 - **Query**: interrogazioni sui dati presenti nel database
 - **Control**: controllo sugli accessi ai dati del database
- Spesso con "query" si intende (in maniera inesatta) un qualsiasi comando

CRUD: premessi

- Per ognuno degli elementi del database server (database, tabelle, righe, viste) è utile pensare a chi ha il permesso di eseguire le quattro operazioni possibili:
 - Create: creare un nuovo elemento
 - Read: leggere lo stato corrente di un elemento
 - Update: aggiornare lo stato corrente di un elemento
 - Dele~~t~~e: cancellare lo stato corrente di un elemento
- Queste quattro operazioni sono in gergo riferite con l'acronimo CRUD (composto dalle iniziali delle operazioni)
- Di solito, alcuni utenti hanno tutti i diritti (amministratore del DB), altri possono solo leggere (read), alcuni possono solo creare la struttura (create), ecc

SQL: creazione di un database

- Creazione di un database:

CREATE DATABASE nomedatabase;

- **nomedatabase** è il nome del database
- es: **CREATE DATABASE** esami;

- Creazione di un utente:

CREATE USER 'nome'@'hostname' **IDENTIFIED BY** 'password';

- **nome** è il nome dell'utente
- **hostname** è il nome del server dove viene ospitato il database (può essere localhost)
- **password** è la password di accesso per l'utente
- es: **CREATE USER** 'carlo'@'localhost' **IDENTIFIED BY** 'FinalmenteRe';

- Come tutti i comandi (compresi quelli che vedremo in seguito), può non andare a buon fine per vari motivi

- il database specifica un codice di errore, che possiamo utilizzare per identificarne la causa del problema

Utilizzo di un database

- Assegnare diritti di lettura e scrittura ad un utente

```
GRANT ALL [PRIVILEGES] ON esami.* TO 'carlo'@'localhost';
```

- **ALL** sta per tutti i diritti
 - ne esistono molti come **ALTER**, **CREATE**, **DELETE**, **SELECT**, **INSERT**, e non solo (si possono elencare più diritti separandoli con una virgola)
 - Con ***** indichiamo tutte le tabelle contenute nel database
 - Con **REVOKE** seguito dalla stessa sintassi si revocano i privilegi concessi
-
- Selezionare un database per i comandi successivi:
USE nomedatabase;
- **nomedatabase** è il nome del database selezionato
 - da questo comando in poi tutte le istruzioni successive verranno eseguite sul database identificato da **nomedatabase**

SQL: Tipi di dato

- MySQL (ma non solo) può salvare all'interno delle tabelle diversi tipi di dato:
- Stringhe e testi
- Numeri
- Date
- Formati binari
- In aggiunta, possiede dei tipi per gestire gli identificatori univoci per rappresentare le chiavi delle righe di una tabella
 - **SERIAL**

Dati testuali

- Per i dati testuali si utilizzano principalmente due tipi:
- **CHAR** e **VARCHAR**

Data type	Bytes used	Examples
CHAR(<i>n</i>)	Exactly <i>n</i> (≤ 255)	CHAR(5): "Hello" uses 5 bytes CHAR(57): "New York" uses 57 bytes
VARCHAR(<i>n</i>)	Up to <i>n</i> (≤ 65535)	VARCHAR(100): "Greetings" uses 9 bytes plus 1 byte overhead VARCHAR(7): "Morning" uses 7 bytes plus 1 byte overhead

- Per entrambi si specifica il numero di caratteri che possono contenere
- **CHAR** utilizza sempre tutti i caratteri
 - se specificate una stringa più corta viene riempita con spazi
- **VARCHAR** usa al più i caratteri specificati
 - non riempie la stringa con spazi
- Entrambe troncano la stringa se troppo lunga

Dati numerici

Data type	Bytes used	Minimum value (signed/unsigned)	Maximum value (signed/unsigned)
TINYINT	1	−128	127
		0	255
SMALLINT	2	−32768	32767
		0	65535
MEDIUMINT	3	−8388608	8388607
		0	16777215
INT or INTEGER	4	−2147483648	2147483647
		0	4294967295
BIGINT	8	−9223372036854775808	9223372036854775807
		0	18446744073709551615
FLOAT	4	−3.402823466E+38	3.402823466E+38
		(no unsigned)	(no unsigned)
DOUBLE or REAL	8	−1.7976931348623157E+308	1.7976931348623157E+308
		(no unsigned)	(no unsigned)

- Per i booleani c'è **BOOL**, ma è un alias di **TINYINT**
- **SERIAL** è un alias di **BIGINT UNSIGNED** che viene utilizzato per gli ID univoci (chiavi)

