Teatro SQL

1. **Panoramica**
    - Tipologie di basi di dati
    - Perchè usare un DBMS
    - Quando non usare un DBMS
    - Processo di sviluppo di basi di dati
    - Linguaggi per l'accesso alle basi di dati
2. **Progettazione di basi di dati relazionali**
    - Raccolta e analisi dei requisiti
        - Caso di studio: requisiti
    - Progettazione concettuale
        - Progettazione dei dati: il modello Entità-Relazione
            - Diagramma Entità-Relazione
            - Documentazione e regole aziendali
        - Progettazione delle transazioni
        - Qualità di uno schema concettuale
        - Caso di studio: schema concettuale
            - Il diagramma ER
            - Documentazione e regole aziendali
            - Progetto delle transazioni
            - Verifica di qualità
        - Esercitazione
    - Progettazione logica
        - Il modello relazionale
            - Strutture relazionali
            - Vincoli di integrità
        - Traduzione del modello ER nel modello relazionale
        - Documentazione del modello relazionale
        - Caso di studio: schema logico
            - Schema relazionale, vincoli e viste
            - Documentazione dello schema relazionale
        - Esercitazione
    - Progettazione fisica
        - Il linguaggio SQL
            - Interrogazione della base di dati
            - Aggiornamento della base di dati
            - Definizione dei dati
                - Tabelle
                - Vincoli di integrità
                - Viste
            - Perchè SQL?
        - Strutture di memorizzazione e di accesso ai dati

- Il DBMS MySQL
- Caso di studio: schema fisico
- Esercitazione

Un **dato** (in inglese *data*) è un fatto noto che può essere memorizzato.

Una **base di dati** (*database*) è una collezioni di dati logicamente correlati. Essa modella un **micro-mondo** (o universo del discorso), cioè una porzione della realtà (ad esempio un teatro). Deve essere consistente, cioè rappresentare fedelmente la realtà modellata. Ogni cambiamento nel micro-mondo deve riflettersi al più presto nella base di dati e ogni modifica della base di dati deve essere consistente con la realtà modellata.

Una base di dati è costruita ad uno scopo preciso (ad esempio informatizzare l'attività di un teatro), ha un determinato gruppo di **utenti** (ad esempio le persone interessate a prenotare un biglietto per uno spettacolo teatrale) e un insieme di **applicazioni** a cui questi utenti sono interessati (ad esempio una pagina Web che consente la prenotazione dei biglietti).

Un **sistema di gestione di basi di dati** (*database management system*, DBMS) è un insieme di programmi che permettono di creare, popolare, manipolare e condividere una base di dati.

I DBMS commerciali più popolari sono: Oracle Database, IBM DB2, e Microsoft SQL Server. I DBMS open source più famosi sono: PostgreSQL e MySQL.

Tipologie di basi di dati

Le basi di dati più usate o oggetto principale di questo corso si fondano sul **modello dei dati relazionale**. Esse modellano la realtà mediante un insieme di relazioni o tabelle. Una relazione possiede degli attributi o colonne dotati di un nome e un tipo. Le colonne contengono valori atomici del corrispondente tipo. Il modello relazionale è stato proposto da **Edgar F. Codd** nel celebre articolo *A relational Model for Large Shared Data Banks* apparso sulla rivista Communications of the ACM nel 1970. Solo a metà degli anni 80 le basi di dati relazionali sono diventate competitive e quindi usate su larga scala.

Modelli dei dati storici e oramai obsoleti sono quello **gerarchico** e quello **reticolare**. Il modello gerarchico si è sviluppato negli anni 60 e rappresenta i dati mediante strutture gerarchiche ad albero, mentre quello reticolare è basato sull'uso di grafi e si sviluppa agli inizi degli anni 70. Un forte svantaggio dei DBMS basati su questi modelli è che non implementano nessun livello di astrazione e di indipendenza dei dati. Risulta quindi difficile adattare la base di dati a nuovi requisiti dell'applicazione.

Il **modello dei dati ad oggetti** si è sviluppato negli anni 80 come tentativo di estendere le basi di dati relazionali. Esso si basa sul paradigma della programmazione orientata agli oggetti che descrive la realtà modellata in termini di oggetti dotati di proprietà e operazioni. Oggetti dello stesso tipo formano una classe e le classi sono organizzate in gerarchie. Una applicazione di tale modello consiste nella **memorizzazione persistente di oggetti** di programmi scritti in linguaggi

orientati agli oggetti (C++, Java). In tal caso, una base di dati ad oggetti offre compatibilità tra le strutture di dati dei programmi e quelle delle basi di dati. Nonostante le aspettative, tale modello ha avuto poca penetrazione nel mercato delle basi di dati.

Il **modello dei dati semistrutturati** descrive la realtà mediante strutture gerarchiche flessibili che consentono di modellare dati privi di una struttura rigida. Ad esempio, in questo modello, elementi dello stesso tipo possono avere attributi differenti oppure un elemento può avere più attributi con lo stesso nome. Inoltre gli attributi di un elemento possono essere altri elementi generando in questo modo elementi composti che formano una gerarchia ad albero. Questa flessibilità ha reso questo modello molto popolare per lo **scambio di dati su Web**.

Una base di dati semistrutturati fa solitamente uso del linguaggio **XML** (Extensible Markup Language) per rappresentare i dati. La base di dati consiste di una collezione di documenti XML, salvati come documenti di testo sfruttando il sistema operativo sottostante o archiviati mediante qualche struttura di dati specializzata. Esistono linguaggi standard per definire lo schema di una base XML (DTD, XSchema), per interrogarla (XPath, XQuery). Sono inoltre state sviluppate interfacce per accedere a documenti XML mediante linguaggi di programmazione (Java API for XML Processing).

Un vantaggio delle basi XML rispetto alle basi relazionali è che non necessitano per forza di un DBMS (strumento complesso e costoso) per essere operative. Esse infatti possono sfruttare il file system per memorizzare dati e schemi della base e sfruttare specifiche applicazioni (spesso non proprietarie) per validare e interrogare la base di dati. Naturalmente questo approccio non garantisce una serie di servizi che un DBMS offre quali, ad esempio, l'accesso concorrente ai dati.

Per approfondire l'approccio basato su XML si veda Caffè XML o il corso di Tecnologie XML.

Perchè usare un DBMS

Un DBMS permette di accedere in modo semplice e efficiente ad una base di dati mantenendone la consistenza, la privatezza e l'affidabilità. I vantaggi dell'uso di un DBMS sono i seguenti:

- **Accesso ai dati tramite un linguaggio universale**. Ogni DBMS di una certa tipologia mette a disposizione un **linguaggio di interrogazione** (SQL nel caso relazionale). Tale linguaggio permette la creazione delle strutture che contengono i dati, l'inserimento, la cancellazione, l'aggiornamento dei dati e il recupero delle informazioni dalla base di dati.
- **Accesso efficiente ai dati**. Un DBMS ha molti modi per ottimizzare l'accesso all'informazione. La base di dati è solitamente memorizzata in memoria secondaria (disco rigido). Un DBMS permette di creare dei file ausiliari (**indici**) che permettono l'accesso veloce ai dati su disco. Inoltre spesso un DBMS mantiene porzioni della base di dati in memoria centrale velocizzando in questo modo l'accesso ai dati. Infine ogni interrogazione prima di essere eseguita viene ottimizzata scegliendo un piano efficiente di esecuzione sulla base degli indici esistenti.
- **Indipendenza dei dati**. Un DBMS mantiene diversi livelli di astrazione dei dati. Il livello più basso è detto **fisico** (o interno) e descrive *come* i dati sono fisicamente archiviati. Il livello immediatamente superiore è detto **logico** e descrive *quali* dati sono archiviati e quali

vincoli di integrità essi devono soddisfare. Un DBMS permette di accedere ai dati logici indipendentemente dalla loro rappresentazione fisica. Quest'ultima può cambiare senza che i metodi di accesso ai dati logici debbano essere modificati. Si parla di **indipendenza fisica dei dati**. Non sempre l'intera base di dati deve essere visibile a tutti gli utenti. Il livello **esterno** descrive porzioni della base di dati (**viste**) accessibili da particolari gruppi di utenti.

- **Controllo della ridondanza dei dati**. Ogni **dato logico** dovrebbe essere memorizzato in un solo posto nella base di dati. Avere più copie della stessa informazione ha i seguenti svantaggi:
    1. vi è un maggior uso di memoria;
    2. le modifiche della stessa informazione debbono essere effettuate diverse volte;
    3. ci possono essere fenomeni di **inconsistenza** dei dati qualora gli aggiornamenti dei dati vengano eseguiti in modo indipendente. Dunque la stessa informazione potrebbe assumere valori diversi.

  Talvolta viene consentita una forma di **ridondanza controllata** in un DBMS allo scopo di migliorare le prestazioni delle interrogazioni. Il DBMS stesso controlla che tale ridondanza non generi inconsistenze.

- **Imposizione di vincoli di integrità sui dati**. Un DBMS permette di specificare diversi tipi di vincoli per mantenere l'integrità dei dati e controlla che tali vincoli siano soddisfatti quando la base di dati cambia. Ad esempio, il vincolo che un voto di un esame universitario sia un intero tra 18 e 30 o il vincolo che i valori del codice fiscale di un una persona siano univoci.
- **Atomicità delle operazioni**. Un DBMS permette di effettuare sequenze di operazioni (**transazioni**) in modo **atomico**. Ciò significa che l'intera sequenza di operazioni viene eseguita con successo oppure nessuna di queste operazioni ha alcun effetto sui dati della base. L'atomicità delle transazioni permette di mantenere una stato della base di dati consistente con la realtà modellata. Si pensi ad esempio ad un bonifico da un conto A ad un conto B. Se la transazione non fosse atomica, potrebbe accadere che dopo aver prelevato dal conto A il sistema per qualche motivo fallisca senza aver accreditato il conto B.
- **Accesso concorrente ai dati**. Un DBMS permetta a più utenti di accedere contemporaneamente alla base di dati. Più utenti possono accedere nello stesso istante a dati diversi. Inoltre, un DBMS fa in modo che l'accesso concorrente agli stessi dati non generi **anomalie**, cioè inconsistenza nello stato della base di dati rispetto alla realtà modellata. Si pensi ad esempio a due utenti A e B che vogliono prelevare rispettivamente 100 e 200 Euro dallo stesso conto che ha un saldo di 1000 Euro. Una possibile anomalia avviene se entrambi gli utenti leggono lo stesso saldo (1000). Successivamente l'utente A preleva 100 e aggiorna il saldo a 900 (1000 - 100). Infine l'utente B preleva 200 e aggiorna il saldo a 800 (1000 - 200). Il saldo reale è invece 700 (1000 - 100 - 200).
- **Privatezza dei dati**. Un DBMS permette un accesso protetto ai dati. Utenti diversi possono avere accesso a diverse porzioni della base di dati e possono essere abilitati a diverse

operazioni sui di esse. Ad esempio, alcuni utenti possono essere impediti a leggere dei dati sensibili oppure possono leggere ma non modificare certi dati.

- **Affidabilità dei dati**. Un DBMS offre dei metodi per salvare copie dei dati (**backup**) e per ripristinare lo stato della base di dati in caso di guasti software e hardware (**recovery**).

Quando non usare un DBMS

Un DBMS è un prodotto **complesso** e quindi **costoso** in termini finanziari (se il DBMS è commerciale) e in ogni caso in termini di formazione all'uso dello strumento.

Un DBMS inoltre fornisce in forma integrata una serie di servizi difficilmente scorporabili. Ciò comporta una riduzione delle prestazioni e un aumento dei costi se questi servizi non sono indispensabili.

E' preferibile **non usare** un DBMS quando l'applicazione è semplice, relativamente stabile nel tempo e non prevede accesso concorrente da parte di più utenti. In tal caso meglio usare i file ordinari per archiviare i dati e delle procedure apposite per accedere ai dati.

Un approccio intermedio tra i file ordinari e i DBMS consiste nell'uso di **documenti XML**. L'approccio basato su XML elimina gli svantaggi citati dell'uso dei DBMS in quanto un documento XML è semplicemente un file di testo che può essere archiviato sfruttando direttamente il file system. Inoltre, tale approccio offre strumenti oramai consolidati (e non proprietari) per definire gli schemi, interrogare, trasformare, e accedere ai dati tramite linguaggi di programmazione. Come detto questo approccio non offre i vantaggi di concorrenza, consistenza, privatezza e affidabilità dei dati tipici dei DBMS.

Processo di sviluppo di basi di dati

La creazione di una base di dati è un processo complesso articolato in cascata nelle seguenti fasi:

1. **Raccolta e analisi dei requisiti**. Il risultato di questa fase è una descrizione in linguaggio naturale del **micro-mondo** (**universo del discorso**) che si vuole modellare. Vanno descritti sia gli aspetti statici (dati) che quelli dinamici (operazioni o transazioni sui dati). Solitamente questa fase avviene a stretto contatto con il cliente del sistema inserendo i progettisti del DBMS nell'ambiente di lavoro in cui l'applicazione dovrà essere realizzata (progettazione contestuale). Inoltre occorre interagire con gli utenti potenziali del sistema raccogliendo le modalità con cui essi intendono usare il sistema. Una volta raccolti i requisiti, occorre analizzarli ottenendo una descrizione sintetica e precisa che isoli le **entità** essenziali dell'universo del discorso e le **relazioni** che intercorrono tra di esse.
2. **Progettazione concettuale della basi di dati**. L'obiettivo di questa fase è la formalizzazione dei requisiti della base di dati in uno **schema concettuale** indipendente sia dal tipo che dallo specifico DBMS che verrà utilizzato. Il **modello Entità-Relazione** (ER) è solitamente usato durante la progettazione concettuale per definire gli aspetti statici

del sistema, cioè i dati. Esso è un modello diagrammatico che descrive le entità da modellare (ad esempio, in un contesto universitario, studente e corso), gli attributi delle entità (ad esempio matricola per lo studente e nome per corso), le relazioni che intercorrono tra le entità (ad esempio la relazione esame che associa studenti a corsi il cui esame è stato superato), e le cardinalità delle relazioni (uno studente può aver superato zero o più corsi e un corso può essere stato superato da zero o più studenti).

Il modello ER non consente la specifica delle transazioni, le quali vengono descritte in termini del loro input, output e comportamento funzionale. Un modello concettuale che permette di specificare in modo integrato gli aspetti statici e dinamici del sistema è *Unified Modeling Language* (UML).

3. **Scelta della tipologia della base di dati**. Prima di iniziare la progettazione logica occorre scegliere il tipo di base di dati da utilizzare, cioè il **modello dei dati** che si vuole adottare. La scelta dello specifico DBMS viene effettuata dopo la progettazione logica e prima di quella fisica.
4. **Progettazione logica della base di dati**. Durante questa fase lo schema concettuale (riferito ai dati) viene tradotto in uno **schema logico** che rispecchia il modello dei dati scelto al passo precedente. Lo schema logico è indipendente dallo specifico DBMS che verrà scelto dopo questa fase. Il modello dei dati delle basi relazionali prende il nome di **modello relazionale**. Il modello relazione dei dati usa una collezione di **tabelle** per modellare sia le entità che le relazioni del modello concettuale. Le **colonne** delle tabelle corrispondono agli attributi delle entità.

   Ad esempio, potremmo tradurre l'entità studente in una tabella omonima con, tra le altre, la colonna matricola, l'entità corso in una tabella omonima con, tra le altre, la colonna nome, e la relazione esame in una terza tabella omonima con colonne matricola dello studente, nome del corso, data e voto dell'esame. Inoltre durante la progettazione logica si definiscono gli schemi esterni (**viste**) per le specifiche applicazioni.
5. **Scelta di un DBMS**. In questa fase occorre scegliere lo specifico DBMS appartenente alla tipologia individuata precedentemente. La scelta di un DBMS è influenzata da fattori tecnici (ad esempio le strutture di memorizzazione e di accesso ai file offerte dal DBMS) e da fattori economici e organizzativi (ad esempio, costo del sistema e assistenza post-vendita).
6. **Progettazione fisica della base di dati**. Durante questa fase lo schema logico della base di dati viene tradotto in uno **schema fisico** costituito dalle definizioni delle tabelle, dei relativi vincoli e delle eventuali viste in un linguaggio formale (SQL nelle basi relazionali). Inoltre, in questa fase vengono scelte, tra quelle messe a disposizione dal DBMS, le strutture di memorizzazione e di accesso alle tabelle con l'obiettivo di garantire l'efficienza del sistema. Infine, le transazioni vengono implementate in un linguaggio formale (SQL nelle basi relazionali).
7. **Realizzazione e ottimizzazione del sistema della base di dati**. In questa fase vengono fisicamente create le tabelle, i vincoli di integrità sui dati e le viste sfruttando le definizioni dello schema fisico. Le tabelle vengono quindi popolate inserendo i dati veri e propri. A questo punto ha inizio la fase ottimizzazione del sistema (*tuning*): le transazioni vengono

eseguite sui dati e le prestazioni del sistema vengono monitorare, eventualmente modificando i parametri scelti durante la progettazione fisica.

L'esecuzione di queste fasi è teoricamente sequenziale. In realtà, durante lo sviluppo del sistema ci possono essere dei **cicli di feedback** durante i quali si modifica una fase precedente durante una fase successiva.

E' importante notare che le fasi più importanti sono quelle più generali, cioè quelle iniziali. Infatti un errore in una fase iniziale, ad esempio dimenticarsi una relazione importante tra due entità durante l'analisi dei requisiti, si ripercuote a cascata in tutte le fasi successive. Viceversa un errore in una fase terminale, ad esempio un errore di programmazione in una transazione, si può correggere facilmente senza influenzare le fasi precedenti.

Linguaggi per l'accesso alle basi di dati

I linguaggi per l'accesso alle basi di dati comprendono i seguenti linguaggi:

Linguaggio di definizione dei dati (*data-definition language*, DDL)
: Questo linguaggio permette di definire i tre livelli di astrazione della base di dati: esterno, logico e fisico.

Linguaggio di manipolazione dei dati (*data-manipulation language*, DML)
: Questo linguaggio permette di inserire, cancellare, modificare e reperire i dati. Questo linguaggio può essere **procedurale** (l'utente specifica che dati recuperare e come farlo) o **dichiarativo** (l'utente specifica che dati recuperare ma non dice come. Il sistema si occupa di implementare le richieste dell'utente).

Il **linguaggio SQL** è un linguaggio dichiarativo globale che permette sia la definizione dei dati che la loro manipolazione in una base di dati relazionale. Una **interrogazione** (**query**) è una istruzione ben formata del linguaggio SQL.

Solitamente l'accesso diretto alla base di dati con l'interprete SQL è effettuato dall'amministratore, dai progettisti e dagli utenti esperti della base. L'accesso più tipico ad una base di dati avviene invece attraverso applicazioni scritte in un linguaggio ad alto livello (C, C++, Java) che richiamano interrogazioni SQL. L'integrazione tra il linguaggio ospite ad alto livello e SQL può avvenire in due modi:

SQL Embedded
: Questo metodo prevede di introdurre direttamente nel programma sorgente scritto in linguaggio ad alto livello le istruzioni SQL distinguendole con un opportuno separatore. Prima della compilazione del codice un precompilatore traduce le istruzioni SQL in istruzioni del linguaggio ospite.

Call Level Interface
: Questa soluzione consiste nel mettere a disposizione una libreria di procedure scritte nel linguaggio ad alto livello per realizzare il dialogo con la base di dati. Due esempi di interfacce per la connettività sono **ODBC** (Open Database Connectivity), sviluppato originariamente da Microsoft e diventato in seguito uno standard, e **JDBC** (Java Database

Connectivity), ideato dalla Sun Microsystems per gestire la connettività di Java verso le basi di dati relazionali.

Progettazione di basi di dati relazionali

La progettazione si divide in **progettazione dei dati** e **progettazione delle transazioni**. Nella prima si individuano la struttura e l'organizzazione dei dati, nella seconda si definiscono le caratteristiche delle operazioni che usano i dati. Le due attività sono interconnesse e dovrebbero procedere in parallelo.

Esiste una **metodologia** consolidata di progettazione di basi di dati che consiste nelle seguenti fasi:

1. **raccolta e analisi dei requisiti**
2. **progettazione concettuale**
3. **progettazione logica**
4. **progettazione fisica**

Tale metodologia si fonda sul principio della separazione del **cosa** rappresentare in una base di dati (fasi 1 e 2) dal **come** farlo (fasi 3 e 4).

Terminata la progettazione si passa alla **realizzazione** della base di dati che consiste nella creazione fisica delle strutture dei dati e nell'implementazione delle applicazioni che useranno la base di dati.

Raccolta e analisi dei requisiti

La **raccolta dei requisiti** consiste nella individuazione delle caratteristiche statiche (dei dati) e dinamiche (delle operazioni) dell'applicazione da realizzare. I requisiti vengono raccolti in specifiche espresse in linguaggio naturale e per questo motivo spesso ambigue e disorganizzate. L'**analisi dei requisiti** consiste nel chiarimento e nell'organizzazione delle specifiche raccolte.

I requisiti dell'applicazione provengono da diverse fonti, tra le quali: gli *utenti* dell'applicazione, la *documentazione* esistente attinente al problema allo studio, eventuali *realizzazioni preesistenti* dell'applicazione.

Il coinvolgimento del cliente nel processo di sviluppo aumenta il suo livello di soddisfazione nei confronti del sistema che gli verrà consegnato e agevola il lavoro di raccolta dei requisiti dei progettisti. Spesso i progettisti vengono inseriti direttamente nell'ambiente di lavoro in cui l'applicazione dovrà essere utilizzata (si parla di *progettazione contestuale*). E' prassi percorrere assieme al cliente il flusso di lavoro e gli scenari di esecuzione delle transazioni del sistema proposto, oppure creare un prototipo dimostrativo dell'applicazione.

Alcune regole generali per ottenere una specifica precisa e priva di ambiguità:

- Evitare termini troppo generici o troppo precisi. Mantenere un livello di astrazione costante.
- Evitare l'uso di sinonimi (termini diversi con il medesimo significato) e omonimi (termini uguali con differenti significati). Riferirsi allo stesso concetto sempre nello stesso modo.
- Usare frasi brevi e semplici possibilmente uniformandone la struttura.
- Dividere il testo in paragrafi. Dedicare ogni paragrafo alla descrizione di una specifica entità della realtà modellata. Evidenziare l'entità descritta in ogni paragrafo.

E' fondamentale fin da questa fase differenziare le seguenti componenti del modello:

1. **Entità**. Una entità è un concetto complesso e di rilievo che descrive classi di oggetti con esistenza autonoma. Esempi di entità nel contesto di una rete di teatri potrebbero essere teatro, dipendenti, spettacoli.
2. **Attributi delle entità**. Un attributo è un concetto che ha una struttura semplice e non possiede proprietà rilevanti associate. Ad esempio, il nome del teatro, il codice fiscale del dipendente, il titolo dello spettacolo.
3. **Relazioni tra entità**. Queste sono associazioni tra due o più entità. Ad esempio una relazione che associa teatri e relativi dipendenti. Delle relazioni occorre identificare anche le **cardinalità** con cui le entità vi partecipano. Ad esempio, ogni teatro può avere *più* dipendenti e ogni dipendente può lavorare per *più* teatri.

Accanto alle specifiche sui dati vanno raccolte le specifiche sulle **transazioni tipiche del sistema**, cioè le operazioni che si stimano essere più frequenti. Non tutte le transazioni sono note in fase di progettazione ma esiste una regola informale che afferma che l'80% del carico del sistema è prodotto dal 20% delle transazioni. Una operazione tipica deve essere efficiente. E' possibile fare delle particolari scelte nelle successive fasi della progettazione al fine di migliorare l'efficienza delle transazioni tipiche. Nella descrizione informale delle transazioni bisogna adottare la medesima terminologia usata per i dati.

La progettazione delle transazioni è importante quanto quella dei dati e dovrebbe procedere di pari passo. Conoscere le transazioni tipiche permette di capire come modellare i dati affinchè queste transazioni possano essere eseguite (efficientemente). Ad esempio, se una tipica transazione richiede il numero di posti liberi per uno spettacolo, occorre registrare nella base di dati la capienza degli spazi e il numero di posti prenotati per uno spettacolo. Inoltre, se una transazione tipica richiede gli spettacoli ordinati cronologicamente, è bene che gli spettacoli vengano mantenuti ordinati (creando un indice sul campo data dello spettacolo). In realtà, spesso la progettazione dei dati, affidata ai progettisti della base, è disgiunta dalla progettazione delle transazioni svolta dagli ingegneri del software.

Caso di studio: requisiti

Vediamo un esempio concreto di analisi dei requisiti per una base di dati che vuole modellare la realtà di una **rete teatrale**. Occorre innanzitutto raccogliere i requisiti. Le fonti delle informazioni possono essere: documentazione relativa all'attività dei teatri coinvolti, ad esempio i siti Web dei

teatri. Qualche dipendente del teatro che conosca bene sia le dinamiche del teatro che le aspettative del sistema da realizzare. Un gruppo di potenziali spettatori del teatro. Il sistema informatico esistente e le interazioni che dovrà avere con la nuova applicazione.

Supponiamo di aver raccolto, dopo la consultazione delle fonti di informazione, i seguenti **requisiti per i dati**:

*Un **teatro** della rete è identificato da: nome, indirizzo fisico, telefono, fax, indirizzo di posta elettronica e pagina web.*

*Un teatro ha degli **spazi teatrali** in cui vanno in scena gli spettacoli. Tali spazi hanno un nome, un indirizzo, una pianta (in formato grafico) e una capienza. Gli spazi possono essere condivisi da più teatri della rete.*

*Ogni teatro ha le proprie biglietterie. Ogni biglietteria è associata ad un unico teatro. Le **biglietterie** sono identificate da un nome, un indirizzo fisico, un telefono, un indirizzo di posta elettronica e gli orari di apertura per ogni giorno settimanale.*

*Un teatro ha dei **dipendenti** suddivisi in: (i) coordinamento artistico, (ii) produzione, organizzazione e distribuzione, (iii) comunicazione, ufficio stampa e relazioni con il pubblico, (iv) amministrazione, controllo e finanza. Ogni dipendente è identificato da: codice fiscale, nome, cognome, residenza, data e luogo di nascita, età, numero di telefono fisso, numero di telefono cellulare, indirizzo di posta elettronica, data di assunzione, ruolo, stipendio attuale e passato. I dipendenti con almeno 10 anni di attività in un teatro possono concorrere ad un posto nel Consiglio di Amministrazione di quel teatro. Un dipendente può lavorare per più teatri della rete.*

*Un teatro propone le proprie **stagioni teatrali**. Ogni stagione è identificata da un nome e un biennio ed è composta da un certo numero di spettacoli. Vi possono essere più stagioni associate allo stesso biennio. Lo stesso spettacolo può essere incluso in più stagioni differenti (anche di teatri diversi).*

*Uno **spettacolo** è descritto da: titolo, anno di produzione, descrizione, interpreti e produttori.*

*Un **interprete** è descritto da nome, ruolo (ad esempio, attore, regista, autore), e un CV in formato*

testo.

*Un produttore può essere un teatro della rete o un ente esterno. Per un **produttore esterno** si memorizza il nome, l'indirizzo fisico e di posta elettronica, e il numero di telefono. Un teatro non può mettere in scena più di due spettacoli di propria produzione all'interno della stessa stagione teatrale.*

*Uno spettacolo viene **messo in scena** in una certa data, ora e spazio teatrale. Per ogni messa in scena vengono archiviate eventuali immagini e spezzoni video e viene fissato un prezzo del biglietto (intero, ridotto e studenti): il prezzo ridotto corrisponde all'80% prezzo intero, quello per studenti è il 50% del prezzo intero. Viene inoltre mantenuto il numero di posti disponibili calcolato come differenza tra la capienza dello spazio teatrale e il numero di prenotazioni effettuate.*

*Uno spettacolo messo in scena può essere prenotato. Per ogni **prenotazione** vengono registrati: un numero progressivo, data e ora, un codice identificativo del posto in sala, il tipo di biglietto (intero, ridotto, studenti) e il prezzo del biglietto. Una prenotazione può essere effettuata solo se vi sono ancora posti disponibili. Il prezzo di una prenotazione per un biglietto di un certo tipo è uguale al prezzo del biglietto di quel tipo fissato per lo spettacolo prenotato.*

*Esiste un blog nel quale gli utenti possono inviare un commento ad uno spettacolo messo in scena. Ogni **commento** è identificato da: pseudonimo dell'autore, data e ora, testo del commento. Ogni commento può essere nuovo o la risposta a qualche altro commento.*

*Infine, un teatro può mantenere una newsletter contenente notizie sull'attività dei teatri della rete. Per ogni **notizia** viene memorizzata la data, l'ora, l'oggetto e il testo della notizia.*

Supponiamo inoltre di aver identificato per la nostra rete di teatri le seguenti **transazioni tipiche**:

i. Inserire una nuova stagione teatrale di un teatro. Vanno inseriti anche tutti i nuovi interpreti e produttori degli spettacoli della stagione non presenti nella base di dati.
ii. Inserire una prenotazione, se possibile.
iii. Inserire un commento ad uno spettacolo messo in scena.
iv. Inserire una notizia.
v. Inserire/rimuovere un dipendente.

vi. Trovare gli spettacoli messi in scena in una stagione teatrale di un teatro. Per ogni spettacolo occorre stampare il titolo, una descrizione, data, ora e luogo di scena, posti disponibili e prezzi.
vii. Trovare gli attori che hanno lavorato in una stagione di un teatro. La stessa transazione per i registi e gli autori.
viii. Trovare i dipendenti di un teatro.
ix. Trovare i commenti riferiti ad uno spettacolo.
x. Trovare le notizie riferite ad un teatro.
xi. Trovare gli orari di apertura delle biglietterie di un teatro.
xii. Generare una statistica sull'andamento delle stagioni teatrali della rete. Per ogni stagione si vuole calcolare la media degli spettatori paganti, la media dell'affluenza, cioè del rapporto tra paganti e capienza e la media degli incassi riferite agli spettacoli della stagione. Si vuole inoltra stampare gli spettacoli di una particolare stagione in ordine di: (i) paganti, (ii) affluenza, (iii) incasso.

Progettazione concettuale

La progettazione concettuale consiste nella parziale formalizzazione dei requisiti sui dati e sulle transazioni che sono stati raccolti e analizzati nella fase precedente. In particolare la progettazione concettuale consiste delle seguenti fasi:

1. formalizzare i requisiti sui dati in un **diagramma Entità-Relazione**;
2. **documentare** il diagramma;
3. aggiungere le **regole aziendali** non codificabili nel diagramma;
4. progettare le **transazioni**;
5. **verificare la qualità** degli schemi concettuali prodotti.

Progettazione dei dati: il modello Entità-Relazione

Il **modello Entità-Relazione** (modello ER) è un modello concettuale dei dati. Come tale descrive ad alto livello la realtà modellata indipendentemente da come i dati verranno logicamente e fisicamente rappresentati.

Il modello ER definisce uno **schema** concettuale dei dati. Uno schema descrive la struttura o forma dei dati, ma non descrive i dati veri e propri, vale a dire le **istanze** o occorrenze dello schema. Ad esempio, lo schema ER può dire che l'entità teatro ha gli attributi nome e indirizzo ma non dice quali sono gli effettivi valori di questi attributi. I valori verranno inseriti alla fine della progettazione in fase di popolamento della base di dati. Facendo un paragone con la programmazione orientata agli oggetti, lo schema corrisponde alla nozione di **classe** e una istanza dello schema è un **oggetto** della classe.

Lo schema ER è costituito da due componenti:

1. **diagramma ER**: è una rappresentazione diagrammatica della realtà che evidenzia le entità, le relazioni tra entità e gli attributi di entrambi;

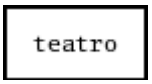
2. **documentazione**: una spiegazione (in linguaggio naturale) dei concetti presenti nel diagramma ER;
3. **regole aziendali**: sono delle regole che aggiungono dei vincoli sui dati che non sono rappresentabili con il modello ER.

Diagramma Entità-Relazione

I costrutti del diagramma ER sono i seguenti:

Entità

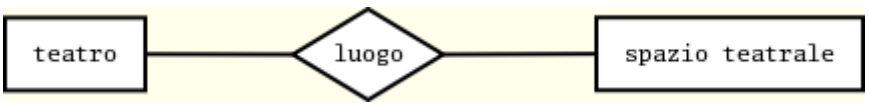
Rappresentano concetti complessi e di rilievo che descrivono classi di oggetti con esistenza autonoma. Una istanza di una entità è un oggetto della classe rappresentata. Ad esempio, possibili entità della rete teatrale sono: teatro, spazio teatrale, dipendente, spettacolo, attore, notizia. Una entità ha un nome univoco all'interno dello schema concettuale e viene rappresentata nel diagramma ER con un rettangolo con il nome dell'entità all'interno:



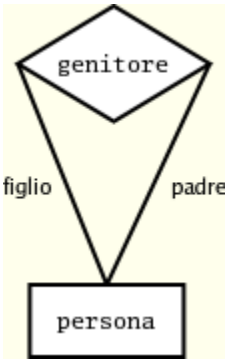
Relazioni

Rappresentano legami logici, significativi per la realtà modellata, tra due o più entità. Una istanza di una relazione è una coppia (ennupla in generale) di istanze di entità partecipanti alla relazione. Formalmente, una relazione è un sottoinsieme del prodotto cartesiano delle classi che la compongono. Essendo un insieme, tutti le istanze di una relazione sono distinte.

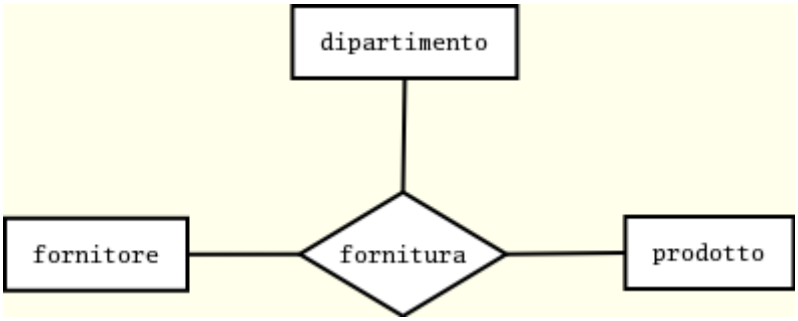
Ad esempio, esiste una relazione, che possiamo chiamare luogo, tra le entità teatro e spazio teatrale. Questa relazione associa ad un teatro gli spazi teatrali che il teatro usa per gli spettacoli e viceversa essa associa ad uno spazio teatrale i teatri che usano quello spazio. Ogni relazione ha un nome univoco tra le relazioni e le entità dello schema. E' preferibile usare un sostantivo come nome per non indurre ad individuare un verso nella relazione. Una relazione viene rappresentata mediante un rombo con il suo nome all'interno e da linee che connettono la relazione con le entità componenti:



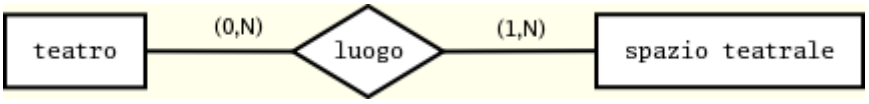
E' possibile avere **relazioni ricorsive**, cioè che legano la stessa entità. Ad esempio, la relazione collega definita sull'entità dipendente o la relazione genitore sull'entità persona. La relazione collega è simmetrica, la relazione genitore non lo è. Nel caso di relazioni non simmetriche, occorre definire i **ruoli di partecipazione** dell'entità alla relazione. Nel caso di genitore, i ruoli sono padre e figlio:



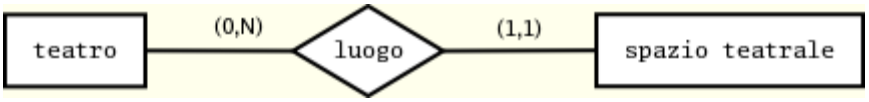
Inoltre è possibile avere relazioni che associano più di due entità. Ad esempio, la relazione fornitura associa un fornitore che rifornisce di un certo prodotto un qualche dipartimento:



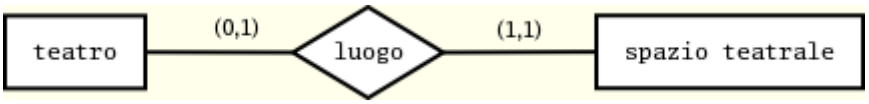
Per ogni entità partecipante ad una relazione viene specificata una **cardinalità di relazione**. Essa è una coppia di numeri naturali che specifica il numero minimo e massimo di istanze di relazione a cui una istanza dell'entità può partecipare. E' possibile indicare con la costante N un numero generico maggiore di uno quando la cardinalità non è nota con precisione. Nel diagramma ER, la cardinalità di relazione di una entità etichetta l'arco che lega l'entità alla relazione. Seguono alcuni esempi:



In questo caso un teatro ha nessun, uno o molti spazi teatrali e uno spazio teatrale ha uno o più teatri associati. Questo tipo di relazione in cui entrambe le cardinalità massime sono maggiori di uno si chiama **molti a molti**. La partecipazione dell'entità teatro si dice **opzionale** (cardinalità minima uguale a zero) e la partecipazione dell'entità spazio teatrale si dice **obbligatoria** (cardinalità minima uguale a uno).

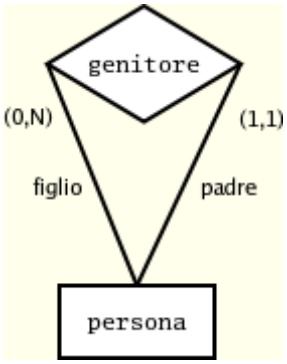


In questo caso un teatro ha nessun, uno o molti spazi teatrali e uno spazio teatrale ha esattamente un teatro associato. Dunque non vi sono spazi condivisi da più teatri. Questo tipo di relazione in cui una cardinalità massima è pari a uno e l'altra è maggiore di uno si chiama **uno a molti**.

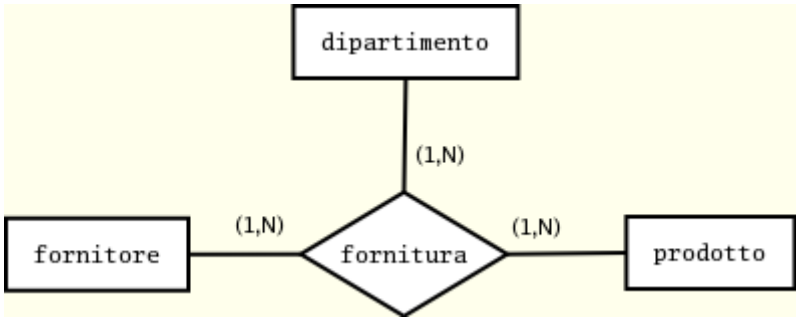


In questo caso un teatro ha nessun o uno spazio teatrale e uno spazio teatrale ha esattamente un teatro associato. Dunque non vi sono teatri con più spazi e neppure spazi condivisi da più teatri. Questo tipo di relazione in cui entrambe le cardinalità massime sono uguali a uno si chiama **uno a uno**.

Di seguito proponiamo un esempio di relazione ricorsiva con ruoli e cardinalità. Ogni persona ha esattamente un padre e zero o più figli:



Infine mostriamo un esempio di relazione ternaria con cardinalità:



Si ricordi che la cardinalità di una entità si riferisce al numero di istanze della relazione a cui quella entità può partecipare. Ad esempio, nel caso di fornitore, la cardinalità si riferisce al numero di *coppie* prodotto e dipartimento che il fornitore rifornisce. Dunque una cardinalità massima pari a uno per l'entità fornitore si ha quando un fornitore rifornisce un solo dipartimento di un solo prodotto.

Attributi

Un attributo è un concetto che ha una struttura semplice e non possiede proprietà rilevanti associate. Un attributo non ha esistenza autonoma ma è associato ad una entità o ad una relazione. Un attributo può essere semplice, multivalore, composto o calcolato:

- un **attributo semplice** ha, per ogni istanza dell'entità associata, un unico valore atomico di un certo tipo di base. Ad esempio, attributi semplici possono essere nome, indirizzo di posta elettronica e pagina web;
- un **attributo multivalore** può avere, per ogni istanza dell'entità associata, più valori. Ad esempio, un attributo multivalore potrebbe essere telefono per il quale possono essere noti più numeri;

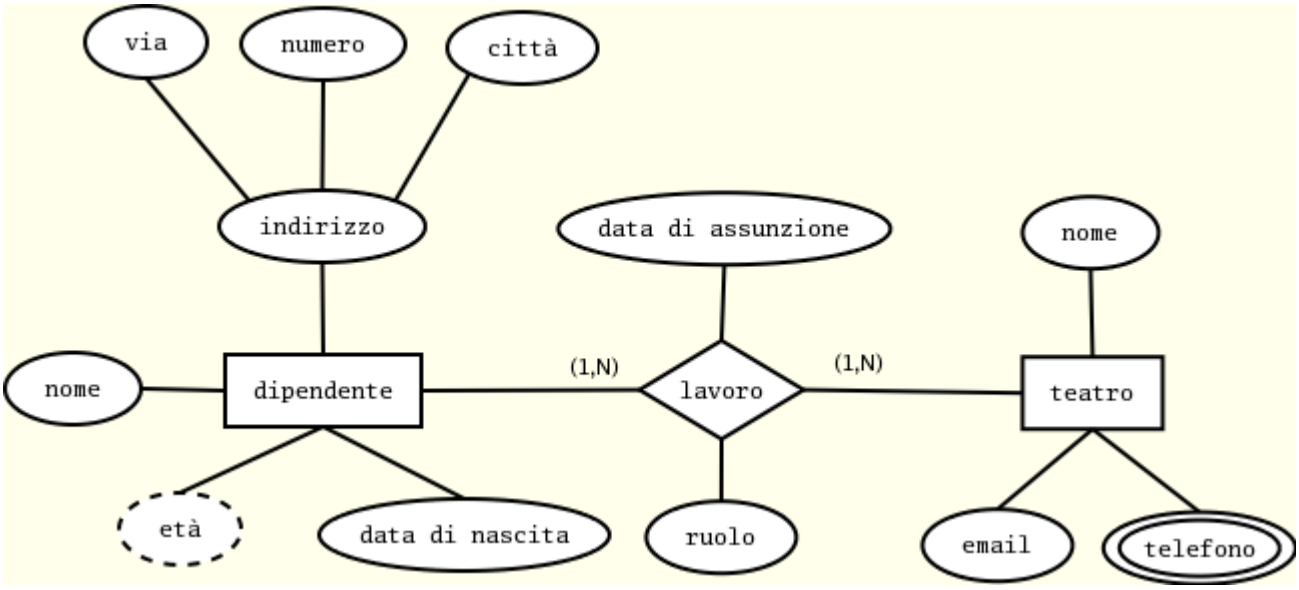
- un **attributo composto** è costituito da più attributi correlati. Ad esempio un attributo composto potrebbe essere indirizzo formato dagli attributi via, numero, e città;
- un **attributo calcolato** è un attributo il cui valore è ottenuto calcolandolo da quello di altri attributi. Ad esempio l'incasso totale si ottiene moltiplicando il numero di spettatori per il prezzo del biglietto (supponendo che il prezzo sia fisso).

Un attributo può essere inoltre obbligatorio o opzionale. Un attributo **obbligatorio** deve sempre avere un valore valido. Un attributo **opzionale** può non avere un valore a causa di uno dei seguenti motivi:

- il valore dell'attributo non esiste. Ad esempio, nel caso di un attributo email per una persona, la persona non possiede un indirizzo di posta elettronica;
- il valore dell'attributo esiste ma non è noto. Quindi, la persona ha un indirizzo di posta elettronica ma non ci è noto quale sia;
- non è noto se il valore dell'attributo esista. Quindi, non sappiamo se la persona abbia un indirizzo di posta elettronica.

Un attributo può essere associato ad una entità o ad una relazione. Nel secondo caso esso assume un valore per ogni coppia (o ennupla) di istanze partecipanti alla relazione. Ad esempio, una relazione che lega i dipendenti ai teatri in cui essi lavorano può avere attributi data di assunzione e ruolo.

Un attributo ha un **nome univoco** all'interno dell'insieme degli attributi della *stessa* entità o relazione. Dunque attributi di entità o relazioni diverse possono essere omonimi. Un attributo si rappresenta graficamente con una ellisse (con una doppia ellisse se multivalore, con una ellisse tratteggiata se calcolato) con il nome all'interno collegata all'entità, alla relazione o all'attributo composto di appartenenza. Segue un esempio:



Chiavi

Per ciascuna entità occorre specificare una chiave primaria. Una **chiave** è un insieme di uno o più attributi con due caratteristiche:

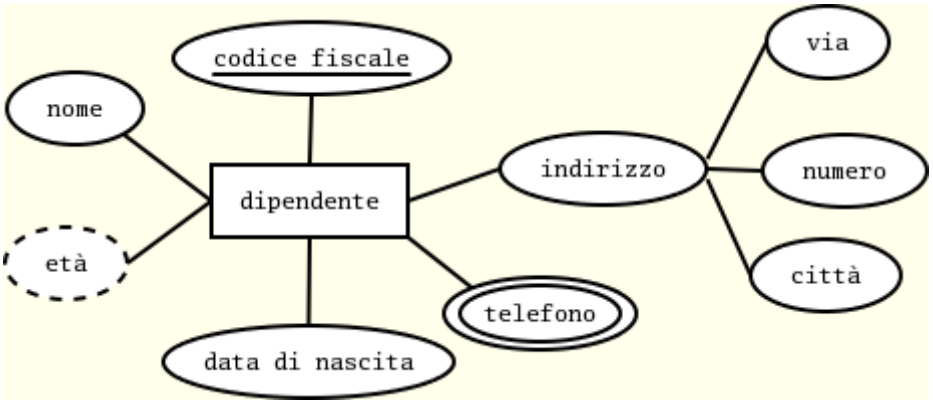
- **univocità**: i valori degli attributi identificano univocamente le istanze dell'entità *nel micro-mondo modellato*. Vale a dire che nell'universo del discorso dell'applicazione non possono esistere due istanze diverse con lo stesso valore per la chiave;
- **minimalità**: rimuovendo un qualsiasi attributo dall'insieme perdiamo il requisito di univocità.

Ogni insieme di attributi che verifica la prima proprietà è detto **superchiave**. Una chiave è dunque una superchiave minimale. E' possibile che esistano più chiavi per una entità. Tali chiavi sono dette **chiavi candidate**. Tra queste occorre sceglierne una che è detta **chiave primaria**. Esistono dei criteri di decisione per la scelta della chiave primaria:

- gli attributi opzionali non possono far parte della chiave primaria. Dunque chiavi che contengono attributi opzionali non possono diventare primarie;
- sono preferibili, per motivi di efficienza, chiavi piccole, cioè con pochi attributi;
- è preferibile, per motivi di efficienza, scegliere chiavi con attributi che vengono utilizzate da molte operazioni o da operazioni molto frequenti;

Se non si riesce a trovare una chiave che soddisfa questi requisiti è possibile introdurre un ulteriore attributo *sintetico* che serve solo ad identificare le istanze dell'entità. Questi attributi sono solitamente detti **codici**.

Facciamo un esempio. In un opportuno micro-mondo, uno studente può essere identificato dalla chiave matricola e dalla chiave nome e cognome. Possiamo scegliere come chiave primaria la matricola che è più breve. Una superchiave di studente è matricola (o nome e cognome) unita con un qualsiasi insieme di attributi di studente. Si noti che il concetto di chiave è definito *nel micro-mondo modellato*. Vedremo tra breve un esempio in cui matricola non è una chiave per studente. Nel diagramma ER, il nome gli attributi che formano la chiave primaria viene sottolineato:

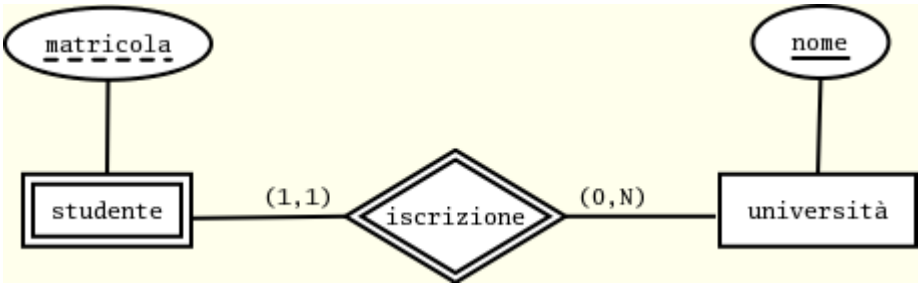


Talvolta una entità non ha una chiave ma è identificata componendo una propria **chiave parziale** con la chiave di un'altra entità alla quale è associata mediante una relazione.

L'entità priva di chiave propria si chiama **entità debole**, l'entità associata a questa si chiama **entità proprietario** e la relazione che le lega si dice **identificante**. Affinchè l'entità debole possa essere univocamente identificata, essa deve partecipare alla relazione identificante con vincolo di partecipazione (1,1). Vale a dire, per ogni istanza dell'entità debole, deve esistere esattamente una istanza dell'entità proprietario (che la identifica). Una identificazione esterna può coinvolgere più entità proprietarie. Inoltre, una entità proprietario può essere a sua volta debole purchè non si formino cicli di entità deboli.

Ad esempio, in una base di dati che modella una rete di università, vi sono le entità studente e università. Ogni studente è dotato di una matricola, che lo identifica *solo* all'interno dell'università in cui studia. Lo studente viene identificato univocamente componendo la matricola con la chiave dell'entità università (ad esempio nome). In tal caso, l'entità debole è studente, l'entità proprietaria è università, la chiave parziale di studente è matricola e la chiave di studente è matricola e nome dell'università.

Una entità debole viene rappresentata con un doppio rettangolo; la chiave parziale viene sottolineata tratteggiandola. Inoltre, la relazione che la lega all'entità proprietario viene disegnata con un doppio rombo. Segue un esempio:

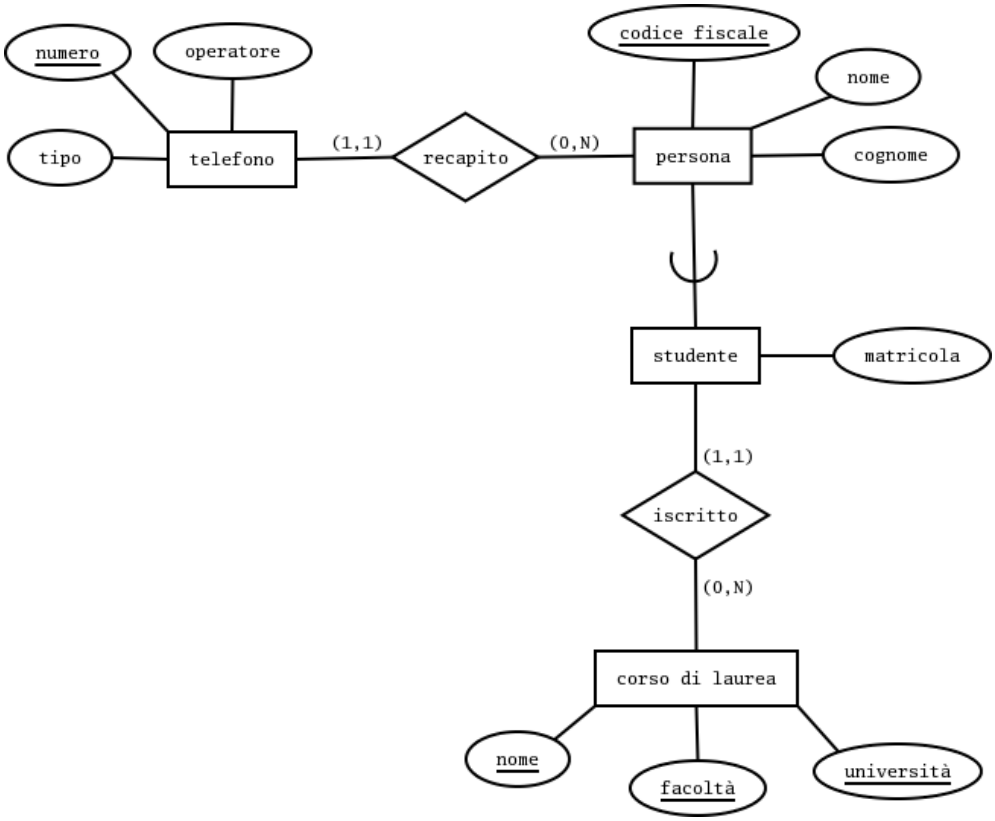


Specializzazioni

Talvolta una entità può essere specializzata in una o più sotto-entità. Le sotto-entità **ereditano** dall'entità genitore i suoi attributi e le sue relazioni. Inoltre, le sotto-entità possono definire **nuovi attributi e nuove relazioni** non presenti nell'entità genitore. L'insieme delle istanze di una sotto-entità è un sottoinsieme dell'insieme delle istanze dell'entità genitore. Il concetto di sotto-entità è affine al concetto di sotto-classe nei linguaggi di programmazione orientati agli oggetti. Talvolta, il modello ER esteso con la specializzazione prende il nome di **modello Entità-Relazione esteso** (*Enhanced Entity-Relationship Model*, EER)

Vediamo un esempio. Una persona è identificata da codice fiscale, nome e cognome e ha alcuni recapiti telefonici, di cui vuole registrare il numero, l'operatore e il tipo (fisso o mobile). Uno studente universitario è una persona per la quale si vuole memorizzare la matricola e il corso di laurea a cui è iscritto. Uno corso di laurea è identificato da nome,

facoltà e università. Possiamo rappresentare questi requisiti in questo diagramma ER esteso con la specializzazione.



L'entità studente eredita dall'entità persona gli attributi codice fiscale, nome e cognome e la relazione recapito. In particolare la chiave di studente è quella di persona, cioè codice fiscale. Inoltre, studente definisce un nuovo attributo matricola e una nuova relazione iscritto. Si noti il simbolo di sottoinsieme sull'arco che connette l'entità persona alla sotto-entità studente.

Una entità può specializzarsi in più sotto-entità. La specializzazione è **totale** se ogni istanza dell'entità genitore corrisponde ad una istanza di qualche sotto-entità, oppure **parziale** altrimenti. Una specializzazione è **disgiunta** se gli insiemi di istanze delle sotto-entità sono disgiunti, altrimenti è **sovrapposta**. Ad esempio, supponiamo di estendere il precedente diagramma con l'entità lavoratore, sotto-entità di persona. Un lavoratore è caratterizzato da una professione e da un impiego presso un'azienda. Il diagramma ER risultante è il seguente.

La lettera O all'interno del cerchio da cui partono gli archi verso le sotto-entità studente e lavoratore sta ad indicare che la specializzazione è di tipo sovrapposto (*overlapping*); infatti esistono studenti lavoratori. Per specializzazioni di tipo disgiunto si usa la lettere D (*disjoint*). La specializzazioni totali vengono disegnate con una doppia linea che connette l'entità genitore al cerchio di diramazione dei collegamenti alle sotto-entità, quelle parziali da una linea singola. Nel nostro caso, la specializzazione è parziale in quanto esistono

persone che non sono ne studenti e ne lavoratori.

Infine, una sotto-entità può avere più entità genitore. Essa eredita gli attributi e le relazioni da tutti i genitori. In tal caso occorre fare attenzione a non ereditare due volte lo stesso attributo. Questo accede se le entità genitore sono a loro volta sotto-entità di una stessa entità nonno, e quindi ereditano le stesse proprietà da quest'ultima. Inoltre occorre rinominare eventuali attributi diversi ma omonimi. Questo accede se l'insieme unione di tutti gli attributi delle entità genitore contiene attributi differenti ma omonimi. L'**ereditarietà multipla** generalmente non viene ammessa per le classi dei linguaggi orientati agli oggetti.

Estendiamo il precedente diagramma con l'entità studente-lavoratore, sotto-entità di studente e di lavoratore. Per uno studente lavoratore viene specificata una percentuale di rivalutazione del voto di laurea, in funzione dell'intensità dell'impegno lavorativo. Otteniamo il seguente diagramma ER.

Uno studente lavoratore eredita gli attributi e le relazioni di persona (una sola volta), quelli di studente e quelli di lavoratore. Inoltre definisce un nuovo attributo rivalutazione.

Il modello ER può essere descritto in termini di sè stesso mediante il seguente diagramma ER. Si noti che l'informazione sulla chiave è stata modellata mediante un attributo chiave della relazione appartenenza-E che associa gli attributi alle entità corrispondenti. Possiamo immaginare che, per ogni coppia entità-attributo della relazione tale attributo abbia valore vero se e soltanto se l'attributo fa parte della chiave dell'entità.

Documentazione e regole aziendali

E' bene che il diagramma ER sia accompagnato da una **documentazione** che lo descrive in tutte le sue componenti. Tale documentazione è utile in quanto, ad esempio, il significato dei concetti presenti nel diagramma spesso richiede una spiegazione che va oltre il nome del concetto. Inoltre, può accadere che per mantenere la leggibilità del diagramma alcuni costrutti (tipicamente attributi) vengano omessi dal diagramma.

Inoltre, come ogni formalismo, anche il modello concettuale ER ha una espressività limitata. Ne deriva che esistono vincoli della realtà modellata che non possono essere espressi con il modello ER. Ad esempio, non sono codificabili nel modello ER la regola che afferma che un dipendente può partecipare al consiglio di amministrazioni di un teatro solo dopo 10 anni di attività e la regola che dice che un teatro non può mettere in scena più di due produzioni proprie all'interno della stessa stagione teatrale. Inoltre, il diagramma ER dice quali sono gli attributi calcolati ma non specifica come calcolarli. Il diagramma ER deve quindi essere integrato da un insieme di **regole aziendali** (*business rules*) che lo completano.

La documentazione del diagramma ER deve contenere:

- per ogni **entità**: il nome, una descrizione, la lista degli attributi, la chiave, il tipo (normale o debole) e eventuali specializzazioni;
- per ogni **relazione**: il nome, una descrizione, le entità partecipanti e le relative cardinalità, gli eventuali attributi e il tipo (normale o identificante);
- per ogni **attributo**: il nome, una descrizione, il tipo (semplice, multivalore, composto, calcolato) e il fatto che sia obbligatorio o opzionale;
- le **regole aziendali**.

Proponiamo una soluzione per descrivere uno schema concettuale basata su XML. Particolari applicazioni (linguaggi) XML possono essere definite nel linguaggio Document Type Definition (DTD). Proponiamo di seguito una DTD per descrivere uno schema concettuale basato sul modello ER:

```
<!ELEMENT er      (entità | relazione | regola)*>
<!ATTLIST er      diagramma CDATA #IMPLIED>

<!ELEMENT entità  (descrizione?, attributo*, specializzazione*)>
<!ATTLIST entità  nome ID #REQUIRED
                  tipo (normale | debole) #REQUIRED
                  relazioni IDREFS #IMPLIED>

<!ELEMENT relazione  (descrizione?, partecipazione, attributo*)>
<!ATTLIST relazione  nome ID #REQUIRED
                     tipo (normale | identificante) #REQUIRED>

<!ELEMENT partecipazione  (partecipante, partecipante, partecipante*)>
<!ELEMENT partecipante EMPTY>
<!ATTLIST partecipante entità IDREF #REQUIRED
                       cardinalitàMin CDATA #REQUIRED
                       cardinalitàMax CDATA #REQUIRED
                       ruolo CDATA #IMPLIED>

<!ELEMENT attributo  (nome, descrizione?)>
<!ATTLIST attributo  tipo (semplice | multivalore | composto | calcolato) #REQUIRED
                     opzionale (si | no) #REQUIRED
                     chiave (si | no) #REQUIRED>

<!ELEMENT specializzazione EMPTY>
<!ATTLIST specializzazione  sotto-entità IDREFS #REQUIRED
                            tipo (TO | TD | PO | PD) #REQUIRED>

<!ELEMENT descrizione (#PCDATA)>
<!ELEMENT nome        (#PCDATA)>
<!ELEMENT regola      (#PCDATA)>
```

La soluzione basata su XML numerosi vantaggi, fra questi:

1. non necessita di alcun DBMS;
2. la documentazione può essere interrogata mediante linguaggi di interrogazione per XML, quali XPath e XQuery;
3. la documentazione può essere validata rispetto allo schema DTD definito. Inoltre, è facile estendere il processo di validazione con delle procedure che verificano vincoli ulteriori, non catturati dalla DTD proposta. Ad esempio, il fatto che non esistano cicli di entità deboli;

4. la documentazione può essere trasformata, mediante XSLT, in altri formati, quali LaTeX e HTML.

Progettazione delle transazioni

Questa fase specifica a livello concettuale le transazioni individuate durante la fase di raccolta e analisi dei requisiti in funzione dello schema concettuale dei dati.

Occorre innanzitutto suddividere le transazioni in:

- **Transazioni di aggiornamento**. Sono operazioni di inserimento, modifica o cancellazione dei dati;
- **Transazioni di interrogazione**. Sono operazioni di reperimento dei dati;
- **Transazioni miste**. Sono operazioni che aggiornano e reperiscono i dati.

Per ogni transazione occorre specificare: **dati di input**, **dati di output** e **comportamento funzionale**dell'operazione. Questa specifica deve essere indipendente dal sistema di basi di dati che verrà impiegato. In particolare, il comportamento funzionale delle operazioni va descritto ad alto livello e non implementato in un qualche linguaggio per basi di dati.

E' molto importante fare una **verifica di fattibilità** delle transazioni sul modello concettuale dei dati prodotto. Per ogni transazione, occorre seguirne il flusso operazionale sul diagramma ER e verificare se l'operazione è fattibile. In caso contrario, lo schema dei dati deve essere alterato in maniera da poter specificare la transazione tipica. Tale verifica deve essere effettuata anche per le regole aziendali.

Qualità di uno schema concettuale

Il passo finale della progettazione concettuale consiste nella verifica di qualità degli schemi concettuali prodotti per i dati e per le transazioni. Tale verifica deve concentrarsi almeno sulle seguenti proprietà:

Correttezza
: Uno schema è corretto quando utilizza nel giusto modo i costrutti messi a disposizione del modello.

Completezza
: Uno schema è completo quando rappresenta tutti i requisiti sui dati e quando tutte le transazioni e le regole aziendali possono essere eseguite a partire dallo schema.

Minimalità
: Uno schema è minimale quando non è possibile eliminare alcun concetto dallo schema senza per questo rinunciare alla sua completezza. Dunque uno schema non minimale contiene concetti superflui oppure ridondanti.

Una volta ottenuto uno schema concettuale di qualità è possibile passare alla fase successiva della **progettazione logica**.

Caso di studio: schema concettuale

Riprendiamo il caso di studio della rete teatrale di cui abbiamo già specificato i requisiti. L'obiettivo è produrre uno schema a livello concettuale che consista delle seguenti componenti:

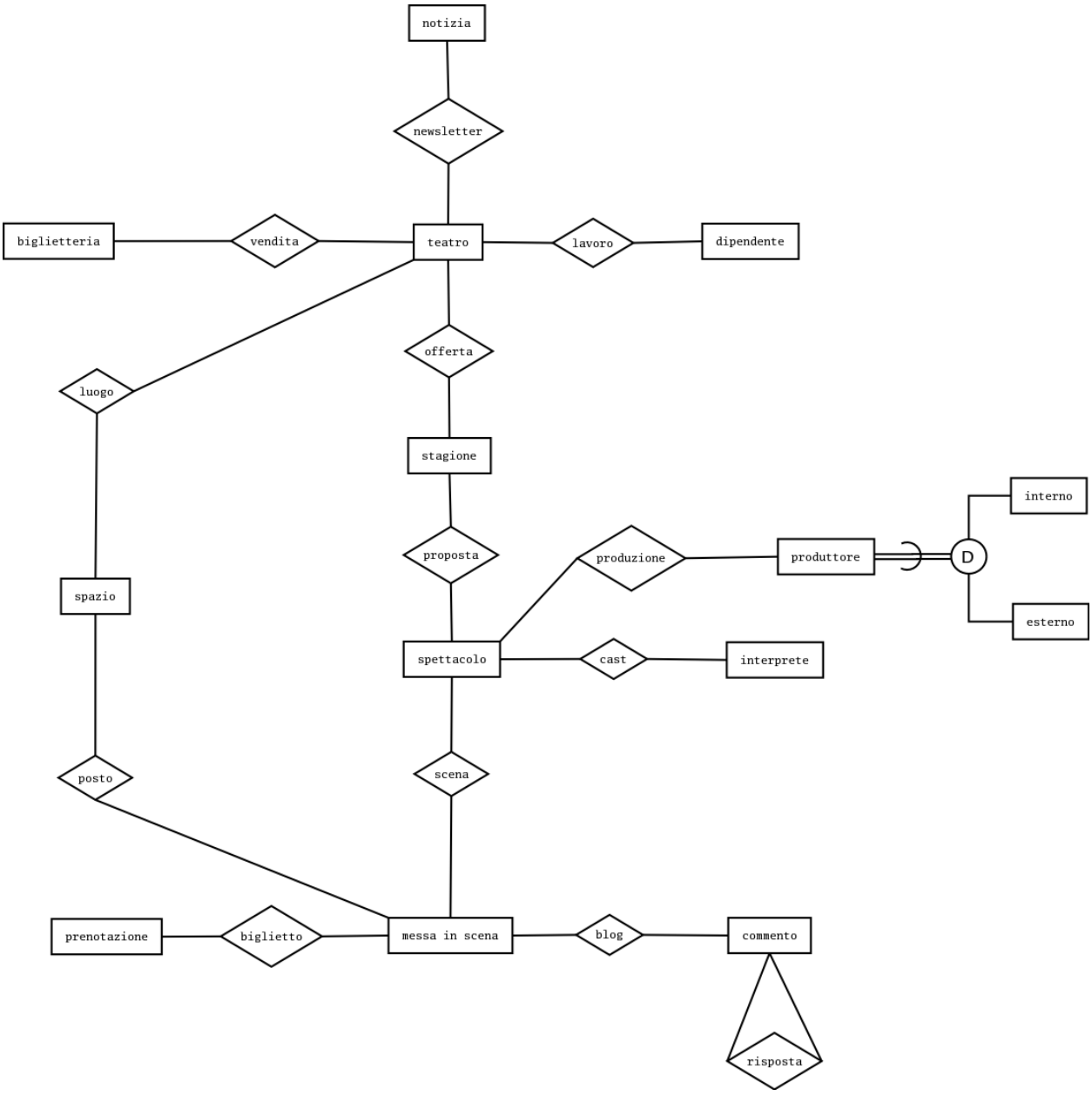
1. progetto dei dati mediante: diagramma ER, relativa documentazione e regole aziendali;
2. progetto e verifica delle transazioni e delle regole aziendali.

Il diagramma ER

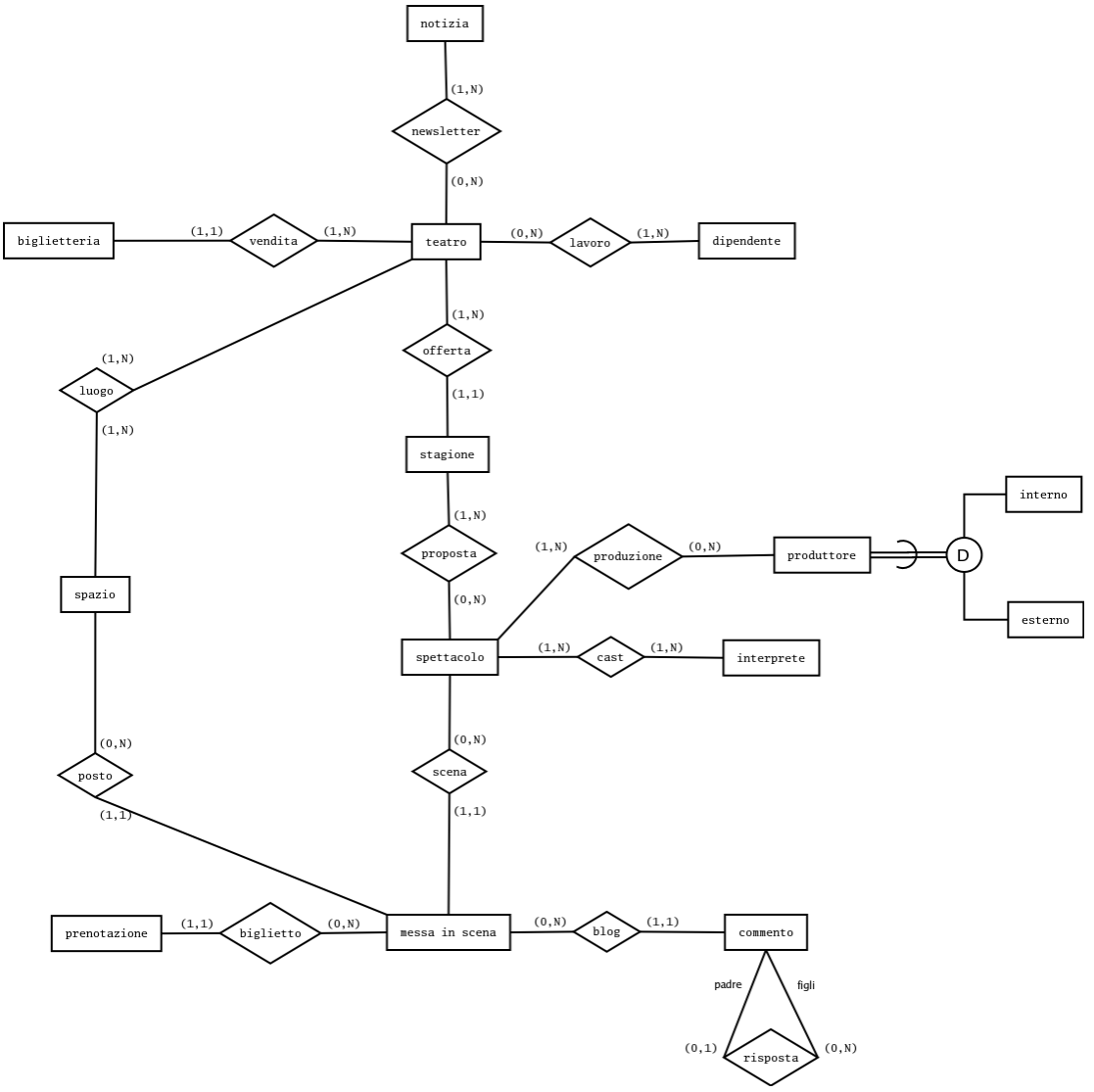
La strategia che seguiremo per la traduzione dei requisiti del caso di studio in un diagramma ER è costituita dalle seguenti fasi:

1. identificare le **entità** e le **relazioni** che le collegano.
2. aggiungere i **ruoli** e le **cardinalità delle relazioni**;
3. aggiungere gli **attributi** e il loro tipo per le entità e per le relazioni. In questa fase inoltre vengono determinate le **chiavi**, identificate le **entità deboli** e le corrispondenti **relazioni identificanti**;

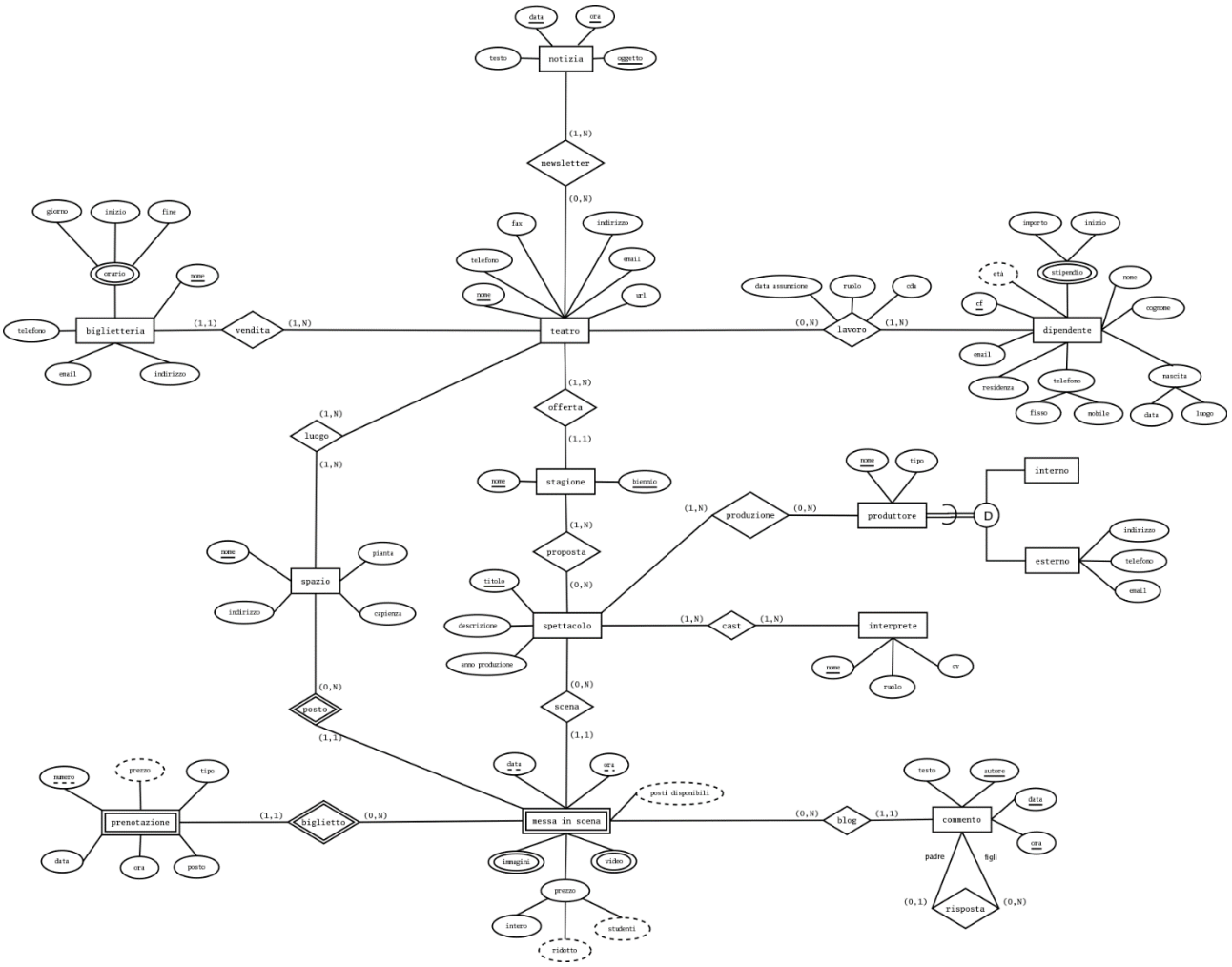
La **prima fase** serve per definire lo *scheletro* del diagramma ER finale. Tale scheletro definisce solo le entità e le loro relazioni. In particolare, lo scheletro non contiene le seguenti componenti del modello ER: il tipo delle entità (normali o deboli) e delle relazioni (normali o identificanti), i ruoli delle relazioni ricorsive non simmetriche, le cardinalità delle relazioni e gli attributi. Solitamente si procede a *macchia d'olio*, cioè partendo da una entità, possibilmente rilevante (ad esempio teatro o spettacolo nel nostro caso di studio), e si procede scoprendo tutte le entità collegate all'entità radice mediante una qualche relazione. Quindi si ripete il lavoro per tutte le nuove entità scoperte finchè non vi sono più entità e relazioni da introdurre. Il risultato della prima fase per il nostro caso di studio è il seguente scheletro del diagramma ER.



Durante la **seconda fase** si aggiungono le cardinalità delle relazioni. A tal fine, se vi sono relazioni ricorsive non simmetriche (come la relazione risposta dell'entità commento), occorre definire i ruoli di partecipazione delle entità. Il risultato della seconda fase per il nostro caso di studio è il seguente scheletro del diagramma ER.



La **terza fase** completa il diagramma con gli attributi e il loro tipo (semplice, composto, multivalore, derivato). Inoltre, per ogni entità, vengono selezionati gli attributi chiave. Se una entità non ha una chiave, occorre associarla ad una entità proprietario mediante una relazione identificate di tipo uno a uno ed eventualmente selezionare una chiave parziale tra i suoi attributi. Il risultato della terza fase per il nostro caso di studio è il seguente diagramma ER.



La strategia in tre fasi sopra delineata può essere eseguita in maniera differente: si identifica una entità radice. Quindi, per ogni relazione che collega questa entità ad altre entità, si procede localmente con la seconda e la terza fase della strategia. Vale a dire si aggiungono le cardinalità per la relazione individuata e gli attributi per la relazione e le entità coinvolte. Inoltre si determinano le chiavi e eventualmente le entità deboli e proprietario. Quindi si procede a macchia d'olio con le nuove entità scoperte fino ad ottenere lo schema completamente specificato.

Documentazione e regole aziendali

Discutendo della documentazione del diagramma ER abbiamo proposto di usare a tale scopo il linguaggio XML e abbiamo fornito una corrispondente DTD. Qui mostriamo un estratto della documentazione in formato XMLconforme alla DTD proposta. L'estratto contiene le entità teatro e dipendente e la relazione lavoro che le lega.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE er SYSTEM "er.dtd">

<er diagramma="teatro3.jpg">
  <entità nome="teatro" tipo="normale" relazioni="lavoro">
    <descrizione>Un teatro della rete</descrizione>
    <attributo chiave="si" tipo="semplice" opzionale="no">
      <nome>nome</nome>
      <descrizione>Il nome del teatro</descrizione>
    </attributo>
```

```
    <attributo chiave="no" tipo="semplice" opzionale="no">
      <nome>indirizzo</nome>
      <descrizione>L'indirizzo fisico del teatro</descrizione>
    </attributo>
    <attributo chiave="no" tipo="semplice" opzionale="si">
      <nome>email</nome>
      <descrizione>Un indirizzo di posta elettronica del teatro</descrizione>
    </attributo>
  </entità>

  <entità nome="dipendente" tipo="normale" relazioni="lavoro">
    <descrizione>Un dipendente di un teatro della rete</descrizione>
    <attributo chiave="si" tipo="semplice" opzionale="no">
      <nome>cf</nome>
      <descrizione>Il codice fiscale del dipendente</descrizione>
    </attributo>
    <attributo chiave="no" tipo="semplice" opzionale="no">
      <nome>nome</nome>
      <descrizione>Il nome del dipendente</descrizione>
    </attributo>
    <attributo chiave="no" tipo="semplice" opzionale="no">
      <nome>cognome</nome>
      <descrizione>Il cognome del dipendente</descrizione>
    </attributo>
  </entità>

  <relazione nome="lavoro" tipo="normale">
    <descrizione>Associa un teatro ai suoi dipendenti</descrizione>
    <partecipazione>
      <partecipante entità="teatro" cardinalitàMin="0"
                    cardinalitàMax="N"></partecipante>
      <partecipante entità="dipendente" cardinalitàMin="1"
                    cardinalitàMax="N"></partecipante>
    </partecipazione>
    <attributo tipo="semplice" opzionale="no">
      <nome>assunzione</nome>
      <descrizione>La data di assunzione del dipendente nel teatro</descrizione>
    </attributo>
    <attributo tipo="semplice" opzionale="no">
      <nome>ruolo</nome>
      <descrizione>Il ruolo del dipendente nel teatro</descrizione>
    </attributo>
    <attributo tipo="semplice" opzionale="no">
      <nome>cda</nome>
      <descrizione>Indica se il dipendente fa parte del CDA del teatro</descrizione>
    </attributo>
  </relazione>