Date

Data type	Time/date format
DATETIME	'0000-00-00 00:00:00'
DATE	'0000-00-00'
TIMESTAMP	'0000-00-00 00:00:00'
TIME	'00:00:00'
YEAR	0000 (Only years 0000 and 1901 - 2155)

- Piccole differenze
 - **TIMESTAMP** arriva massimo al 2038
 - ma può essere utilizzato per includere la data corrente in automatico
 - **DATETIME** può assumere valori più lontani nel tempo

Dati binari

Data type	Bytes used	Attributes
TINYBLOB(<i>n</i>)	Up to <i>n</i> (≤ 255)	Treated as binary data—no character set
BLOB(<i>n</i>)	Up to <i>n</i> (≤ 65535)	Treated as binary data—no character set
MEDIUMBLOB(<i>n</i>)	Up to <i>n</i> (≤ 16777215)	Treated as binary data—no character set
LOB(<i>n</i>)	Up to <i>n</i> (≤ 4294967295)	Treated as binary data—no character set

- È possibile salvare all'interno del database dei file di tipo binario
 - immagini
 - video
 - ...
- Si utilizzano le diverse varianti del tipo **BLOB**
- Non sempre questo tipo di file si salva nel DB

SQL: Manipolazione delle tabelle

- Una volta che abbiamo definito (progettato) la struttura delle tabelle, è possibile crearle tramite la sintassi SQL

- Creazione di una tabella:

```
CREATE TABLE studenti (  
    id SERIAL,  
    nome VARCHAR(128),  
    cognome VARCHAR(128),  
    matricola INT );
```

- Si specifica prima il nome della tabella
- Poi si elencano, fra parentesi tonde, i nomi delle colonne con i rispettivi tipi di dato
- Utilizzare il tipo **SERIAL** imposta in modo automatico anche la chiave primaria della tabella
- Una tabella si cancella con la seguente sintassi
DROP TABLE studenti;

SQL: Manipolazione delle tabelle

- Una volta creata, possiamo visualizzare la struttura di una determinata tabella con il comando

```
SELECT * from studente  
WHERE FALSE;
```

id [PK] integer 	nome character varying (128) 	cognome character varying (128) 	matricola integer 
------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------

SQL: Chiave primaria (se NON serial)

Opzione 1 (chiave primaria formata da un campo):

```
CREATE TABLE esami (  
    id CHAR(10) NOT NULL PRIMARY KEY,  
    voto INT);
```

Opzione 2 (chiave primaria formata da più campi)

```
CREATE TABLE persone (  
    nome VARCHAR(50) NOT NULL,  
    cognome VARCHAR(50) NOT NULL,  
    data DATE  
    CONSTRAINT PRIMARY KEY(nome, cognome) );
```

Opzione 3 (specificare la chiave in una tabella già creata) usando il comando ALTER

```
ALTER TABLE studente ADD PRIMARY KEY (id);
```

SQL: Chiave esterna

Creiamo una tabella che abbia una chiave esterna verso la tabella studente

Opzione 1: aggiunta della chiave esterna durante la creazione della tabella

```
CREATE TABLE esame (  
    id SERIAL PRIMARY KEY,  
    voto INT,  
    studente_id INT,  
    FOREIGN KEY studente_id REFERENCES studente (id)  
    ON UPDATE CASCADE  
    ON DELETE SET NULL);
```

Opzione 2: aggiunta della chiave esterna dopo la creazione della tabella:

```
ALTER TABLE esame ADD CONSTRAINT fk_esame_studente FOREIGN KEY  
(studente_id) REFERENCES studente (id);
```


SQL: Manipolazione delle tabelle

- Nei comandi precedenti, prima creiamo la tabella esami
- Poi aggiungiamo un vincolo di chiave esterna:
 - **studenti_fk** è il nome del vincolo di chiave esterna (mantenuto dal database)
 - **studente_id** è il nome del campo sulla tabella **esami** che contiene il riferimento alla tabella **studenti**
 - **studenti** è il nome della tabella verso cui la chiave esterna punta
 - **id** è il nome del campo della tabella **studenti** che viene puntato da **esami**
- La parte **ON UPDATE** specifica cosa succede nel caso venga modificato il campo **id** di uno studente puntato, o cancellata la riga (come si propaga la modifica):
 - **CASCADE**: la modifica viene effettuata anche nella tabella esami, in cascata
 - **SET NULL**: il campo della tabella esami viene messo a NULL (non specificato)
 - **NO ACTION**: non si fa nulla