  <regola>
    I dipendenti con almeno 10 anni di attività in un teatro possono
    concorrere ad un posto nel Consiglio di Amministrazione di quel teatro.
  </regola>
  <regola>
    L'età di un dipendente si calcola come differenza tra la data
    corrente e la data di nascita.
  </regola>
</er>
```

Le **regole aziendali** individuate per il nostro caso di studio solo le seguenti. Esse includono vincoli di integrità e regole di derivazione:

i. I dipendenti con almeno 10 anni di attività in un teatro possono concorrere ad un posto nel Consiglio di Amministrazione di quel teatro.
ii. Un teatro non può mettere in scena più di due spettacoli di propria produzione all'interno della stessa stagione teatrale.
iii. Una prenotazione può essere effettuata solo se vi sono ancora posti disponibili per lo spettacolo. Il numero di posti disponibili per uno spettacolo si ottiene come differenza tra la capienza dello spazio teatrale e il numero di prenotazioni effettuate.
iv. Il prezzo ridotto corrisponde all'80% prezzo intero, quello per studenti è il 50% del prezzo intero.
v. Il prezzo di una prenotazione per un biglietto di un certo tipo (intero, ridotto, studenti) è uguale al prezzo del biglietto di quel tipo fissato per lo spettacolo prenotato.
vi. L'età di un dipendente si calcola come differenza tra la data corrente e la data di nascita.

Progetto e verifica delle transazioni

Le transazioni tipiche sono state raccolte nei requisiti del sistema. A livello concettuale, occorre simulare il loro flusso di esecuzione verificandone la fattibilità. Ad esempio, consideriamo la transazione:

Generare una statistica sull'andamento delle stagioni teatrali della rete. Per ogni stagione si vuole calcolare la media degli spettatori paganti, la media dell'affluenza, cioè del rapporto tra paganti e capienza e la media degli incassi riferite agli spettacoli della stagione. Si vuole inoltra stampare gli spettacoli di una particolare stagione in ordine di: (i) paganti, (ii) affluenza, (iii) incasso.

I dati in ingresso di questa transazione di interrogazione sono una particolare stagione teatrale. E' possibile accedere ai relativi spettacoli tramite le relazioni proposta e alle corrispondenti messe in scena attraverso la relazione scena. Per ogni spettacolo, il numero di spettatori paganti si ottiene contando il numero di prenotazioni associate ad ogni messa in scena dello spettacolo. L'incasso dello spettacolo si ottiene sommando i prezzi delle prenotazioni. L'affluenza si ottiene come rapporto tra paganti e capienza dello spazio teatrale della messa in scena. Per ogni spettacolo è dunque possibile definire funzioni per il calcolo dei paganti, dell'affluenza e dell'incasso. Le medie possono dunque essere calcolate e gli spettacoli possono dunque essere ordinati secondo i valori di queste funzioni. Dunque la transazione è fattibile.

Inserire una nuova prenotazione, se possibile.

Questa è una transazione di aggiornamento che ha in ingresso una prenotazione riferita ad uno spettacolo. La prenotazione si può inserire se ci sono ancora posti a disposizione per quello spettacolo. L'attributo calcolato posti disponibili contiene il numero di posti liberi per uno spettacolo.

Trovare gli spettacoli andati in scena che sono prodotti da un teatro della rete.

Questa è una transazione di interrogazione senza dati in ingresso. Occorre restituire gli spettacoli appartenenti a qualche stagione teatrale che partecipano alla relazione produzione con un produttore interno.

Trovare tutti gli incrementi stipendiali di un certo dipendente ordinati cronologicamente.

Questa è una transazione di interrogazione con un dipendente in ingresso. Occorre trovare tutti gli stipendi raggiungibili tramite la relazione paga ordinati secondo la data di inizio dello stipendio. Ogni coppia consegutiva di stipendi corrisponde ad un incremento salariale pari alla differenza dell'importo.

Anche le **regole aziendali** dovranno essere implementate come le transazioni. Dunque è necessario verificare la loro fattibilità. Ad esempio, consideriamo la regola:

Un teatro non può mettere in scena più di due spettacoli di propria produzione all'interno della stessa stagione teatrale.

Quando una stagione teatrale di un teatro viene inserita (o modificata) occorre verificare che non vi siamo più di due spettacoli che compaiono come produzioni di quel teatro. E' possibile controllare se uno spettacolo è prodotto da un teatro della rete verificando se esso partecipa alla relazione produzione con un produttore interno.

I dipendenti con almeno 10 anni di attività in un teatro possono concorrere ad un posto nel Consiglio di Amministrazione di quel teatro.

Quando un dipendente entra nel CDA di un teatro occorre verificare che la sua data di assunzione in quel teatro sia almeno 10 anni fa. Questo è fattibile in quanto esiste l'attributo data di assunzione della relazione lavoro.

Verifica di qualità

Uno schema di qualità è corretto, completo e minimale. Un metodo per verificare la **correttezza** dello schema consiste nel validare la documentazione dello schema in formato XML rispetto allo schema logico corrispondente. Ad esempio, la DTD che abbiamo fornito impone che una relazione abbia almeno due entità partecipanti. Il modello ER infatti non ammette relazioni unarie. Una relazione unaria genererebbe un errore in fase di validazione del documento XML. Eventuali vincoli di correttezza non esprimibili nel modello logico per XML possono essere verificati mediante un validatore apposito.

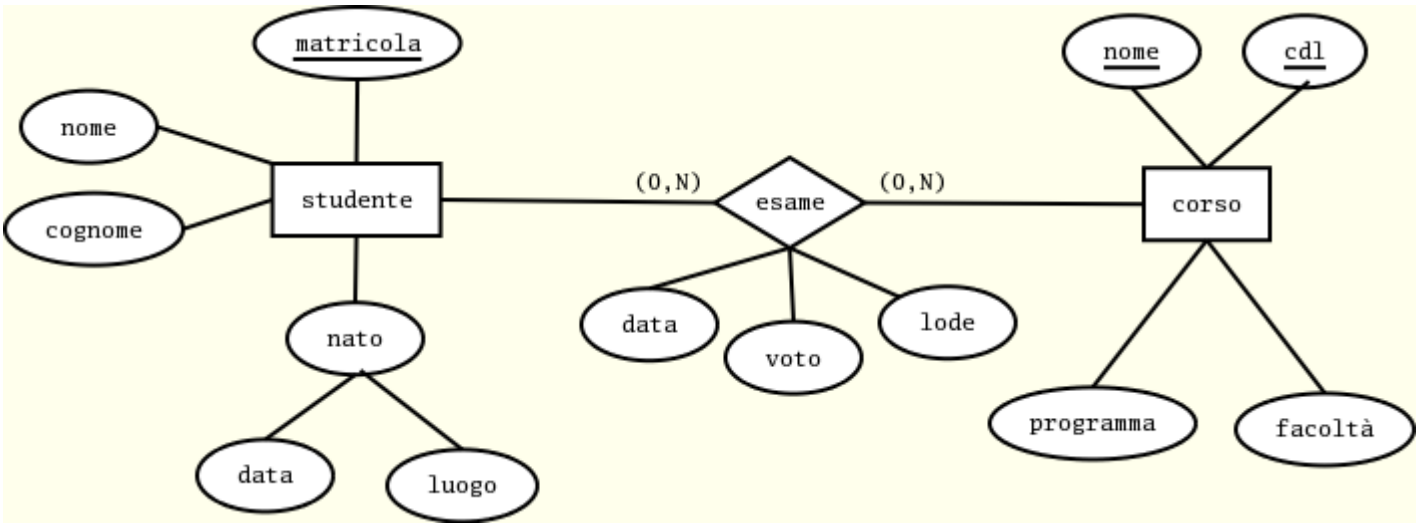
La verifica di **completezza** può essere effettuata controllando che tutti i concetti espressi nei requisiti dei dati trovino un posto nel diagramma ER, nella sua documentazione o nelle regole aziendali associate. Inoltre tutte le transazioni e le regole aziendali debbone essere fattibili sul diagramma ER proposto.

Infine occorre verificare che lo schema ER sia **minimale**, cioè non contenga informazione superflua o ripetuta.

Esercitazione

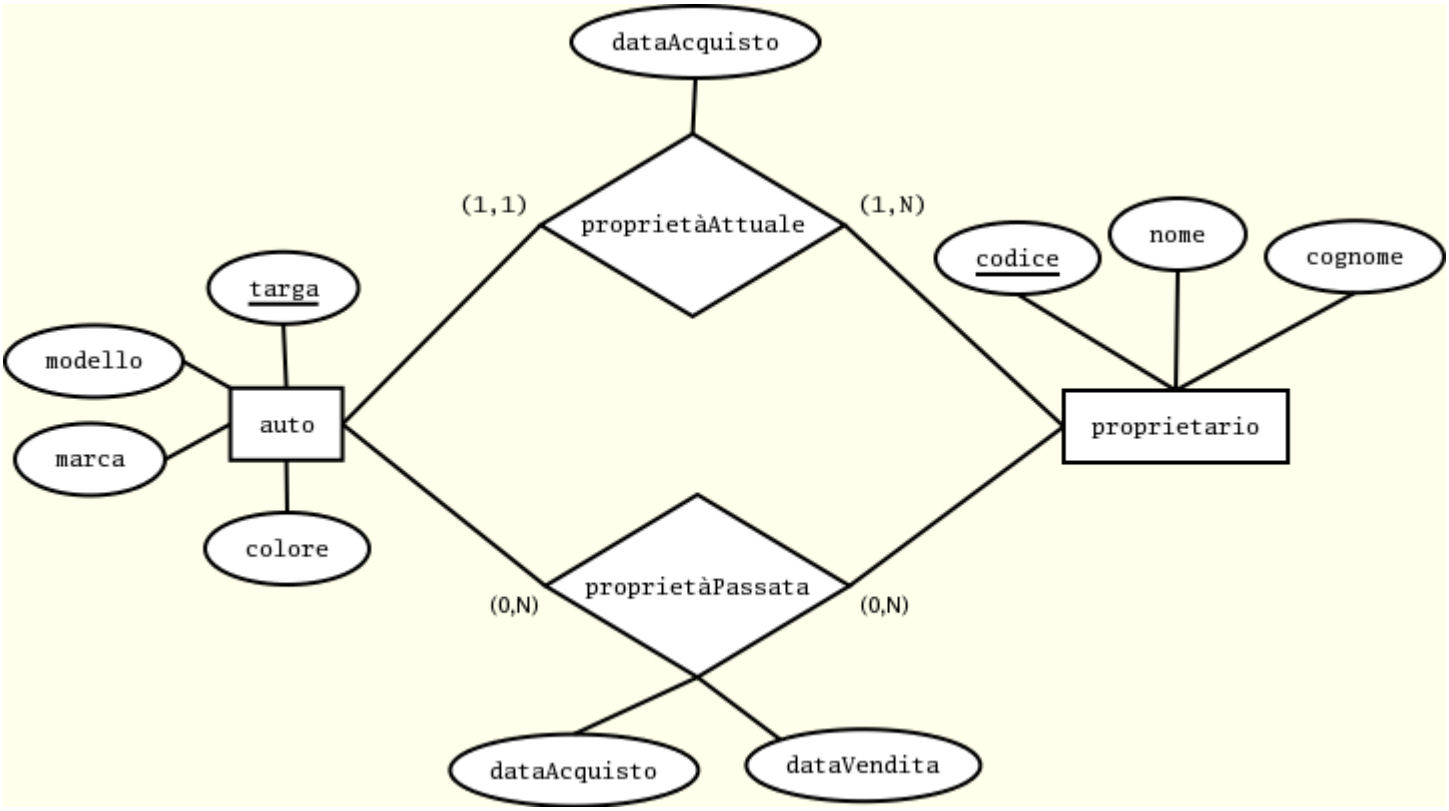
Esercizio 1. Disegnare un diagramma ER che rappresenti i seguenti requisiti:

Uno studente universitario è identificato da una matricola e viene descritto da un nome, cognome, data e luogo di nascita. Per un corso universitario si vogliono registrare il nome, un programma, il corso di laurea e la facoltà di appartenenza. Il nome del corso e il relativo corso di laurea identificano univocamente un corso. Per ogni corso occorre registrare gli studenti che hanno superato il relativo esame, la data e il voto (con eventuale lode) dell'esame.



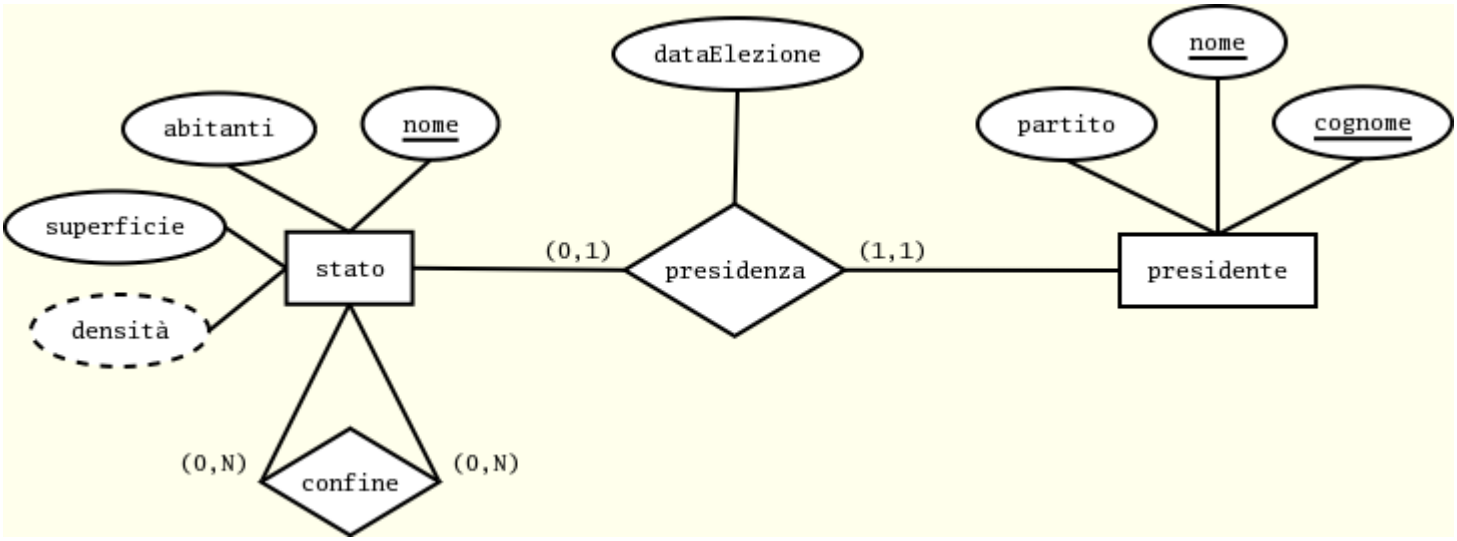
Esercizio 2. Disegnare un diagramma ER che rappresenti i seguenti requisiti:

Una automobile è descritta da una targa, un modello, una marca, e un colore. Ogni auto ha un unico proprietario attuale, descritto da codice fiscale, nome e cognome. Un proprietario può possedere più automobili. Si vuole tener traccia anche dei proprietari passati di un'auto e dell'intervallo temporale in cui hanno avuto la proprietà dell'auto.



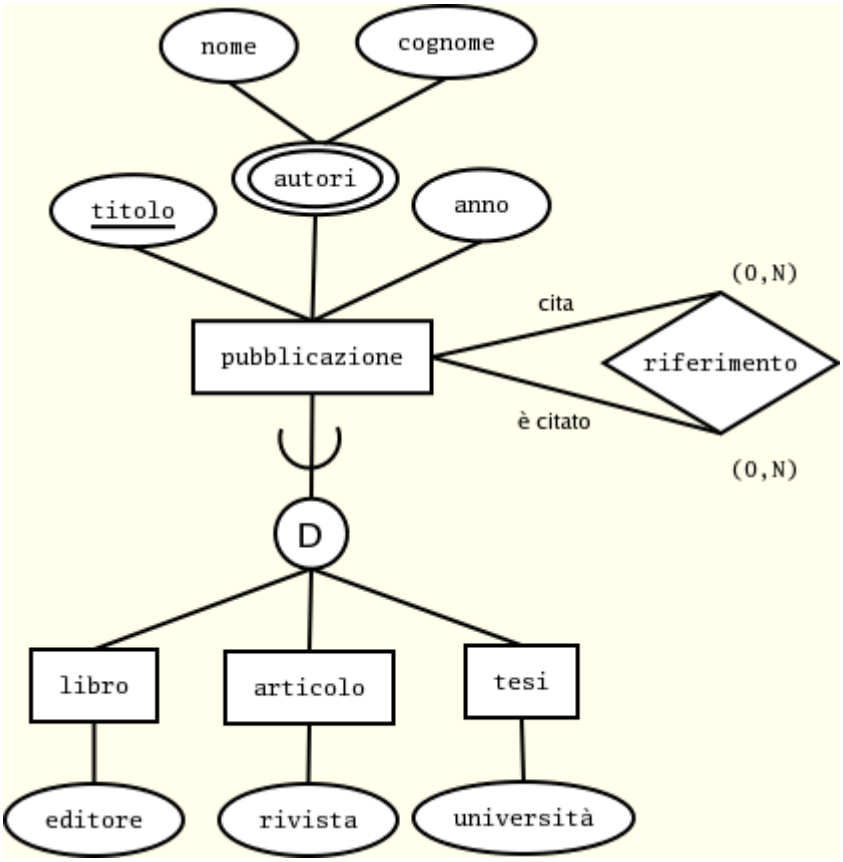
Esercizio 3. Disegnare un diagramma ER che rappresenti i seguenti requisiti:

Uno Stato ha un nome, un numero di abitanti, una superficie e una densità. Si registrano anche gli Stati confinanti. Ogni Stato ha un presidente, di cui si conosce il nome, il cognome e l'eventuale partito politico di appartenenza e la data di elezione.



Esercizio 4. Disegnare un diagramma ER che rappresenti i seguenti requisiti:

Una pubblicazione ha un titolo, una lista di autori, di cui si conoscono solo il nome e il cognome, e una data di pubblicazione. Una pubblicazione può contenere dei riferimenti bibliografici ad altre pubblicazioni. Una pubblicazione può appartenere ad una ed una sola delle seguenti categorie: libro, di cui si conosce l'editore, articolo su rivista, di cui si conosce il nome della rivista, tesi di laurea, di cui si conosce l'università.



Esercizio 5. Si vuole progettare una base di dati per memorizzare le informazioni relative ai corsi offerti nell'anno accademico corrente dai corsi di laurea dalle diverse facoltà presenti in una università. I requisiti della base di dati sono i seguenti:

*Una **facoltà** è caratterizzata da un nome, che la identifica, una o più **sedi** e un preside, che deve essere un docente ordinario della facoltà. Ogni sede è descritta da un indirizzo e da un numero di telefono e può ospitare più facoltà.*

*Una facoltà offre almeno un **corso di laurea**. Un corso di laurea è descritto da un nome, da una descrizione degli obiettivi e finalità del corso e da un presidente, che deve essere un docente ordinario della facoltà relativa al corso di laurea.*

*Un corso di laurea offre più **corsi** insegnati nell'anno accademico corrente. Ogni corso è identificato, all'interno del corso di laurea di appartenenza, da un nome e possiede una descrizione, un programma e un anno di corso. Alcuni corsi di un dato corso di laurea possono essere mutuati da uno o più corsi appartenenti al altri corsi di laurea. Per ogni corso, si vuole registrare l'orario delle lezioni. Ogni lezione si tiene in una fascia oraria, in un giorno della settimana, e in un'aula. Non vi possono essere collisioni di orario tra corsi dello stesso corso di laurea insegnati allo stesso anno di corso. Naturalmente, non vi possono essere più corsi insegnati nella stessa fascia oraria dello stesso giorno e nella stessa aula.*

*Un corso viene insegnato da un unico **docente**. E' possibile inserire corsi scoperti, cioè che non hanno ancora un docente. Ogni docente afferisce ad una facoltà, è identificato da un codice fiscale ed è descritto da nome, cognome, indirizzo di posta elettronica, sito personale, numero di telefono e ufficio. Inoltre, ogni docente ha una qualifica che può essere ricercatore, associato e ordinario. Docenti ricercatori non possono tenere corsi, docenti associati devono insegnare almeno due corsi e docenti ordinari devono insegnare almeno tre corsi. Inoltre, per i soli docenti ricercatori si vuole registrare se essi sono confermati in ruolo o meno.*

*Uno **studente** è identificato da una matricola e descritto da un nome, cognome e data di immatricolazione. Ogni studente è iscritto presso un corso di laurea e redige un piano di studi formato da corsi offerti dal proprio corso di laurea.*

Si richiede di:

1. produrre un corrispondente diagramma ER;
2. identificare le regole aziendali dell'applicazione;
3. documentare lo schema concettuale usando XML (il documento deve risultare valido rispetto alla DTD proposta);
4. ideara alcune transazioni tipiche e verificarne la fattibilità.

Segue un possibile diagramma ER per l'esercizio proposto (per semplicità sono stati aggiunti solo gli attributi chiave). Con riferimento a questo diagramma ER sono state identificate le seguenti regole aziendali:

1. il preside di una facoltà deve afferire alla facoltà;
2. il presidente di un corso di laurea deve afferire alla facoltà che contiene il corso di laurea;
3. un corso può mutuare su o essere mutuato da altri corsi solo se tali corsi appartengono ad altri corsi di laurea;
4. non vi possono essere collisioni di orario tra corsi dello stesso corso di laurea insegnati allo stesso anno di corso;
5. un corso deve essere tenuto da al più un docente;
6. uno studente può inserire nel proprio piano di studi solo corsi offerti dal proprio corso di laurea.

Progettazione logica

Prima di iniziare la progettazione logica occorre scegliere il tipo di base di dati da utilizzare, cioè il **modello dei dati** che si vuole adottare. In questo corso assumiamo che la scelta ricada sul modello relazionale dei dati su cui si fondano le basi di dati relazionali.

La progettazione logica della base di dati consiste nella traduzione dello schema concettuale dei dati in uno schema logico che rispecchia il modello dei dati scelto, cioè, nel nostro caso, il **modello relazionale**. Lo schema logico risultante è indipendente dallo specifico DBMS che verrà scelto al termine della progettazione logica. Inoltre vengono definiti i **vincoli di integrità** sui dati. Infine durante la progettazione logica si definiscono eventuali schemi esterni (**viste**) per le specifiche applicazioni.

Il modello relazionale

Il modello relazionale è oggi quello più diffuso. Esso fu proposto nel 1970 da **Edgar F. Codd** nel suo articolo *A Relational Model for Large Shared Data Banks* apparso sulla rivista Communications of the ACM. Codd metteva in risalto i limiti dei modelli utilizzati in quegli anni (modello reticolare e gerarchico), in particolare in fatto che tali modelli non distinguessero il livello logico e quello fisico dei dati. In tali modelli l'accesso ai dati avveniva sfruttando la rappresentazione fisica dei dati, ad esempio l'indirizzo di memoria di un certo dato. Non era dunque possibile cambiare la rappresentazione fisica dei dati senza necessariamente cambiarne i metodi di accesso ai dati.

Il modello relazionale risponde al requisito dell'**indipendenza fisica dei dati**. L'accesso ai dati avviene a livello logico astraendosi dalla loro rappresentazione fisica. L'affermazione del modello relazionale è stata però lenta, in quanto la proprietà di indipendenza fisica dei dati rese difficile una implementazione efficiente delle basi di dati relazionali. Solo verso la metà degli anni 80, quindi dopo 15 anni rispetto alla proposta di Codd, le basi di dati relazionali sono diventate competitive e quindi usate su larga scala.

Il modello relazionale può essere suddiviso in due componenti principali:

- le **strutture** che permettono di organizzare i dati;
- i **vincoli di integrità** che permettono di inserire solo dati corretti per la realtà modellata.

Strutture relazionali

Il modello relazionale si fonda sul concetto di **relazione** (da non confondere con la relazione concettuale del modello ER), la cui rappresentazione è una **tabella**. Il concetto di relazione è **formale** e proviene dalla teoria degli insiemi, una parte della matematica. Il concetto di tabella è **intuitivo** ed è usato in vari contesti che prescindono dalle basi di dati. Il successo delle basi di dati relazionali sta appunto nella congiunzione di un concetto formale, la relazione, che ha permesso lo sviluppo di una teoria delle basi di dati relazionali con risultati di impatto pratico, con un concetto intuitivo, la tabella, che ha reso comprensibile il modello relazionale anche ad utenti finali senza alcuna nozione matematica. L'approccio seguito in questo capitolo sarà di tipo *intuitivo* e non formale. Un buon testo per la teoria delle basi di dati relazionali è l'articolo *Elements of Relational Database Theory* di Paris C. Kanellakis.

Consideriamo un esempio di relazione che descrive gli orari dei treni rappresentata dalla tabella che segue:

orario

treno	città di partenza	ora di partenza	città di arrivo	ora di arrivo
IC129	Udine	19:00	Milano	22:30
IR567	Milano	11:40	Roma	17:30
ES21	Roma	8:40	Napoli	10:30

Le seguenti proprietà caratterizzano il modello relazionale (useremo i termini relazione e tabella in modo intercambiabile):

- una relazione è composta da **righe** e da **colonne**. Le colonne, dette anche **attributi**, hanno un nome che le identifica. Tali nomi devono essere distinti uno dall'altro all'interno della tabella. Ogni riga (oltre l'intestazione) è detta **tupla** e contiene un dato per ogni attributo della tabella. Ogni tupla corrisponde ad un elemento della relazione. Gli elementi di una tupla sono dati *in relazione* tra loro. Ad esempio, il primo elemento di ogni tupla si riferisce ad un treno e il secondo alla città di partenza di quel treno. Il numero di righe di dati viene detto **cardinalità** della relazione, il numero di colonne è chiamato **grado** della relazione.
- ogni tupla della relazione è distinta dalle altre. L'ordine delle righe nella tabella non è rilevante. L'ordine delle colonne nella tabella non è significativo in quanto le colonne sono identificate dai corrispondenti attributi. Dunque **scambiando righe e colonne non cambia la relazione** rappresentata dalla tabella;

- ogni attributo della relazione viene associato un **dominio**, cioè un insieme di valori. Gli attributi possono assumere valori solo nel corrispondente dominio. Ad esempio, l'attributo treno ha come dominio l'insieme delle stringhe e ora di partenza ha come dominio l'insieme delle ore ben formate. Un dominio nel modello relazionale deve contenere solo valori atomici. Un **valore atomico** è indivisibile, per lo meno per quanto riguarda il modello relazionale. Ad esempio, un valore atomico di un attributo non può essere una riga di una tabella o una tabella intera. La nozione di dominio corrisponde a quella di tipo di dato semplice nei linguaggi di programmazione;
- ogni relazione deve avere una **chiave primaria**, cioè un insieme di attributi che identificano univocamente ogni tupla della relazione. Nel caso della relazione orario, una chiave può essere la coppia treno e orario di partenza (infatti lo stesso treno non può partire da città diverse alla stessa ora).
- gli attributi possono avere un valore non noto o non esistente. In tal caso si dice che l'attributo ha **valore nullo** e si scrive nella tabella la costante NULL. Questa costante non deve appartenere a nessuno dei domini usati dalla base di dati. Gli attributi della chiave primaria non possono avere valori nulli.

In una relazione si distingue il suo schema dal suo contenuto:

- uno **schema di relazione** R(X) è formato da un simbolo R, detto nome della relazione, e da un insieme di attributi X, ognuno dei quali è associato ad un dominio.
- una **istanza di relazione** (o relazione) sullo schema R(X) è un insieme di tuple definite su X.

Riprendendo la tabella degli orari dei treni, lo schema di relazione corrisponde al nome della tabella e alla riga di intestazione e viene indicato in questo modo:

orario({treno, città di partenza, ora di partenza, città di arrivo, ora di arrivo})

Si noti che il nome degli attributi che formano la chiave primaria viene sottolineato. Per comodità le parentesi graffe che identificano l'insieme vengono di solito omesse. Inoltre è preferibile usare identificatori che non contengono spazi. Ad esempio, invece di città di partenza possiamo usare cittàDiPartenza. Una istanza della relazione orario corrisponde a un insieme (possibilmente vuoto) di tuple corrispondenti.

Solitamente, l'informazione contenuta in una base di dati viene rappresentata in più tabelle. Tali tabelle possono contenere valori comuni che servono come associazione tra dati diversi. Definiamo quindi:

- uno **schema di base di dati** è un insieme di schemi di relazione con nomi diversi. Attributi di schemi di relazione diversi possono avere lo stesso nome;
- una **istanza di base di dati** (o base di dati) su uno schema di base di dati è un insieme di istanze di relazione, una istanza per ogni schema di relazione dello schema della base di dati.

Facciamo un esempio tratto dal nostro caso di studio. Consideriamo uno schema di base di dati composto dai seguenti schemi di relazione:

teatro(nome, città, email)

dipendente(cf, nome, cognome, dataDiNascita, età)

lavoro(teatro, dipendente, ruolo)

Di seguito mostriamo una istanza di base di dati sullo schema proposto:

teatro

nome	**città**	**email**
CSS	Udine	css@gmail.com
Litta	Milano	litta@gmail.com
Eliseo	Roma	NULL

dipendente

cf	**nome**	**cognome**	**dataDiNascita**	**età**
ELSDLL72	Elisa	D'Allarche	29/04/1972	35
FRNDPP76	Fernanda	D'Ippoliti	11/03/1976	31
MRCDLL70	Marco	Dall'Aglio	09/01/1970	37

lavoro

teatro	**dipendente**	**ruolo**
CSS	ELSDLL72	relazioni
Litta	FRNDPP76	finanza
Eliseo	FRNDPP76	controllo

Eliseo	MRCDLL70	direzione

Facciamo una osservazione fondamentale. Nel modello relazionale tutta l'informazione viene rappresentata tramite relazioni che corrispondono a tabelle con attributi. Quindi, in questo modello, sia le entità che le relazioni dello schema concettuale vengono implementate in relazioni dello schema relazionale. In particolare le associazioni concettuali tra entità vengono realizzate a livello logico mediante tabelle che contengono **valori comuni** ad altre tabelle a cui fanno riferimento.

Ad esempio, la relazione concettuale lavoro che associa le entità dipendente e teatro viene rappresentata da una relazione lavoro nella quale l'attributo teatro identifica l'entità teatro e l'attributo dipendente identifica l'entità dipendente. In particolare, per dire che il dipendente ELSDLL72 (Elisa D'Allarche) lavora per il teatro CSS scriviamo questi valori nelle corrispondenti colonne della tabella lavoro. Gli attributi teatro e dipendente della relazione lavoro prendono il nome di **chiavi esterne** (*foreign keys*) in quanto fanno riferimento a chiavi di altre tabelle.

Questo modello per stabilire corrispondenze tra i dati prende il nome di **modello basato su valori**. Esso si distingue dal **modello basato su riferimenti** tipico dei modelli dei dati reticolare e gerarchico. In quest'ultimo le corrispondenze tra i dati vengono implementate da puntatori, cioè variabili che contengono indirizzi di memoria dei dati. Il modello basato su valori ha due vantaggi rispetto a quello basato su riferimenti:

1. rappresenta solo informazione rilevante per la realtà modellata;
2. l'indipendenza fisica dei dati viene mantenuta permettendo di cambiare la rappresentazione fisica dei dati (ad esempio spostando i dati in una diversa zona di memoria) senza mutare quella logica.

Vincoli di integrità

I vincoli di integrità servono per mantenere la consistenza della base di dati. Infatti, non tutte le istanze della base di dati sono lecite. Sono ammissibili solo quelle che soddisfano tutti i vincoli di integrità specificati per la base di dati. Riprendiamo lo schema di base di dati:

teatro(nome, città, email)

dipendente(cf, nome, cognome, dataDiNascita, età)

lavoro(teatro, dipendente, ruolo)

Consideriamo la seguente istanza di base di dati definita sullo schema precedente:

teatro		
nome	**città**	**email**
CSS	Udine	css@gmail.com
NULL	Milano	litta@gmail.com
Eliseo	Roma	NULL

dipendente				
cf	**nome**	**cognome**	**dataDiNascita**	**età**
ELSDLL72	Elisa	D'Allarche	31/04/1972	35
FRNDPP76	Fernanda	D'Ippoliti	11/03/1976	31
FRNDPP76	Marco	Dall'Aglio	09/01/1970	47

lavoro		
teatro	**dipendente**	**ruolo**
CSS	ELSDLL72	relazioni
Litta	FRNDPP76	finanza
Eliseo	FRNDPP76	controllo
Eliseo	MRCDLL70	direzione

I dati inconsistenti sono stati evidenziati in rosso nell'istanza che segue:

teatro		
nome	**città**	**email**
CSS	Udine	css@gmail.com

NULL	Milano	litta@gmail.com
Eliseo	Roma	NULL

dipendente				
cf	**nome**	**cognome**	**dataDiNascita**	**età**
ELSDLL72	Elisa	D'Allarche	31/04/1972	35
FRNDPP76	Fernanda	D'Ippoliti	11/03/1976	31
FRNDPP76	Marco	Dall'Aglio	09/01/1970	47

lavoro		
teatro	**dipendente**	**ruolo**
CSS	ELSDLL72	relazioni
Litta	FRNDPP76	finanza
Eliseo	FRNDPP76	controllo
Eliseo	MRCDLL70	direzione

Le situazioni inconsistenti sono le seguenti:

1. la seconda riga della tabella teatro ha una chiave primaria (nome) con valore nullo. Ciò non è possibile perchè la chiave primaria è costituita da attributi obbligatori;
2. la prima riga della tabella dipendente ha un valore non ammesso per data di nascita. Infatti non esiste il 31 Aprile;
3. la seconda e la terza riga della tabella dipendente hanno lo stesso valore per la chiave primaria (cf). Ciò è inammissibile perchè la chiave identifica le tuple.
4. la terza riga della tabella dipendente ha valori inconsistenti, rispetto alla data odierna, per data di nascita e età;
5. la seconda riga della tabella lavoro ha un valore per teatro che non corrisponde ad alcun teatro nella tabella teatro;
6. la quarta riga della tabella lavoro ha un valore per dipendente che non corrisponde ad alcun dipendente nella tabella dipendente.

Alcune delle inconsistenze evidenziate corrispondono a vincoli tipici del modello relazionale. Questi sono:

- **Vincoli di dominio**: specificano che un attributo associato ad un certo dominio deve assumere valori in quel dominio. Ad esempio, l'inconsistenza numero 2 viola tale vincolo.
- **Vincoli di chiave**: specificano che una chiave deve avere valori univoci e una chiave primaria deve avere valori univoci e non nulli. Ad esempio, le inconsistenze numero 1 e 3 violano questi vincoli.
- **Vincoli di chiave esterna**: specificano che i valori *non nulli* della chiave esterna devono corrispondere a valori della chiave riferita. Si noti che il vincolo deve essere soddisfatto solo per i valori non nulli. Inoltre, la chiave riferita dalla chiave esterna può essere primaria o candidata. Ad esempio, le inconsistenze numero 5 e 6 violano questi vincoli.

Un vincolo di chiave esterna è specificato associando una sequenza di attributi $X_1, X_2,..., X_n$, che formano la chiave esterna di una tabella T, ad una sequenza di attributi $Y_1, Y_2,..., Y_n$, che formano la chiave (primaria o candidata) di una tabella R. T e R possono essere la stessa tabella. Un tale vincolo si rappresenta in questo modo:

T($X_1, X_2,..., X_n$) --> **R**($Y_1, Y_2,..., Y_n$)

Il vincolo è soddisfatto se per ogni sequenza di valori $x_1, x_2,..., x_n$, ognuno dei quali è non nullo, per gli attributi $X_1, X_2,..., X_n$ di T esiste una uguale sequenza di valori $y_1, y_2,..., y_n$ per gli attributi $Y_1, Y_2,..., Y_n$ di R.

Ad esempio, i vincoli di chiave esterna per lo schema di sopra vengono scritti come segue:

lavoro(teatro) --> **teatro**(nome)

lavoro(dipendente) --> **dipendente**(cf)

Ulteriori vincoli di integrità possono essere specificate mediante le **regole aziendali** contenute nello schema concettuale. Ad esempio, il fatto che l'età di una persona sia calcolata come differenza tra la data odierna e la data di nascita (l'inconsistenza numero 4 viola tale vincolo). Un qualsiasi altro vincolo che regoli l'applicazione può essere asserito. Ad esempio, il fatto che un dipendente che si occupa di controllo deve avere almeno 40 anni.

Solitamente, i vincoli del modello relazionale (di dominio, di chiave, di chiave esterna) vengono controllati automaticamente dal DBMS dopo ogni aggiornamento (inserimento, modifica, cancellazione) della base di dati. Gli altri vincoli devono essere implementati dal progettista nel linguaggio di definizione dei dati o, se ciò non è possibile, in un linguaggio di programmazione.

Ad esempio, immaginiamo una applicazione che permette di inserire via Web dati in una base. Quando i dati vengono inseriti dall'utente, l'applicazione verifica i vincoli di integrità che non sono codificabili nel linguaggio di definizione dei dati del DBMS. Se tali vincoli sono soddisfatti, l'informazione viene passata al DBMS per l'inserimento nella base di dati. Prima di inserire i dati, il DBMS verifica i vincoli implementati nel linguaggio di definizione dei dati e i vincoli tipici del modello relazionale. Se anche questi vincoli sono soddisfatti, l'inserimento viene effettuato e la

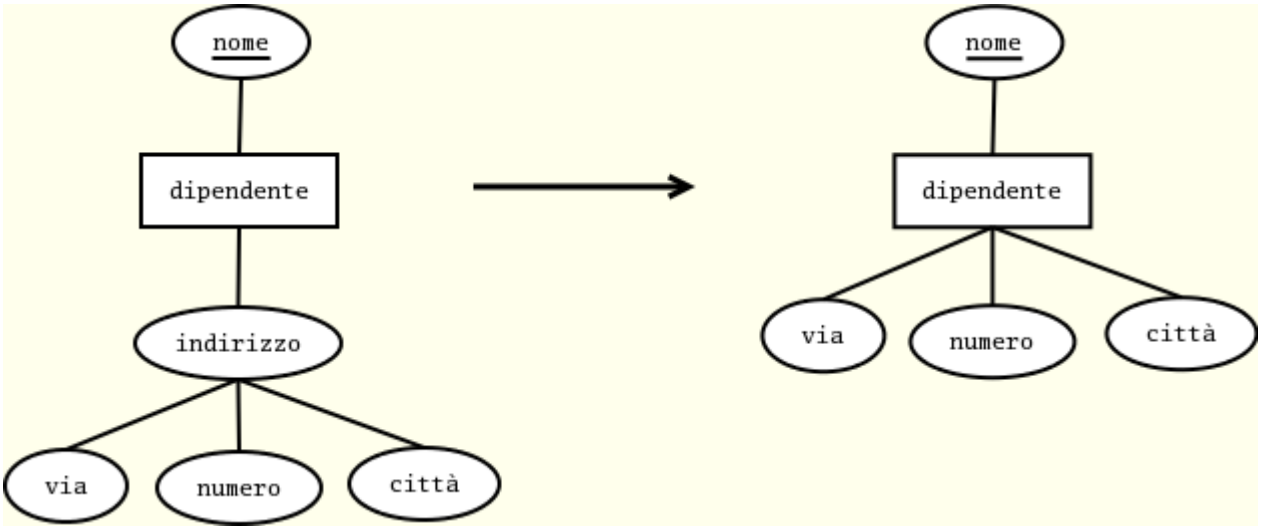
base di dati rimane in uno stato consistente. Altrimenti l'aggiornamento è rifiutato avvisando l'utente delle inconsistenze.

Traduzione del modello ER nel modello relazionale

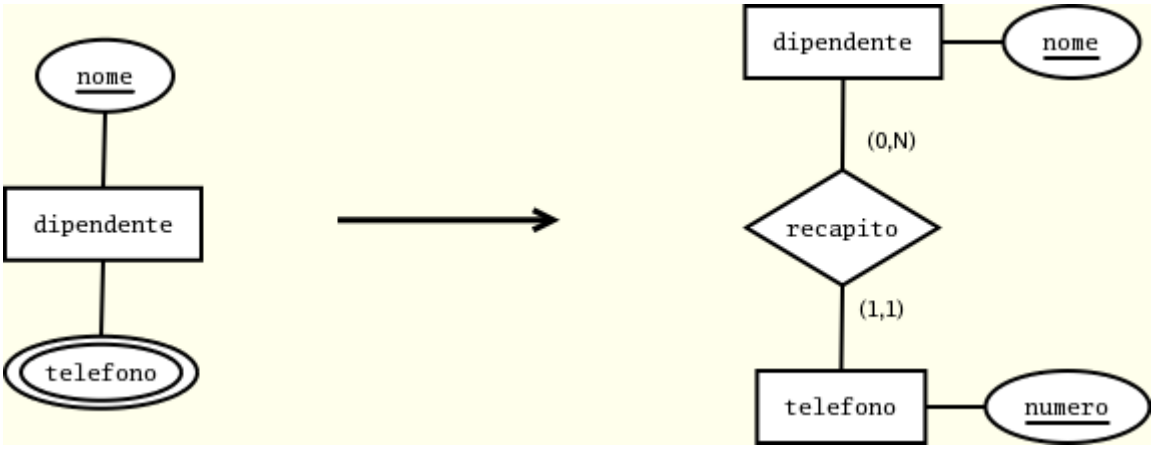
Esistono delle regole per tradurre uno schema ER in uno schema relazionale equivalente, cioè che rappresenta la stessa informazione. Questa traduzione si muove dall'astratto al concreto. Il risultato della traduzione è quindi un passo più lontano dal livello concettuale e un passo più vicino al livello fisico dei dati. In particolare, il risultato è organizzato secondo il modello dei dati che è stato scelto (il modello relazionale nel nostro caso).

Prima di iniziare la traduzione occorre notare che, mentre gli attributi del modello relazionale assumono solo valori atomici, il modello ER permette di specificare **attributi composti** (che assumono una sequenza di valori non omogenei) e **attributi multivalore** (che assumono una sequenza di valori omogenei). Occorre dunque rimuovere questi attributi mediante una fase preliminare di ristrutturazione del modello ER.

La rimozione degli attributi composti è semplice. Essi vengono sostituiti con gli attributi componenti come nel seguente caso:



Eventuali gerarchie di attributi composti si traducono similmente procedendo dalle foglie verso la radice. A questo punto tutti gli attributi sono semplici, derivati, oppure multivalore. Per rimuovere gli attributi multivalore si crea una nuova entità che contiene i valori dell'attributo e la si collega all'entità che possedeva l'attributo mediante una nuova relazione uno a molti o molti a molti, a seconda dei casi. Segue un esempio:

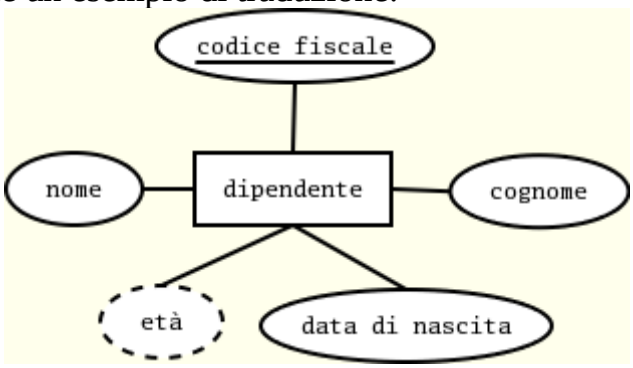


Se l'attributo telefono è opzionale, il vincolo di cardinalità (1,1) va sostituito con (0,1). Se inoltre un numero di telefono può essere condiviso da più persone (ad esempio, il telefono di casa), allora il vincolo di cardinalità (1,1) va sostituito con (0,N).

Vediamo ora quali sono le regole di traduzione assumendo che non ci siano attributi composti o multivalore. Ogni regola si applica ad un frammento di schema ER ed ha come risultato un insieme di schemi di relazione corrispondenti al frammento ER in ingresso.

Entità

Una entità viene tradotta in una relazione con lo stesso nome, gli stessi attributi e la stessa chiave dell'entità. Segue un esempio di traduzione:

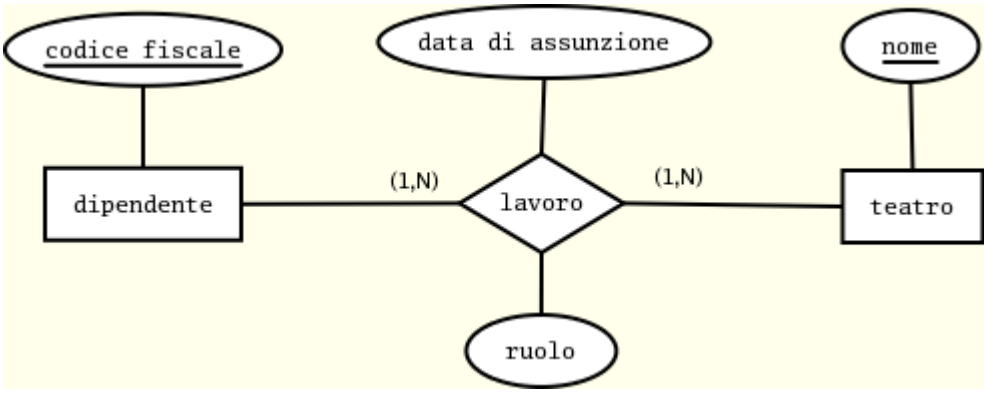


dipendente(codiceFiscale, nome, cognome, dataDiNascita, età)

Per gli attributi calcolati va aggiunto un vincolo di integrità che specifica il metodo di calcolo. Nel caso di età, il vincolo afferma che il valore di tale attributo si ottiene come differenza tra la data odierna e la data di nascita di una persona, se tale informazione esiste, altrimenti il suo valore è nullo.

Relazione molti a molti

Una relazione molti a molti tra due entità si traduce con una relazione con lo stesso nome avente come attributi le chiavi delle entità coinvolte, che formano la chiave della relazione, più eventuali attributi della relazione. Gli attributi possono eventualmente essere rinominati per maggiore chiarezza. Segue un esempio di traduzione:



dipendente(codiceFiscale)

teatro(nome)

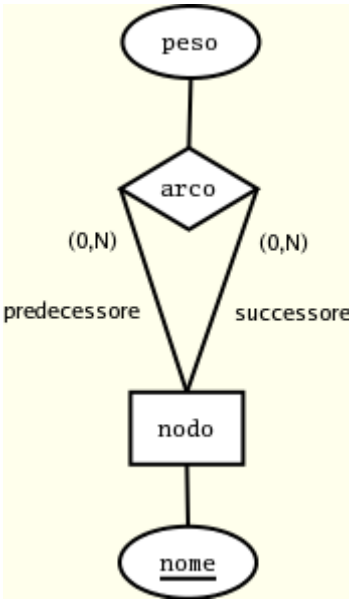
lavoro(dipendente, teatro, dataDiAssunzione, ruolo)

lavoro(teatro) --> **teatro**(nome)

lavoro(dipendente) --> **dipendente**(codiceFiscale)

Nello schema di relazione lavoro abbiamo rinominato l'attributo codice fiscale di dipendente con dipendente e l'attributo nome di teatro con teatro. Esistono inoltre vincoli di chiave esterna tra l'attributo dipendente della relazione lavoro e la relazione dipendente e tra l'attributo teatro della relazione lavoro e la relazione teatro. La traduzione mostrata rimane valida anche nel caso di cardinalità minima diversa da 1, e in particolare quando essa vale 0.

Similmente si traducono le relazioni binarie ricorsive di tipo molti a molti. In tal caso la rinominazione degli attributi è essenziale al fine di evitare attributi con lo stesso nome nello stesso schema di relazione:



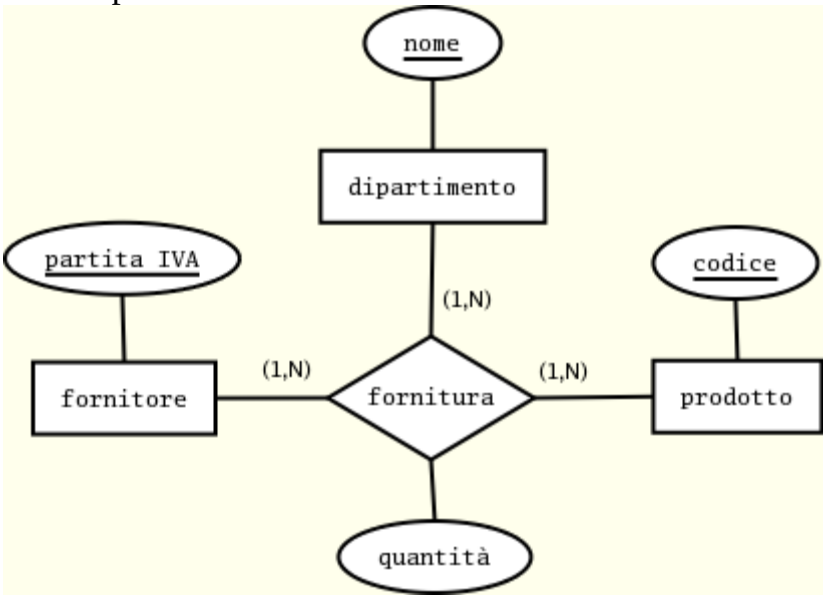
nodo(nome)

arco(predecessore, successore, peso)

arco(predecessore) --> **nodo**(nome)

arco(successore) --> **nodo**(nome)

Le relazioni con più di due entità partecipanti di tipo molti a molti si traducono in modo analogo. Segue un esempio di traduzione:



fornitore(partitaIVA)

dipartimento(nome)

prodotto(codice)

fornitura(fornitore, dipartimento, prodotto, quantità)

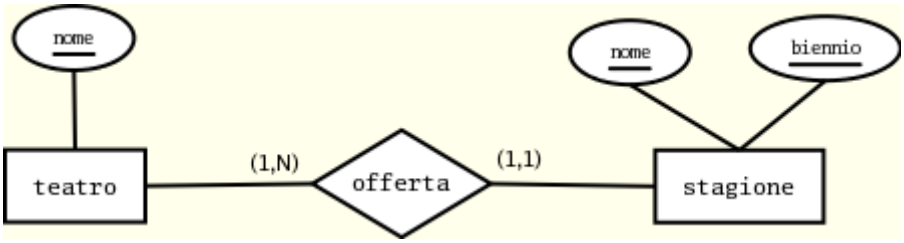
fornitura(fornitore) --> **fornitore**(partitaIVA)

fornitura(dipartimento) --> **dipartimento**(nome)

fornitura(prodotto) --> **prodotto**(codice)

Relazione uno a molti

Trattiamo solo il caso binario. Il caso con più di due relazioni di tipo uno a molti è raro e comunque traducibile in modo simile. In questo caso uno dei due vincoli di cardinalità è di tipo (1,1) oppure (0,1), mentre l'altro vincolo è di tipo (k,N), con k un naturale fissato. Se il primo vincolo di cardinalità è di tipo (1,1) la traduzione rappresenta la relazione concettuale sfruttando l'entità che partecipa alla relazione con cardinalità massima pari a uno. Segue un esempio:



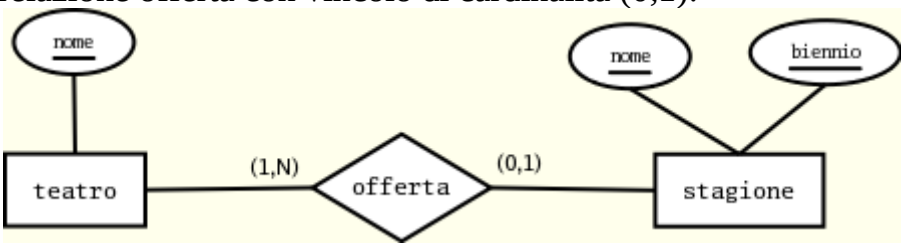
teatro(<u>nome</u>)

stagione(<u>nome</u>, <u>biennio</u>, teatro)

stagione(teatro) --> **teatro**(nome)

Si noti che in questo caso l'attributo teatro di stagione non assume mai valori nulli in quanto ogni stagione ha esattamente un teatro associato.

Supponiamo che una stagione teatrale possa non avere ancora un teatro associato e dunque partecipi alla relazione offerta con vincolo di cardinalità (0,1):



In tal caso abbiamo due possibili traduzioni. La prima è quella appena vista. La seconda, simile alla traduzione di una relazione molti a molti, è la seguente:

teatro(<u>nome</u>)

stagione(<u>nome</u>, <u>biennio</u>)

offerta(<u>nomeStagione</u>, <u>biennioStagione</u>, teatro)

offerta(nomeStagione, biennioStagione) --> **stagione**(nome, biennio)

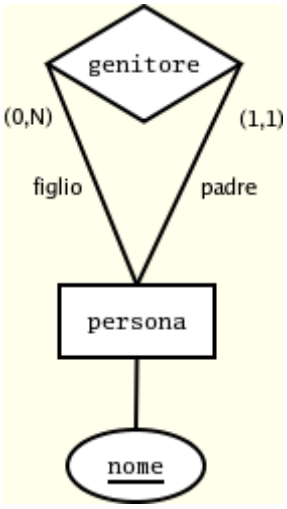
offerta(teatro) --> **teatro**(nome)

Si noti una differenza rispetto alla traduzione molti a molti: la chiave della relazione offerta è la chiave dell'entità stagione (cioè l'entità che partecipa alla relazione con vincolo massimo pari a 1). Infatti, esiste al massimo un teatro associato ad una stagione.

La prima traduzione ha il vantaggio di creare una tabella in meno. D'altronde, essa ha come svantaggio di dover inserire valori nulli per l'attributo teatro di stagione per quelle stagioni che non hanno associato un teatro. Questi valori nulli sono invece esclusi nella seconda traduzione in quanto la relazione offerta registra solo le associazioni esistenti tra stagione e teatro. Un criterio di scelta tra le due traduzioni è la valutazione del rapporto tra istanze di stagione con un teatro e istanze di stagione. Se questo rapporto è significativo (cioè se il

numero di valori nulli da inserire è scarso) allora è preferibile la prima traduzione, altrimenti è meglio la seconda.

Mostriamo ora la traduzione di una relazione binaria ricorsiva di tipo uno a molti:



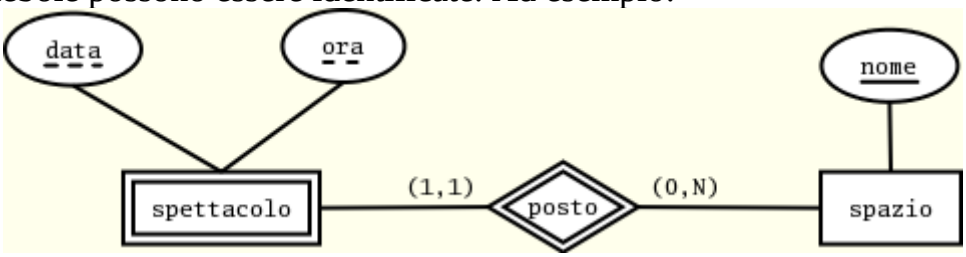
persona(<u>nome</u>, padre)

persona(padre) --> **persona**(nome)

In realtà in questo caso l'espressione chiave *esterna* non è esattamente appropriata in quanto padre fa parte della stessa tabella a cui fa riferimento.

Entità debole

La traduzione per una entità debole è simile alla prima traduzione per relazioni uno a molti. Infatti una entità debole è identificata da una entità proprietario mediante una relazione a cui partecipa con vincolo (1,1). Occorre però aggiungere alla chiave parziale della relazione dell'entità debole la chiave dell'entità proprietario, in quanto solo in questo modo le istanze dell'entità debole possono essere identificate. Ad esempio:



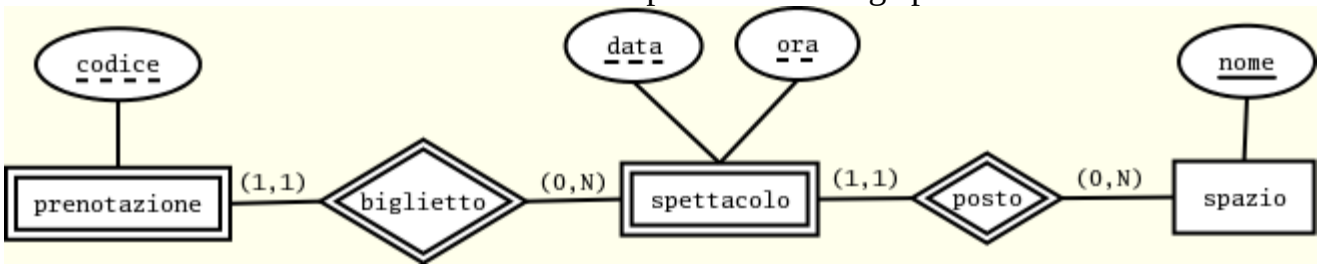
spazio(<u>nome</u>)

spettacolo(<u>data</u>, <u>ora</u>, <u>spazio</u>)

spettacolo(spazio) --> **spazio**(nome)

Se abbiamo un cammino di entità deboli E_1, E_2,..., E_n, dove le prime n-1 entità del cammino sono deboli e l'ultima è il proprietario, allora dobbiamo fare attenzione ad applicare la regola di traduzione a partire dalla fine del cammino, cioè dalla coppia (E_{n-1}, E_n), per poi proseguire all'indietro. In questo modo è possibile specificare una chiave per ognuna delle entità deboli. Si noti che tutte le entità coinvolte devono essere distinte altrimenti avremo

un ciclo di entità deboli e sarebbe impossibile l'identificazione di una chiave per le entità deboli conivolte nel ciclo. Vediamo un esempio che coinvolge più di una entità debole:



spazio(nome)

spettacolo(data, ora, spazio)

prenotazione(codice, data, ora, spazio)

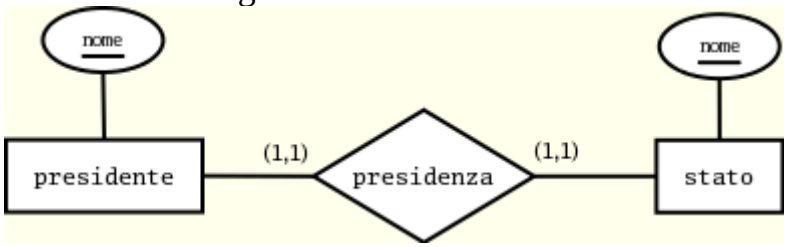
spettacolo(spazio) --> **spazio**(nome)

prenotazione(data, ora) --> **spettacolo**(data, ora)

prenotazione(spazio) --> **spazio**(nome)

Relazione uno a uno

Trattiamo solo il caso binario. Il caso con più di due relazioni di tipo uno a uno è molto raro e comunque traducibile in modo analogo. Vi sono quattro casi a seconda delle cardinalità minime (0 o 1) delle due entità partecipanti. Supponiamo che la partecipazione delle entità alla relazione sia obbligatoria:



Abbiamo due possibulità equivalenti. La prima è questa:

stato(nome)

presidente(nome, stato)

presidente(stato) --> **stato**(nome)

La seconda possibilità è simmetrica:

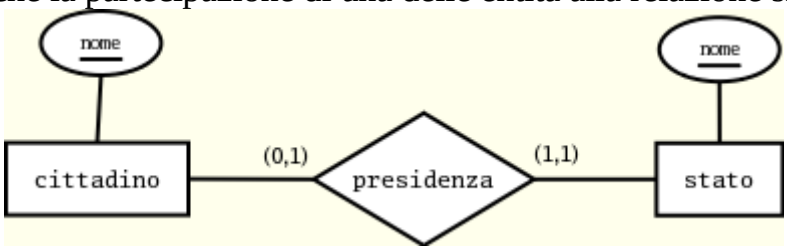
presidente(nome)

stato(nome, presidente)

stato(presidente) --> **presidente**(nome)

Dato che la relazione presidenza è biunivoca, una terza possibilità è quella di usare una unica relazione (presidente o stato) che raggruppa tutti e tre i concetti del frammento ER. Questa soluzione è tecnicamente giusta ma metodologicamente sbagliata. Infatti, a livello concettuale, i concetti di stato e presidente sono stati separati (per qualche motivo) e tale separazione deve persistere anche a livello logico.

Supponiamo ora che la partecipazione di una delle entità alla relazione sia opzionale:



La traduzione da preferire è la seguente:

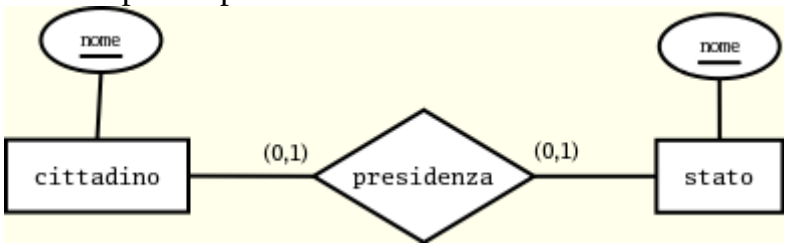
cittadino(<u>nome</u>)

stato(<u>nome</u>, presidente)

stato(presidente) --> **cittadino**(nome)

Questa traduzione ha il vantaggio che non vengono inseriti valori nulli nelle relazioni.

Supponiamo infine che la partecipazione di entrambe le entità alla relazione sia opzionale:



In questo caso abbiamo una terza possibilità:

cittadino(<u>nome</u>)

stato(<u>nome</u>)

presidenza(<u>stato</u>, <u>presidente</u>)

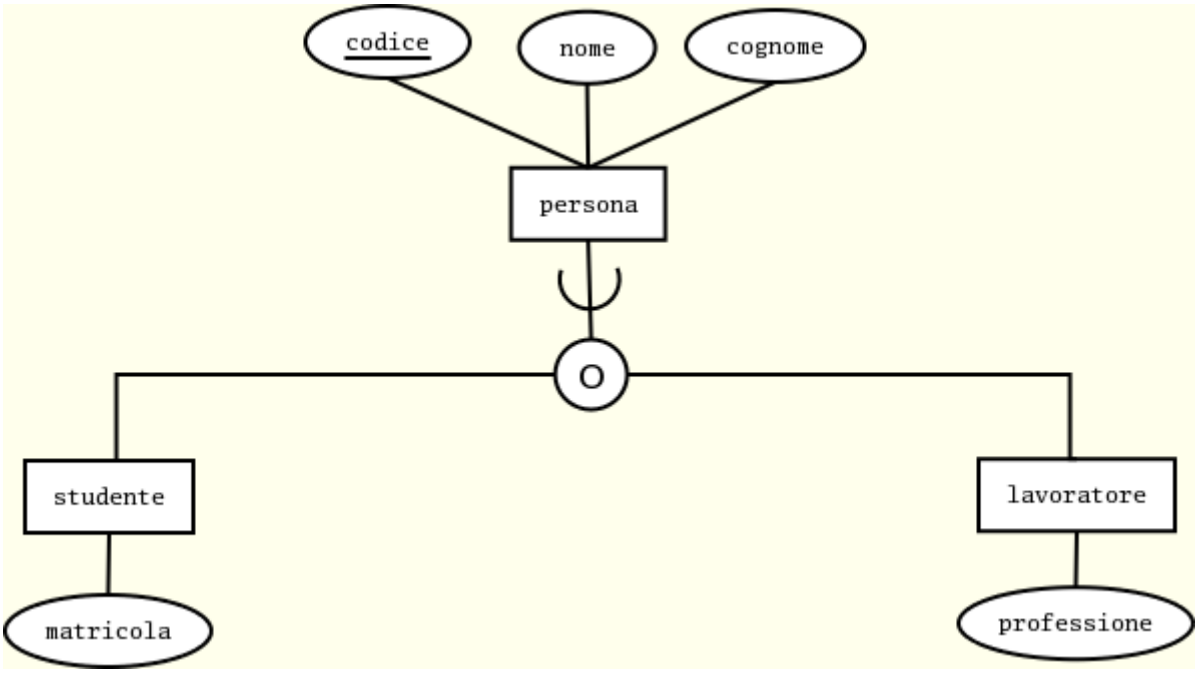
presidenza(stato) --> **stato**(nome)

presidenza(presidente) --> **cittadino**(nome)

Quest'ultima traduzione è da preferire se il numero di istanze della relazione presidenza è piccolo rispetto al numero di istanze delle entità che vi partecipano.

Specializzazione

Vi sono diverse traduzioni possibili in questo caso, ognuna con pregi e difetti. Consideriamo il seguente esempio:



Una traduzione che va bene *qualsiasi sia il tipo* della specializzazione è la seguente:

persona(codice, nome, cognome)

studente(codice, matricola)

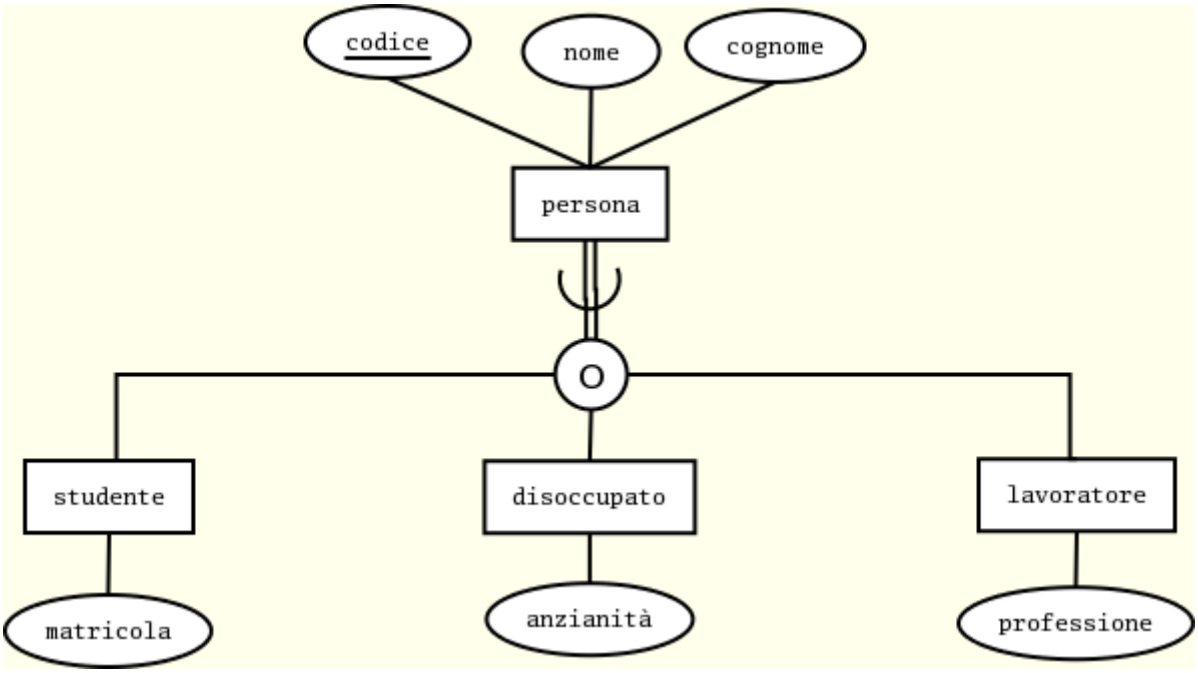
lavoratore(codice, professione)

studente(codice) --> **persona**(codice)

lavoratore(codice) --> **persona**(codice)

Viene creata una tabella per ogni entità coinvolta nella specializzazione. Le tabelle delle sotto-entità ereditano dall'entità genitore la chiave, che diventa la loro chiave e anche chiave esterna riferita all'entità genitore. Il vincolo di chiave esterna realizza il vincolo di sotto-insieme tra istanze delle entità. Uno studente lavoratore è contenuto in entrambe le tabelle studente e lavoratore. Si noti che le tabelle delle sotto-entità non contengono gli altri attributi di persona (nome e cognome), e neppure le eventuali relazioni a cui persona partecipa. Infatti tali attributi e relazioni sono raggiungibili mediante la chiave esterna delle sotto-entità.

Rendiamo la specializzazione totale aggiungendo una sotto-entità disoccupato:



Una traduzione alternativa che va bene quando la specializzazione è *totale* è la seguente:

studente(codice, nome, cognome, matricola)

disoccupato(codice, nome, cognome, anzianità)

lavoratore(codice, nome, cognome, professione)

L'entità genitore non viene tradotta ma tutti i suoi attributi vengono ereditati dalle tabelle delle entità figlie. Il vantaggio di questa traduzione è che, eliminando l'entità genitore, si riduce il numero di collegamenti tra tabelle necessari per il recupero dell'informazione. Inoltre, non impone vincoli di chiave esterna. Lo svantaggio è che istanze che appartengono a più sotto-entità vengono duplicate in più tabelle. Inoltre, eventuali relazioni a cui partecipa l'entità genitore devono essere duplicate in ogni entità figlia. Si noti che se la specializzazione fosse stata parziale, la traduzione di sopra è scorretta in quanto fa perdere le istanze dell'entità genitore non incluse in qualche sotto-entità.

Un'altra traduzione dell'ultimo schema proposto che rimane valida in *tutti i casi* è la seguente:

persona(codice, nome, cognome, tipo, matricola, anzianità, professione)

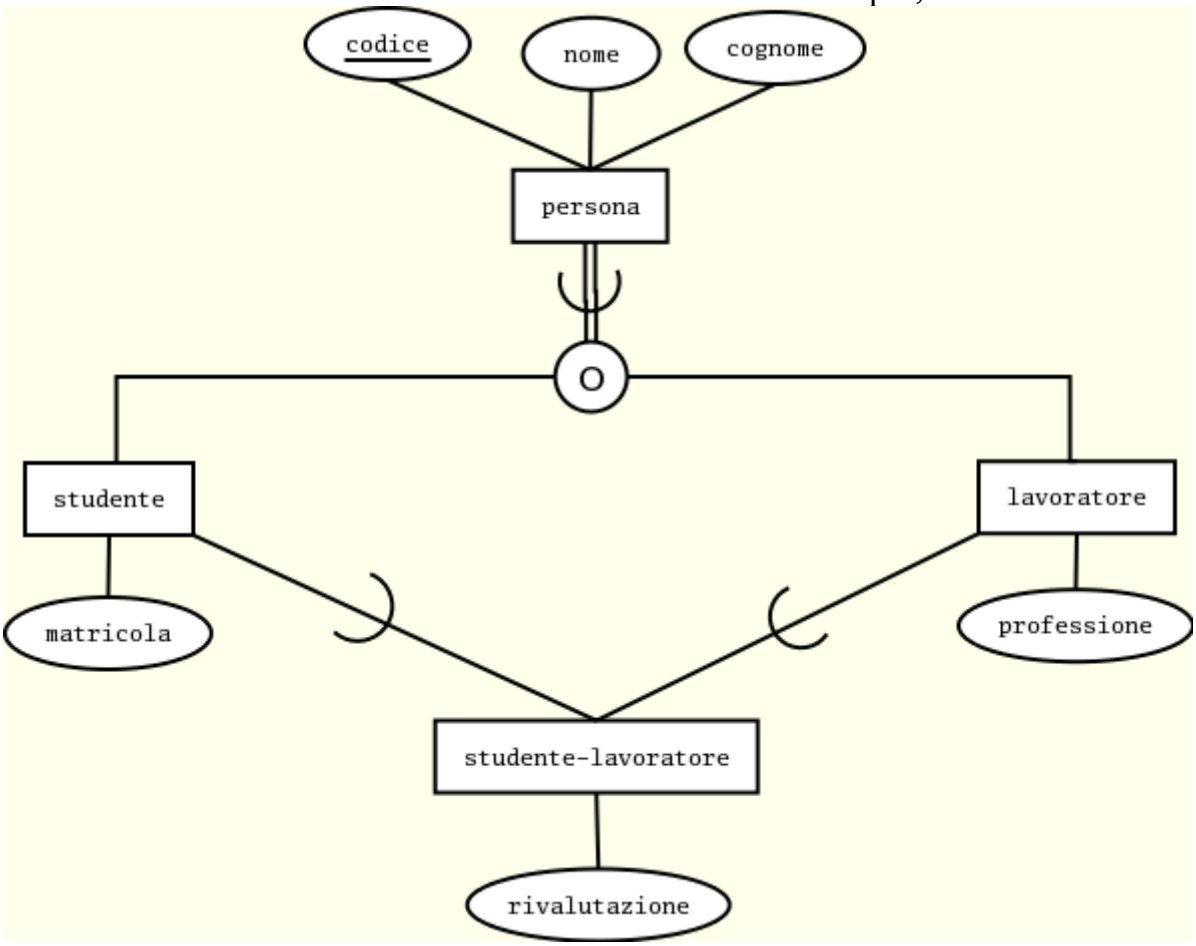
Abbiamo inglobato le entità figlie nell'entità genitore. Inoltre è stato aggiunto un nuovo attributo (tipo), che discrimina la sotto-entità a cui l'istanza appartiene. Ad esempio, per una istanza di studente, l'attributo tipo identifica la sotto-entità studente, l'attributo matricola contiene la matricola dello studente, e gli attributi anzianità e professione sono nulli.

Il vantaggio di questa traduzione è quello di usare una unica tabella. Questo riduce le operazioni di collegamento tra tabelle nel recupero dell'informazione. Inoltre, non impone

vincoli di chiave esterna. Infine, rispetto alla soluzione precedente, possiamo accedere ad una unica tabella qualora conosciamo il codice di una persona ma non il suo tipo. Lo svantaggio è che ci potrebbero essere molti valori nulli in corrispondenza degli attributi specifici delle sotto-entità oppure quando la specializzazione è parziale. Si noti che anche eventuali relazioni associate alle entità figlie devono essere portate verso l'entità genitore, modificando la cardinalità minima di partecipazione dell'entità genitore a 0. Ad esempio, se uno studente partecipa alla relazione iscritto ad un unico corso di laurea con vincolo (1,1), una persona, che non è necessariamente uno studente, dovrà partecipare alla stessa relazione con vincolo (0,1). Questo cambiamento può introdurre ulteriori valori nulli.

Qualora la specializzazione sia sovrapposta, occorre scegliere con attenzione il dominio dell'attributo discriminante da aggiungere (qualora non esista). Una possibile soluzione è quella di usare come valore una sequenza di n bit, dove n è il numero di entità figlie. L'i-esimo bit identifica la partecipazione dell'istanza alla i-esima sotto-entità. Un'altra possibilità è quella di usare n campi distinti di tipo Booleano.

Specializzazioni multiple si traducono come le normali specializzazioni, facendo attenzione ad ereditare solo una volta attributi comuni. Ad esempio, lo schema:



si traduce come segue:

persona(<u>codice</u>, nome, cognome)

studente(<u>codice</u>, matricola)

lavoratore(<u>codice</u>, professione)

studenteLavoratore(<u>codice</u>, rivalutazione)

studente(codice) --> **persona**(codice)

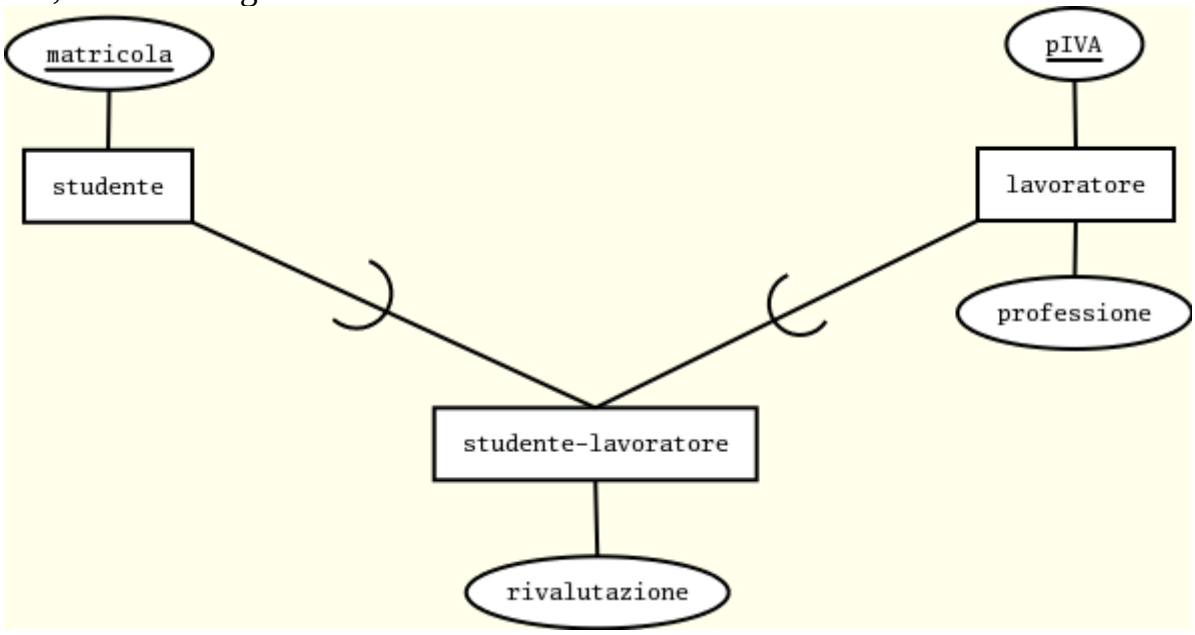
lavoratore(codice) --> **persona**(codice)

studenteLavoratore(codice) --> **studente**(codice)

studenteLavoratore(codice) --> **lavoratore**(codice)

Si noti che studenteLavoratore eredita l'attributo codice una sola volta ma esistono due vincoli di chiave esterna riferiti a studente e lavoratore. In tal modo affermiamo che un'istanza di studenteLavoratore appartiene a studente e anche a lavoratore.

Un caso particolare si ha quando l'ereditarietà multipla proviene da entità con chiavi diverse, come nel seguente caso:



Una traduzione possibile è la seguente:

studente(<u>matricola</u>)

lavoratore(<u>pIVA</u>, professione)

studenteLavoratore(<u>matricola</u>, pIVA, rivalutazione)

studenteLavoratore(matricola) --> **studente**(matricola)

studenteLavoratore(pIVA) --> **lavoratore**(pIVA)

Si noti che la chiave di studenteLavoratore può essere scelta tra la chiave di studente e quella di lavoratore.

Nel caso di traduzione di un reticolo di specializzazioni, le varie traduzioni proposte possono essere combinate.

Documentazione del modello relazionale

Come per lo schema concettuale, anche lo schema logico deve essere documentato. In particolare lo schema di base di dati non contiene le seguenti informazioni:

- il significato delle relazioni e dei loro attributi;
- il tipo degli attributi (semplice o calcolato) e la loro obbligatorietà;
- eventuali chiavi candidate;
- le relazioni concettuali che sono state rimosse durante la fase di traduzione dallo schema ER allo schema relazionale;
- le regole aziendali. In particolare tali regole contengono le regole di derivazione degli attributi calcolati.

Come fatto per il modello concettuale, proponiamo una soluzione basato XML per documentare lo schema logico. La seguente DTD descrive un possibile modello per la documentazione di uno schema relazionale:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT schema (relazione+, regola*)>

<!ELEMENT relazione  (descrizione?, attributo+, chiavePrimaria,
                      chiaveCandidata*, chiaveEsterna*)>
<!ATTLIST relazione  nome ID #REQUIRED>

<!ELEMENT attributo  (nome, descrizione?)>
<!ATTLIST attributo  id ID #REQUIRED
                     tipo (semplice | calcolato) #REQUIRED
                     opzionale (si | no) #REQUIRED>

<!ELEMENT chiavePrimaria EMPTY>
<!ATTLIST chiavePrimaria attributi IDREFS #REQUIRED>

<!ELEMENT chiaveCandidata EMPTY>
<!ATTLIST chiaveCandidata attributi IDREFS #REQUIRED>

<!ELEMENT chiaveEsterna EMPTY>
<!ATTLIST chiaveEsterna attributi IDREFS #REQUIRED
                        relazione IDREF  #REQUIRED
                        associazione CDATA #IMPLIED>

<!ELEMENT descrizione (#PCDATA)>
<!ELEMENT nome (#PCDATA)>
<!ELEMENT regola (#PCDATA)>
```

Seguono alcune osservazione sulla DTD proposta:

- Gli elementi attributo che formano chiavi esterne devono essere inseriti nell'elemento relazione nell'ordine in cui sono inseriti gli elementi attributo della corrispondente chiave primaria. Questo al fine di mantenere una corrispondenza tra attributi della chiave esterna e attributi della chiave primaria. Si noti che in XML gli elementi sono ordinati secondo l'ordine in cui appaiono nel documento;
- l'attributo id dell'elemento attributo è formato concatenando il nome della relazione e il nome dell'attributo con iniziale maiuscola, ad esempio personaTelefono per la relazione persona e l'attributo telefono. In questo modo tale attributo ha valore univoco nel documento;
- l'attributo attributi dell'elemento chiavePrimaria (chiaveCandidata, chiave Esterna) è una lista di identificatori dei corrispondenti attributi che formalo la chiave;
- l'attributo relazione dell'elemento chiaveEsterna è un identificatore della relazione a cui la chiave esterna si riferisce;
- l'attributo opzionale associazione dell'elemento chiaveEsterna contiene il nome della corrispondente relazione concettuale se questa è stata rimossa nella traduzione dallo schema ER a quello relazionale.

Caso di studio: schema logico

Riprendiamo il caso di studio della rete teatrale di cui avevamo specificato lo schema ER.

Proponiamo di seguito una traduzione nel modello relazionale, evidenziando le strutture e i vincoli di integrità utilizzati. Inoltre vediamo come sia possibile documentare lo schema logico ottenuto mediante XML secondo la DTD che abbiamo proposto.

Schema relazionale, vincoli e viste

Vediamo una traduzione dello schema ER del caso di studio in uno schema relazionale equivalente. Come primo passo rimuoviamo dallo schema ER gli attributi multivalore. In particolare:

1. per gli attributi multivalore video e immagini dell'entità messa in scena creiamo una unica entità materiale con attributi file (percorso e nome del file che contiene l'informazione), tipo (immagine o video e relativo tipo) e dimensione (dimensione del file in MB) e la colleghiamo con una relazione uno a molti all'entità di appartenenza;
2. l'attributo multivalore orario dell'entità biglietteria viene trasformato in un'entità debole di biglietteria con attributi giorno, inizio e fine e chiave parziale giorno e inizio;
3. l'attributo multivalore stipendio dell'entità biglietteria viene trasformato in un'entità debole di dipendente con attributi importo e inizio e chiave parziale inizio.

La traduzione dello schema ER risultante è la seguente:

teatro(nome, telefono, fax, indirizzo, email, url)

biglietteria(nome, indirizzo, email, telefono, teatro)

orario(biglietteria, giorno, inizio, fine)

notizia(data, ora, oggetto, testo)

newsletter(teatro, data, ora, oggetto)

dipendente(cf, nome, cognome, dataDiNascita, luogoDiNascita, residenza, telefonoFisso, telefonoMobile, email)

lavoro(teatro, dipendente, dataDiAssunzione, ruolo, cda)

stipendio(dipendente, inizio, importo)

spazio(nome, indirizzo, pianta, capienza)

luogo(teatro, spazio)

stagione(nome, biennio, teatro)

spettacolo(titolo, descrizione, annoProduzione)

proposta(nomeStagione, biennioStagione, spettacolo)

messaInScena(data, ora, spazio, spettacolo, postiDisponibili, prezzoIntero, prezzoRidotto, prezzoStudenti)

materiale(file, tipo, dimensione, dataSpettacolo, oraSpettacolo, spazioSpettacolo)

prenotazione(dataSpettacolo, oraSpettacolo, spazioSpettacolo, numero, data, ora, posto, tipo, prezzo)

commento(autore, data, ora, testo, dataSpettacolo, oraSpettacolo, spazioSpettacolo)

risposta(autoreRisposta, dataRisposta, oraRisposta, autore, data, ora)

produzione(produttore, spettacolo)

produttore(nome, tipo, indirizzo, telefono, email)

interprete(nome, ruolo, cv)

cast(spettacolo, interprete)

biglietteria(teatro) --> **teatro**(nome)

orario(biglietteria) --> **biglietteria**(nome)

newsletter(teatro) --> **teatro**(nome)

newsletter(data, ora, oggetto) --> **notizia**(data, ora, oggetto)

lavoro(teatro) --> **teatro**(nome)

lavoro(dipendente) --> **dipendente**(cf)

stipendio(dipendente) --> **dipendente**(cf)

luogo(teatro) --> **teatro**(nome)

luogo(spazio) --> **spazio**(nome)

stagione(teatro) --> **teatro**(nome)

messaInScena(spazio) --> **spazio**(nome)

messaInScena(spettacolo) --> **spettacolo**(titolo)

materiale(dataSpettacolo, oraSpettacolo, spazioSpettacolo) --> **messaInScena**(data, ora, spazio)

prenotazione(dataSpettacolo, oraSpettacolo, spazioSpettacolo) --> **messaInScena**(data, ora, spazio)

commento(dataSpettacolo, oraSpettacolo, spazioSpettacolo) --> **messaInScena**(data, ora, spazio)

risposta(autoreRisposta, dataRisposta, oraRisposta) --> **commento**(autore, data, ora)

risposta(autore, data, ora) --> **commento**(autore, data, ora)

produzione(produttore) --> **produttore**(nome)

produzione(spettacolo) --> **spettacolo**(titolo)

cast(spettacolo) --> **spettacolo**(titolo)

cast(interprete) --> **interprete**(nome)

Seguono alcune osservazioni:

- un contributo multimediale associato ad una messa in scena viene inserito nella tabella materiale mediante un riferimento al nome del file che lo contiene, piuttosto che con l'uso di campi binari. Questo nome può essere usato come chiave;
- la specializzazione di produttore in interno e esterno viene tradotta in una unica tabella produttore. L'attributo tipo di tale tabella discrimina se il produttore è interno o esterno. In caso di produttore esterno possono essere usati i campi indirizzo, telefono e email;
- alcune tabelle derivanti da entità concettuali hanno chiavi lunghe. Le tabelle che corrispondono a relazioni molti a molti che coinvolgono queste entità soffrono dello stesso problema. Esempi sono messaInScena, commento, notizia e la relazione newsletter. Per ragioni di efficienza dell'implementazione è possibile introdurre, per queste relazioni incriminate, degli **attributi codice** che vanno a sostituire le chiavi lunghe. Questa soluzione ha il vantaggio di ridurre la dimensione delle chiavi e lo svantaggio di introdurre un attributo sintetico che non corrisponde ad una informazione presente nella realtà modellata. Inoltre, il codice deve essere mantenuto univoco per ogni istanza dell'entità;

I **vincoli di dominio** verranno aggiunti in fase di progettazione fisica quando gli attributi delle tabelle verranno associati ai loro domini. Altri vincolo che verranno realizzati durante la progettazione fisica sono le regole aziendali identificate nel modello ER.

Infine possiamo definire le seguenti **viste** sui dati:

1. una vista spettacoli che descrive, per ogni teatro e relativa stagione, i corrispondenti spettacoli messi in scena. Ogni spettacolo deve essere descritto da: titolo, descrizione, data, ora, luogo, indirizzo, posti disponibili e prezzi per i diversi tipi di biglietto. Le porzione di base di dati coinvolta è formata dalle seguenti tabelle: stagione, proposta, spettacolo, scena, messaInScena e spazio. La vista può essere accessibile sia dagli utenti esterni della rete (i clienti dei teatri) che dai dipendenti interni alla rete;
2. una vista interpreti che descrive, per ogni teatro e relativa stagione, gli interpreti che hanno partecipato agli spettacoli. Le tabelle coinvolte sono: stagione, proposta, spettacolo, cast e interprete. La vista è accessibile solo dai dipendenti interni alla rete del settore comunicazione, ufficio stampa e relazioni con il pubblico;
3. una vista dipendenti che descrive, per ogni teatro, i relativi dipendenti. Per ogni dipendente si vuole esporre: cognome, nome, cf, età, telefono, email, ultimo stipendio, data di assunzione, ruolo nel teatro e se il dipendente fa parte del CdA del teatro. Le tabelle coinvolte sono: lavoro, dipendente e stipendio. La vista è accessibile dai dipendenti interni alla rete del settore amministrazione, controllo e finanza;
4. una vista che, per ogni spettacolo terminato, mostra il numero di spettatori paganti, l'affluenza (il rapporto tra paganti e capienza) e l'incasso. Le tabelle usate sono spettacolo, scena, messaInScena, spazio e prenotazione. La vista è riservata ai dipendenti interni alla rete del settore amministrazione, controllo e finanza.

Si noti che le viste definite possono essere usate in due modi: per far accedere un gruppo di utenti ad una porzione specifica dei dati e, come vedremo in fase di progettazione fisica, per realizzare alcune transazioni tipiche dell'applicazione.

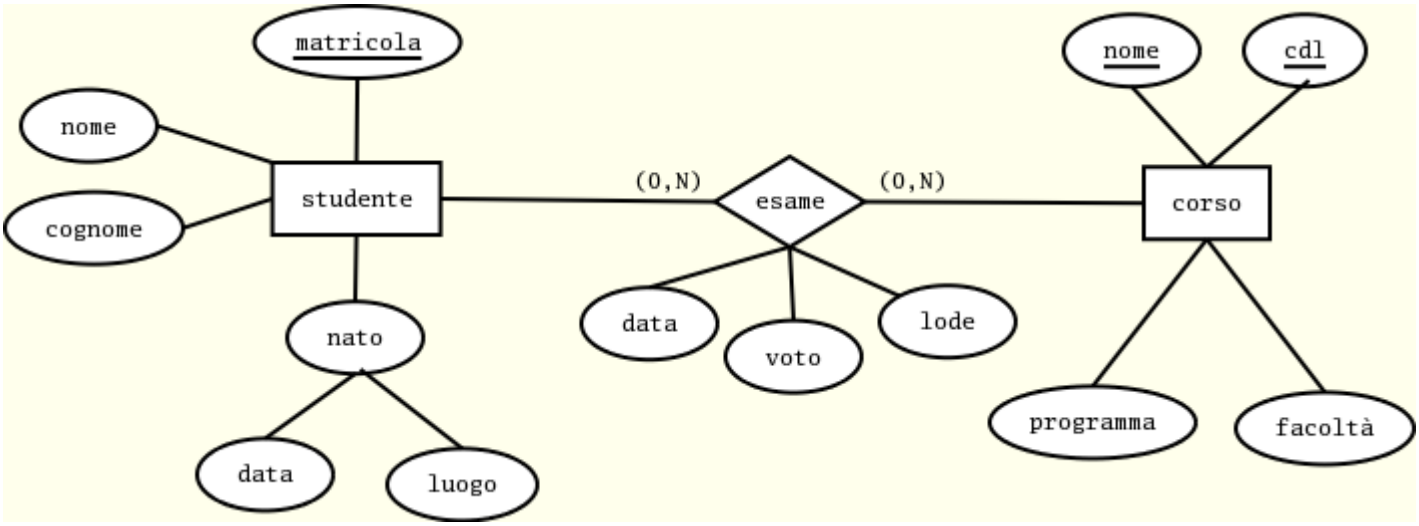
Documentazione dello schema relazionale

Vediamo un estratto della documentazione dello schema relazionale proposto per il nostro caso di studio. Lo schema è documentato da un file XML che risponde alla DTD proposta a tale scopo. Un frammento di tale documento XML è il seguente:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE schema SYSTEM "relazionale.dtd">

<schema>

  <relazione nome="teatro">
    <attributo id = "teatroNome" tipo="semplice" opzionale="no">
      <nome>nome</nome>
    </attributo>
    <attributo id="teatroTelefono" tipo="semplice" opzionale="no">
      <nome>telefono</nome>
    </attributo>
    <attributo id="teatroFax" tipo="semplice" opzionale="si">
      <nome>fax</nome>
    </attributo>
    <attributo id="teatroIndirizzo" tipo="semplice" opzionale="no">
      <nome>indirizzo</nome>
    </attributo>
    <attributo id="teatroEmail" tipo="semplice" opzionale="si">
      <nome>email</nome>
    </attributo>
    <attributo id="teatroUrl" tipo="semplice" opzionale="si">
      <nome>url</nome>
    </attributo>
    <chiavePrimaria attributi="teatroNome"/>
  </relazione>

  <relazione nome="biglietteria">
    <attributo id="biglietteriaNome" tipo="semplice" opzionale="no">
      <nome>nome</nome>
    </attributo>
    <attributo id="biglietteriaIndirizzo" tipo="semplice" opzionale="no">
      <nome>indirizzo</nome>
    </attributo>
    <attributo id="biglietteriaEmail" tipo="semplice" opzionale="si">
      <nome>email</nome>
    </attributo>
    <attributo id="biglietteriaTelefono" tipo="semplice" opzionale="no">
      <nome>telefono</nome>
    </attributo>
    <attributo id="biglietteriaTeatro" tipo="semplice" opzionale="no">
      <nome>teatro</nome>
    </attributo>
    <chiavePrimaria attributi="biglietteriaNome"/>
    <chiaveCandidata attributi="biglietteriaIndirizzo"/>
    <chiaveEsterna attributi="biglietteriaTeatro"
                   relazione="teatro"
                   associazione="vendita"/>
  </relazione>