SQL: Inserimento dei dati

- Una volta che la struttura dei dati è stata tradotta in un insieme di tabelle, bisogna iniziare a popolarle con dei dati
- Utilizziamo il comando SQL apposito **INSERT**
- Inserimento di una riga
INSERT INTO studenti
(id, indirizzo, cognome, matricola, nome)
VALUES
(**default**, "Via Londra 1", "Middleton", 250573, "Catherine");
- Le colonne possono essere specificate in qualsiasi ordine
- **default** è una parola chiave che inserisce il valore di default per il tipo selezionato
 - zero per gli interi, NULL per le stringhe, ecc.
 - per i **SERIAL**, inserisce il prossimo identificatore progressivo valido

Ricerca di dati (la vera query)

```
SELECT id, nome, cognome, matricola  
FROM studenti  
WHERE nome = "Camilla" AND id = 4;
```

- Restituisce una lista di righe
 - nessuna nel caso non vi siano dati che soddisfano la ricerca
- Mettiamo dopo il **FROM** il nome della tabella su cui si vuole cercare
- Dopo la **SELECT** la lista delle colonne da selezionare
 - si utilizza ***** per specificare tutte le colonne della tabella
- La clausola **WHERE** è un predicato che deve essere soddisfatto da tutte le righe che vengono restituite
 - serve per filtrare le righe che non lo soddisfano
 - è un'espressione booleana
 - che effettua dei test sui valori delle colonne

Modifica di un dato

- La modifica di un dato procede logicamente in due passi:
 - prima si ricerca una lista di righe
 - poi, su tutte le righe trovate, si modifica il contenuto di una o più colonne

UPDATE studenti

SET matricola = 13579, nome = "George"

WHERE id = 4 **AND** cognome = "Mountbatten";

- La tabella da modificare si specifica dopo la parola chiave **UPDATE**
- Dopo di che si specifica una lista di *nome-colonna = valore*, separati da virgole
 - rappresentano la lista di modifiche da fare alle righe
- Come nella **SELECT**, la clausola **WHERE** filtra le righe della tabella con un predicato booleano

Cancellazione di dati

- Come per la update, si procede in due passi:
 - si seleziona una lista di righe
 - e si cancella

```
DELETE FROM studenti  
WHERE id = 9 AND cognome = "Windsor";
```

- La tabella dove si vuole cancellare viene specificata dopo **DELETE FROM**
- Le righe si selezionano specificando la clausola **WHERE**
- La cancellazione, e in generale la modifica dei dati, devono rispettare i vincoli dello schema del database
 - altrimenti viene segnalato un errore
 - es: solitamente non si può cancellare una riga "puntata" da una chiave esterna

Ordinamento

- È possibile farsi restituire le righe di una tabella, ordinate per una certa colonna o gruppo di colonne
- L'ordinamento si effettua dopo che gli elementi sono stati filtrati
- **SELECT * FROM studenti ORDER BY cognome;**
 - restituisce tutti gli studenti ordinati per cognome in modo ascendente, dal più piccolo al più grande (dalla A alla Z)
 - ovviamente vale per qualsiasi tipo su cui sia definito un ordinamento (es: interi)
- **SELECT * FROM studenti ORDER BY cognome, nome;**
 - restituisce tutti gli studenti ordinati prima per cognome e, a parità di cognome, ordinati per nome
- **SELECT * FROM studenti ORDER BY cognome DESC;**
 - restituisce tutti gli studenti ordinati per cognome in modo discendente (dalla Z alla A)

Contare elementi

- Delle volte, è necessario semplicemente contare **quante** righe corrispondono ad una certa ricerca
- Per esempio vogliamo conoscere quanti studenti abbiamo sul database che abbiano più di 20 anni
- Per questo si utilizza **COUNT**

```
SELECT COUNT(*) FROM studenti  
WHERE eta > 20;
```

- Restituisce una sola riga con il numero di studenti
- Al posto di *, si può specificare il nome di una colonna

Limitare il numero di righe

- Soprattutto quando si devono visualizzare dei risultati su una pagina web, è inutile elencarli con una enorme lista unica
 - in alcuni casi i risultati potrebbero essere migliaia
- Spesso si usa una lista di pagine, che l'utente può scorrere in avanti e indietro
- In questi casi è inutile farsi restituire tutte le righe dal database per usarne una piccola parte
- es: Google...