</schema>
```

Esercitazione

Esercizio 1. Tradurre in uno schema relazionale il seguente diagramma ER:



studente(<u>matricola</u>, nome, cognome, data, luogo)

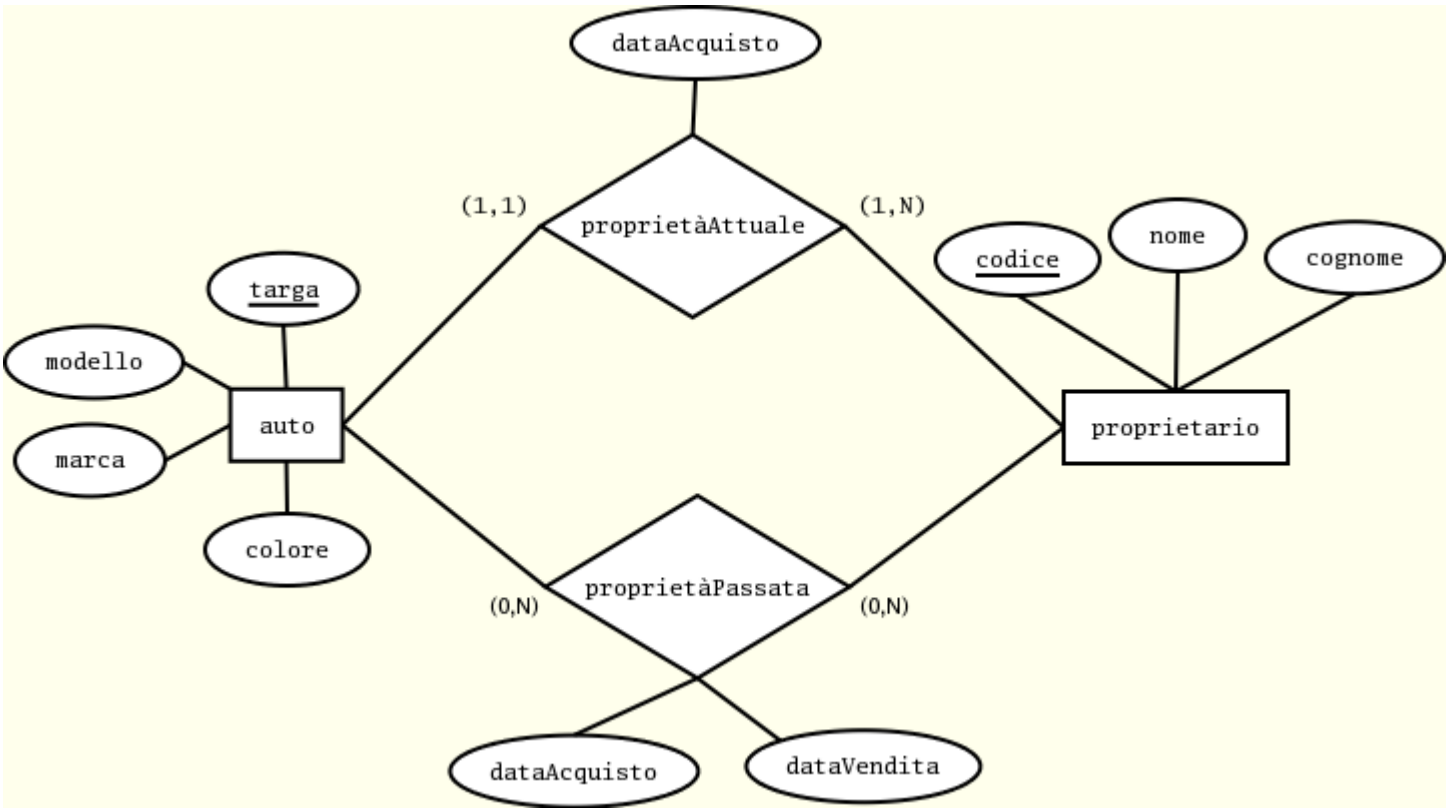
corso(<u>nome</u>, <u>cdl</u>, facoltà, programma)

esame(<u>studente</u>, <u>corso</u>, <u>cdl</u>, data, voto, lode)

esame(studente) -> **studente**(matricola)

esame(corso, cdl) -> **corso**(nome, cdl)

Esercizio 2. Tradurre in uno schema relazionale il seguente diagramma ER:



auto(<u>targa</u>, modello, marca, colore, proprietario, dataAcquisto)

proprietario(<u>codice</u>, nome, cognome)

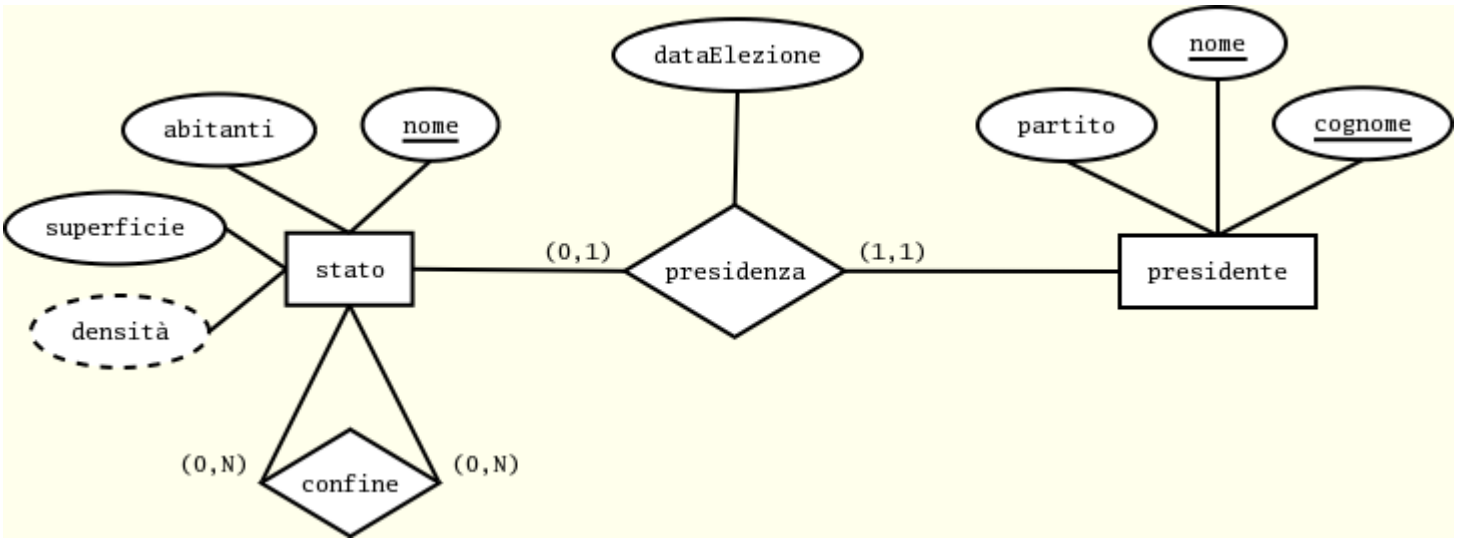
proprietàPassata(<u>auto</u>, <u>proprietario</u>, dataAcquisto, dataVendita)

auto(proprietario) -> **proprietario**(codice)

proprietàPassata(auto) -> **auto**(targa)

proprietàPassata(proprietario) -> **proprietario**(codice)

Esercizio 3. Tradurre in uno schema relazionale il seguente diagramma ER:



stato(<u>nome</u>, abitanti, superficie, densità)

presidente(<u>nome</u>, <u>conome</u>, partito, stato, dataElezione)

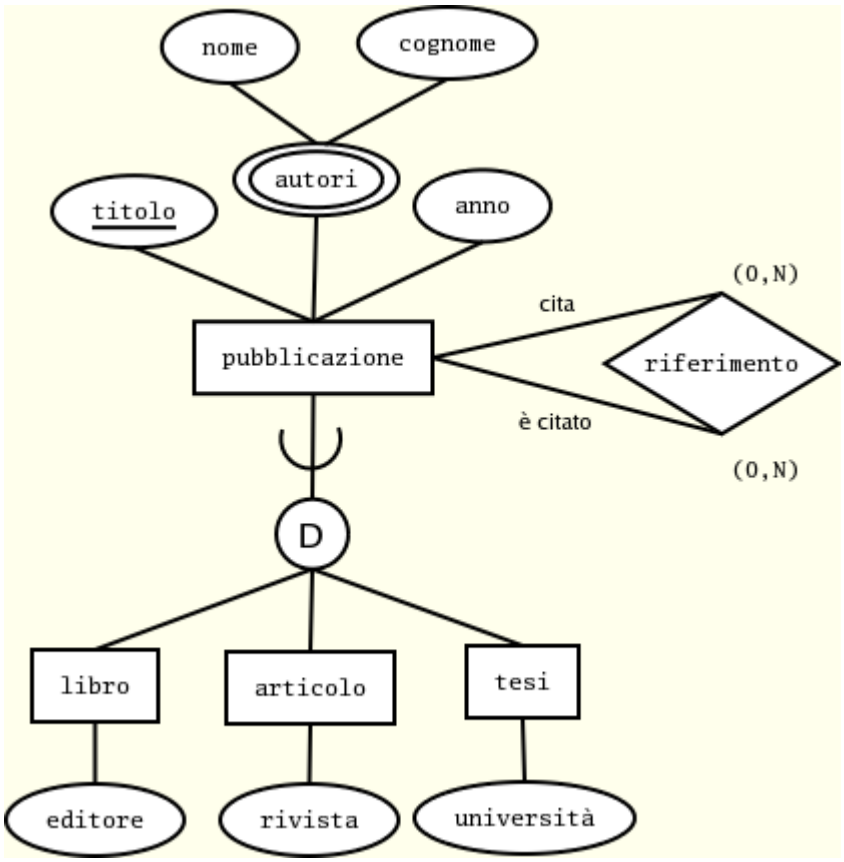
confine(<u>stato</u>, <u>statoConfinante</u>)

presidente(stato) -> **stato**(nome)

confine(stato) -> **stato**(nome)

confine(statoConfinante) -> **stato**(nome)

Esercizio 4. Tradurre in uno schema relazionale il seguente diagramma ER:



pubblicazione(titolo, anno)

autore(nome, cognome)

testo(nome, cognome, titolo)

riferimento(pubblicazione, citazione)

libro(titolo, editore)

articolo(titolo, rivista)

tesi(titolo, università)

testo(nome, cognome) -> **autore**(nome, cognome)

testo(titolo) -> **pubblicazione**(titolo)

riferimento(pubblicazione) -> **pubblicazione**(titolo)

riferimento(citazione) -> **pubblicazione**(titolo)

libro(titolo) -> **pubblicazione**(titolo)

articolo(titolo) -> **pubblicazione**(titolo)

tesi(titolo) -> **pubblicazione**(titolo)

Esercizio 5. Creare un diagramma ER che corrisponda al seguente schema relazionale:

paziente(codice, cognome, nome)

reparto(codice, nome, primario)

medico(matricola, cognome, nome, reparto)

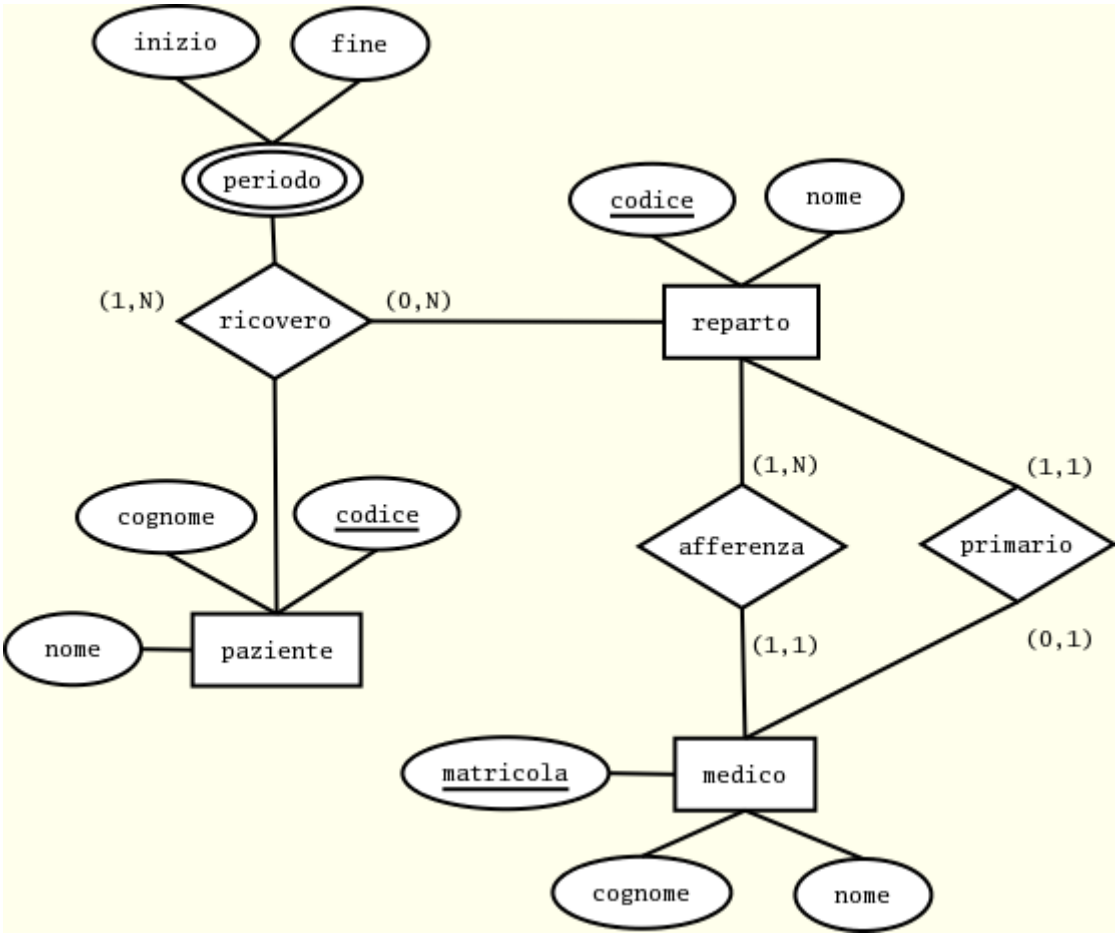
ricovero(paziente, reparto, inizio, fine)

reparto(primario) -> **medico**(matricola)

medico(reparto) -> **reparto**(codice)

ricovero(paziente) -> **paziente**(codice)

ricovero(reparto) -> **reparto**(codice)



Esercizio 6. Si consideri lo schema concettuale per la base di dati universitaria. Si richiede di:

1. produrre un corrispondente schema relazionale;
2. documentare lo schema relazionale usando XML (il documento deve risultare valido rispetto alla DTD proposta).

Lo schema relazionale è il seguente:

facoltà(nome, preside)

sede(indirizzo, telefono)

dislocazione(facoltà, sede)

cdl(nome, descrizione, presidente, facoltà)

studente(matricola, nome, cognome, dataImmatricolazione, cdl)

corso(nome, cdl, descrizione, programma, anno, docente)

mutuo(corso, cdl, corsoMutuo, cdlMutuo)

pianoDiStudi(studente, corso, cdl)

lezione(giorno, fascia, aula, corso, cdl)

docente(codice, tipo, nome, cognome, email, sito, telefono, ufficio, facoltà)

facoltà(preside) -> **docente**(codice)

dislocazione(facoltà) -> **facoltà**(nome)

dislocazione(sede) -> **sede**(indirizzo)

cdl(presidente) -> **docente**(codice)

cdl(facoltà) -> **facoltà**(nome)

studente(cdl) -> **cdl**(nome)

corso(cdl) -> **cdl**(nome)

corso(docente) -> **docente**(codice)

mutuo(corso, cdl) -> **corso**(nome, cdl)

mutuo(corsoMutuo, cdlMutuo) -> **corso**(nome, cdl)

pianoDiStudi(corso, cdl) -> **corso**(nome, cdl)

pianoDiStudi(studente) -> **studente**(matricola)

lezione(corso, cdl) -> **corso**(nome, cdl)

docente(facoltà) -> **facoltà**(nome)

Progettazione fisica

L'ultima fase della progettazione di una basi di dati è la **progettazione fisica**. Prima di inniziare la progettazione fisica occorre **scegliere un DBMS** che implementi il modello dei dati dello schema logico. La progettazione fisica consiste delle seguenti attività:

1. scelta delle **strutture di memorizzazione** delle tabelle e delle **strutture ausiliarie di accesso** ai dati (indici). Queste ultime servono per rendere più efficiente l'accesso ai dati contenuti in tabelle usate di frequente. Le strutture di memorizzazione e di accesso sono valutate tra quelle messe a disposizione dal DBMS scelto;
2. traduzione dello schema logico dei dati in uno **schema fisico dei dati** contenente le definizioni delle tabelle, dei relativi vincoli di integrità e delle viste espresse in SQL;
3. implementazione delle **transazioni** in SQL.

Terminata questa fase la base di dati è stata completamente progettata e si passa alla sua **realizzazione**, cioè alla costruzione fisica delle tabelle e all'implementazione delle applicazioni della base di dati. Le applicazioni sono scritte in linguaggi di programmazione ad alto livello (C++, Java) e possono riutilizzare il codice SQL scritto per le transazioni. La fase di realizzazione è spesso seguita da una **fase di ottimizzazione** in cui le prestazione del DBMS sulla specifica base di dati vengono valutate e sono possibili cambiamenti dei parametri decisi durante la progettazione fisica (ad esempio, l'aggiunta di un nuovo indice).

Il linguaggio SQL

SQL è il linguaggio di **definizione** e **manipolazione** dei dati universalmente usato nelle basi di dati relazionali. Esso ha le seguenti funzioni:

- **definizione dei dati**: SQL permette di definire, in modo integrato, i tre livelli della base di dati: il livello esterno (viste), il livello logico (struttura delle tabelle e vincoli di integrità) e il livello fisico (memoriazzazione delle tabelle e indici) della base di dati;
- **aggiornamento dei dati**: SQL permette di aggiornare l'istanza della base mediante inserimenti, cancellazioni e modifiche di tuple;
- **interrogazione sui dati**: SQL consente di estrarre informazione presente nella base di dati.

Queste tre componenti del linguaggio, che esamineremo separatamente, sono in realtà interconnesse: la definizione e l'aggiornamento dei dati possono far uso delle interrogazioni sui dati. Inoltre, da definizione di particolari vincoli sui dati (le regole attive) coinvolge le operazioni di aggiornamento. Logicamente, la definizione dei dati precede l'aggiornamento che precede l'interrogazione. Didatticamente, data le dipendenze citate, è conveniente descrivere queste componenti in ordine inverso, cioè interrogazione, aggiornamento e definizione dei dati.

SQL è l'acronimo di *Structured Query Language* ed era originariamente il linguaggio di *System R*, uno dei primi prototipi di DBMS relazionale svuluppato dall'IBM a metà degli anni 70. Il successo di SQL è dovuto principalmente alla sua standardizzazione presso ANSI (American National Standards Institute) e ISO (International Standard Organization), processo a cui hanno partecipato gran parte dei produttori di DBMS relazionali. Esistono tre grosse versioni di SQL:

- **SQL-1**. Questa versione è composta da SQL-86 (1986 è l'anno di emanazione) e SQL-89;
- **SQL-2**. Questa versione è anche nota come SQL-92;
- **SQL-3**. Anche questa versione è suddivisa in due sotto-versioni: SQL:1999 e SQL:2003.

E' importante notare il comportamento dei sistemi commerciali a fronte di questi standard. Le funzionalità di base sono state implementate da quasi tutti i DBMS commerciali secondo lo standard. D'altronde, le funzionalità più avanzate sono raramente state realizzate. In particolare, SQL-3 è ben lontano da essere comunemente adottato, mentre SQL-2 è sostanzialmente stato implementato dai produttori del settore. Inoltre, alcuni DBMS hanno introdotto costrutti (ad esempio, domini) non presenti negli standard o modificato la sintassi dei comandi SQL. Il risultato è che ogni DBMS ha un proprio dialetto di SQL che condivide con lo standard le parti di base ma può aggiungere o mancare di altre funzionalità. In questo corso cercheremo di mostrare la parte dello standard SQL comunemente implementata.

Interrogazione della base di dati

SQL è un linguaggio di definizione e di manipolazione dei dati. In quanto linguaggio di manipolazione, SQL permette di **selezionare dati di interesse** dalla base e di aggiornarne il contenuto. In questa sezione vedremo la parte più interessante e usata del linguaggio SQL, vale a dire le interrogazioni per selezionare i dati. Tali interrogazioni sono usate sia nei costrutti SQL di definizione dei dati che in quelli di aggiornamento della base di dati. E' bene quindi iniziare lo studio di SQL dalle interrogazioni di selezione di dati.

SQL è un **linguaggio dichiarativo**: esso permette di specificare *cosa* cercare senza dire *come*. Quando una interrogazione (*query*) viene eseguita dall'elaboratore di interrogazioni (*query processor*), essa viene tradotta in un **linguaggio procedurale interno** al sistema il quale permette di specificare come accedere ai dati. Esistono in generale più possibili traduzioni di una interrogazione SQL nel linguaggio procedurale. Il compito dell'ottimizzatore delle interrogazioni (*query optimizer*) è scegliere il piano di esecuzione più efficiente.

E' quindi bene che l'utente si concentri sull'obiettivo della propria ricerca, cercando di scrivere interrogazioni leggibili e modificabili, delegando tutti gli aspetti procedurali e di efficienza al DBMS. Il vantaggio di poter scrivere interrogazioni ad alto livello (pensando allo schema logico e non alla sua realizzazione fisica) è una conseguenza dell'**indipendenza fisica dei dati** dei DBMS relazionali.

Una interrogazione SQL viene eseguita su una base di dati, quindi su un insieme di tabelle collegate tra loro mediante il meccanismo delle chiavi esterne. Il risultato di una interrogazione è una tabella. E' bene chiarire subito che esistono due differenze importanti tra relazioni del modello relazionale e tabelle SQL. Una tabella SQL può contenere **righe duplicate** e **colonne omonime**. Le colonne vengono identificate univocamente dalla loro posizione. Questo è vero per le **tabelle risultato** dalle interrogazioni (vedremo degli esempi). Le **tabelle di base**, cioè quelle che fanno parte della base di dati, non possono avere colonne omonime. Inoltre, se esiste una chiave per la tabella di base (questa è la norma), non possono esistere righe duplicate nella tabella. Dunque tabelle di base dotate di una chiave corrispondono a relazioni del modello relazionale. Come vedremo, esiste un modo per far si che una tabella non contenga duplicati. La presenza di duplicati nelle tabelle risultato è motivata come segue:

- in certe occasioni rimuovere le righe duplicare corrisponde ad una perdita di informazione. Si pensi ad esempio ad una tabella che contiene una sola colonna con dei dati numerici. Supponiamo di voler calcolare la media sui dati di questa colonna. L'eliminazione dei duplicati falserebbe il risultato;
- l'eliminazione dei duplicati dal risultato è in generale una operazione costosa (un modo per implementarla consiste nell'ordinare le righe e poi eliminare i duplicati).

Introdurremo SQL *by example*, cioè mostrando esempi sempre più ricchi e complessi di interrogazione.

Interrogazioni di base

Consideriamo la seguente tabella teatro:

teatro		
nome	**città**	**email**
CSS	Udine	css@gmail.com
Litta	Milano	litta@gmail.com
Piccolo	Milano	piccolo@gmail.com
Eliseo	Roma	eliseo@gmail.com

L'interrogazione più semplice che si possa scrivere è la seguente:

```
select *
from teatro
```

Il risultato è l'intera tabella teatro. La prima riga dell'interrogazione è detta **clausola select** e serve per selezionare le colonne della tabella che ci interessano. L'operatore `*` permette di selezionare tutte le colonne. La seconda riga dell'interrogazione è detta **clausola from** e serve per indicare quali tabelle usare. La clausola select e quella from sono obbligatorie in una interrogazione.

Se siamo interessati solo al nome e all'email dei teatri, possiamo selezionarli in questo modo:

```
select nome, email
from teatro
```

Il risultato è la seguente tabella:

nome	email
CSS	css@gmail.com

Litta	litta@gmail.com
Piccolo	piccolo@gmail.com
Eliseo	eliseo@gmail.com

Per chiarire la differenza tra relazione del modello realzionale e tabella SQL, vediamo una semplice interrogazione che genera una tabella con due colonne con lo stesso nome:

```
select nome, nome
from teatro
```

Il risultato è la seguente tabella:

nome	**nome**
CSS	CSS
Litta	Litta
Piccolo	Piccolo
Eliseo	Eliseo

Inoltre, mostriamo una semplice interrogazione che genera una tabella con due righe uguali:

```
select città
from teatro
```

Il risultato è la seguente tabella:

città
Udine
Milano
Milano
Roma

E' possibile specificare la parola chiave `distinct` dopo la parola chiave `select` per eliminare i duplicati.

Introduciamo ora la clausola `where`:

```
select nome
from teatro
where città = 'Milano'
```

Il risultato è la seguente tabella:

nome
Litta
Piccolo

La clausola `where` definisce un **predicato sui valori degli attributi** delle tabelle selezionate. Le righe che soddisfano il predicato, cioè per le quali in predicato è vero, vengono inserite nella tabella risultato.

Un predicato è formato combinando predicati atomici con gli operatori Booleani `and`, `or` e `not`. Il `not` ha precedenza sugli altri due operatori ma non è stata definita una precedenza tra `or` e `and`. Un predicato atomico è ottenuto confrontando due espressioni sui valori degli attributi mediante i seguenti operatori di confronto: =, <>, <, >, <=, >=. Esempi di espressioni sono un attributo e una costante. Le costanti di tipo stringa, tempo e data si scrivono tra apici, come nell'esempio di sopra. Non servono gli apici per i numeri. Inoltre, l'operatore `like`permette un confronto con stringhe che contengono i valori speciali _ (un carattere qualsiasi) e `%` (una sequenza arbitraria, eventualmente vuota, di caratteri). E' possibile confrontare il valore di un'espressione con il valore nullo con i predicati `is null` e `is not null`.

Facciamo qualche esempio sulla seguente tabella:

dipendente

cf	nome	cognome	dataDiNascita	stipendio
ELSDLL72	Elisa	D'Allarche	1972-04-29	2500
FRNDPP76	Fernanda	D'Ippoliti	1976-03-11	2100
MRCDLL70	Marco	Dall'Aglio	1970-01-09	2700

Di seguito scriveremo prima la query in linguaggio naturale, poi la sua traduzione in SQL e infine il suo risultato.

Il nome, il cognome e lo stipendio dei dipendenti con uno stipendio di almeno 2500 Euro.

```
select nome, cognome, stipendio
from dipendenti
where stipendio >= 2500
```

nome	cognome	stipendio
Elisa	D'Allarche	2500
Marco	Dall'Aglio	2700

Il nome e il cognome dei dipendenti nati dopo il 1975.

```
select nome, cognome
from dipendenti
where dataDiNascita > '1975-12-31'
```

nome	cognome
Fernanda	D'Ippoliti

Il nome e il cognome dei dipendenti con il nome che finisce con la lettera 'a' e con uno stipendio di almeno 2500 Euro.

```
select nome, cognome
from dipendente
where (stipendio >= 2500) and (nome like '%a')
```

nome	cognome
Elisa	D'Allarche

Il codice fiscale e lo stipendio annuale di Elisa D'Allarche.

```
select cf as codiceFiscale, stipendio * 12 as stipendioAnnuale
from dipendente
where nome = 'Elisa' and cognome = 'D\'Allarche'
```

codiceFiscale	stipendioAnnuale
ELSDLL72	30000

Alcune osservazioni sull'ultima query: nelle espressioni possiamo usare le 4 operazioni *, /, +, -. Possiamo inoltre rinominare le colonne. Infine occorre far precedere il carattere apice (') da una barra (\) all'interno delle stringhe come in 'D\'Allarche'.

A partire da SQL-2, la logica dei predicati in SQL è a **tre valori**: vero (V), falso (F) e sconosciuto (*unknown*, indicato con U). Le regole per usare i tre valori di verità sono le seguenti:

- ogni confronto in cui almeno una componente ha valore sconosciuto ha come risultato il valore sconosciuto. Fanno eccezione i predicati `is null` e `is not null`, il cui valore è sempre o vero o falso, anche se il valore di confronto è sconosciuto;

- gli operabori Booleani vengono estesi al valore U nel seguente modo:

  not U = U

  V and U = U, F and U = F, U and U = U

  V or U = V, F or U = U, U or U = U

  In sostanza, not A è vero se e solo se A è falso, A and B è vero se e solo se entrambi A e B sono veri e A or B è vero se e solo almeno uno tra A e B è vero;
- una tupla soddisfa un predicato se il valore del predicato per quella tupla è *vero*.

Ad esempio, la query:

```
select *
from dipendente
where (età >= 30)
```

seleziona i dipendenti la cui età *è nota* (non nulla) e il suo valore è maggiore o uguale a 30. Inoltre:

```
select *
from dipendente
where (età < 30) or (età >= 30)
```

seleziona i dipendenti la cui età *è nota*, qualsiasi sia in suo valore. Si noti che il risultato non contiene tutti i dipendenti, ma, *giustamente*, vengono esclusi quelli che hanno valore sconosciuto per l'attributo età. In realtà questo risultato sembra contrario all'intuizione, in quanto *noi sappiamo* che ogni persona ha un'età e qualsiasi valore abbia è sicuramente un valore che soddisfa il predicato dato. L'intuizione è però fuorviante in tal caso in quanto assume che l'attributo età abbia un valore, anche se sconosciuto. Invece, il valore nullo per un attributo significa che: (i) il valore dell'attributo non esiste, oppure (ii) il valore dell'attributo esiste ma non è noto, oppure (iii) non è noto se il valore dell'attributo esista. L'interprete SQL non assume nessuno di questi tre casi. Per capire meglio, consideriamo la stessa query riferita all'attributo opzionale email:

```
select *
from dipendente
where (email like '%@gmail.com') or not (email like '%@gmail.com')
```

Un impiegato senza indirizzo di posta elettronica viene giustamente escluso dal risultato, in quanto è falso che il suo indirizzo appartiene al dominio gmail.com ed è falso pure che il suo indirizzo non appartiene a tale dominio. Semplicemente, il suo indirizzo non esiste.

Interrogazioni di congiunzione (join)

Finora abbiamo visto interrogazioni su una singola tabella. Nel modello relazionale, ogni concetto indipendente viene rappresentato con una tabella e le corrispondenze tra i dati in tabelle diverse vengono realizzate mediante il confronto tra i valori degli attributi (generalmente tra una chiave esterna e una chiave primaria). In particolare, le interrogazioni di congiunzione (più comunemente note con l'espressione inglese *join*) sfruttano il **modello di corrispondenza basato su valori**,

tipico del modello relazionale, per recuperare dati correlati ma distribuiti in più di una tabella. Si considerino le seguenti tre tabelle:

teatro

nome	**città**	**email**
CSS	Udine	css@gmail.com
Litta	Milano	litta@gmail.com
Eliseo	Roma	eliseo@gmail.com

dipendente

cf	**nome**	**cognome**	**dataDiNascita**	**stipendio**
ELSDLL72	Elisa	D'Allarche	29/04/1972	2500
FRNDPP76	Fernanda	D'Ippoliti	11/03/1976	2100
MRCDLL70	Marco	Dall'Aglio	09/01/1970	2700

lavoro

teatro	**dipendente**	**ruolo**
CSS	ELSDLL72	relazioni
Litta	FRNDPP76	finanza
Eliseo	FRNDPP76	controllo
Eliseo	MRCDLL70	direzione

Supponiamo di voler recuperare il nome e cognome di tutti i dipendenti del teatro CSS. Queste informazioni sono distribuite in due tabelle: dipendente e lavoro. Occorre dunque *congiungere* queste due tabelle mediante la chiave esterna dipendente della tabella lavoro:

```
select nome, cognome
from lavoro, dipendente
where (teatro = 'CSS') and (dipendente = cf)
```

Per aumentare la leggibilità, possiamo riscrivere la query facendo precedere ad ogni attributo il nome della corrispondente tabella:

```
select dipendente.nome, dipendente.cognome
from lavoro, dipendente
where (lavoro.teatro = 'CSS') and (lavoro.dipendente = dipendente.cf)
```

Similmente, possiamo recuperare tutte le città in cui lavora il dipendente identificato dal codice FRNDPP76 nel seguente modo:

```
select teatro.città
from lavoro, teatro
where (lavoro.dipendente = 'FRNDPP76') and (lavoro.teatro = teatro.nome)
```

L'aspetto procedurale di interrogazioni che coinvolgono più tabelle è utile per meglio capirne il significato. Consideriamo l'ultimo join tra lavoro e teatro. L'esecuzione procede in questo modo: la tabella **prodotto cartesiano** delle tabelle lavoro e teatro viene calcolata. Tale tabella contiene righe formate giustapponendo ogni riga di lavoro ad ogni riga di teatro:

lavoro x teatro

teatro	dipendente	ruolo	nome	città	email
CSS	ELSDLL72	relazioni	CSS	Udine	css@gmail.com
CSS	ELSDLL72	relazioni	Litta	Milano	litta@gmail.com
CSS	ELSDLL72	relazioni	Eliseo	Roma	eliseo@gmail.com
Litta	FRNDPP76	finanza	CSS	Udine	css@gmail.com
Litta	FRNDPP76	finanza	Litta	Milano	litta@gmail.com
Litta	FRNDPP76	finanza	Eliseo	Roma	eliseo@gmail.com
Eliseo	FRNDPP76	controllo	CSS	Udine	css@gmail.com
Eliseo	FRNDPP76	controllo	Litta	Milano	litta@gmail.com
Eliseo	FRNDPP76	controllo	Eliseo	Roma	eliseo@gmail.com
Eliseo	MRCDLL70	direzione	CSS	Udine	css@gmail.com
Eliseo	MRCDLL70	direzione	Litta	Milano	litta@gmail.com
Eliseo	MRCDLL70	direzione	Eliseo	Roma	eliseo@gmail.com

Ad ogni riga della tabella prodotto viene applicato il predicato della clausola `where` e le righe che lo soddisfano vengono selezionate. In particolare la condizione di join lavoro.teatro = teatro.nome permette di associare righe di lavoro a corrispondenti righe di teatro. Tali righe sono colorate in blu nella tabella di sopra. Di queste, solo le righe con lavoro.dipendente = 'FRNDPP76' vengono trattenute:

teatro	dipendente	ruolo	nome	città	email
Litta	FRNDPP76	finanza	Litta	Milano	litta@gmail.com
Eliseo	FRNDPP76	controllo	Eliseo	Roma	eliseo@gmail.com

Infine viene proiettata solo la colonna città specificata nella clausola `select`:

città
Milano
Roma

Una visione operazionale alternativa del join è la seguente. Supponiamo di dover fare un join tra due tabelle R e S con condizione di join theta. Se dovessimo programmare questa operazione, dovremmo scrivere **due cicli for annidati**. Il ciclo esterno scandisce le righe di R. Per ogni riga r di R, il ciclo interno scandisce le righe s di S e verifica se la riga congiunta rs soddisfa theta. In tal caso la riga viene selezionata.

E' possibile fare il join di più di due tabelle. Supponiamo di voler selezionare il nome, il cognome e la città di lavoro di ogni dipendente. Queste informazioni sono sparse su due tabelle (dipedente e teatro) che devono essere collegate attraverso una terza tabella, lavoro. Dunque tre tabelle sono coinvolte nel join:

```
select dipendente.nome, dipendente.cognome, teatro.città
from lavoro, dipendente, teatro
where (lavoro.dipendente = dipendente.cf) and (lavoro.teatro = teatro.nome)
```

Il risultato è la seguente tabella:

nome	cognome	città
Elisa	D'Allarche	Udine
Fernanda	D'Ippoliti	Roma
Fernanda	D'Ippoliti	Milano

Marco	Dall'Aglio	Roma

Abbiamo visto che tecnica di denominazione degli attributi facendoli precedere la nome della tabella è utile per aumentare la leggibilità dell'interrogazione. Inoltre, questa tecnica è indispensabile per individuare un attributo senza ambiguità se vi sono attributi con lo stesso nome in tabelle diverse. Si noti che l'ambiguità rimane se le tabelle hanno lo stesso nome, cioè sono istanze diverse della stessa tabella. E' infatti possibile concatenare nella clausola `from` più istanze della stessa tabella. Per eliminare l'ambiguità in questo caso è possibile rinominare le istanze della stessa tabella con il costrutto `as`.

Vediamo un esempio. Supponiamo di aggiungere alla tabella dipendente un attributo capo che contenga il codice fiscale del capo del dipendente. In particolare, capo è una chiave esterna e si riferisce alla chiave primaria cf della tabella dipendente stessa. Supponiamo che un dipendente possa avere o meno un capo e che un capo possa dirigere zero o più dipendenti:

dipendente			
cf	**nome**	**cognome**	**capo**
ELSDLL72	Elisa	D'Allarche	NULL
FRNDPP76	Fernanda	D'Ippoliti	ELSDLL72
MRCDLL70	Marco	Dall'Aglio	ELSDLL72

Cerchiamo il nome e cognome dei capi di tutti i dipendenti:

```
select d1.nome, d1.cognome, d2.nome as nomeCapo, d2.cognome as cognomeCapo
from dipendente as d1, dipendente as d2
where d1.capo = d2.cf
```

Abbiamo usato la parola chiave `as`, che può essere omessa, per rinominare sia gli attributi che le tabelle. Il risultato del join è il seguente:

dipendente							
cf	**nome**	**cognome**	**capo**	**cf**	**nome**	**cognome**	**capo**
ELSDLL72	Elisa	D'Allarche	NULL	ELSDLL72	Elisa	D'Allarche	NULL
FRNDPP76	Fernanda	D'Ippoliti	ELSDLL72	FRNDPP76	Fernanda	D'Ippoliti	ELSDLL72
MRCDLL70	Marco	Dall'Aglio	ELSDLL72	MRCDLL70	Marco	Dall'Aglio	ELSDLL72

Il risultato dell'interrogazione è il seguente:

nome	cognome	nomeCapo	cognomeCapo
Fernanda	D'Ippoliti	Elisa	D'Allarche
Marco	Dall'Aglio	Elisa	D'Allarche

In SQL-2 è stata definita una sintassi particolare per i join e sono stati distinti 4 diversi tipi di join. La sintassi del join è la seguente:

```
select d1.nome, d1.cognome, d2.nome as nomeCapo, d2.cognome as cognomeCapo
from dipendente d1 join dipendente d2 on d1.capo = d2.cf
```

La tabella risultato dell'operazione di join è detta **tabella congiunta** (*joined table*). Le tabelle argomento dell'operazione di join possono essere lore stesse tabelle congiunte. Ad esempio, il seguente join recupera i dipendenti con stipendio inferiore a 1000 e i capi dei loro capi:

```
select d1.nome, d1.cognome, d3.nome as nomeSuperCapo, d3.cognome as cognomeSuperCapo
from (dipendente d1 join dipendente d2 on d1.capo = d2.cf)
      join dipendente d3 on d2.capo = d3.cf
where d1.stipendio < 1000
```

Un vantaggio di questa sintassi è che la condizione di join viene isolata dalle altre condizioni di selezione delle righe, che possono essere aggiunte mediante la clausola `where`. Precisamente, una **condizione di join** tra due tabelle R e S è una congiunzione (secondo l'operatore Booleano `and`) di condizioni atomiche di tipo A theta B, dove A è un attributo di R, B è un attributo di S, e theta è un operatore di confronto. Di solito, ma non necessariamente, A è una chiave primaria, B è una chiave esterna riferita ad A e theta è l'operatore di uguaglianza.

Un secondo vantaggio di questa sintassi per il join è che ci permette di distinguere diverse forme di join. Si noti che nell'ultima interrogazione fatta Elisa non viene selezionato nel risultato come dipendente. Questo perchè ella non ha un capo e dunque la sua riga nell'istanza d1 di dipendente non incontra nessuna riga nell'istanza d2 di dipendente che verifichi la condizione di join d1.capo = d2.cf e dunque non viene inserita nel risultato. Questa situazione è spiacevole perchè non ci informa che Elisa non ha capi. E' possibile risolvere questo problema usando un **left join**:

```
select d1.nome, d1.cognome, d2.nome as nomeCapo, d2.cognome as cognomeCapo
from dipendente d1 left join dipendente d2 on d1.capo = d2.cf
```

che produce il seguente risultato:

nome	cognome	nomeCapo	cognomeCapo
Elisa	D'Allarche	NULL	NULL
Fernanda	D'Ippoliti	Elisa	D'Allarche
Marco	Dall'Aglio	Elisa	D'Allarche

Similmente, i dipendenti che non sono a capo di alcun dipendente (Fernanda e Marco) non fanno parte del risultato come capi. E' possibile risolvere questo problema usando un **right join**:

```
select d1.nome, d1.cognome, d2.nome as nomeCapo, d2.cognome as cognomeCapo
from dipendente d1 right join dipendente d2 on d1.capo = d2.cf
```

nome	cognome	nomeCapo	cognomeCapo
Fernanda	D'Ippoliti	Elisa	D'Allarche
Marco	Dall'Aglio	Elisa	D'Allarche
NULL	NULL	Fernanda	D'Ippoliti
NULL	NULL	Marco	Dall'Aglio

Si noti che lo stesso risultato si avrebbe con il seguente left join a tabelle invertite:

```
select d1.nome, d1.cognome, d2.nome as nomeCapo, d2.cognome as cognomeCapo
from dipendente d2 left join dipendente d1 on d1.capo = d2.cf
```

Un **full join** unisce un left join e un right join:

```
select d1.nome, d1.cognome, d2.nome as nomeCapo, d2.cognome as cognomeCapo
from dipendente d1 full join dipendente d2 on d1.capo = d2.cf
```

nome	cognome	nomeCapo	cognomeCapo
Elisa	D'Allarche	NULL	NULL
Fernanda	D'Ippoliti	Elisa	D'Allarche
Marco	Dall'Aglio	Elisa	D'Allarche
NULL	NULL	Fernanda	D'Ippoliti
NULL	NULL	Marco	Dall'Aglio

Ordinamento delle tuple

Una tabella contiene tuple non ordinate. E' possibile ordinare le tuple secondo determinati criterio con il costrutto `order by`. Ad esempio, la seguente interrogazione ordina le tuple della tabella dipendente in ordine crescente per nome e cognome, e in ordine decrescente per stipendio:

```
select *
from dipendente
order by nome, cognome, stipendio desc
```

Le tuple vengono ordinate in ordine crescente secondo i valori dell'attributo nome. Due dipendenti con lo stesso nome vengono ordinati secondo il loro cognome in senso crescente. Infine due

dipendenti con lo stesso nome e cognome vengono ordinati secondo lo stipendio in ordine discendente.

Operatori aggregati

Un operatore aggregato è una funzione che si applica ad un insieme di tuple di una tabella e ha come risultato un valore atomico. Lo standard SQL prevede i seguenti operatori aggregati:

count

Questo operatore serve per contare le tuple di una tabella. Può essere usato nei seguenti tre modi:

```
select count(*)
from dipendente
```

Il risultato è il numero di tuple della tabella dipendente.

```
select count(all nome)
from dipendente
```

Il risultato è il numero di valori **non nulli** dell'attributo nome della tabella dipendente. La parola chiave `all` può essere omessa ottenendo lo stesso risutato.

```
select count(distinct nome)
from dipendente
```

Il risultato è il numero di valori **distinti e non nulli** dell'attributo nome della tabella dipendente.

min e **max**

Restituiscono rispettivamente il minimo e il massimo di una espressione valutata sulle tuple di una tabella. L'espressione deve restituire valori su cui è definito un ordinamento (numeri, stringhe, istanti temporali). Ad esempio, la seguente interrogazione restituisce lo stipendio massimo di un dipendente:

```
select max(stipendio)
from dipendente
```

La seguente interrogazione restituisce la data di nascita più remota di un dipendente:

```
select min(dataDiNascita)
from dipendente
```

sum e **avg**

Restituiscono rispettivamente la somma e la media di una espressione valutata sulle tuple di una tabella. L'espressione deve restituire valori su cui è definita la somma (numeri). Ad esempio, la seguente interrogazione restituisce la somma degli stipendi dei dipendenti:

```
select sum(stipendio)
from dipendente
```

Le seguenti due interrogazioni restituiscono entrambe la media degli stipendi dei dipendenti:

```
select avg(stipendio)
from dipendente
```

```
select sum(stipendio)/count(stipendio)
from dipendente
```

E' possibile usare la parole chiave `distinct` per rimuovere i valori duplicati dal conteggio effettuato dagli operatori `sum` e `avg`.

Si noti che non è possibile mescolare nella clausola `select` espressioni aggregate, cioè espressioni che restituiscono un valore per un insieme di tuple, e espressioni di tupla, cioè espressioni che restituiscono un valore per ogni singola tuple. Dunque la sequente query è scorretta:

```
select nome, max(stipendio)
from dipendente
```

Raggruppamenti

Gli esempi di operatori aggregati visti fin ora operano sull'insieme di *tutte* le righe di una tabella. La clausola `group by` permette di partizionare le righe di una tabella in sottoinsiemi e applicare gli operatori aggregati ai singoli sottoinsiemi. Tale clausola ha come argomento un insieme di attributi e raggruppa le righe che posseggono lo stesso valore per gli attributi dell'insieme argomento. Nella clausola `select` posso usare operatori aggregati e attributi ma quest'ultimi devono essere inclusi nella clausola `group by`. Si consideri la seguente tabella lavoro:

lavoro

teatro	dipendente	ruolo
CSS	ELSDLL72	finanza
CSS	ABCDEF74	direzione
Litta	FRNDPP76	finanza
Litta	GHILMN77	finanza
Eliseo	FRNDPP76	controllo
Eliseo	MRCDLL70	direzione

Supponiamo di voler calcolare il numero di dipendenti per ogni ruolo. Possiamo scrivere la seguente interrogazione:

```
select ruolo, count(*) as numero
from lavoro
group by ruolo
```

La query raggruppa le tuple di lavoro secondo i valori dell'attributo ruolo. Vengono creati tre insiemi corrispondenti ai valori di finanza (3 righe), direzione (2 righe) e controllo (1 riga). Su ogni insieme viene selezionato il valore di ruolo e il numero di righe mediante l'operatore

aggretato `count`. Si noti che, per ogni sottoinsieme identificato, esiste un unico valore di ruolo, che quindi può essere resituito. Non è così per gli attributi teatro e dipendente, che non possono essere usati nella clausola `select` in questo caso. Il risultato è la seguente tabella:

ruolo	numero
finanza	3
direzione	2
controllo	1

Se vogliamo calcolare il numero di dipendenti per ogni ruolo del teatro CSS possiamo scrivere la seguente interrogazione:

```
select ruolo, count(*) as numero
from lavoro
where teatro = 'CSS'
group by ruolo
```

ruolo	numero
finanza	1
direzione	1

Si noti che la tabella su cui opera la clausola `group by` può essere il risultato di una query, ad esempio una tabella risultato di un join come nel seguente esempio:

```
select ruolo, count(*), sum(stipendio)
from lavoro, dipendente
where lavoro.dipendente = dipendente.cf
group by ruolo
```

Se vogliamo selezionare solo i gruppi che soddisfano un certo predicato possiamo usare la clausola `having`. Un predicato sui gruppi può usare operatori aggregati e attributi della clausola che compaiono in `group by`. Ad esempio, se vogliamo calcolare il numero di dipendenti per ogni ruolo selezionando solo i gruppi di almeno due dipendenti possiamo scrivere nel seguente modo:

```
select ruolo, count(*) as numero
from lavoro
group by ruolo
having count(*) >= 2
```

Si badi bene alla differenza tra le clausole `where` e `having`: la clausola `where` contiene **predicati di tupla** e serve per filtrare le tuple delle tabelle, la clausola `having` contiene **predicati di gruppo** e serve per filtrare i gruppi ottenuti mediante la clausola `group by`. Concettualmente, prima si applica il filtro `where` e poi quello `having`.

Le clausole `select`, `from`, `where`, `group by`, `having`, `order by` debbono essere usate in quest'ordine e sono tutte le clausole che si possono usare in SQL. Solo le clausole `select` e `from` sono obbligatorie. La clausola `having`prevede l'uso di `group by`. Come esempio riepilogativo selezioniamo il numero di dipendenti per ogni ruolo del teatro CSS per gruppi maggiori di 1 ordinando il risultato per ruolo:

```
select ruolo, count(*) as numero
from lavoro
where teatro = 'CSS'
group by ruolo
having count(*) > 1
order by ruolo
```

Concettualmente la query viene eseguita nel seguente modo:

1. vengono selezionate le righe della tabella lavoro (`from`);
2. vengono filtrate le righe che corrispondono al teatro CSS (`where`);
3. le righe vengono suddivise in gruppi che hanno lo stesso valore di ruolo (`group by`);
4. vengono filtrati i gruppi di cardinalità maggiore di 1 (`having`);
5. i gruppi risultati vengono ordinati lessicograficamente secondo il valore di ruolo (`order by`);
6. per ogni gruppo viene selezionato il valore di ruolo e calcolato il numero di righe (`select`);

Si noti che la clausola `select` viene scritta per prima ma concettualmente eseguita per ultima. Inoltre, non è detto che la sequenza concettuale descritta corrisponda alla sequenza operazionale implementata dal DBMS il quale, per ragioni di efficienza, può eseguire la query in un ordine differente.

Operatori insiemistici

SQL permette di fare l'unione (`union`), l'intersezione (`intersect`) e la differenza (`except`) tra insiemi di righe di tabelle. Questi operatori richiedono che gli schemi su cui operano abbiano lo stesso numero di attributi e che i loro domini siano compatibili. Per default, questi operatori eliminano i duplicati dal risultato, restituendo quindi un insieme nel senso matematico del termine. E' possibile recuperare i duplicati facendo seguire la parola chiave `all` al nome dell'operatore. Vediamo alcuni esempi sulle seguenti tabelle:

attore

nome	**dataDiNascita**	**luogoDiNascita**
Elisa Bottega	1972-04-29	Treviso
Lorenzo Vignando	1970-05-29	Venezia
Barbara Altissimo	1982-03-01	Torino

autore		
nome	**dataDiNascita**	**luogoDiNascita**
Vittorio Cortellessa	1969-11-29	Napoli
Lorenzo Vignando	1970-05-29	Venezia
Antinisca Di Marco	1972-08-01	L'Aquila

Selezioniamo tutte le persone che sono attori oppure autori:

```
select nome
from attore
   union
select nome
from autore
```

nome
Elisa Bottega
Lorenzo Vignando
Barbara Altissimo
Vittorio Cortellessa
Antinisca Di Marco

Si noti che l'autore e attore Lorenzo Vignando compare solo una volta nel risultato.

Selezioniamo tutte le persone che sono attori e autori:

```
select nome
from attore
   intersect
select nome
from autore
```

nome
Lorenzo Vignando

Selezioniamo tutte le persone che sono attori ma non sono autori:

```
select nome
```

```
from attore
   except
select nome
from autore
```

nome
Elisa Bottega
Barbara Altissimo

Interrogazioni nidificate

Una **interrogazione nidificata** è una interrogazione che contiene un'altra interrogazione. SQL non pone limiti al livello di annidamento ma solitamente più di due annidamenti rendono incomprensibile una query ad un umano (una macchina, invece, non ha problemi di comprensione in questo caso).

E' possibile nidificare una interrogazione nella clausola `where` (vedremo un'altra forma di annidamento parlando delle viste). In particolare una espressione può essere confrontata mediante gli usuali operatori di confronto con una interrogazione. L'operatore di confronto è seguito dalla parola chiave `any` oppure `all`. Nel primo caso, il predicato è vero se ha successo il confronto (secondo l'operatore usato) tra il valore dell'espressione e **almeno uno** tra i valori risultato della query. Nel secondo caso, il predicato è vero se ha successo il confronto (secondo l'operatore usato) tra il valore dell'espressione e **tutti** i valori risultato della query. L'espressione e il risultato dell'interrogazione devono essere compatibili, cioè avere lo stesso numero di attributi e gli attributi devono evere domini compatibili.

Vediamo alcuni esempi. Consideriamo le tabelle attore a autore usate nelle interrogazioni insiemistiche. La seguente interrogazione seleziona tutti gli attori che sono anche autori. Si noti che abbiamo già scritto la stessa query usando l'operatore `intersect`:

```
select nome
from attore
where nome = any (select nome
                  from autore)
```

Si noti che la query interna è scritta tra parentesi tonde. La seguente interrogazione seleziona tutti gli attori che non sono autori. Si noti che abbiamo già scritto la stessa query usando l'operatore `except`:

```
select nome
from attore
where nome <> all (select nome
                   from autore)
```

Selezionare tutti gli attori o autori è invece possibile solo usando l'operatore `union`. Le combinazioni `= any` e `<> all` sono molto frequenti e hanno un nome: nel primo caso posso usare la parola chiave `in` e nel secondo `not in`. Ad esempio:

```
select nome
from attore
where nome not in (select nome
                   from autore)
```

Se il nome e il cognome di un attore fosse stato specificanto usando due attributi invece che uno soltato, avremmo potuto usare il costruttore di tupla `(nome, cognome)` in questo modo:

```
select nome, cognome
from attore
where (nome, cognome) not in (select nome, cognome
                              from autore)
```

Il nome del dipendente con lo stipendio massimo può essere estratto nei seguenti due modi:

```
select nome, stipendio
from dipendente
where stipendio >= all (select stipendio
                        from dipendente)
select nome, stipendio
from dipendente
where stipendio = (select max(stipendio)
                   from dipendente)
```

Si noti che nel secondo caso la parola chiave `all` o `any` può essere omessa in quanto la query interna restituisce un solo elemento.

L'interrogazione interna può essere sostituita con un **insieme esplicito**. L'interrogazione seguente recupera tutti i nomi dei dipendenti in un certo insieme esplicito:

```
select nome
from dipendente
where nome in ("Elisa Bottega", "Lorenzo Vignando", "Barbara Altissimo")
```

Le interrogazioni nidificate viste fin ora possono essere risolte indipendentemente dall'interrogazione che le contiene. Ad esempio, nell'ultima interrogazione scritta, è possibile prima di tutto risolvere la query interna, cioè calcolare il massimo stipendio, e poi elaborare la query esterna confrontando il massimo stipendio con il valore dello stipendio di ogni dipendente. Questa soluzione è più efficiente rispetto a quella che valuta la query interna per ogni tupla della query esterna. Questa soluzione non può però sempre essere seguita. Consideriamo una interrogazione che restituisce gli attori per i quali esiste un altro attore con la medesima data di nascita. Osserviamo innanzitutto che è possibile risolvere l'interrogazione con il seguente join:

```
select distinct A1.nome
from attore A1 join attore A2 on
(A1.dataDiNascita = A2.dataDiNascita) and
(A1.nome <> A2.nome)
```

Vediamo ora una soluzione che usa una query nidificata e l'operatore `exists`. Questo operatore ha come argomento una query interna e restituisce vero se e soltanto se il risultato della query argomento contiene qualche elemento:

```
select A1.nome
from attore A1
```

```
where exists (select A2.nome
              from attore A2
              where (A1.dataDiNascita = A2.dataDiNascita) and
                    (A1.nome <> A2.nome))
```

Questo tipo di query si dicono **query nidificate con passaggio di variabili**. In particolare, la variabile A1, creata nella query esterna per la prima istanza della tabella attore, viene passata alla query interna che ne fa uso. In tal caso la query interna non può essere valutata indipendentemente da quella esterna in quanto si serve di variabili definite a livello di query esterna. Dunque l'unico modo di risolvere questa query consiste nel valutare la query interna per ogni tupla della query esterna. La **visibilità delle variabili** in SQL segue la seguente semplice regola: una variabile è visibile nella query che l'ha definita o in una query nidificata in essa (ad un qualsiasi livello).

Vediamo un esempio che usa la negazione dell'operatore `exists`: gli attori per i quali non esiste un altro attore con la medesima data di nascita:

```
select A1.nome
from attore A1
where not exists (select A2.nome
                  from attore A2
                  where (A1.dataDiNascita = A2.dataDiNascita) and
                        (A1.nome <> A2.nome))
```

Una formulazione che non usa le query nidificate è la seguente:

```
select nome
from attore
  except
select A1.nome
from attore A1 join attore A2 on
(A1.dataDiNascita = A2.dataDiNascita) and
(A1.nome <> A2.nome)
```

Aggiornamento della base di dati

SQL permette di aggiornare lo stato della base di dati mediante inserimenti, modifiche e cancellazioni di righe di tabelle.

Inserimento

E' possibile inserire una o più righe in una tabella con il comando `insert`. Consideriamo il seguente schema di relazione:

dipendente(cf, nome, cognome, stipendio)

Il seguente comando inserisce un nuovo dipendente:

```
insert into dipendente(cf, nome, cognome, stipendio)
values ('ALSBRT69', 'Alessio', 'Bertallot', '1000')
```

Il seguente comando inserisce come dipendenti tutti gli attori presenti nella tabella attore:

```
insert into dipendente(cf, nome, cognome)
select cf, nome, cognome
from attore
```

Se in un inserimento non vengono specificati tutti gli attributi della tabella, gli attributi non specificati assumono il valore di default, se definito, oppure il valore nullo.

Cancellazione

E' possibile cancellare una o più righe da una tabella con il comando `delete`. Ad esempio, per cancellare tutti i dipendenti posso scrivere il seguente comando:

```
delete from dipendente
```

Si noti che questo comando cancella il contenuto della tabella ma, a differenza del comando `drop`, non rimuove la tabella dallo schema della base di dati. Per cancellare solo alcune righe, posso aggiungere la clausola `where` che segue la sintassi già vista per le interrogazioni di selezione di dati. Ad esempio, per rimuovere i dipendenti disoccoupati, cioè che non partecipano alla tabella lavoro, posso scrivere:

```
delete from dipendente
where cf not in (select dipendente
                 from lavoro)
```

Modifica

E' possibile modificare una o più righe di una tabella con il comando `update`. Per incrementare del 10% lo stipendio del dipendente identificato dal codice ALSBRT69 posso scrivere come segue:

```
update dipendente
set stipendio = stipendio * 1.1
where cf = 'ALSBRT69'
```

Posso incrementare lo stipendio di tutti gli impiegati con stipendio inferiore a 1000:

```
update dipendente
set stipendio = 1000
where stipendio < 1000
```

Posso modificare più attributi per ogni tupla inserendo la lista di assegnamenti, separati da uan virgola, nella clausola `set`. Ogni attributo può essere assegnato al valore di una espressione, al valore di una interrogazione, al valore di default (scrivendo `default`) e al valore nullo (scrivendo `null`). Ad esempio, il seguente comando assegna lo stipendio medio a tutti i dipendenti (si noti che in assenza della clausola `where` tutte le righe della tabella vengono modificate):

```
update dipendente
set stipendio = (select avg(stipendio)
                 from dipendente)
```

Supponiamo infine di voler incrementare del 20% gli stipendi inferiori a 1000, del 10% quelli tra 1000 e 2000, e lasciare invariati gli altri. Posso in questo caso usare una interrogazione che fa uso del costrutto `case`:

```
update dipendente
set stipendio =
case
 when (stipendio < 1000)
 then stipendio = stipendio * 1.2
 when ((stipendio >= 1000) and (stipendio <= 2000))
 then stipendio = stipendio * 1.1
 else stipendio
end
```

Si noti che, grazie al comando `case`, ogni riga viene incrementata una sola volta. Si avrebbe un effetto diverso usando i seguenti due comandi in cascata:

```
update dipendente
```

```
set stipendio = stipendio * 1.2
where stipendio < 1000

update dipendente
set stipendio = stipendio * 1.1
where (stipendio >= 1000) and (stipendio <= 2000)
```

Infatti, uno stipendio di 900 verrebbe incrementato due volte.

Definizione dei dati

SQL permette di definire le strutture e i vincoli sui dati. In particolare, esso consente:

1. la definizione della **struttura delle tabelle**. In particolare, occorre specificare il nome della tabella e, per ogni attributo, il nome, il dominio e un possibile valore predefinito;
2. la definizione dei **vincoli di integrità**. Tali vincoli si distinguono in vincoli tipici del modello relazionale (chiavi e obligatorietà degli attributi) e vincoli di integrità generici (regole aziendali);
3. la definizione delle **viste sui dati**.

Tabelle

A differenza di una definizione di relazione a livello logico, la definizione di una tabella presuppone anche l'associazione delle colonne ai relativi domini atomici. Un **dominio atomico** corrisponde sostanzialmente ad un tipo di dato semplice dei linguaggi di programmazione. I principali domini atomici offerti da SQL sono:

- **stringhe**. Per definire una stringa di esattamente n caratteri si scrive `char(n)`, per una stringa di al massino n caratteri si scrive `varchar(n)`. `char` è una abbreviazione di `char(1)`;
- **numeri**. Per i numeri interi si usa il dominio `integer` oppure `smallint`, per i numeri frazionari in virgola fissa si usa `decimal(p,s)`, dove p è la precisione (il numero di cifre decimali utilizzate per il numero) e s è la scala (il numero di cifre decimali della parte frazionaria). Ad esempio, `decimal(4,2)` rappresenta tutti i numeri da -99,99 a +99,99 con incrementi pari a un centesimo. Per i numeri frazionari in virgola mobile si possono usare i domini `float`, `real` e `double precision`;
- **istanti temporali**. E' possibile usare il dominio `date` per le date (YYYY-MM-DD), `time` per le ore (HH:MM:SS) e `timestamp` per gli istanti temporali composti da una data e un'ora (YYYY-MM-DD HH:MM:SS). Il tipo `time(n)` specifica una precisione di n cifre dopo la virgola per i secondi. Ad esempio, 13:24:50,25 appartiene a `time(2)`

SQL-3 ha anche introdotto il dominio `boolean` con valore `true` e `false`, il dominio `BLOB` (*Binary Large Object*) per oggetti di gradi dimensioni costituiti da sequenze di valori binari (ad esempio una immagine) e il dominio `CLOB` (*Character Large Object*) per oggetti di grandi dimensioni costituiti da sequenze di caratteri (ad esempio un documento di testo). I valori di tipo BLOB e CLOB solitamente memorizzati in una zona apposita separata dagli altri dati.

E' possibile costruire nuovi domini *atomici* con il costrutto `create domain`. Seguono due esempi:

```
create domain voto as smallint
check (voto >= 18 and voto <=30)
```

Una colonna con dominio voto può assumere solo valori interi da 18 a 30.

```
create domain tipoBiglietto as varchar(8)
default "Intero"
check (tipoBiglietto in ("Intero", "Ridotto", "Studenti"))
```

Una colonna con dominio tipoBiglietto può assumere solo le stringhe 'Intero', 'Ridotto' o 'Studenti'. Se in ingresso il valore non è specificato si assume il valore 'Intero'.

Vediamo quindi come è possibile definire le tabelle in SQL. Innanzitutto, è possibile **definire uno schema** di base di dati con il comando `create schema`:

```
create schema TeatroSQL
{definizione delle componenti}
```

Il comando ha creato uno schema dal nome TeatroSQL. Tra parentesi graffe vanno inserite le definizioni delle varie componenti, vale a dire domini, tabelle, vincoli di integrità e viste. Le componenti possono essere associate ad uno schema anche in un secondo momento. I nomi delle componenti all'interno di uno schema devono essere univoci. Schemi diversi possono però avere componenti con lo stesso nome. Gli schemi possono dunque essere usati per mantenere versioni diverse della stessa base di dati. Se non viene specificato nessuno schema viene usato uno schema di default associato all'utente che si è connesso alla base di dati.

Per **definire una tabella** esiste il comando `create table`. Occorre specificare il nome della tabella e, per ogni attributo, il nome, il dominio e un possibile valore predefinito. Segue un esempio:

```
create table dipendente
(
  cf               char(16),
  nome             varchar(20),
  cognome          varchar(20),
  dataDiNascita    date,
  luogoDiNascita   varchar(20),
  età              smallint,
  sesso            char,
  statoCivile      varchar(10) default 'libero'
)
```

Il valore predefinito (*default*) è il valore che assume l'attributo qualora il suo valore non sia specificato in fase di inserimento di una riga nella tabella. Se il valore di dafault non è specificato, si assume che esso sia il valore nullo (NULL). Se vogliamo associare la tabella ad uno schema specifico, occorre far precedere al nome della tabella il nome dello schema separato da un punto, ad esempio `TeatroSQL.dipendente`.

Una volta definiti, un dominio e una tabella possono essere modificati con il comando `alter`. In particolare, è possibile aggiungere e rimuovere colonne da una tabella come nel seguente esempio:

```
alter table dipendente add column residenza varchar(30) not null

alter table dipendente drop column statoCivile
```

E' anche possibile rimuovere uno schema, un dominio e una tabella con il comando `drop`. Esistono due opzioni: `restrict`, che rimuove il componente solo se vuoto e `cascade` che rimuove comunque il componente. Quest'ultima opzione è da usare con cautela in quanto genera una reazione a catena per cui tutti i componenti che dipendono dal componente rimosso sono anch'essi cancellati. Segue un esempio:

```
drop table dipendente restrict
```

Vincoli di integrità

In SQL è possibile specificare dei vincoli di integrità sui dati, sia quelli propri del modello relazionale che quelli che specificano le regole aziendali.

Vincoli di dominio. Essi vengono implicitamente specificati quando un attributo viene associato al corrispondente dominio: i valori dell'attributo devono appartenere al relativo dominio.

Obligatorietà degli attributi. In SQL, per default, un attributo (non facente parte della chiave primaria) è opzionale, cioè può assumere valori nulli. Per rendere obbligatorio un attributo, cioè per fare in modo che il suo valore sia sempre diverso da NULL, si usa il vincolo `not null` da usare come nel seguente esempio:

```
nome   varchar(20) not null
```

Se il valore di un attributo obbligatorio non viene specificato in fase di inserimento di una riga, il DBMS genera un errore. Si noti che non è necessario specificare un valore, in fase di inserimento, per un attributo obbligatorio con valore di default diverso da NULL.

Chiave primaria. E' possibile identificare una unica chiave primaria con il vincolo `primary key` da usare nel seguente modo:

```
create table dipendente
(
  cf               char(16) primary key,
  nome             varchar(20),
  cognome          varchar(20),
)
```

Se il vincolo coinvolge più attributi è possibile aggiungerlo dopo la definizione degli attributi come segue:

```
create table dipendente
(
```

```
  nome              varchar(20),
  cognome           varchar(20),
  dataDiNascita     date,
  luogoDiNascita    varchar(20),
  primary key (nome, cognome)
)
```

Gli attributi della chiave primaria devono assumere valori univoci e devono essere obbligatori. Per loro non occorre specificare il vincolo di obligatorietà. Una violazione di questo vincolo genera un errore.

Chiavi candidate. E' possibile specificare altre chiavi candidate con il vincolo `unique`. La specifica delle chiavi candidate avviene come per le chiavi primarie, ad esempio:

```
create table dipendente
(
  cf                char(16) primary key,
  nome              varchar(20),
  cognome           varchar(20),
  unique (nome, cognome)
)
```

Si noti che per la chiave primaria non occorre specificare il vincolo di univocità. I valori assunti da una chiave candidata devono essere univoci ma possono essere nulli. (a differenza di quanto accade per la chiave primaria). Una violazione di questo vincolo genera un errore.

Chiavi esterne. Il vincolo di chiave esterna (*foreign key*) coinvolge due tabelle (che possono essere due istanze della stessa tabella). La tabella che contiene la chiave esterna è detta **secondaria** (*slave*), mentre la tabella a cui fa riferimento la chiave esterna è detta **principale** (*master*). Gli attributi riferiti nella tabella principale devono formare una *chiave candidata*, di solito la chiave primaria. Il vincolo specifica che ogni valore *non nullo* della chiave esterna nella tabella secondaria deve corrispondere ad un valore nella tabella principale. La definizione del vincolo di chiave esterna usa il costrutto `foreign key` come segue:

```
create table teatro
(
  nome              varchar(20) primary key,
  indirizzo         varchar(40) not null,
  email             varchar(20) unique
)

create table biglietteria
(
  nome              varchar(20) primary key,
  indirizzo         varchar(40) not null,
  teatro            varchar(20) foreign key references teatro(nome)
)
```

Nell'esempio la tabella principale è teatro, la tabella secondaria è biglietteria e il vincolo lega l'attributo teatro di biglietteria all'attributo nome di teatro. Se il vincolo coinvolge più attributi è possibile specificarlo dopo la definizione degli attributi della tabella nel seguente modo:

```
foreign key(nomeDipendente, cognomeDipendente)
references dipendente(nome, cognome)
```

A differenza degli altri vincoli di integrità discussi, una violazione di un vincolo di chiave esterna non genera necessariamente un errore. E' infatti possibile stabilire diverse politiche per reagire ad una violazione. La violazione può avvenire in due modi:

1. nella tabella secondaria, inserisco una nuova riga o modifico la chiave esterna. Si noti che la cancellazione di una riga dalla tabella secondaria non viola mai il vincolo di chiave esterna;
2. nella tabella principale, cancello una riga o modifico la chiave riferita. Si noti che l'inserimento di una nuova riga dalla tabella principale non viola mai il vincolo di chiave esterna.

Nel primo caso viene sempre generato un errore. Nel secondo posso stabilire le delle **politiche di reazione**. Per le operazione di modifica vi sono le seguenti politiche:

- `cascade`: il nuovo valore dell'attributo della tabella principale viene riportato su tutte le corrispondenti righe della tabella secondaria;
- `set null`: alla chiave esterna della tabella secondaria viene assegnato il valore nullo al posto del valore modificato nella tabella principale;
- `set default`: alla chiave esterna della tabella secondaria viene assegnato il corrispondente valore di default al posto del valore modificato nella tabella principale;
- `no action`: nessuna azione viene intrapresa e viene generato un errore.

Per le operazione di cancellazione posso reagire come segue:

- `cascade`: le corrispondenti righe della tabella secondaria vengono cancellate;
- `set null`: alla chiave esterna della tabella secondaria viene assegnato il valore nullo al posto del valore cancellato nella tabella principale;
- `set default`: alla chiave esterna della tabella secondaria viene assegnato il corrispondente valore di default al posto del valore cancellato nella tabella principale;
- `no action`: nessuna azione viene intrapresa e viene generato un errore.

La sintassi per specificare queste politiche usa i costrutti `on update` e `on delete` come segue:

```
create table biglietteria
(
  nome              varchar(20) primary key,
  indirizzo         varchar(40) not null,
  teatro            varchar(20) foreign key references teatro(nome)
                                on update cascade
                                on delete set null
)
```

Come regola generale, per le modifiche si usa la politica `on update cascade`. Per le cancellazioni si usa la politica `on delete cascade` per chiavi esterne di tabelle che corrispondono a relazioni concettuali (ad esempio la tabella lavoro nel nostro caso di studio) oppure ad entità deboli (ad esempio la tabella prenotazione) e la politica `on delete set`

`null` negli altri casi. La ragione è che nel primo caso vi è un forte collegamento tra la tabella master e la tabella slave e dunque una cancellazione nella tabella master dovrebbe provocare corrispondenti cancellazioni nella tabella slave.

Regole aziendali. Ulteriori vincoli, detti **vincoli di integrità generici** in quanto non legati al modello relazionale, sono quelli imposti dalle regole aziendali. Tali vincoli possono essere rappresentati in SQL in più modi: mediante il costrutto `check` nella definizione di una tabella, mediante le asserzioni, oppure attraverso l'uso di regole attive (*trigger*).

E' bene chiarire in anticipo che i vincoli di integrità generici rappresentano un argomento contrastato. A differenza dei vincoli relazionali, gli strumenti per specificare vincoli generici non sono stabilmente inseriti nello standard SQL (ad esempio i trigger sono stati aggiunti sono nell'ultima versione di SQL dopo essere stati disponibili per molto tempo nei DBMS). Di conseguenza, mentre i vincoli tipici del modello relazionale sono supportati efficientemente da tutti i DBMS relazionali, gli strumenti per specificare vincoli generici variano notevolmente tra i vari DBMS disponibili e non sempre garantiscono l'efficienza del sistema risulante. E' quindi fortemente consigliato accertarsi di quali siano e di come funzionino gli strumenti per vincoli generici supportati dal DBMS prescelto.

Il costrutto `check` permette di specificare, mediante una condizione come quella che può apparire nella clausola `where` di una interrogazione SQL, vincoli generici a livello di di tabella o, mediante le asserzioni, a livello di schema di base di dati.

Si noti che un uso indiscriminato di questi vincoli appesantisce il carico del sistema in quanto, solitamente, i DBMS garantiscono una implementazione efficiente solo per i vincoli propri del modello relazionale.

Vediamo alcuni esempi. Supponiamo di avere le tabelle dipendente, teatro e lavoro. La tabella lavoro associa i dipendenti ai relativi teatri. Un teatro può avere più dipendenti e un dipendente può lavorare per più teatri. Supponiamo di voler esprimere un **vincolo massimo di partecipazione** per un dipendente rispetto alla relazione lavoro: un dipendente non può lavorare per più di due teatri. Questo vincolo può essere specificato sulla tabella lavoro nel seguente modo:

```
create table teatro
(
  nome              varchar(20) primary key,
  indirizzo         varchar(40) not null
)

create table dipendente
(
  cf                      char(16) primary key,
  nome                    varchar(20) not null,
  cognome                 varchar(20) not null,
  dataDiNascita           date,
  luogoDiNascita          varchar(30),
  capo                    char(16),
  foreign key capo references dipendente(cf)
)

create table lavoro
(
```

```
  teatro              varchar(20),
  dipendente          char(16),
  primary key(teatro, dipendente),
  foreign key teatro references teatro(nome),
  foreign key dipendente references dipendente(cf),
  check(2 >= (select count(*)
              from lavoro L
              where dipendente = L.dipendente))
)
```

Il vincolo afferma che per ogni dipendente non ci possono essere più di due righe nella tabella lavoro, quindi più di due teatri per cui il dipendente lavoro. Nella query di definizione del vincolo si può usare il nome degli attributi sui quali si sta definenendo il vincolo (dipendente in questo caso).

Lo stesso vincolo può essere espresso mediante una **asserzione**. Solitamente viene scritta una interrogazione SQL che seleziona le righe della base di dati che violano il vincolo. La condizione dell'asserzione viene fomata mettendo tale interrogazione come argomento del predicano `not exists`. Dunque il vincolo di integrità specificato dall'asserzione è verificato che il risultato della interrogazione è vuoto, cioè se non esistono righe che violano il vincolo. Vediamo un esempio:

```
create assertion limitaImpieghi check (not exists(

  select dipendente
  from lavoro
  group by dipendente
  having count(*) > 2

))
```

Supponiamo ora di voler affermare il seguente **vincolo minimo di partecipazione** per un dipendente rispetto alla relazione lavoro: ogni dipendente deve essere assunto presso almeno un teatro. Possiamo imporre questo vincolo sull'attributo `cf` della tabella dipendente:

```
create table dipendente
(
  cf                    char(16) primary key,
  nome                  varchar(20) not null,
  cognome               varchar(20) not null,
  dataDiNascita         date,
  luogoDiNascita        varchar(30),
  capo                  varchar(20),
  foreign key capo references dipendente(cf),
  check (cf in (select dipendente from lavoro))
)
```

Lo stesso vincolo può essere espresso mediante una asserzione come segue:

```
create assertion disoccupato check (not exists (

  select cf
  from dipendente
  where cf not in (select dipendente from lavoro)

))
```

Vediamo un vincolo che coinvolge più attributi della stessa tabella. Ad esempio, supponiamo di voler affermare che i dipendenti nati a Milano devono essere nati prima del 1970. Possiamo riscrivere la definizione della tabella dipendente come segue:

```
create table dipendente
(
  cf                    char(16) primary key,
  nome                  varchar(20) not null,
  cognome               varchar(20) not null,
  dataDiNascita         date,
  luogoDiNascita        varchar(30),
  capo                  varchar(20),
  foreign key capo references dipendente(cf),
  check (luogoDiNascita <> "Milano" or
         dataDiNascita < '1970-01-01')
)
```

Vediamo un esempio di vincolo che coinvolge più tabelle. Supponiamo di voler specificare il seguente vincolo minimo di partecipazione: un teatro deve avere almeno 5 dipendenti. Possiamo scrivere la seguente asserzione:

```
create assertion vincoloDipendentiTeatro check (

  not exists (select nome
              from teatro
              where nome not in (select teatro from lavoro))

  and

  not exists (select count(*)
              from lavoro
              group by teatro
              having count(*) < 5)

)
```

Il vincolo asserisce che non esistono teatri privi di dipendenti e, tra quelli che hanno almeno un dipendente, non esistono teatri con meno di 5 dipendenti. Dunque tutti i teatri hanno almeno 5 dipendenti.

Inoltre, vediamo un vincolo sulla cardinalità di una tabella. La seguente asserzione afferma che ci devono essere almeno 3 teatri nella rete:

```
create assertion vincoloTeatriRete check (

  3 <= (select count(*) from teatro)

)
```

Le **regole attive** (*trigger*) permettono di *gestire* i vincoli di integrità. La differenza rispetto agli strumenti fin ora introdotti per specificare vincoli di integrità (relazionali o generici) è la seguente: un trigger specifica una azione da intraprendere qualora in vincolo non sia soddisfatto, solitamente una azione riparatrice della integrità violata.

Un trigger segue il **paradigma evento-condizione-azione**: se un certo evento si verifica, la relativa condizione viene controllata e, se soddisfatta, l'azione viene intrapresa. Un evento è

solitamento un aggiornamento della base di dati (insert, update, delete). Una condizione è un predicato espresso in SQL. Una azione è una interrogazione SQL (solitamente di aggiornamento della base di dati) oppure una eccezione che annulla gli effetti dell'operazione che ha attivato il trigger riportando la base di dati allo stato precedente a tale operazione (*rollback*). Il trigger può essere attivato prima o dopo l'evento.

Si noti che ci possono essere più trigger associati ad un evento. L'**ordine di esecuzione** dei trigger in tal caso è gestito dal sistema e generalmente tiene conto dell'ordine di creazione dei trigger. Un trigger che come azione aggiorna lo stato della base di dati può a sua volta innescare altri trigger, che a loro volta possono attivare altri trigger, con la possibilità di avere **reazioni a catena infinite**. Inoltre, l'azione di un trigger può violare vincoli di integrità. La violazione di un vincolo di integrità di chiave esterna può causare, come conseguenza delle politiche di gestione di tali vincoli, ulteriori modifiche alla base di dati che al loro volta possono scatenare altri trigger, oppure violare altri vincoli di integrità. Si badi bene che la violazione di un vincolo di integrità non gestito, a qualsiasi livello della catena di attivazione, produce un **annullamento degli effetti** di tutte le operazioni innescate dalla primitiva madre che ha generato la catena di trigger, compresi gli effetti della primitiva madre stessa. I trigger sono dunque strumenti semplici da scrivere in modo indipendente ma difficili da gestire in modo integrato.

Supponiamo di voler specificare un trigger per il vincolo che afferma che lo stipendio di un dipendente non può essere incrementato più del 20%:

```
create trigger LimitaIncrementoStipendio
after update of stipendio on dipendente
for each row
when (New.stipendio > Old.Stipendio * 1.2)
update dipendente
set New.stipendio = Old.Stipendio * 1.2
where cf = New.cf
```

Il trigger LimitaIncrementoStipendio viene attivato dall'evento modifica (update) dello stipendio di un dipendende. Per ogni riga modificata, se il nuovo stipendio è stato incrementato più del 20% rispetto al vecchio (condizione when), allora lo stipendio viene incrementato del massimo possibile senza violare il vincolo di integrità. Si noti che è possibile usare le variabili di tupla `New` e `Old` per riferirsi, rispettivamente, alla tupla dopo e prima la modifica. Per gli eventi di inserimento, solo New è accessibile, per gli eventi di cancellazione, solo Old è accessibile.

Vediamo un altro esempio. Vogliamo modellare la regola che dice che una prenotazione per uno spettacolo può essere effettuata solo se vi sono ancora posti a disposizione in sala. Usiamo i seguenti quattro trigger:

```
create trigger disponibilità-1
after insert on messaInScena
for each row
update messaInScena
set postiDisponibili = (select capienza
                        from spazio
                        where nome = New.spazio)
where (data = New.data and
       ora = New.ora
       spazio = New.spazio)
```

```
create trigger disponibilità-2
after insert on prenotazione
for each row
update messaInScena
set postiDisponibili = postiDisponibili - 1
where (data = New.dataSpettacolo and
       ora = New.oraSpettacolo
       spazio = New.spazioSpettacolo)


create trigger disponibilità-3
after delete on prenotazione
for each row
update messaInScena
set postiDisponibili = postiDisponibili + 1
where (data = Old.dataSpettacolo and
       ora = Old.oraSpettacolo
       spazio = Old.spazioSpettacolo)

create trigger disponibilità-4
before insert on prenotazione
for each row
when (0 = (select postiDisponibili
          from messaInScena
          where (data = New.dataSpettacolo and
                 ora = New.oraSpettacolo
                 spazio = New.spazioSpettacolo)))
rollback("Posti esauriti")
```

Per specificare la regola aziendale sui posti disponibili abbiamo usato i seguenti trigger:

1. disponibilità-1, che imposta il numero di posti disponibili alla capienza dello spazio teatrale quando uno spettacolo viene inserito;
2. disponibilità-2, che decrementa di uno i posti disponibili quando una prenotazione viene inserita;
3. disponibilità-3, che incrementa di uno i posti disponibili quando una prenotazione viene cancellata;
4. disponibilità-4, che controlla, *prima* dell'inserimento della prenotazione nella base di dati, se esistono posti disponibili. Se non ne esistono, esso annulla l'operazione di inserimento e avvisa l'utente che i posti sono esauriti.

Si noti che la soluzione funziona assumendo che disponibilità-4 venga eseguito prima di disponibilità-2 (di solito è così in quanto disponibilità-4 è di tipo before e disponibilità-2 è di tipo after).

I trigger sono anche utili per specificare le regole di calcolo degli **attributi calcolati**. Supponiamo che il prezzo ridotto di uno spettacolo debba essere scontato del 20% rispetto a quello intero. Dunque l'attributo prezzo ridotto è calcolato rispetto al prezzo intero. I trigger per gestire questo vincolo seguono:

```
create trigger CalcolaPrezzoRidottoInsert
after insert on messaInScena
for each row
update messaInScena
set New.prezzoRidotto = New.prezzoIntero * 0.8
where codice = New.codice
```

```
create trigger CalcolaPrezzoRidottoUpdate
after update of prezzoIntero on messaInScena
for each row
update messaInScena
set New.prezzoRidotto = New.prezzoIntero * 0.8
where codice = New.codice
```

Non tutti i vincoli di integrità possono essere descritti a livello di schema in SQL. Solitamente, quando un vincolo non è descrivibile in SQL, esso viene catturato a livello di applicazione implementandolo in qualche linguaggio di programmazione. E' bene che tutti i vincoli esprimibili in SQL vengano definiti a livello di schema in modo da renderli condivisi da tutte le applicazioni invece che replicare il vincolo per ogni applicazione. In tal modo le modifiche di un vincolo sono gestite a livello di schema senza modificare le applicazioni. Si parla in tal caso di **indipendenza dalla conoscenza**, dove per conoscenza si intende l'insieme delle regole codificate nei vincoli che regolano l'integrità della base.

E' possibile aggiungere e rimuovere vincoli di integrità definiti su una tabella mediante il comando `alter`. Per rimuovere un vincolo occorre averlo definito per nome mediante il costrutto `constraint`. Ad esempio:

```
create table dipendente
(
  cf                    char(16) primary key,
  nome                  varchar(20) not null,
  cognome               varchar(20) not null,
  indirizzo             varchar(30),
  constraint chiaveCandidata unique(nome, cognome)
)

alter table dipendente drop constraint chiaveCandidata
alter table dipendente add constraint chiaveCandidata unique(indirizzo)
```

Per rimuovere una asserzione o un trigger occorre usare il comando `drop` seguito dal nome del costrutto.

Concludiamo la parte sulla definizione dei dati in SQL parlando brevemente del **catalogo dei dati**. Il catalogo dei dati è una base relazionale per archiviare lo schema fisico di una base di dati; tale base contiene una descrizione dei dati e non i dati veri e propri. Ad esempio, il catalogo dei dati contiene una tabella per gli attributi delle tabelle di uno schema fisico. Ogni riga della tabella specifica, tra l'altro, il nome dell'attributo, la tabella di appartenenza, il suo valore di default e l'obligatorietà.

Il catalogo dei dati viene solitamente mantenuto dal DBMS e non deve essere creato o modificato dall'utente. Il catalogo dei dati può però essere interrogato dall'utente. Questo offre la possibilità interessante di costruire interrogazioni che accedano sia ai dati che ai metadati. E' bene che i dati e i metadati vengano organizzati nel medesimo modello dei dati (relazionale, ad oggetti, XML). In questo modo è possibile archiviare dati e metadati con le stesse strutture e interrogarli con lo stesso linguaggio. Questa caratteristica prende il nome di **riflessività**.

Viste

Le viste sono **tabelle virtuali** contenenti dati provenienti da altre tabelle della base di dati. Le viste hanno due principali funzioni:

- definire una porzione della base di dati accessibili da un particolare gruppo di utenti. Questa caratteristica contribuisce a realizzare la **privatezza dei dati**, una delle qualità offerte da un DBMS;
- estendere il potere espressivo di SQL permettendo un annidamento delle interrogazioni più sofisticato rispetto a quello già discusso oppure la realizzazione di attributi calcolati che non dipendono totalmente dai dati presenti nella base.

Per quanto riguarda la prima funzione, consideriamo il caso di studio della rete teatrale. Si consideri una vista che mostra, per ogni spettacolo, i dati relativi al numero di paganti, all'affluenza e all'incasso. Tale vista sfrutta una porzione limitata della base e deve essere accessibile solo da un gruppo di utenti interni alla rete e non, ad esempio, da uno spettatore. Possiamo definire tale vista in due passi: prima definiamo una vista StatScena che calcola le statistiche per una messa in scena e poi usiamo tale vista nella definizione della vista statSpettacolo che mostra le statistiche per uno spettacolo.

```
create view statScena(spettacolo, paganti, affluenza, incasso) as
select Sce.spettacolo, count(*), count(*) / Spa.capienza, sum(P.prezzo)
from messaInScena Sce, spazio Spa, prenotazione P
where (Sce.spazio = Spa.nome) and
      (P.dataSpettacolo = Sce.data) and
      (P.oraSpettacolo = Sce.ora) and
      (P.spazioSpettacolo = Sce.spazio)
group by Sce.data, Sce.ora, Sce.spazio

create view statSpettacolo(spettacolo, paganti, affluenza, incasso) as
select Spe.titolo, avg(Sce.paganti), avg(Sce.affluenza), avg(Sce.incasso)
from spettacolo Spe, statScena Sce
where (Spe.titolo = Sce.spettacolo)
group by Spe.titolo
```

Naturalmente, gli attributi della vista devono essere compatibili con gli attributi dell'interrogazione che la definisce. Le viste possono essere usate per la definizione di altre viste o nelle interrogazioni come se fossero tabelle fisiche.

Vediamo ora come si possono utilizzare le viste per formulare interrogazioni che altrimenti non sarebbero definibili in SQL. L'idea è la seguente: dato che una vista è una tabella (virtuale), posso usarla come tale, dunque anche nella clausola `from` di una query SQL. Questo meccanismo permette una nuova forma di annidamento di interrogazioni oltre a quella già vista che consente di nidificare query nei predicati della clausola `where`. Vediamo qualche esempio:

Vogliamo calcolare, per ogni stagione teatrale, la media degli spettatori paganti, la media dell'affluenza e la media degli incassi riferite agli spettacoli della stagione. Possiamo sfruttare la vista statSpettacolo sopra definita nella seguente interrogazione:

```
select nomeStagione, biennioStagione, avg(paganti) as mediaPaganti,
       avg(affluenza) as mediaAffluenza, avg(incasso) as mediaIncasso
from statSpettacolo SS, proposta P
where SS.spettacolo = P.spettacolo
group by nomeStagione, biennioStagione
```

La seguente ordina gli spettacoli di una certa stagione per incasso:

```
select SS.spettacolo, incasso
from statSpettacolo SS, proposta P
where (SS.spettacolo = P.spettacolo) and
      (nomeStagione = 'Contatto') and
      (biennioStagione = '2006/2007')
order by incasso desc
```

Supponiamo ora di lavorare sul seguente schema:

teatro(<u>nome</u>, indirizzo)

dipendente(<u>cf</u>, nome, cognome)

lavoro(<u>teatro</u>, <u>dipendente</u>, ruolo)

Vogliamo identificare il teatro con il numero massimo di dipendenti. Intuitivamente, vogliamo prima contare il numero di dipendenti di ogni teatro e poi prendere il massimo. Si potrebbe pensare di scrivere in questo modo:

```
select max(count(*))
from lavoro
group by teatro
```

In realtà questa query è scorretta perchè in SQL gli operatori di aggregazione non possono essere annidati. Possiamo provare la seguente soluzione:

```
select teatro, count(*)
from lavoro
group by teatro
having count(*) >= all (select count(*)
                        from lavoro
                        group by teatro)
```

Questa soluzione è formalmente corretta però non potrebbe essere riconosciuta da qualche interprete SQL in quanto fa una di un annidamento di query nella clausola `having`. Una soluzione alternativa che fa uso delle viste è la seguente:

```
create view numeroDipendenti(teatro, numero) as
select teatro, count(*)
from lavoro
group by teatro

select teatro, numero
from numeroDipendenti
where numero = (select max(numero)
                from numeroDipendenti)
```

L'esempio che segue seleziona il numero medio di dipendenti dei teatri della rete. Si noti che vengono inclusi nel conteggio anche i teatri con nessun dipendente.

```
create view numeroDipendenti(teatro, numero) as
select nome, count(all dipendente)
from teatro left join lavoro on (teatro.nome = lavoro.teatro)
group by nome

select avg(numero) as numeroMedioDipendenti
from numeroDipendenti
```

Una vista, creata con il comando `create`, è una componente facente parte dello schema della base di dati così come i domini, le tabelle, le asserzioni e i trigger. In particolare, una vista, una volta definita, può essere rimossa mediante il comando `drop`. In realtà, quando una vista viene usata per esprimere una interrogazione, la sua natura è temporanea. Una **vista temporanea** è una vista definita nel contesto di una interrogazione. Essa è accessibile dall'interrogazione nella quale appare ma non appartiene allo schema della base di dati. In particolare, non può essere usata da altre interrogazioni oltre a quella a cui appartiene e non può essere rimossa dallo schema. E' possibile definire viste temporanee con la clausola `with`. In particolare, l'ultima interrogazione può essere riscritta come segue:

```
with numeroDipendenti(teatro, numero) as
select nome, count(all dipendente)
from teatro left join lavoro on (teatro.nome = lavoro.teatro)
group by nome

select avg(numero) as numeroMedioDipendenti
from numeroDipendenti
```

Un altro modo in cui le viste aumentano l'espressività di SQL è mediante le viste ricorsive, introdotte in SQL-3. Una **vista ricorsiva** è una vista V definita usando V stessa oppure usando un'altra vista V' che usa, direttamente o indirettamente, la vista V. Vediamo un esempio. Prendiamo lo schema di relazione che segue:

dipendente(<u>cf</u>, nome, cognome, capo)

dove capo è una chiave esterna di dipendente. Supponiamo che un dipendente senza capi abbia come capo sè stesso. Vogliamo recuperare tutti i superiori, diretti o indiretti, di un certo dipendente dato. Si noti che non è possibile farlo in SQL senza la ricorsione, perchè avremmo bisogno di un numero non prevedibile a priori di join della tabella dipendente con sè stessa. Vediamo una soluzione che prima definisce una vista ricorsiva temporanea e poi la usa per recuperare i capi del dipendente con codice 'ELSDLL72'.

```
with recursive superiore(cf, nome, cognome, capo) as
(
(select *
 from dipendente)
   union
(select dipendente.cf, dipendente.nome, dipendente.cognome, superiore.capo
 from dipendente, superiore
 where dipendente.capo = superiore.cf)
)

select capo
```

```
from superiore
where cf = 'ELSDLL72'
```

La vista ricorsiva superiore contiene i dipendenti (identificati da cf, nome e cognome) e i loro capi diretti o indiretti. Si noti che per ogni dipendente ci possono essere più capi, dunque cf non identifica le tuple della vista superiore. In generale non è definito il concetto di chiave per le viste. La definizione ricorsiva di superiore specifica un caso base (i capi diretti) e un caso ricorsivo (i capi indiretti) in cui la vista superirore viene riusata. Infine l'interrogazione di partenza si risolve semplicemente selezionando i capi del dipendente con codice 'ELSDLL72'.

E' istruttivo analizzare il metodo di calcolo di una vista ricorsiva. L'algoritmo è simile a quello che risolve il **problema di raggiungibilità** su un grafo: a partire da un insieme iniziale di nodi, trovare l'insieme dei nodi raggiungibili da questi attraverso un cammino arbitrario. Come primo passo si applica da definizione base e si ottiene un primo risultato X (i dipendenti e i loro capi diretti nel nostro esempio). A questo punto si applica la definizione ricorsiva in cui la tabella superiore contiene tutte e sole le tuple in X. In questo modo si ottiene un secondo risultato Y (i dipendenti e i capi dei loro capi). Se Y contiene qualche tupla non presente in X, allora si applica la definizione ricorsiva in cui la tabella superiore contiene tutte e sole le tuple in Y meno X (le tuple nuove), ottenendo un nuovo risultato Z (i dipendenti e i capi dei capi dei loro capi). Se Z contiene qualche tupla non presente in X unito Y, allora si procede similmente finchè si ottiene un risultato che non aggiunge nessuna riga rispetto a ciò che è stato calcolato fino a quel momento. Il risultato della valutazione ricorsiva è l'unione insiemistica di tutti i risultati parziali ottenuti (X unito Y unito Z e così via).

Ad esempio, se A ha come capo B, B ha come capo C, C ha come capo D, e D non ha capi (cioè ha come capo sè stesso), allora la prima iterazione produce l'insieme di coppie (dipendente, capo) pari a X = {(A,B), (B,C), (C,D), (D,D)}. La seconda iterazione produce l'insieme Y = {(A,C), (B,D), (C,D)}. Solo le prime due coppie sono nuove. La terza iterazione produce Z = {(A,D)}. La successiva iterazione non produce nulla e dunque il risultato finale è l'unione di X, Y e Z, cioè {(A,B), (B,C), (C,D), (D,D), (A,C), (B,D), (A,D)}

Infine, le viste sono utili per modellare attributi, come l'età di una persona, il cui valore non dipende completamente dai dati della base. In tal caso, il valore di età dipende anche dalla data corrente. Vediamo un esempio di vista che, per ogni teatro, seleziona i rispettivi dipendenti calcolando, tra l'altro, la loro età e il loro stipendio corrente:

```
create view
dipendenti(teatro, cognome, nome, cf, età, telefonoFisso, telefonoMobile, email,
           stipendio, dataDiAssunzione, ruolo, cda) as
  select L.teatro, D.cognome, D.nome, D.cf,
    case
      when ((month(current_date) > month(D.dataDiNascita)) or
            (month(current_date) = month(D.dataDiNascita) and
            day(current_date) >= day(D.dataDiNascita)))
      then year(current_date) - year(D.dataDiNascita)
      else year(current_date) - year(D.dataDiNascita) - 1
    end,
    D.telefonoFisso, D.telefonoMobile, D.email, S.stipendio,
    L.dataDiAssunzione, L.ruolo, L.cda
  from lavoro L, dipendente D, stipendio S
  where (L.dipendente = D.cf) and
        (S.dipendente = D.cf) and
```