Limitare il numero di righe

- È possibile specificare dei limiti al numero di righe nella ricerca

```
SELECT * FROM studenti LIMIT 15;
```

- Restituisce le prime 15 righe della tabella studenti

```
SELECT * FROM studenti LIMIT 10,10;
```

- Restituisce le righe che vanno dalla undicesima alla ventesima della tabella studenti
 - la prima riga ha indice 0 (quindi quella di indice 10 è la undicesima)
 - il primo numero è l'offset (indice di riga da cui partire)
 - il secondo numero è il numero di righe da restituire

Ricerche con pattern

- A volte è necessario fare delle ricerche all'interno di alcuni campi, senza conoscerne l'esatto valore
- Nelle query SQL l'operatore **LIKE** nella **WHERE** permette di "specificare" solo una parte del valore
- Il pattern viene specificato tramite due *wildcards* (caratteri che si sostituiscono ad altri)
 - il carattere **_** sostituisce **un solo carattere** qualsiasi
 - il carattere **%** sostituisce **una sequenza di caratteri** qualsiasi
 - si possono inserire prima e/o dopo una stringa fissa
- Esempi
 - **LIKE** 'Antoni_' seleziona sia Antoniaa che Antonioa, ma non Antonietta
 - **LIKE** 'Antoni%' seleziona Antoniaa, Antonioa e Antonietta
 - **LIKE** '%Antoni%' seleziona "Caro Antonio" e "Ciao Antonietta"

Ricerche con pattern

- Ricerchiamo tutti gli studenti il cui nome contenga una "i"

```
SELECT * FROM docenti  
WHERE nome LIKE '%i%';
```

id	nome	cognome	matricola
1	Caterina	Fenu	1394586
2	Silvia	Columbu	3496038
3	Maria	Infusino	2592049

Ricerche e relazioni

- Abbiamo visto che quando ci sono delle relazioni, i dati sono suddivisi su più tabelle
- Come si fa a selezionare questi dati?
- La sintassi **JOIN** . . . **ON** permette di unire "virtualmente" più tabelle per effettuare delle query sulla loro unione
- Per esempio consideriamo queste due chiavi esterne della tabella esami:
 - la prima verso la tabella insegnamenti
 - la seconda verso la tabella studenti
- Vogliamo l'elenco di tutti gli studenti che hanno superato l'esame di FPW, con il relativo voto
 - supponiamo che l'insegnamento FPW abbia id=19

Ricerche e relazioni

```
SELECT studenti.cognome,  
       studenti.nome,  
       esami.voto
```

```
FROM studenti JOIN esami
```

```
ON studenti.id = esami.studente_id
```

```
WHERE esami.insegnamento_id = 19;
```

- Si seleziona una tabella su cui fare la ricerca tra le due (o più) da unire
 - si preferisce la più piccola dal punto di vista delle righe per una questione di efficienza
- Nella **JOIN** si specifica quale sia l'altra tabella da unire
- Con **ON** si specifica quali siano i valori da utilizzare per l'unione (uguaglianza tra campi delle due tabelle)
- Questo crea una "**tabella virtuale**" unica su cui poi si esegue il filtro della **WHERE**
- Per distinguere tra i campi delle due tabelle (in caso di ambiguità) si può far precedere il nome del campo dal nome della tabella, seguito dal punto

Ricerche e relazioni

1

Studenti

id	nome	cognome	matricola
1	Piero	Angela	123456
2	Luciano	Onder	654321

JOIN

Esami

id	studente_id	insegnamento_id	voto
1	1	19	24
2	2	19	27
3	1	23	30
4	1	25	18
5	2	23	30

Ricerche e relazioni

2

studenti JOIN esami ON studenti.id = esami.studente_id

studenti.id	studenti.nome	studenti.cognome	studenti.matricola	esami.id	esami.studente_id	esami.insegnamento_id	esami.voto
1	Piero	Angela	123456	1	1	19	24
2	Luciano	Onder	654321	2	2	19	27
1	Piero	Angela	123456	3	1	23	30
1	Piero	Angela	123456	4	1	25	18
2	Luciano	Onder	654321	5	2	23	30

Ricerche e relazioni

3

```
... WHERE esami.insegnamento_id = 19;
```

studenti.id	studenti.nome	studenti.cognome	studenti.matricola	esami.Id	esami.studente_id	esami.insegnamento_id	esami.voto
1	Piero	Angela	123456	1	1	19	24
2	Luciano	Onder	654321	2	2	19	27

4

```
SELECT studenti.cognome, studenti.nome, esami.voto ...
```

studenti.nome	studenti.cognome	esami.voto
Piero	Angela	24
Luciano	Onder	27

Ricerche e relazioni

- È possibile fare la **JOIN** con più di una tabella
- Selezioniamo per ogni esame il nome dell'insegnamento, nome e cognome dello studente ed il voto

```
SELECT insegnamento.nome,  
       studente.cognome,  
       studente.nome,  
       esami.voto  
  
FROM esami JOIN insegnamento  
ON esami.insegnamento_id = insegnamento.id  
JOIN studenti  
ON studenti.id = esami.studente_id
```