```
        (S.inizio = (select max(S2.inizio)
                     from stipendio S2
                     where S2.dipendente = D.cf))
  order by L.teatro, D.cognome, D.nome
```

Vi sono due approcci complementari per implementare le viste. Il primo consiste nel **riscrivere le interrogazioni** che usano viste in termini di interrogazioni che usano tabelle di base. La seconda soluzione consiste nel **materializzare le viste**, cioè nel calcolare le relative tabelle virtuali, e usare queste tabelle per risolvere le interrogazioni che le usano. Il primo approccio ha il vantaggio di non dover salvare i dati delle tabelle virtuali associate alle viste. D'altronde tale approccio risulta inefficiente quando vi sono molte query che si riferiscono alla medesima vista. Il secondo approccio risolve questi problemi di efficienza ma ha lo svantaggio di dover mantenere la materializzazione della vista e aggiornala quando le tabelle di base su cui la vista è definita vengono modificate.

Perchè SQL?

E' lecito chiedersi perchè SQL offra esattamente le funzionalità descritte e non altre. Una risposta a questa domanda viene fornita dalla teoria delle basi di dati relazionali.

Come abbiamo già detto, la forza del modello relazionale sta nella sua dualità pratico-teorica: una tabella è al tempo stesso un oggetto concreto semplice da descrivere e capire ma anche la rappresentazione di una relazione matematica. La **teoria delle basi di dati relazionali** è una teoria matematica che studia le proprietà (come espressività e complessità) del modello relazionale e dei relativi linguaggi di interrogazione.

Vediamo quindi di reinterpretare SQL all'interno di questa teoria. La discussione sarà necessariamente informale e intuitiva. Per una trattazione formale si vedano le Sezioni 2.1 e 2.2 dell'articolo *Elements of Relational Database Theory* di Paris C. Kanellakis.

Nelle interrogazioni SQL per il recupero di informazione si nascondono alcune **operazioni fondamentali**:

Proiezione
: E' l'operazione di selezione di alcune **colonne** a partire da una tabella. Corrisponde alla clausola `select` di SQL;

Selezione
: E' l'operazione di selezione delle **righe** di una tabella che soddisfano un certo predicato. Corrisponde alla clausola `where` di SQL;

Join
: E' l'operazione di congiunzione di due tabelle rispetto ad un predicato di join che confronta coppie formate da attributi delle due tabelle. In altri termini, l'operazione di join realizza un prodotto cartesiano delle due tabelle e seleziona le righe che soddisfano il predicato di join. Corrisponde al costrutto `join` da usare nella clausola `from` di SQL ed è esprimibile anche mediante la clausola `where`;

Unione
E' l'operazione di unione di due tabelle. Corrisponde al costrutto `union` di SQL;

Differenza
E' l'operazione di differenza di due tabelle. Corrisponde al costrutto `except` di SQL;

Rinomina
E' l'operazione di rinomina di attributi di una tabella. Corrisponde al costrutto `as` di SQL.

Queste operazioni costituiscono il nucleo centrale del linguaggio e sono la base per la definizione di altri costrutti SQL (ad esempio l'operazione di intersezione può essere definita in termini di unione e differenza). Queste operazioni formano l'**algebra relazionale**. Data una base di dati, una **espressione** dell'algebra relazionale è formata a partire dalle relazioni della base di dati e dalle operazioni fondamentali descritte. Ogni espressione dell'algebra relazionale calcola (cioè ha come valore) una relazione risultato.

L'algebra relazionale è un **linguaggio procedurale**: un programma (espressione) dell'algebra specifica *come*calcolare il risultato, e non *quale* risultato si vuole raggiungere, come accade per i linguaggi dichiarativi (ad esempio SQL). Dunque l'algebra relazionale è una controparte procedurale (o operazionale) del linguaggio SQL. Possiamo quindi rispondere alla domanda iniziale, cioè perchè SQL ha proprio quelle funzionalità e non altre, concentrandoci sulla sua controparte procedurale. In particolare, la scelta delle operazioni fondamentali dell'algebra relazionale non è dettata dal caso ma trova giustificazione nelle seguenti tre proprietà dell'algebra:

Espressività
Una proprietà interessante dell'algebra relazionale è la seguente:

L'algebra relaziona ha l'espressività del calcolo dei predicati al prim'ordine.

Questo risultato fissa l'espressività dell'algebra relazionale: le operazioni fondamentali su cui l'algebra è definita e che sono alla base di SQL permettono di specificare tutte le formule del calcolo dei predicati al prim'ordine. Inoltre questo risultato fissa anche i limiti dell'algebra relazionale (e di SQL). Ad esempio, è ben noto che l'operazione di chiusura transitiva di una relazione non è definibile al prim'ordine. Tale operazione dunque non sarà definibile nè in algebra relazionale nè in SQL.

Complessità
Il problema della valutazione, cioè del calcolo del risultato, di una espressione dell'algebra relazionale rispetto ad una base di dati è completo nella classe di complessità PSPACE, cioè nella classe dei problemi risolubili in *spazio* polinomiale. Questa sembra essere una brutta notizia, in quanto significa che non esistono con ogni probabilità algoritmi polinomiali per tale problema. In realtà non lo è fino in fondo. Infatti, la complessità della valutazione di una espressione è esponenziale nella dimensione della espressione e **polinomiale nella dimensione della base di dati**. La dimensione della base di dati è di gran lunga superiore a quella dell'interrogazione, tanto che spesso si considera la complessità dell'interrogazione come una costante e non più come un parametro di complessità. Fissando dunque la dimensione dell'interrogazione come costante, e quindi

riferendoci solo alla complessità dei dati, possiamo affermare che:

Il problema della valutazione dell'algebra relazionale è in PTIME, la classe dei problemi risolubili in tempo polinomiale. In particolare, tale problema sta nella classe LOGSPACE, una sottoclasse di PTIME, che corrisponde ai problemi risolubili in spazio logaritmico.

La classe **LOGSPACE** è una classe molto bassa nella gerarchia delle classi di complessità computazionale. In particolare i problemi contenuti in questa classe si prestano ad essere risolti in modo parallelo.

Ottimizzazione

Questa proprietà è connessa al risultato di espressività già esposto. In particolare abbiamo che:

Ogni espressione di SQL può essere trasformata, con complessità polinomiale rispetto alla sua dimensione, in una equivalente espressione del dell'algebra relazionale e viceversa.

Questa proprietà caratterizza l'algebra relazionale come la **controparte procedurale** di SQL. Si noti che la trasformazione da SQL a algebra è efficiente.

Questo risultato apre le porte ad una serie di **ottimizzazioni** che è possibile intraprendere durante la valutazione di una interrogazione SQL. Una interrogazione SQL, per essere valutata, viene prima trasformata una espressione in algebra relazionale. Tale espressione viene poi riscritta in qualche versione equivalente ma ottimizzata rispetto a qualche misura di complessità. Dato che l'operazione più costosa dell'algebra relazionale è il join e che la complessità del join dipende dalla cardinalità delle tabelle argomento, l'obiettivo della riscrittura delle espressioni relazionali è quello di minimizzare il numero di operazioni di join e di eseguire tali operazioni su tabelle di piccole dimensioni. Quest'ultimo risultato si ottiene spingendo la selezione all'interno dei join in modo da filtrare le tabelle prima di sottoporle all'operazione di congiunzione.

In sostanza, le motivazioni teoriche che stanno dietro al successo di SQL sono la sua espressività e la possibilità di risolvere efficientemente le sue interrogazioni.

Strutture di memorizzazione e di accesso ai dati

Un compito importante della progettazione fisica è quello di definire le **strutture di memorizzazione** delle tabelle (si parla di organizzazione primaria dei dati) e le **strutture ausiliarie di accesso** ai dati (organizzazione secondaria dei dati).

Una struttura di memorizzazione per una tabella è una struttura di dati che consente di salvare in memoria i dati della tabella. La memoria degli attuali calcolatori di divide in memoria primaria e memoria secondaria:

- la **memoria primaria** permette un uso diretto e un accesso veloce da parte del processore, è costosa e volatile (cioè funziona con la corrente elettrica e dunque perde il suo contenuto se la corrente viene a mancare). Esempi di memoria primaria sono la memoria cache (o RAM statica) e la memoria principale (o memoria centrale, o RAM dinamica);
- la **memoria secondaria** *non* permette un uso diretto da parte del processore, ma il suo contenuto, per poter essere elaborato, deve prima essere caricato in memoria primaria. L'accesso ai dati in memoria secondaria è dunque lento. La memoria secondaria è poco costosa e permanente (cioè il suo contenuto non dipende dalla presenza di corrente elettrica). Esempi sono i dischi magnetici, i dischi ottici e i nastri.

Una base di dati viene tipicamente memorizzata in memoria secondaria, e in particolare su dischi magnetici. I motivi sono i seguenti:

- le basi di dati sono solitamente **molto grandi** e eccedono la memoria primaria a disposizione;
- le basi di dati contengono informazioni che devono essere memorizzate in modo **permanente** per lunghi periodi di tempo;
- il **costo** della memoria secondaria è un ordine di grandezza inferiore rispetto al costo della memoria principale.

Un **disco magnetico** memorizza una singola unità di informazione (**bit**) magnetizzando opportunamente un'area del disco. In tal modo è possibile distinguere il valore 0 dal valore 1. I bit sono organizzati in **byte**, normalmente formati da 8 bit. Il disco è suddiviso in **tracce** concentriche le quali sono suddivise in **blocchi** di dimensione fissa che contengono i byte di informazione (di solito da 512 a 8192 byte). Il blocco è l'**unità elementare di trasferimento** di dati tra la memoria secondaria e principale. Ciò significa che ogni trasferimento di dati coinvolge uno o più blocchi, ma non una frazione. Per leggere un blocco, una testina deve posizionarsi sulla traccia del blocco, aspettare che il disco ruoti sul blocco da leggere, e infine trasferire il contenuto del blocco in memoria centrale. Un blocco è quindi identificato da un **indirizzo fisico** formato da un numero di traccia e un numero di blocco all'interno della traccia.

La lettura di un blocco coinvolge operazioni meccaniche (lo spostamento della testina, la rotazione del disco) e dunque è una operazione molto lenta rispetto ad una lettura in memoria primaria. Per questo le letture su disco rappresentano il collo di bottiglia delle applicazioni per basi di dati. Scopo della progettazione fisica è trovare le opportune strutture di dati per memorizzare e accedere ai dati che **minimizzano il numero di accessi al disco**.

Una tabella viene memorizzata per righe su un insieme di blocchi. Nel contesto della progettazione fisica delle strutture di memorizzazione, un attributo viene chiamato **campo**, una riga viene chiamata **record** e una tabella viene chiamata **file**. Un file è dunque un insieme di record formati da campi. I record possono avere **dimensione fissa o variabile** all'interno del file. Un record a

dimensione variabile contiene campi a dimensione variabile (ad esempio, di tipo varchar), oppure campi opzionali, cioè che possono assumere il valore nullo. In questi casi viene usato un separatore per indicare la terminazione del campo.

L'**allocazione dei record** nei blocchi può essere **unspanned**, in cui ogni record è interamente contenuto in un blocco, oppure **spanned**, in cui è possibile allocare un record a cavallo di più blocchi. L'allocazione spanned può essere adottata per risparmiare spazio di memoria (evitando che si formino aree di blocco inutilizzate qualora la dimensione del blocco non è un multiplo della dimensione dei record) e *deve* essere adottata quando la dimensione dei record è maggiore della dimensione dei blocchi.

L'**allocazione dei blocchi** di un file su disco può essere:

- **contigua**: i blocchi di un file sono allocati consecutivamente;
- **concatenata**: i blocchi non sono necessariamente consecutivi e sono collegati tra loro mediante puntatori;
- **segmentata**: insiemi di blocchi vengono allocati consecutivamente in segmenti. I segmenti sono collegati da puntatori;
- **indicizzata**: uno o più blocchi indice contengono i puntatori ai blocchi di dati del file.

Ogni file è dotato di un **descrittore** che contiene l'informazione necessaria alle applicazioni per accedere ai record del file. Questa informazione include gli indirizzi dei blocchi che contengono i record e il formato dei record.

Esistono tre metodi principali per **organizzare i record nei file**:

- **file non ordinato**: i record sono inseriti nel file nell'ordine in cui sono inseriti nella base di dati, quindi, in generale, senza un ordine preciso. Per cercare un record è necessaria una scansione sequenziale del file con costo lineare;
- **file ordinato**: i record sono inseriti nel file secondo l'ordine dei valori di un determinato campo, detto campo di ordinamento. Per cercare un record di cui è noto il valore del campo di ordinamento è possibile usare una ricerca binaria con costo logaritmico;
- **file ad accesso calcolato (*hash*)**: i record sono inseriti nel file nell'ordine determinato applicando una funzione di hash ai valori di un campo, detto campo di hash. Per cercare un record di cui è noto il valore del campo di hash è sufficiente calcolare la funzione di hash su quel valore. Mediamente, questa operazione ha costo costante. Occorre risolvere con qualche tecnica il problema delle collisioni (due record con lo stesso valore di hash). Inoltre, di solito la dimensione del file è fissa.

Una **struttura ausiliaria di accesso**, più comunemente detta **indice**, è una struttura di dati che serve a velocizzare l'accesso ai record in risposta a certe interrogazioni. Gli indici sono strutture ausiliarie e opzionali rispetto alle strutture di memorizzazione dei file e non vanno ad influire sull'organizzazione primaria dei record nei file. Un indice è definito su uno o più attributi di una tabella e, in generale, le interrogazioni che coinvolgono gli attributi indice beneficiano di una maggiore efficienza. Infatti, al fine di trovare un record di cui è noto il valore del campo indice, invece di scandire il file dei dati, è possibile cercare nel file indice, di dimensione molto inferiore

rispetto al file di dati, l'indirizzo del blocco che contiene il record. Il relativo blocco è poi caricato in memoria centrale e il record viene cercato nel blocco in memoria centrale.

Il principio su cui si basano gli indici è simile a quello dell'indice analitico di un libro di testo, cioè una lista ordinata di termini associati alle pagine in cui tali termini compaiono nel testo. Per trovare nel testo un certo argomento, è possibile scandire sequenzialmente il libro, oppure, più efficientemente, trovare il termine nell'indice, recuperare il numero di pagina (indirizzo) e infine accedere direttamente alla pagina (blocco) nel testo (file). L'accesso tramite l'indice è più veloce per almeno due ragioni: l'indice contiene meno informazioni del testo. Inoltre i termini nell'indice sono ordinati, dunque è possibile usando una ricerca dicotomica (binaria). Si ricordi che il fattore critico dell'accesso alle basi di dati è il numero di accessi a blocchi del disco. Un indice è sostanzialmente più leggero del relativo file e dunque può essere memorizzato in meno blocchi di disco. L'accesso ai record tramite indice riduce dunque il numero letture su disco.

Si noti che in questo caso tutti i termini del testo compaiono nell'indice. Questo tipo di indici viene detto **denso**. Inoltre, i termini nel testo non sono ordinati, ad esempio il termine 'relazione' può essere contenuto in una pagina precedente al termine 'entità'. Dunque l'indice è definito su un campo non di ordinamento del file. Indici su campi non di ordinamento vengono detti **indici secondari**.

Se la struttura primaria di memorizzazione è un file ordinato rispetto ad un campo, un indice sul campo di ordinamento può migliorare comunque l'efficienza dell'accesso ai dati. Prendiamo l'esempio di un dizionario in cui i termini sono inseriti in ordine. Supponiamo di creare un indice analitico dei termini. Nell'indice inseriamo, in modo ordinato, il primo termine di ogni pagina e la relativa pagina. Quindi i termini inseriti nell'indice sono un sottoinsieme dei termini dei dizionario di cardinalità pari al numero delle pagine del dizionario. La ricerca di un termine può dunque avvenire in due modi: mediante una ricerca dicotomica nel dizionario, oppure cercando, mediante una ricerca dicotomica nell'indice, la pagina del termine e successivamente cercando il termine nella relativa pagina del dizionario (anche in quest'ultimo caso si può usare una ricerca dicotomica). Il secondo metodo è più efficiente in quanto l'indice contiene meno termini e, per ogni termine, l'indice contiene meno informazione. Anche in questo caso l'indice occupa meno blocchi di disco e dunque l'accesso tramite indice riduce il numero di letture su disco. Indici su campi di ordinamento vengono detti **indici primari**. Un indice primario è anche detto **sparso** in quanto contiene un parte dei valori del campo presenti nel file.

Gli indici visti fin ora sono detti indici di singolo livello. Gli **indici multi-livello** nascono dalla seguente osservazione. Un indice non è altro che un file ordinato di dati. In particolare, deve essere memorizzato su un insieme di blocchi del disco. Se l'indice è grande, nulla vieta di creare un indice (primario) definito sull'indice stesso. L'indice ottenuto è detto di secondo livello ed è lui stesso un file ordinato di dati. Il processo può essere iterato finchè l'ultimo livello dell'indice è contenuto interamente in un blocco. La struttura risultante assume la forma di un albero in cui la radice è l'ultimo livello e le foglie stanno al primo livello dell'indice. Gli indici multi-livello possono essere definiti a partire da indici primari o secondari e vengono implementati da una struttura di dati nota come **B-tree**.

Riprendendo l'esempio del dizionario, supponiamo che l'indice analitico occupi più pagine. E' dunque possibile creare un indice dell'indice. L'indice di secondo livello contiene il primo termine

per ogni pagina dell'indice di primo livello e la rispettiva pagina nell'indice di primo livello. Supponiamo che l'indice di secondo livello sia contenuto in un'unica pagina, e dunque che il processo di indicizzazione per livelli possa terminare. Per accedere ad un termine del dizionario, procedo come segue: cerco il termine nell'indice di secondo livello ottenendo la pagina relativa all'indice di primo livello, quindi cerco il termine in questa pagina, ottenendo la pagina relativa al dizionario, infine cerco il termine nella pagina del dizionario. Questo processo sembra complicato per un umano ma in realtà per una macchina risulta più efficiente.

E' infine possibile creare indici basati sulla struttura di dati **hash**. In generale, dato un file, è possibile definire un solo indice primario (di singolo o multi-livello) sul campo di ordinamento (se esiste) e altri indici secondari (di singolo o multi-livello) su campi diversi da quello di ordinamento.

Le interrogazioni più frequenti in una base di dati sono quelle che coinvolgono operazioni di **selezione** di tuple che soddisfano un certo predicato e operazioni di **join** tra due tabelle. Ha quindi senso definire degli indici per velocizzare queste operazioni. Ad esempio, consideriamo un tipico esempio di selezione:

```
select nome, cognome
from dipendente
where cf = 'ABCDEF89'
```

Supponiamo che sulla tabella dipendente non vi siano indici. La ricerca del dipendente con codice dato deve necessariamente procedere in modo sequenziale leggendo, nel caso peggiore, tutti i blocchi occupati dalla tabella. Supponiamo ora che vi sia un indice sul campo cf di dipendente. In tal caso, è possibile trovare velocemente il record associato al dipendente con codice dato accedendo all'indice. Nel caso peggiore, solo i blocchi occupati dall'indice più quello che contiene il record cercato vengono letti.

Vediamo un esempio di join:

```
select lavoro.teatro, dipendente.nome, dipendente.cognome,
from lavoro join dipendente on (lavoro.dipendente = dipendente.codice)
```

Supponiamo che non vi siano indici sulle tabelle lavoro e dipendente. In tal caso, per ogni riga della tabella lavoro devo scandire linearmente la tabella dipendente cercando la riga corrispondente. Occorre dunque leggere tutti i blocchi di disco di entrambe le tabelle. Inoltre, il costo computazionale in memoria centrale di questa strategia è pari al prodotto delle cardinalità delle due tabelle coinvolte.

Supponiamo ora che vi sia un indice sul campo codice, chiave primaria della tabella dipendente. Posso scandire la tabella lavoro e per ogni riga accedere alla tabella dipendente sfruttando l'indice definito sul capo codice. Vi è un risparmio sia in termini di letture su disco (l'indice è più piccolo della tabella e accedo alla tabella dipendente solo per i record che soddisfano la condizione di join) che in termini di costo computazionale in memoria centrale (ad esempio, nel caso di indice di singolo livello, il costo è pari alla cardinalità della tabella lavoro per il logaritmo della cardinalità della tabella dipendente).

Se inoltre vi fosse un indice anche sul campo dipendente, chiave esterna della tabella lavoro, potrei scorrere tale indice invece di scandire la tabella lavoro e accedere ai blocchi della tabella lavoro solo per i record che soddisfano la condizione di join. Avrei dunque un ulteriore riduzione degli accessi al disco.

Vediamo un esempio di ordinamento:

```
select nome, cogmome, dataDiNascita
from dipendente
order by dataDiNascita
```

Se non vi sono indici, le righe della tabella dipendente devono essere ordinate per data di nascita. La presenza di un indice sul campo dataDiNascita implica che i record nell'indice sono ordinati secondo questo campo e dunque non è necessario ordinare le righe della tabella dipendente.

Infine un esempio selezione di tuple in un intervallo:

```
select nome, cogmome, dataDiNascita
from dipendente
where dataDiNascita > '1970-01-01'
```

Se vi è un indice sul campo dataDiNascita, è possibile trovare il primo record che corrisponde alla data '1970-01-01' e poi fornire in risultato tutti i record da questo in poi secondo l'ordine dell'indice.

Si badi bene che gli indici, in quanto strutture ausiliarie, devono essere mantenuti aggiornati qualora la tabella venga aggiornata in modifica, inserimento e cancellazione. In generale gli indici usano strutture di dati, come i B-tree o le tabelle hash, che permettono di implementare in modo efficiente sia la ricerca che le operazioni di aggiornamento dei record. Dunque gli indici favoriscono le interrogazioni di ricerca ma sfavoriscono quelle di aggiornamento (che non coinvolgono ricerche).

Concludiamo con alcuni consigli generali sull'uso degli indici:

- definire un indice sulla chiave primaria di ogni tabella e sulle chiavi esterne. Infatti, spesso le operazioni di selezione e di join coinvolgono questi attributi. Solitamente gli indici sulla chiave primaria sono creati in automatico dal DBMS;
- definire un indice su altri attributi usati di frequenza in interrogazioni che prevedono le operazioni di selezione, join o ordinamento. Quando le condizioni usate in queste operazioni sono solo di uguaglianza (cioè confronti con l'operatore =) e non è previsto l'ordinamento, definire un indice di tipo hash;
- se la tabella è soggetta a frequenti aggiornamenti, soprattutto rispetto alle altre interrogazioni che riceve, ridurre al minimo l'uso degli indici sulla tabella, in quanto gli indici devono essere aggiornati dopo ogni inserimento, cancellazione e modifica degli attributi indice.

Lo standard SQL non prevede comandi per la definizione di strutture di memorizzazione e di indici. Il motivo è che queste strutture sono strettamente legate all'implementazione del sistema e dunque difficilmente uniformabili. Naturalmente, ogni DBMS offre una varietà di strutture di

memorizzazione e di indicizzazione e inoltre permette di usare comandi SQL (non standard) per definire entrambi. Generalmente, la struttura di memorizzazione di una tabella viene associata in fase di creazione della tabella. Un indice viene creato col comando `create index` seguito dal nome dell'indice, nome della tabella e lista degli attributi da indicizzare e rimosso col comando `drop index` seguito dal nome dell'indice.

MySQL

MySQL è un DBMS relazionale creato da Michael "Monty" Widenius nel 1995 e attualmente posseduto e sviluppato dall'azienda svedese **MySQL AB** (AB sta per *aktiebolag*, cioè società per azioni).

MySQL viene distribuito con una **doppia licenza**: una licenza commerciale (a pagamento), che consente di includere le funzionalità di MySQL nello sviluppo di un proprio software e vendere tale software con licenza commerciale. Una licenza libera (GNU General Public License, GPL), che consente si scaricare liberamente i sorgenti e gli eseguibili, modificare i sorgenti e ridistribuirli a patto che il prodotto creato sia distribuito con la licenza GPL. In ogni caso, MySQL AB mantiene i diritti sui sorgenti e sul marchio MySQL, che quindi non può essere usato come nome nel software distribuito.

MySQL funziona su diverse piattaforme, in particolare, Linux, Mac OS X e MS Windows. E' possibile scaricare il DBMS dal sito di MySQL AB. Il sito di MySQL AB contiene una estesa documentazione sul DBMS, in particolare spiega come installarlo.

Il pacchetto MySQL contiene diversi programmi. I principali sono i seguenti:

- `mysqld`. Questo è il server MySQL che accoglie le richieste (interrogazioni) dei client. Lo script `mysqld_safe` è il modo più comune di lanciare il server, in quanto lo fa ripartire qualora esso termini in modo anomalo;
- `mysql`. Questo è il client MySQL che permette connettersi al server e eseguire le interrogazioni SQL;
- `mysqladmin`. Serve per amministrare il server MySQL, ad esempio per terminarlo;
- `mysqlshow`. Serve per ottenere informazioni sulla base di dati, sulle tabelle e sulle colonne;
- `mysqldump`. Serve per esportare schemi e tabelle su file di testo. Il client `mysql` oppure il programma `mysqlimport` sono in grado di caricare tabelle leggendole dai file esportati;
- `mysqlhotcopy`. Serve per esportare una base di dati o singole tabelle nel formato interno usato da MySQL. Per recuperare la base di dati basta copiare i file esportati nella cartella dei dati di MySQL.

Tutti i comandi che seguono sono stati testati su MySQL versione 5.0.37. Di seguito, assumeremo che la cartella bin della directory di installazione di MySQL sia nella variabile d'ambiente PATH

del sistema operativo. Per eseguire dei comandi SQL occorre che il server sia acceso. Per avviare il server, spostarsi nella cartella di installazione di MySQL e usare il comando:

```
mysqld_safe
```

Per spegnere il server, usare il comando:

```
mysqladmin -u user -p shutdown
```

dove `user` è un utente con il privilegio di `shutdown` (verrà richiesta la parola chiave per l'utente).

Se il server è acceso, è possibile connettere un client al server con il comando:

```
mysql -h host -u user -p
```

dove `host` è la macchina su cui gira il server di MySQL e `user` è l'utente che vuole connettersi. La parola chiave può essere specificata dopo l'opzione `-p` (senza spazi di separazione) oppure, più saggiamente, è possibile evitare di scriverla in chiaro. In tal caso viene richiesto l'inserimento della parola chiave in modo criptato. Se il server gira sulla macchina su cui si sta lavorando (identificata con *localhost*), l'opzione `-h host` può essere omessa. Esiste un utente root, la cui parola chiave è inizialmente nulla, e un utente anonimo, con nome utente e parola chiave entrambi nulli. Ne deriva che la prima volta è possibile avviare il client con i comandi `mysql -u root`(accedendo come root) oppure `mysql` (accedendo come utente anonimo).

Una volta avviato, il client è pronto a ricevere dall'utente i comandi da rivolgere al server. I comandi possono essere inseriti in tre modi:

1. scrivendo il comando direttamente dal monitor di MySQL, cioè dalla linea di comando che si ottiene dopo aver avviato il client. Un comando SQL può essere spezzato su più righe e termina sempre con il simbolo *;* oppure `\g` oppure `\G` (quest'ultimo mostra il risultato verticalmente). Un comando può essere annullato con `\c`;
2. scrivendo il comando in un file e, dal monitor di MySQL, eseguendo il file con il comando source. Ad esempio:
3. `source data/univ/sql/insert.sql;`
4. dalla linea di comando della shell del sistema operativo, in questo modo:

   ```
   mysql -e "query" database
   ```

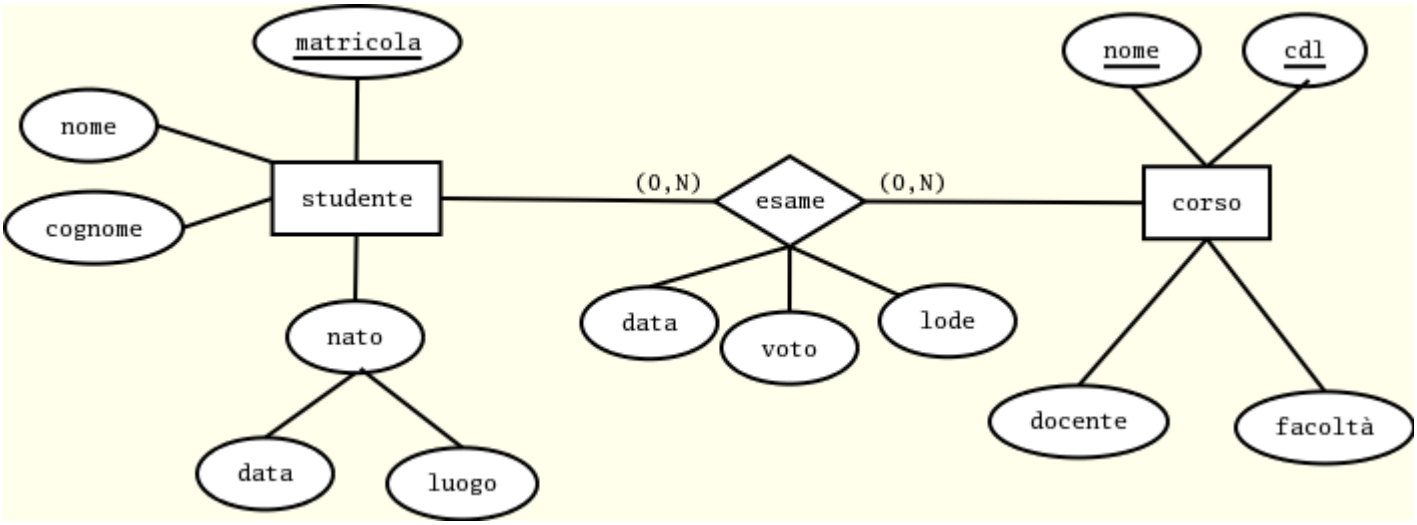
   dove `query` è l'interrogazione SQL e `database` è il nome della base di dati da interrogare. E' anche possibile specificare host, user e password, se necessari.

5. dalla linea di comando della shell del sistema operativo, in quest'altro modo:

   ```
   mysql database < queryFile
   ```

dove `queryFile` è il nome di un file che contiene l'interrogazione SQL e `database` è il nome della base di dati da interrogare. E' anche possibile specificare host, user e password, se necessari.

Di seguito, simuliamo la creazione, il popolamento e l'interrogazione di una semplice base di dati denominata univ che registra gli esami superati dagli studenti di una facoltà universitaria:



studente(<u>matricola</u>, nome, cognome, data, luogo)

corso(<u>nome</u>, <u>cdl</u>, facolta, docente)

esame(<u>studente</u>, <u>corso</u>, <u>cdl</u>, data, voto, lode)

esame(studente) --> **studente**(matricola)

esame(corso, cdl) --> **corso**(nome, cdl)

Useremo comandi SQL di MySQL appartenenti alle seguenti categorie ordinate logicamente:

Amministrazione degli utenti

Questi comandi permettono di creare e rimuovere utenti, assegnare loro privilegi e parola chiave.

Definizione dello schema

Questi comandi permettono di creare, modificare, e rimuovere le varie componenti dello schema della base di dati.

Informazioni sullo schema

Questi comandi permettono di stampare informazioni sullo schema della base di dati.

Aggiornamento e interrogazione

Questi comandi permettono di aggiornare il contenuto della base di dati e di interrogarla.

Salvataggio e ripristino

Questi comandi permettono di salvare una base di dati e di ripristinarla a partire dai file salvati.

Il primo passo da fare è **creare un nuovo utente** e assegnargli i privilegi. Per entrambe le cose esiste il comando grant. Ad esempio, per creare un utente alan con parola chiave nala e assegnargli tutti i privilegi su tutti i database, inclusa la possibilità di usare il comando grant, su può usare il seguente comando:

```
grant all privileges on *.*
to 'alan'@'localhost' identified by 'nala'
with grant option;
```

E' possibile essere più selettivi dando solo alcuni privilegi su alcune tabelle (o colonne) di alcune basi di dati. Ad esempio, il seguente crea (se già non esiste) l'utente alan e gli assegna il solo privilegio di effettuare select sulla tabella esame del database univ:

```
grant select on univ.esame
to 'alan'@'localhost' identified by 'nala'
```

E' possibile usare il privilegio usage per creare un utente senza privilegi. Le informazioni su utenti e i loro privilegi vengono archiviate nelle tabelle dei privilegi (grant tables) della base di dati mysql, creata in fase di installazione del DBMS. Altri comandi utili per amministrare gli utenti sono:

- revoke: revoca i privilegi.
- drop user: per eliminare un utente.
- set password: per associare una parola chiave ad un utente.
- show grants: per mostrare i comandi grant eseguiti per un utente. Utile per duplicare i privilegi di un utente per un altro utente.
- show privileges: per mostrare tutti i privilegi che possono essere assegnati in generale.

E' giunta l'ora di **creare il database** univ secondo lo schema relazionale sopra descritto. Per far ciò possiamo usare il comando create database:

```
create database univ;
```

Occorre essere un utente con il privilegio create sul database creato. E' possibile specificare il character set (l'insieme di caratteri da usare) e il relativo collation (l'ordinamento dei caratteri nell'insieme scelto) come segue:

```
create database univ
character set utf8
collate utf8_general_ci;
```

Alcuni comandi connessi alla creazione di un database sono:

- show databases: mostra la lista dei database presenti;
- drop database: rimuove un database e tutto il suo contenuto.
- show character set: mostra i character set disponibili;
- show collate: mostra i collate disponibili.

E' possibile selezionare il database corrente su cui lavorare con il comando use:

```
use univ;
```

Ora possiamo **creare le tabelle** usando la parte di definizione dei dati di SQL. Per creare una tabella si usa il comando create table. Ad esempio, i seguenti comandi creano le tre tabelle della base di dati univ e i relativi vincoli di integrità:

```
create table studente
  (
    matricola        varchar(8) primary key,
    nome             varchar(20) not null,
    cognome          varchar(20) not null,
    data             date,
    luogo            varchar(30)
  )
comment = 'Questa tabella contiene studenti universitari'
engine = InnoDB;

create table corso
  (
    nome             varchar(30),
    cdl              varchar(30),
    facolta          varchar(30) not null,
    docente          varchar(30),
    primary key      (nome, cdl)
  )
comment = 'Questa tabella contiene corsi universitari'
engine = InnoDB;

create table esame
  (
    studente         varchar(8),
    corso            varchar(30),
    cdl              varchar(30)
    data             date not null,
    voto             tinyint not null,
    lode             enum('si','no') not null default 'no',
    primary key      (studente, corso, cdl),
    foreign key (studente) references studente(matricola)
      on update cascade on delete cascade,
    foreign key (corso, cdl) references corso(nome, cdl)
      on update cascade on delete cascade
```

```
  )
comment = 'Questa tabella contiene esami sostenuti da studenti universitari'
engine = InnoDB;
```

Seguono alcune osservazioni:

- Le tabelle possono essere create con diversi tipi (detti anche *engine*). Il tipo della tabella viene specificato dal parametro engine del comando create, oppure in fase di avvio del server usando l'opzione `--default-storage-engine=type`, dove `type` va sostituito con il tipo che deve essere assegnato alla tabelle create durante la sessione. Tabelle di tipo diverso possono coesistere nella medesima base di dati. I principali engine offerti da MySQL sono:
  - MyISAM: questo è il tipo di default. Non supporta le chiavi esterne e le transazioni e quindi, se tali caratteristiche non servono, è più efficiente e richiede meno spazio di memoria. Permette di effettuare ricerche fulltext e di definire indici fulltext per agevolare tali ricerche. Permette inoltre di creare indici di tipo btree;
  - InnoDB: è ideale per ambienti multi-utente ad alte prestazioni. Gestisce l'integrità referenziale, le transazioni e il locking delle righe. Permette di creare indici del solo tipo btree;
  - MEMORY: è un engine non-transazionale che usa solo la memoria centrale per salvare i dati delle tabelle. Quindi può essere usato per tabelle temporanee non persistenti (lo schema è comunque salvato in memoria secondaria). Permette di creare indici di tipo hash o btree.
- il formato di archiviazione delle righe viene specificato dal parametro row_format, aggiunto dopo la definizione delle colonne e dei vincoli della tabella, e può assumere i valori di dynamic (a lunghezza variabile), fixed (a lunghezza prefissata) e compressed (compresso).
- MySQL offre numerosi tipi di dato da associare agli attributi, tra cui boolean, bit[m] (sequenza di bit di lunghezza m), blob (oggetto binario di grosse dimensioni), text (testo di grosse dimensioni), enum (tipo enumerativo), set (tipo insieme). Ad esempio, abbiamo usato il tipo enum per l'attributo lode della tabella esame.
- il costrutto index permette di creare indici, anche su più colonne. Un indice può avere un nome e un tipo. Gli indici btree sono supportati da tutte le tabelle. Le tabelle di tipo MEMORY possono avere indici hash. Le tabelle MyISAM permettono inoltre indici fulltext (per indicizzare del testo) e spatial (per indicizzare dati geografici). Si noti che un indice viene creato automaticamente per gli attributi della chiave primaria, per quelli dei chiavi candidate (creati con il vincolo unique), e per quelli di chiavi esterne.
- MySQL non implementa il costrutto check (anche se presente nella sintassi), dunque non è possibile creare domini, vincoli generici e asserzioni come in SQL standard.
- E' possibile definire attributi codice specificando il flag auto_increment dopo il tipo dell'attributo. L'attributo codice deve avere un tipo intero.
- E' possibile specificare un commento per la singola colonna o per l'intera tabella mediante l'opzione comment.

E' possibile creare una tabella a partire dallo schema di un'altra tabella, possibilmente importando parte dei dati, come mostrato nei due esempi che seguono:

```
create table corsoInformatica
like corso;

create table corsoInformatica
as select *
   from corso
   where cdl = "Informatica";
```

Nel primo caso, viene creata una tabella vuota corsoInfomatica con lo stesso schema di corso, nel secondo la tabella corsoInformatica viene creata e popolata con i corsi di Informatica.

Alcuni comandi connessi alla creazione delle tabelle seguono:

- alter table: cambia lo schema di una tabella;
- rename table: rinomina tabella;
- drop table: rimuove una tabella e il suo schema;
- show tables: mostra le tabelle presenti nel database corrente;
- show table status: mostra lo stato delle tabelle del database corrente;
- show columns: mostra informazione sulle colonne di una tabella. Il comando `describe table` è una scorciatoria per `show columns from table`;
- show index: mostra informazione sugli indici di una tabella;
- show engines: mostra gli engine disponibili e quello di default;
- show create table: mostra il comando usato per creare una tabella;

Ad esempio, possiamo aggiungere un indice sui campi nome e cognome della tabella studente con il comando alter:

```
alter table studente add index IndNome (nome(5), cognome(5));
```

L'indice definito si chiama IndNome e si basa sui primi 5 caratteri di entrambi gli attributi nome e cognome. Se non specifichiamo un nome, il sistema assegna all'indice un nome univoco, che può essere visto con il comando show create table.

Per vedere le caratteristiche dell'indice creato, in particolare il nome e il tipo, usare il comando show index come segue:

```
show index from studente;
```

Per rimuovere l'indice creato, possiamo usare il comando alter come segue:

```
alter table studente drop index IndNome;
```

Siamo pronti ad **inserire i dati** nelle tabelle. Usiamo il comando insert come segue:

```
insert into studente(matricola, nome, cognome, data, luogo)
values ('ALSBRT66', 'Alessio', 'Bertallot', '1966-12-12', 'Torino'),
       ('PSQMLF56', 'Pasquale', 'Molfetta', '1956-02-11', 'Bari'),
       ('NCLSVN70', 'Nicola', 'Savino', '1970-04-02', NULL);
```

```
insert into corso(nome, cdl, facolta, docente)
values ('Basi di Dati', 'TWM', 'Scienze', 'Franceschet'),
       ('Basi di Dati', 'Informatica', 'Scienze', 'Montanari'),
       ('Tecnologie XML', 'TWM', 'Scienze', 'Franceschet');

insert into esame set
  studente = 'ALSBRT66',
  corso = 'Basi di Dati',
  cdl = 'TWM',
  data = '2007-10-30',
  voto = '30';

insert into esame set
  studente = 'ALSBRT66',
  corso = 'Tecnologie XML',
  cdl = 'TWM',
  data = '2007-09-30',
  voto = '30',
  lode = 'si';

insert into esame set
  studente = 'PSQMLF56',
  corso = 'Tecnologie XML',
  cdl = 'TWM',
  data = '2007-09-18',
  voto = '30';
```

Si noti che la seconda sintassi (usata per la tabella esame) permette di inserire solo i valori per gli attributi che ci interessano. Gli attributi non specificati assumono il valore di default, se è stato specificato, oppure null altrimenti.

Vediamo le operazioni di **aggiornamento dei dati**. Supponiamo di aver sbagliato il voto dell'ultimo inserimento nella tabella esame. Possiamo aggiornare la corrispondente riga con il comando update:

```
update esame
set voto = '18'
where studente = 'PSQMLF56' and
      corso = 'Tecnologie XML' and
      cdl = 'TWM';
```

Per fare uno scherzo ad Alessio Bertallot, potremmo cancellare tutti i sui esami con il seguente comando delete:

```
delete from esame
where studente in (select matricola
                   from studente
                   where nome = 'Alessio' and cognome = 'Bertallot');
```

Supponiamo di aver salvato i dati cancellati in un documento di testo data.txt contenuto nella cartella sql del database univ avente il seguente contenuto:

```
ALSBRT66|Basi di Dati|TWM|2007-10-30|30|no
ALSBRT66|Tecnologie XML|TWM|2007-09-30|30|si
```

Possiamo reinserire i dati nella tabella esame a partire da tale documento con il comando load data infile:

```
load data infile 'univ/sql/data.txt'
into table esame
fields terminated by '|'
lines terminated by '\n';
```

Il carattere `\n` è il separatore di linea di default in Unix, mentre in Windows le linee sono terminate da `\n\r`.

Proviamo a violare i **vincoli di integrità relazionali**. Violiamo un vincolo di dominio:

```
update esame
set voto = 'A'
where studente = 'PSQMLF56' and
      corso = 'Tecnologie XML' and
      cdl = 'TWM';
```

Si noti che l'inserimento viene comunque effettuato usando un valore di default predefinito (0 per gli interi) ma viene segnalato un avvertimento (*warning*) che avvisa del valore scorretto per la colonna voto. Per vedere gli avvertimenti provocati dall'ultimo comando eseguito usare il comando `show warnings`.

Violiamo un vincolo di obbligatorietà per un attributo:

```
update corso
set facolta = NULL
where nome = 'Tecnologie XML' and cdl = 'TWM';
```

Anche in questo caso l'inserimento viene comunque effettuato usando un valore di default predefinito (", la stringa vuota, per le stringhe) ma viene segnalato un avvertimento che avvisa del valore non ammesso per la colonna facolta.

Violiamo ora un vincolo di chiave primaria inserendo uno studente duplicato:

```
insert into studente(matricola, nome, cognome, data, luogo)
values ('ALSBRT66', 'Andrea', 'Collavino', '1986-12-14', 'Udine');
```

In questo caso l'inserimento viene rifiutato e viene segnalata la violazioni di chiave primaria. Per vedere gli errori dell'ultimo comando eseguito usare `show errors`.

Violiamo quindi un vincolo di chiave esterna inserendo un esame per uno studente che non esiste:

```
insert into esame(studente, corso, cdl, data, voto, lode)
values ('MSMFRN72', 'Basi di Dati', 'TWM', '2007-10-30', '30', 'no');
```

Viene segnalato un errore di violazione di integrità referenziale e l'inserimento non viene effettuato. Similmente in caso di modifica.

E' possibile disattivare i controlli referenziali con il comando:

```
SET FOREIGN_KEY_CHECKS = 0;
```

e riattivarli con il comando:

```
SET FOREIGN_KEY_CHECKS = 1;
```

Verifichiamo ora le **politiche referenziali**. Modifichiamo la matricola di uno studente:

```
update studente
set matricola = 'ALSBRT00'
where matricola = 'ALSBRT66';
```

Le modifiche sono riportate a cascata nella tabella esame per le righe della matricola modificata in quanto abbiamo specificato la politica `on update cascade` per il vincolo referenziale **esame**(studente) --> **studente**(matricola).

Ristabiliamo la situazione precedente con una nuova modifica:

```
update studente
set matricola = 'ALSBRT66'
where matricola = 'ALSBRT00';
```

Cancelliamo ora uno studente:

```
delete from studente
where matricola = 'ALSBRT66';
```

La cancellazione ha effetto a cascata nella tabella esame, dove vengono cancellate le tuple che si riferiscono allo studente rimosso. Questo perchè abbiamo specificato la politica `on delete cascade` per il vincolo referenziale **esame**(studente) --> **studente**(matricola).

Ristabiliamo la situazione precedente inserendo le tuple cancellate:

```
insert into studente(matricola, nome, cognome, data, luogo)
values ('ALSBRT66', 'Alessio', 'Bertallot', '1966-12-12', 'Torino');

insert into esame(studente, corso, cdl, data, voto, lode)
values ('ALSBRT66', 'Basi di Dati', 'TWM', '2007-10-30', '30', 'no'),
       ('ALSBRT66', 'Tecnologie XML', 'TWM', '2007-09-30', '30', 'si');
```

Ora possiamo **interrogare la base di dati** creata e popolata con il comando select. Vediamone alcuni esempi:

```
-- where
select nome, cognome
from studente
where data < '1979-12-31' and data > '1970-01-01';

-- order by
select nome, cognome, data
from studente
order by data desc;

-- join
select studente.nome, studente.cognome, esame.corso, esame.voto
from studente join esame on (studente.matricola = esame.studente);

-- left join
select studente.nome, studente.cognome, esame.corso, esame.voto
from studente left join esame on (studente.matricola = esame.studente);
```

```
-- right join
select studente.nome, studente.cognome, esame.corso, esame.voto
from studente right join esame on (studente.matricola = esame.studente);

-- join di più tabelle
select studente.nome, studente.cognome, corso.nome, corso.cdl,
       corso.docente, esame.voto
from (studente join esame on (studente.matricola = esame.studente)) join
     corso on (esame.corso = corso.nome and esame.cdl = corso.cdl);
```

Si noti che MySQL non supporta il full join.

```
-- operatori aggregati
select count(*) as esami, avg(voto) as media, max(voto) as top, min(voto) as bottom
from studente join esame on (studente.matricola = esame.studente)
where studente.nome = 'Alessio' and studente.cognome = 'Bertallot';

-- group by
select cdl, count(*)
from corso
group by cdl
having count(*) > 0
order by count(*) desc;

-- operatori insiemistici
select studente.nome, studente.cognome, esame.corso, esame.voto
from studente left join esame on (studente.matricola = esame.studente)
  union
select studente.nome, studente.cognome, esame.corso, esame.voto
from studente right join esame on (studente.matricola = esame.studente);
```

L'ultima query implementa il full join usando l'operatore union. Gli operatori intersect e except non sono supportati, ma possono essere simulati con le interrogazioni annidate.

```
-- interrogazioni annidate
select nome, cognome
from studente
where matricola in (select studente from esame);

select nome, cognome
from studente
where matricola not in (select studente from esame);

select nome, cognome
from studente join esame on (studente.matricola = esame.studente)
where voto <= all (select voto from esame);

select nome, cognome
from studente join esame on (studente.matricola = esame.studente)
where voto = (select min(voto) from esame);

select S1.nome, S1.cognome
from studente S1
where exists (select *
              from studente S2
              where (S1.data = S2.data) and
                    (S1.matricola <> S2.matricola));
```

E' possibile ridirigere il risultato di una interrogazione in un file come segue:

```
-- output su file
select * from esame
into outfile 'univ/sql/data2.txt'
fields terminated by '|'
lines terminated by '\n';
```

Il comando load into file permette di caricare i dati letti da un file.

Vediamo ora la **definizione di viste** (comando create view) e il loro uso nelle interrogazioni:

```
create view algorithm = merge curriculum (nome, corso, docente, data, voto, lode) as
select concat(studente.nome, ' ', studente.cognome),
       concat(corso.nome, ' (', corso.cdl, ')'),
       corso.docente, esame.data, esame.voto, esame.lode
from (studente join esame on (studente.matricola = esame.studente)) join
     corso on (esame.corso = corso.nome and esame.cdl = corso.cdl);

select nome, avg(voto) as media
from curriculum
where (docente = 'Franceschet')
group by nome
having avg(voto) >= 27
order by avg(voto) desc;
```

MySQL non supporta viste temporanee (clausola with) e neanche viste ricorsive; ogni vista è inserita stabilemente nello schema e può essere rimossa con il comando drop view. E' possibile specificare l'algoritmo con cui la vista viene gestita: le opzioni sono `merge` che riscrive una interrogazione che usa una vista usando la definizione della vista, e `temptable` che crea una tabella temporanea per la vista e usa tale tabella nella valutazione di interrogazioni che ne fanno uso.

Le viste fanno parte dello schema e sono visibili con il comando show create view.

Le **regole attive** (trigger) possono essere create col comando create trigger e rimosse col comando drop trigger. Nella versione usata di MySQL, le regole attive non vengono attivate dalle modifiche referenziali (le modifiche a cascata sulla tabella secondaria come conseguenza delle azioni fatte su una tabella principale).

Vediamo un paio di esempi. Vogliamo aggiungere un attributo calcolato media alla tabella studente che registra la media dei voti degli studenti. Creiamo il campo, lo aggiorniamo allo stato corrente e infine creiamo i trigger per gli eventi di modifica, cancellazione e inserimento di un nuovo esame:

```
-- regole attive
alter table studente
add column media tinyint;

update studente
set media = (select avg(voto)
             from esame
             where matricola = esame.studente);

create trigger updateAvg
after update on esame
for each row
update studente
```

```
set media = (select avg(voto)
             from esame
             where matricola = esame.studente)
where matricola = new.studente;

create trigger deleteAvg
after delete on esame
for each row
update studente
set media = (select avg(voto)
             from esame
             where matricola = esame.studente)
where matricola = old.studente;


create trigger insertAvg
after insert on esame
for each row
update studente
set media = (select avg(voto)
             from esame
             where matricola = esame.studente)
where matricola = new.studente;
```

Proviamo i trigger con i seguenti comandi di aggiornamento:

```
update esame
set voto = '18'
where (studente = 'ALSBRT66') and
      (corso = 'Basi di Dati') and
      (cdl = 'TWM');

delete from esame
where studente = 'ALSBRT66' and
      corso = 'Basi di Dati' and
      cdl = 'TWM';

insert into esame(studente, corso, cdl, data, voto, lode)
values ('ALSBRT66', 'Basi di Dati', 'TWM', '2007-10-30', '30', 'no'),
       ('PSQMLF56', 'Basi di Dati', 'TWM', '2007-10-30', '20', 'no'),
       ('NCLSVN70', 'Basi di Dati', 'TWM', '2007-10-30', '25', 'no');
```

Creiamo quindi un paio di trigger per fa sì che la lode ci sia solo quando il voto è uguale a 30. Si noti che questi trigger vengono chiamati prima dei rispettivi eventi:

```
delimiter //
create trigger insertLode
before insert on esame
for each row
begin
if ((new.voto < 30) and (new.lode = 'si')) then
  set new.lode = 'no';
end if;
end;//
delimiter ;

delimiter //
create trigger updateLode
before update on esame
for each row
begin
if ((new.voto < 30) and (new.lode = 'si')) then
```

```
  set new.lode = 'no';
end if;
end;//
delimiter ;
```

Proviamo gli ultimi due trigger creati:

```
insert into esame(studente, corso, cdl, data, voto, lode)
values ('NCLSVN70', 'Tecnologie XML', 'TWM', '2007-10-30', '27', 'si');

update esame
set lode = 'si'
where (studente = 'NCLSVN70') and
      (corso = 'Tecnologie XML') and
      (cdl = 'TWM');
```

I trigger fanno parte dello schema e sono visibili con il comando show triggers.

MySQL supporta le **transazioni** su tabelle di tipo InnoDB. I relativi comandi sono:

- start transaction: inizia una transazione;
- commit: termina una transazione rendendone persistenti gli effetti;
- rollback: termina una transazione annullandone gli effetti;
- savepoint: stabilisce un punto salvo all'interno di una transazione;
- rollback to savepoint: annulla gli effetti della transazione fino al punto salvo nominato;
- set transaction: assegna il livello di isolamento delle future transazioni;
- lock: blocca l'accesso ad una tabella;
- unlock: sblocca l'accesso ad una tabella.

Seguono due esempi:

```
-- transazioni
start transaction;

insert into studente(matricola, nome, cognome, data, luogo)
values ('ELNDCC76', 'Elena', 'Di Cioccio', '1976-01-01', 'Monza');
insert into esame(studente, corso, cdl, data, voto, lode)
values ('ELNDCC76', 'Basi di Dati', 'TWM', '2007-11-02', '30', 'no');

commit;

start transaction;

insert into studente(matricola, nome, cognome, data, luogo)
values ('FBIVLO72', 'Fabio', 'Volo', '1972-04-18', 'Brescia');
insert into esame(studente, corso, cdl, data, voto, lode)
values ('FBIVLO72', 'Basi di Dati', 'TWM', '2007-11-02', '18', 'no');

rollback;
```

MySQL permetta la creazione di procedure e funzioni definite dall'utente (dette *stored*) in quanto vengono salvate assieme allo schema del database). I comandi `create procedure` e `create function` creano procedure e funzioni. Il loro corpo contiene comandi SQL e possibilmente altri costrutti tipici dei comuni linguaggi di programmazioni. La comunicazione con l'ambiente

chiamante avviene attraverso parametri. Segue un esempio di procedura che calcola il numero di esami superati dallo studente la cui matricola viene passata come parametro:

```
delimiter //
create procedure conta (in matricola varchar(8), out num INT)
begin
  select count(*) into num
  from esame
  where studente = matricola;
end;
//
delimiter ;
```

Si noti che abbiamo dovuto cambiare il delimitatore di comando in `//` in quanto il simbolo `;` viene usato all'interno del corpo della procedura con un'altra semantica. Possiamo invocare una procedura o funzione con il comando `call`. Si noti che abbiamo anche usato in comando `set` per assegnare una variabile:

```
set @studente = 'ALSBRT66';
call conta(@studente, @num);
select @studente, @num;
```

Le procedure fanno parte dello schema e sono visibili con i comandi show procedure code e show procedure status.

Infine, vediamo come **salvare e ripristinare** la nostra base di dati. Per fare il backup possiamo usare il programma mysqldump come segue:

```
mysqldump -u francesc -p --lock-tables --routines univ > /home/francesc/backup/backup.sql
```

L'effetto del comando è di esportare la base di dati univ in un file backup.sql. Tale file conterrà i comandi create e insert per ricreare la base di dati. L'opzione `--lock-tables` blocca le tabelle durante il backup, mentre `--routines` esporta anche le procedure e funzioni (i trigger e le viste vengono esportati di default). L'opzione `--xml`esporta il database in formato XML. Per ripristinare la base di dati è necessario ricrearla con il comando create database e eseguire il file backup.sql ad esempio con il comando source dal monitor di MySQL.

MySQL permette la connettività di applicazioni client sviluppate nel linguaggio Java attraverso un **driver JDBC**, chiamato MySQL Connector/J. MySQL Connector/J è un driver di tipo 4: questo significa che è scritto completamente in Java e che comunica direttamente con il server MySQL attraverso il protocollo MySQL. Si veda il tutorial della Sun dedicato a JDBC

Caso di studio: schema fisico

Concludiamo la progettazione del nostro caso di studio con la progettazione fisica. Non assumeremo l'uso di alcun DBMS specifico ma faremo riferimento ad SQL standard. In particolare questo ci impedirà di definire in SQL le strutture di memorizzazione per le tabelle e

gli indici. Lo schema fisico della base di dati TeatroSQL, in tutte le sue componenti (domini, tabelle, asserzioni, trigger e viste) è il seguente:

```
create schema TeatroSQL
{

  create domain tipoBiglietto as varchar(8)
  default "Intero"
  check (tipoBiglietto in ('Intero', 'Ridotto', 'Studenti'))

  create domain mansione as varchar(5)
  check (mansione in ('CA', 'POD', 'CUSRP', 'ACF'))

  create domain tipoProduttore as varchar(7)
  check (tipoProduttore in ('interno', 'esterno'))

  create table teatro
  (
    nome             varchar(20) primary key,
    telefono         varchar(15),
    fax              varchar(15),
    indirizzo        varchar(40) not null,
    email            varchar(30),
    url              varchar(30)
  )

  create table biglietteria
  (
    nome             varchar(20) primary key,
    indirizzo        varchar(40) not null,
    email            varchar(30),
    telefono         varchar(15),
    teatro           varchar(20) foreign key references teatro(nome)
                                 on update cascade on delete set null
  )

  create table orario
  (
    biglietteria     varchar(20),
    giorno           varchar(10),
    inizio           time,
    fine             time,
    primary key(biglietteria, giorno, inizio),
    foreign key biglietteria references biglietteria(nome)
                on update cascade on delete cascade,
  )

  create table notizia
  (
    data        date,
    ora         time,
    oggetto     varchar(100),
    testo       CLOB,
    primary key(data, ora, oggetto)
  )

  create table newsletter
  (
    teatro      varchar(20),
    data        date,
    ora         time,
    oggetto     varchar(100),
    primary key(teatro, data, ora, oggetto),
```

```
  foreign key teatro references teatro(nome)
              on update cascade on delete cascade,
  foreign key (data, ora, oggetto) references notizia(data, ora, oggetto)
              on update cascade on delete cascade
)

create table dipendente
(
  cf                    char(16) primary key,
  nome                  varchar(20) not null,
  cognome               varchar(20) not null,
  dataDiNascita         date,
  luogoDiNascita        varchar(20),
  residenza             varchar(30),
  telefonoFisso         varchar(15),
  telefonoMobile        varchar(15),
  email                 varchar(30)
)

create table lavoro
(
  teatro              varchar(20),
  dipendente          char(16),
  dataAssunzione      date,
  ruolo               mansione,
  cda                 boolean,
  primary key(teatro, dipendente),
  foreign key teatro references teatro(nome)
              on update cascade on delete cascade,
  foreign key dipendente references dipendente(cf)
              on update cascade on delete cascade,
  constraint RA1 check (
  cda = false or
  (year(current_date) - year(dataAssunzione) > 10) or
  (year(current_date) - year(dataAssunzione) = 10 and
     month(current_date) > month(dataAssunzione)) or
  (year(current_date) - year(dataAssunzione) = 10 and
     month(current_date) = month(dataAssunzione) and
     day(current_date) >= day(dataAssunzione))
  )
)

create table stipendio
(
  dipendente    char(16),
  inizio        date,
  importo       decimal(6,2),
  primary key(dipendente, inizio),
  foreign key dipendente references dipendente(cf)
              on update cascade on delete cascade
)

create table spazio
(
  nome             varchar(20) primary key,
  indirizzo        varchar(40) not null,
  pianta           varchar(20),
  capienza         smallint
)

create table luogo
(
  teatro     varchar(20),
  spazio     varchar(20),
```

```
  primary key(teatro, spazio),
  foreign key teatro references teatro(nome)
              on update cascade on delete cascade
  foreign key spazio references spazio(nome)
              on update cascade on delete cascade
)

create table stagione
(
  nome           varchar(20),
  biennio        char(9),
  teatro         varchar(20),
  primary key(nome, biennio),
  foreign key teatro references teatro(nome)
              on update cascade on delete set null
)

create table spettacolo
(
  titolo             varchar(40) primary key,
  descrizione        CLOB,
  annoProduzione     char(4),
)

create table messaInScena
(
  data               date,
  ora                time,
  spazio             varchar(20),
  spettacolo         varchar(40),
  postiDisponibili   smallint,
  prezzoIntero       decimal(5,2),
  prezzoRidotto      decimal(5,2),
  prezzoStudenti     decimal(5,2),
  primary key (data, ora, spazio),
  foreign key spazio references spazio(nome)
              on update cascade on delete set null,
  foreign key spettacolo references spettacolo(titolo)
              on update cascade on delete set null,
  constraint RA3-1 check (postiDisponibili >= 0)
)

create table materiale
(
  file                   varchar(30) primary key,
  tipo                   varchar(10),
  dimensione             integer,
  dataSpettacolo         date,
  oraSpettacolo          time,
  spazioSpettacolo       varchar(20),
  foreign key (dataSpettacolo, oraSpettacolo, spazioSpettacolo) references
               messaInScena(data, ora, spazio)
               on update cascade on delete set null
)

create table prenotazione
(
  dataSpettacolo     date,
  oraSpettacolo      time,
  spazioSpettacolo   varchar(20),
  numero             smallint,
  data               date,
  ora                time,
  posto              varchar(5),
```

```
  tipo               tipoBiglietto,
  prezzo             decimal(5,2),
  primary key(spettacolo, numero),
  foreign key (dataSpettacolo, oraSpettacolo, spazioSpettacolo) references
              messaInScena(data, ora, spazio)
              on update cascade on delete cascade
)

create table commento
(
  autore             varchar(20),
  data               date,
  ora                time,
  testo              CLOB,
  dataSpettacolo     date,
  oraSpettacolo      time,
  spazioSpettacolo   varchar(20),
  primary key (autore, data, ora),
  foreign key (dataSpettacolo, oraSpettacolo, spazioSpettacolo) references
              messaInScena(data, ora, spazio)
              on update cascade on delete set null
)

create table risposta
(
  autoreRisposta     varchar(20),
  dataRisposta       date,
  oraRisposta        time,
  autore             varchar(20),
  data               date,
  ora                time,
  primary key (autoreRisposta, dataRisposta, oraRisposta),
  foreign key (autoreRisposta, dataRisposta, oraRisposta) references
              commento(autore, data, ora)
              on update cascade on delete cascade,
  foreign key (autore, data, ora) references
              commento(autore, data, ora)
              on update cascade on delete cascade
)

create table produzione
(
  produttore     varchar(20),
  spettacolo     varchar(40),
  primary key(produttore, spettacolo),
  foreign key produttore references produttore(nome)
              on update cascade on delete cascade
  foreign key spettacolo references spettacolo(titolo)
              on update cascade on delete cascade
)

create table produttore
(
  nome           varchar(20) primary key,
  tipo           tipoProduttore,
  indirizzo      varchar(40),
  telefono       varchar(15),
  email          varchar(30),
)

create table interprete
(
  nome                varchar(20),
  ruolo               varchar(15),
```

```
  cv                    CLOB,
  primary key(nome)
)

create table cast
(
  spettacolo               varchar(40),
  interprete               varchar(20),
  primary key(spettacolo, interprete),
  foreign key spettacolo references spettacolo(titolo)
              on update cascade on delete cascade
  foreign key interprete references interprete(nome)
              on update cascade on delete cascade
)

crate assertion RA2 check (
  not exists (
    select stagione.teatro, stagione.nome, stagione.biennio, count(*)
    from (stagione join proposta on
         (stagione.nome = proposta.nomeStagione) and
         (stagione.biennio = proposta.biennioStagione))
         join produzione on
         (proposta.spettacolo = produzione.spettacolo) and
         (stagione.teatro = produzione.produttore)
    group by stagione.teatro, stagione.nome, stagione.biennio
    having count(*) > 2
  )
)

crate trigger RA3-2
after insert on prenotazione
for each row
update messaInScena
set postiDisponibili = postiDisponibili - 1
where (data = New.dataSpettacolo and
       ora = New.oraSpettacolo
       spazio = New.spazioSpettacolo)


crate trigger RA3-3
after delete on prenotazione
for each row
update messaInScena
set postiDisponibili = postiDisponibili + 1
where (data = Old.dataSpettacolo and
       ora = Old.oraSpettacolo
       spazio = Old.spazioSpettacolo)


crate trigger RA3-4
after insert on messaInScena
for each row
update messaInScena
set postiDisponibili = (select capienza
                        from spazio
                        where nome = New.spazio)
where (data = New.data and
       ora = New.ora
       spazio = New.spazio)


crate trigger RA4-1
after insert on messaInScena
for each row
```

```
update messaInScena
set New.prezzoRidotto = New.prezzoIntero * 0.8
where (data = New.data and
       ora = New.ora
       spazio = New.spazio)


crate trigger RA4-2
after insert on messaInScena
for each row
update messaInScena
set New.prezzoStudenti = New.prezzoIntero * 0.5
where (data = New.data and
       ora = New.ora
       spazio = New.spazio)


crate trigger RA4-3
after update of prezzoIntero on messaInScena
for each row
update messaInScena
set New.prezzoRidotto = New.prezzoIntero * 0.8
where (data = New.data and
       ora = New.ora
       spazio = New.spazio)


crate trigger RA4-4
after update of prezzoIntero on messaInScena
for each row
update messaInScena
set New.prezzoStudenti = New.prezzoIntero * 0.5
where (data = New.data and
       ora = New.ora
       spazio = New.spazio)


crate trigger RA5-1
after insert on prenotazione
for each row
when (tipo = "Intero")
update prenotazione
set New.prezzo = (select prezzoIntero
                  from messaInScena
                  where (data = New.dataSpettacolo and
                         ora = New.oraSpettacolo and
                         spazio = New.spazioSpettacolo))
where (dataSpettacolo = New.dataSpettacolo and
       oraSpettacolo = New.oraSpettacolo and
       spazioSpettacolo = New.spazioSpettacolo and
       numero = New.numero)


crate trigger RA5-2
after insert on prenotazione
for each row
when (tipo = "Ridotto")
update prenotazione
set New.prezzo = (select prezzoRidotto
                  from messaInScena
                  where (data = New.dataSpettacolo and
                         ora = New.oraSpettacolo and
                         spazio = New.spazioSpettacolo))
where (dataSpettacolo = New.dataSpettacolo and
```

```
       oraSpettacolo = New.oraSpettacolo and
       spazioSpettacolo = New.spazioSpettacolo and
       numero = New.numero)


crate trigger RA5-1
after insert on prenotazione
for each row
when (tipo = "Studenti")
update prenotazione
set New.prezzo = (select prezzoStudenti
                  from messaInScena
                  where (data = New.dataSpettacolo and
                         ora = New.oraSpettacolo and
                         spazio = New.spazioSpettacolo))
where (dataSpettacolo = New.dataSpettacolo and
       oraSpettacolo = New.oraSpettacolo and
       spazioSpettacolo = New.spazioSpettacolo and
       numero = New.numero)


create view spettacoli(teatro, nomeStagione, biennioStagione,
  titolo, descrizione, data, ora, spazio, indirizzo, postiDisponibili
  prezzoIntero, prezzoRidotto, prezzoStudenti) as
select St.teatro, St.nomeStagione, St.biennioStagione, Spe.titolo,
  Spe.descrizione, Sce.data, Sce.ora, Sce.spazio, Spa.indirizzo,
  Sce.postiDisponibili, Sce.prezzoIntero, Sce.prezzoRidotto, Sce.prezzoStudenti
from stagione St, proposta P, spettacolo Spe, messaInScena Sce, spazio Spa
where (St.nome = P.nomeStagione) and
      (St.biennio = P.biennioStagione) and
      (P.spettacolo = Spe.titolo) and
      (Spe.titolo = Sce.spettacolo)
      (Sce.spazio = Spa.nome)
order by St.teatro, St.nomeStagione, St.biennioStagione, Sce.data


create view attori(teatro, nomeStagione, biennioStagione, nome, ruolo, cv) as
select St.teatro, St.nomeStagione, St.biennioStagione, I.nome, I.ruolo, I.cv
from stagione St, proposta P, spettacolo Sp, cast C, interprete I
where (St.nome = P.nomeStagione) and
      (St.biennio = P.biennioStagione) and
      (P.spettacolo = Sp.titolo) and
      (Sp.titolo = C.spettacolo) and
      (C.interprete = I.nome)
order by St.teatro, St.nomeStagione, St.biennioStagione, I.nome, I.ruolo


create view dipendenti(teatro, cognome, nome, cf, età, telefonoFisso,
  telefonoMobile, email, stipendio, dataDiAssunzione, ruolo, cda) as
select L.teatro, D.cognome, D.nome, D.cf,
  case
    when ((month(current_date) > month(D.dataDiNascita)) or
          (month(current_date) = month(D.dataDiNascita) and
           day(current_date) >= day(D.dataDiNascita)))
    then year(current_date) - year(D.dataDiNascita)
    else year(current_date) - year(D.dataDiNascita) - 1
  end,
  D.telefonoFisso, D.telefonoMobile, D.email, S.stipendio,
  L.dataDiAssunzione, L.ruolo, L.cda
from lavoro L, dipendente D, stipendio S
where (L.dipendente = D.cf) and (S.dipendente = D.cf) and
      (S.inizio = (select max(S2.inizio)
                   from stipendio S2
                   where S2.dipendente = D.cf))
```

```
  order by L.teatro, D.cognome, D.nome


  create view statScena(spettacolo, paganti, affluenza, incasso) as
  select Sce.spettacolo, count(*), count(*) / Spa.capienza, sum(P.prezzo)
  from messaInScena Sce, spazio Spa, prenotazione P
  where (Sce.spazio = Spa.nome) and
        (P.dataSpettacolo = Sce.data) and
        (P.oraSpettacolo = Sce.ora) and
        (P.spazioSpettacolo = Sce.spazio)
  group by Sce.data, Sce.ora, Sce.spazio

  create view statSpettacolo(spettacolo, paganti, affluenza, incasso) as
  select Spe.titolo, avg(Sce.paganti), avg(Sce.affluenza), avg(Sce.incasso)
  from spettacolo Spe, statScena Sce
  where (Spe.titolo = Sce.spettacolo)
  group by Spe.titolo

}
```

Seguono alcune osservazioni:

- compito della progettazione fisica è anche definire le strutture di memorizzazione e di accesso ai dati tra quelle offerte dal DBMS usato. Nel nostro caso assumiamo di definire un indice almeno sulla chiave primaria di tutte le tabelle;
- per quanto riguarda le politiche di integrità referenziale, per le modifiche abbiamo sempre usato la politica `on update cascade`. Per le cancellazioni abbiamo usato la politica `on delete cascade` per chiavi esterne di tabelle che corrispondono a relazioni concettuali (ad esempio lavoro) oppure ad entità deboli (ad esempio prenotazione) e la politica `on delete set null` negli altri casi. Nel primo caso vi è un forte collegamento tra la tabella master e la tabella slave e dunque una cancellazione nella tabella master dovrebbe provocare corrispondenti cancellazioni nella tabella slave;
- i vincoli di integrità corrispondenti alle regole aziendali sono stati nominati con la sigla RA seguida dal numero della regola aziendale, eventualmente seguito da un trattino e da un numero progressivo qualora siano stati necessari pi vincoli per specificare la regola;
- per specificare la prima regola aziendale e l'attributo età della vista dipendenti sono state usate le funzioni `current_date` che restituisce la data corrente, `year`, `month`, e `day` che rispettivamente estraggono l'anno, il mese e il giorno da una data;
- Per specificare la terza regola aziendale sui posti disponibili abbiamo usare i trigger RA3-2 (che decrementa di uno i posti disponibili quando una prenotazione viene inserita), RA3-3 (che incrementa di uno i posti disponibili quando una prenotazione viene cancellata), RA3-4 (che imposta il numero di posti disponibili alla capienza dello spazio teatrale quando uno spettacolo viene inserito). Inoltre, abbiamo usato il vincolo check RA3-1 che specifica che l'attributo postiDisponibili deve essere maggiore o uguale a zero. Dunque, un inserimento di una prenotazione quando i posti sono esauriti innesca il trigger RA3-2 che viola il vincolo RA3-1. Il sistema dunque annulla l'inserimento della prenotazione.

  Una soluzione alternativa consiste nel sostituire il vincolo RA3-1 con il seguente trigger:

-
- `crate trigger RA3-1`
- `  before insert on prenotazione`

```
for each row
when (0 = (select postiDisponibili
            from messaInScena
            where (data = New.dataSpettacolo and
                   ora = New.oraSpettacolo
                   spazio = New.spazioSpettacolo)))
rollback("Posti esauriti")
```

Esso controlla, prima dell'inserimento della prenotazione nella base di dati, che vi siano posti disponibili. Se non ve ne sono, annulla l'operazione di inserimento e avvisa l'utente che i posti sono esauriti. Si noti che la soluzione funziona se RA3-1 viene eseguito prima di RA3-2 (di solito è così in quanto RA3-1 è di tipo before e RA3-2 è di tipo after).

- il dominio mansione, associato all'attributo ruolo della tabella lavoro, assume i seguenti valori corrispondenti alle seguenti mansioni:
  - CA: coordinamento artistico
  - POD: pruduzione, organizzazione e distribuzione
  - CUSRP: comunicazione, ufficio stampa e relazioni con il pubblico
  - ACF: amministrazione, controllo e finanza
- l'attributo età della vista dipendenti è stato calcolato come differenza tra la data corrente e quella di nascita del dipendente. Questa soluzione può essere impiegata per implementare attributi calcolati che dipendono dal tempo: l'attributo viene letto attraverso una vista e il suo valore è calcolato in tempo reale. Si noti che non è possibile usare un trigger per implementare un attributo calcolato come età in quanto il valore dell'età cambia indipendentemente dagli aggiornamenti della base, cioè dagli eventi che attivano i trigger;
- per gli oggetti binari (immagini e video) abbiamo preferito usare attributi di tipo stringa che contengono il percorso del file che contiene l'oggetto piuttosto che l'uso di un attributo di tipo BLOB. Per i documenti si puro testo abbiamo invece usato attributi di tipo CLOB.

Mostriamo infine l'implementazione delle **transazioni tipiche** di interrogazione (quelle di inserimento dipendono dai dati da inserire). Alcune di queste transazioni fanno uso delle viste definite nello schema:

- *Trovare gli spettacoli messi in scena in una stagione teatrale di un teatro. Per ogni spettacolo occorre stampare il titolo, una descrizione, data, ora e luogo di scena, posti disponibili e prezzi.*

  Basta instanziare la vista spettacoli con un teatro e una stagione.
- *Trovare gli attori che hanno lavorato in una stagione di un teatro. La stessa transazione per i registi e gli autori.*

  Basta instanziare la vista interpreti con un particolare ruolo, un teatro e una stagione.

- *Trovare i dipendenti di un teatro.*

  Basta instanziare la vista dipendenti con un teatro.
- *Trovare i commenti riferiti ad uno spettacolo.*

```
select C.autore, C.data, C.ora, C.testo
from messaInScena S, commento C
where (S.data = C.dataSpettacolo) and
      (S.ora = C.oraSpettacolo) and
      (S.spazio = C.spazioSpettacolo) and
      (S.titolo = 'Gli straccioni')
```

- *Trovare le notizie riferite ad un teatro.*

```
select News.nome, N.data, N.ora, N.oggetto, N.testo
from notizia N, newsletter News
where (N.data = News.data) and
      (N.ora = News.ora) and
      (N.oggetto = News.oggetto) and
      (News.teatro = 'CSS')
```

- *Trovare gli orari di apertura delle biglietterie di un teatro.*

```
select O.biglietteria, O.giorno, O.inizio, O.fine
from biglietteria B, orario O
where (B.nome = O.biglietteria) and (B.teatro = 'CSS')
order by O.biglietteria
```

- *Generare una statistica sull'andamento delle stagioni teatrali della rete. Per ogni stagione si vuole calcolare la media degli spettatori paganti, la media dell'affluenza, cioè del rapporto tra paganti e capienza e la media degli incassi riferite agli spettacoli della stagione. Si vuole inoltra stampare gli spettacoli di una particolare stagione in ordine di: (i) paganti, (ii) affluenza, (iii) incasso.*

```
select P.nomeStagione, P.biennioStagione, avg(SS.paganti) as mediaPaganti,
       avg(SS.affluenza) as mediaAffluenza, avg(SS.incasso) as mediaIncasso
from statisticaSpettacolo SS, proposta P
where (SS.spettacolo = P.spettacolo)
group by P.nomeStagione, P.biennioStagione


select SS.spettacolo, SS.paganti
from statisticaSpettacolo SS, proposta P
where (SS.spettacolo = P.spettacolo) and
      (P.nomeStagione = 'Contatto') and
      (P.biennioStagione = '2006/2007')
order by SS.paganti


select SS.spettacolo, SS.affluenza
from statisticaSpettacolo SS, proposta P
where (SS.spettacolo = P.spettacolo) and
      (P.nomeStagione = 'Contatto') and
      (P.biennioStagione = '2006/2007')
order by SS.affluenza


select SS.spettacolo, SS.incasso
from statisticaSpettacolo SS, proposta P
```

```
where (SS.spettacolo = P.spettacolo) and
      (P.nomeStagione = 'Contatto') and
      (P.biennioStagione = '2006/2007')
order by SS.incasso
```

Esercitazione

Si consideri lo schema logico per la base di dati universitaria.

Interrogazione

Scrivere le seguenti interrogazioni in SQL:

1. Nome e cognome di tutti gli studenti ordinati per data di immatricolazione in ordine decrescente

```
select nome, cognome
from studente
order by dataImmatricolazione desc
```

5. Nome, cognome e data di immatricolazione di tutti gli studenti iscritti al corso di laurea di *Tecnologie Web e Multimediali*, immatricolati nel nuovo millennio e ordinati per cognome e nome in ordine crescente

```
select nome, cognome, dataImmatricolazione
from studente
where (cdl = 'Tecnologie Web e Multimediali') and
      (dataImmatricolazione >= '2000-01-01')
order by cognome, nome
```

11. Nome, cognome e corso di laurea di tutti gli studenti iscritti ad un corso di laurea che contiene la parola *Tecnologie* nel nome

```
select nome, cognome, cdl
from studente
where cdl like '%Tecnologie%'
```

15. Il nome di tutti i corsi della laurea specialistica (IV e V anno) che insegnano qualche argomento relativo a *XML*

```
select nome
from corso
where ((anno = 4) or (anno = 5)) and (programma like '%XML%')
```

19. Nome e cognome di tutti i ricercatori della facoltà di *Scienze* che hanno un indirizzo di posta elettronica ma non hanno un sito Web personale

```
select nome, cognome
from docente
where (tipo = 'Ricercatore') and (facoltà = 'Scienze') and
      (email is not null) and (sito is null)
```

24. Il codice identificativo del presidente del corso di laurea dello studente con matricola *84444*

```
select cdl.presidente
from studente join cdl on studente.cdl = cdl.nome
where studente.matricola = '84444'
```

28. Il nome e cognome del presidente del corso di laurea dello studente con matricola *84444*

```
select docente.nome, docente. cognome
from (studente join cdl on studente.cdl = cdl.nome)
```

```
31.       join docente on cdl.presidente = docente.codice
32.  where studente.matricola = '84444'
```

33. Il nome e cognome del preside della facoltà dello studente con matricola *84444*

```
34.  select docente.nome, docente. cognome
35.  from ((studente join cdl on studente.cdl = cdl.nome)
36.       join facoltà on cdl.facoltà = facoltà.nome)
37.       join docente on facoltà.preside = docente.codice
38.  where studente.matricola = '84444'
```

39. Gli studenti che hanno un omonimo nello stesso corso di laurea

```
40.  select distinct s1.matricola, s1.nome, s1.cognome
41.  from studente s1, studente s2
42.  where (s1.nome = s2.nome) and (s1.cognome = s2.cognome) and
43.        (s1.cdl = s2.cdl) and (s1.matricola <> s2.matricola)
```

44. I corsi del medesimo corso di laurea che hanno lezione lo stesso giorno alla stessa ora

```
45.  select distinct l1.corso, l1.cdl
46.  from lezione l1, lezione l2
47.  where (l1.cdl = l2.cdl) and
48.        (l1.giorno = l2.giorno) and
49.        (l1.fascia = l2.fascia) and
50.        (l1.aula <> l2.aula)
```

51. I corsi allo stesso anno di corso del medesimo corso di laurea che hanno lezione lo stesso giorno alla stessa ora

```
52.  select distinct l1.corso, l1.cdl
53.  from corso c1, lezione l1, corso c2, lezione l2
54.  where (c1.nome = l1.corso) and
55.        (c1.cdl = l1.cdl) and
56.        (c2.nome = l2.corso) and
57.        (c2.cdl = l2.cdl) and
58.        (c1.anno = c2.anno) and
59.        (l1.cdl = l2.cdl) and
60.        (l1.giorno = l2.giorno) and
61.        (l1.fascia = l2.fascia) and
62.        (l1.aula <> l2.aula)
```

63. Il numero di corsi insegnati dal docente *Roberto Ranon*

```
64.  select count(*)
65.  from corso join docente on corso.docente = docente.codice
66.  where (docente.nome = 'Roberto') and (docente.cognome = 'Ranon')
```

67. Il numero di corsi insegnati al corso di laurea di *Tecnologie Web e Multimediali* per ogni tipo di docente

```
68.  select docente.tipo, count(*)
69.  from corso join docente on corso.docente = docente.codice
70.  where cdl = 'Tecnologie Web e Multimediali'
71.  group by docente.tipo
```

72. Il numero di studenti per ogni corso di laurea ordinati per facoltà e per corso di laurea

```
73.  select cdl.facoltà, cdl.nome, count(*)
74.  from studente join cdl on studente.cdl = cdl.nome
75.  group by cdl.facoltà, cdl.nome
76.  order by cdl.facoltà, cdl.nome
```

77. Il numero di studenti per ogni corso di laurea della facoltà di *Scienze* per i soli corsi di laurea con almeno 10 studenti ordinati per facoltà e per corso di laurea

```
78.  select cdl.facoltà, cdl.nome, count(*)
79.  from studente join cdl on studente.cdl = cdl.nome
80.  where cdl.facoltà = 'Scienze'
81.  group by cdl.facoltà, cdl.nome
82.  having count(*) >= 10
83.  order by cdl.facoltà, cdl.nome
```

84. I corsi di laurea ordinati a decrescere per numero di studenti (inclusi i corsi privi di studenti)

```
select cdl.nome, count(studente.matricola)
from cdl left join studente on cdl.nome = studente.cdl
group by cdl.nome
order by count(studente.matricola) desc
```

89. Tutti i corsi tenuti da un docente che è preside di qualche facoltà

```
select corso.nome, corso.cdl
from corso
where corso.docente in (select preside
                        from facoltà)
```

94. Tutti i corsi tenuti da un docente che non è preside di alcuna facoltà

```
select corso.nome, corso.cdl
from corso
where corso.docente not in (select preside
                            from facoltà)
```

oppure

```
select corso.nome, corso.cdl
from corso
  except
select corso.nome, corso.cdl
from corso
where corso.docente in (select preside
                        from facoltà)
```

99. Tutti i corsi tenuti da un docente che è preside di qualche facoltà o presidente di qualche corso di laurea

```
select corso.nome, corso.cdl
from corso
where corso.docente in (select preside
                        from facoltà
                          union
                        select presidente
                        from cdl)
```

107. Gli studenti che non hanno omonimi

```
select s1.matricola, s1.nome, s1.cognome
from studente s1
where not exists(select s2.matricola
                 from studente s2
                 where (s1.matricola <> s2.matricola) and
                       (s1.nome = s2.nome) and
                       (s1.cognome = s2.cognome))
```

115. Tutti i corsi che mutuano sul corso di *Basi di Dati* presso *Tecnologie Web e Multimediali*

```
select corso, cdl
from mutuo
where (corsoMutuo = 'Basi di Dati') and
      (cdlMutuo = 'Tecnologie Web e Multimediali')
```

120. Tutti i corsi che mutuano su un corso che mutua sul corso di *Basi di Dati* presso *Tecnologie Web e Multimediali*

```
select m1.corso, m1.cdl
from mutuo m1 join mutuo m2 on (m1.corsoMutuo = m2.corso) and
                               (m1.cdlMutuo = m2.cdl)
where (m2.corsoMutuo = 'Basi di Dati') and
      (m2.cdlMutuo = 'Tecnologie Web e Multimediali')
```

Definizione dei dati

Scrivere i comandi SQL per creare le tabelle e i vincoli di integrità tipici del modello relazionale.

Regole aziendali

Realizzare le seguenti regole aziendali usando il costrutto `check` di SQL:

1. il preside di una facoltà deve afferire alla facoltà;
2. il presidente di un corso di laurea deve afferire alla facoltà che contiene il corso di laurea;
3. un corso può mutuare su altri corsi solo se tali corsi appartengono ad altri corsi di laurea;
4. non vi possono essere collisioni di orario tra corsi dello stesso corso di laurea insegnati allo stesso anno di corso;
5. un corso deve essere tenuto da al più un docente;
6. i ricercatori non tengono corsi, gli associati ne insegnano almeno due, gli ordinari almeno tre;
7. ogni facoltà deve essere dislocata almeno in una sede;
8. uno studente può inserire nel proprio piano di studi solo corsi offerti dal proprio corso di laurea.

```
create table facoltà (
  nome      varchar(30) primary key
  preside   char(16)
            foreign key references docente(codice)
            on update cascade on delete set null,
  -- regola 1
  check (nome = (select docente.facoltà
                 from docente
                 where docente.codice = preside)),
  -- regola 7
  check (nome in (select dislocazione.facoltà from dislocazione)),

)


create table sede (
  indirizzo   varchar(30) primary key,
  telefono    varchar(10)
)


create table dislocazione (
  facoltà      varchar(30) foreign key references facoltà(nome)
               on update cascade on delete cascade,
  sede         varchar(30) foreign key references sede(indirizzo)
               on update cascade on delete cascade,
  primary key (facoltà, sede)
)


create table cdl (
  nome               varchar(30) primary key,
  descrizione        CLOB,
  presidente         char(16)
                     foreign key references docente(codice)
                     on update cascade on delete set null,
```

```
  facoltà            varchar(30)
                     foreign key references facoltà(nome)
                     on update cascade on delete set null,
  -- regola 2
  check (facoltà = (select docente.facoltà
                    from docente
                    where docente.codice = presidente))
)


create table studente (
  matricola                 char(8) primary key,
  nome                      varchar(20) not null,
  cognome                   varchar(20) not null,
  dataImmatricolazione      date not null,
  cdl                       varchar(30)
                            foreign key references cdl(nome)
                            on update cascade on delete set null
)

create table corso (
  nome             varchar(30),
  cdl              varchar(30) foreign key references cdl(nome)
                   on update cascade on delete cascade,
  descrizione      CLOB,
  programma        CLOB,
  anno             smallint,
  docente          char(16)
                   -- regola 5
                   foreign key references docente(codice)
                   on update cascade on delete set null,
  primary key (nome, cdl)
)


create table mutuo (
  corso varchar(30),
  cdl varchar(30),
  corsoMutuo not null,
  cdlMutuo not null,
  primary key (corso, cdl),
  foreign key (corso, cdl) references corso(nome, cdl)
  on update cascade on delete cascade,
  foreign key (corsoMutuo, cdlMutuo) references corso(nome, cdl)
  on update cascade on delete cascade,
  -- regola 3
  check (cdl <> cdlMutuo)
)


create table pianoDiStudi (
  studente    char(8) foreign key references studente(matricola)
                      on update cascade on delete cascade,
  corso       varchar(30),
  cdl         varchar(30),
  foreign key (corso, cdl) references corso(nome, cdl)
  on update cascade on delete cascade,
  primary key (studente, corso, cdl),
  -- regola 8
  check ((corso, cdl) in (select corso.nome, corso.cdl
                          from corso, studente
                          where (corso.cdl = studente.cdl) and
                                (studente.matricola = studente)))
)
```

```
create domain tipoGiorno as varchar(9)
check (tipoGiorno in ("Lunedì", "Martedì", "Mercoledì",
                      "Giovedì", "Venerdì", "Sabato"))


create domain tipoFascia as char(5)
check (tipoGiorno in ("08-10", "10-12", "14-16", "16-18"))


create table lezione (
  giorno      tipoGiorno,
  fascia      tipoFascia,
  aula        varchar(3),
  corso       varchar(30),
  cdl         varchar(30),
  primary key (giorno, fascia, aula),
  foreign key (corso, cdl) references corso(nome, cdl)
  on update cascade on delete set null
)


-- regola 4
create assertion collisione check (not exists(

  select distinct l1.corso, l1.cdl
  from corso c1, lezione l1, corso c2, lezione l2
  where (c1.nome = l1.corso) and
        (c1.cdl = l1.cdl) and
        (c2.nome = l2.corso) and
        (c2.cdl = l2.cdl) and
        (c1.anno = c2.anno) and
        (l1.cdl = l2.cdl) and
        (l1.giorno = l2.giorno) and
        (l1.fascia = l2.fascia) and
        (l1.aula <> l2.aula)

))

create table docente (
  codice       char(16) primary key,
  tipo         char(1),
  nome         varchar(20) not null,
  cognome      varchar(20) not null,
  email        varchar(20),
  sito         varchar(20),
  telefono     varchar(20),
  ufficio      varchar(5),
  facoltà      varchar(30)
               foreign key references facoltà(nome)
               on update cascade on delete set null,
  check (tipo in ("R", "A", "O"))
)


-- regola 6 (ricercatori)
create assertion DidatticaRicercatori check (not exists(

  select codice
  from docente
  where (tipo = "R") and
        (codice in (select corso.docente from corso))
```

```
))


-- regola 6 (associati)
create assertion DidatticaAssociati check (not exists(

  select docente.codice
  from docente left join corso on (docente.codice = corso.docente)
  where (docente.tipo = "A")
  group by docente.codice
  having count(corso.nome) < 2

))


-- regola 6 (ordinari)
create assertion DidatticaOrdinari check (not exists(

  select docente.codice
  from docente left join corso on (docente.codice = corso.docente)
  where (docente.tipo = "O")
  group by docente.codice
  having count(corso.nome) < 3

))
```

Viste

1. Definire una vista permanente che mostra i seguenti dati per ogni studente: matricola, nome, cognome, corso di laurea, facoltà, anno di iscrizione (ottenuto come differenza tra anno corrente e anno di immatricolazione).

```
create view vistaStudente(matricola, nome, cognome, corso, facoltà, anno) as
select studente.matricola, studente.nome, studente.cognome, studente.cdl,
       cdl.facoltà, (year(current_date) - year(studente.dataImmatricolazione) + 1)
from studente join cdl on (studente.cdl = cdl.nome)
```

2. Definire una vista permanente `iscritti` che mostra, per ogni corso di laurea, la facoltà e il numero di iscritti (anche per i corsi con nessun iscritto).

```
create view iscritti(corso, facoltà, numero) as
select cdl.nome, cdl.facoltà, count(*)
from cdl left join studente on (cdl.nome = studente.cdl)
 group by cdl.nome, cdl.facoltà
```

3. Scrivere le seguenti interrogazioni usando la vista `iscritti` definita sopra:
   - Il corso di laurea con il massimo numero di studenti

```
   select corso
   from iscritti
   where numero = (select max(numero) from iscritti)
```

   - Tutti i corsi di laurea con un numero di studenti superiore alla media

```
   select corso
   from iscritti
   where numero > (select avg(numero) from iscritti)
```

4. Usando le viste ricorsive, formulare la seguente interrogazione: tutti i corsi che mutuano, direttamente o indirettamente, sul corso di *Basi di Dati* presso *Tecnologie Web e Multimediali*

```
 with recursive mutuo*(corso, cdl, corsoMutuo, cdlMutuo) as (
   select corso, cdl, corsoMutuo, cdlMutuo
   from mutuo
     union
   select m1.corso, m1.cdl, m2.corsoMutuo, m2.cdlMutuo
```

```
24.    from mutuo m1 join mutuo* m2 on (m1.corsoMutuo = m2.corso) and
25.                                    (m1.cdlMutuo = m2.cdl)
26.  )
27.
28.  select corso, cdl
29.  from mutuo*
30.  where (corsoMutuo = 'Basi di Dati') and
31.        (cdlMutuo = 'Tecnologie Web e Multimediali')
32.
```

Trigger

Si vuole creare un insieme di trigger per calcolare la media dei voti degli studenti. A tal fine:

1. aggiungere allo schema una tabella esame che registra i voti degli studenti per i corsi superati (i voti sono interi dal 18 al 30, possibilmente con lode per i 30);

```
2. create table esame
3.   (
4.     studente         char(8),
5.     corso            varchar(30),
6.     cdl              varchar(30)
7.     data             date not null,
8.     voto             smallint not null,
9.     lode             char(2) not null default 'no',
10.      primary key      (studente, corso, cdl),
11.      foreign key (studente) references studente(matricola)
12.        on update cascade on delete cascade,
13.      foreign key (corso, cdl) references corso(nome, cdl)
14.        on update cascade on delete cascade,
15.      check ((voto >= 18) and (voto <= 30)),
16.      check (lode in ("si", "no")),
17.      check ((lode = "no") or (voto = 30))
18.    )
```

19. aggiungere alla tabella studente un campo media per la media dei voti degli esami superati dallo studente;

```
20.  alter table studente
21.  add column media smallint;
```

22. scrivere una interrogazione di modifica che aggiorna il campo media aggiunto alla tabella studente;

```
23.  update studente
24.  set media = (select avg(voto)
25.               from esame
26.               where matricola = esame.studente);
```

27. definire un insieme di trigger per calcolare il campo media.

```
28.  create trigger updateAvg
29.  after update of voto on esame
30.  for each row
31.  when new.voto <> old.voto
32.  update studente
33.  set media = (select avg(voto)
34.               from esame
35.               where matricola = esame.studente)
36.  where matricola = new.studente;
37.
38.
39.  create trigger deleteAvg
40.  after delete on esame
41.  for each row
42.  update studente
43.  set media = (select avg(voto)
```

```
44.               from esame
45.               where matricola = esame.studente)
46.  where matricola = old.studente;
47.
48.
49.  create trigger insertAvg
50.  after insert on esame
51.  for each row
52.  update studente
53.  set media = (select avg(voto)
54.               from esame
55.               where matricola = esame.studente)
56.  where matricola = new.studente;
```

MySQL

Realizzare la base di dati universitaria usando MySQL. In particolare, decidere le strutture di memorizzazione primarie e ausiliarie usando gli strumenti offerti da MySQL.