Appunti di Basi di dati

Giacomo Simonetto

Primo semestre 2025-26

Sommario

Appunti del corso di Basi di dati della facoltà di Ingegneria Informatica dell'Università di Padova.

Indice

1 Introduzione

Dato vs informazione

- il dato è la registrazione di un veneto effettuata attraverso dei simboli e salvata su un supporto, in un dato puro non c'è nessun contesto per interpretarlo
- l'informazione è il dato con il contesto associato, ovvero è un dato interpretabile

Dati strutturati vs non strutturati

- i dati strutturati sono dati che possono facilmente ed efficientemente essere organizzati in tabelle, sono ad esempio numeri, date, stringhe, ecc., sono facili da memorizzare ed analizzare
- i dati non strutturati sono dati che non possono essere facilmente organizzati in tabelle, sono ad esempio immagini, video, audio, documenti di testo, ecc., richiedono più spazio per poter essere memorizzati e serve un processo di strutturazione per poterli analizzare

Basi di dati

Le basi di dati o database sono collezioni organizzate e permanenti di dati strutturati coerenti che permettono di memorizzare, gestire e recuperare informazioni (dati e contesto) in modo efficiente. Vengono realizzati su misura per rispondere alle esigenze di specifiche applicazioni e utenti. Modellano un piccolo aspetto del mondo reale, ovvero rappresentano un miniworld. I database sono:

- centralizzati (i dati sono memorizzati su un unico server) o distribuiti (i dati sono memorizzati su più server collegati in rete)
- persistenti: i dati rimangono memorizzati anche quando il sistema viene spegnimento
- condivisi: più utenti e applicazioni possono accedere e manipolare i dati contemporaneamente

DataBase Managment System (DBMS)

Il DBMS sono software che permetton di gestire e interagire con i dati all'interno di un database. Garantiscono:

- integrità: mantengono i dati corretti e coerenti secondo le regole definite
- sicurezza: proteggono i dati da accessi non autorizzati
- efficienza: ottimizzano l'archiviazione e il recupero dei dati

Oltre ai dati veri e propri, i DBMS memorizzano anche metadati, ovvero informazioni testuali su come interpretare e organizzare i dati. I DBMS di database relazionali vengono anche detti RDBMS (Relational DataBase Management System).

Progettazione di un database

La progettazione di un database si divide in tre fasi:

- **progettazione concettuale**: si crea un modello concettuale o modello ER (Entity-Relationship) che rappresenta il miniworld di interesse in modo astratto, attraverso entità, relazioni e attributi
- **progettazione logica**: si trasforma il modello concettuale in un modello logico, come il modello relazionale, che specifica come i dati saranno organizzati all'interno del database
- **progettazione fisica**: si definisce come i dati saranno effettivamente memorizzati e gestiti nel DBMS, attraverso un effettivo linguaggio di definizione e manipolazione dei dati, come SQL

2 Modello ER

2.1 Struttura generale del modello ER

Il modello ER (Entity-Relationship) è un modello concettuale utilizzato per rappresentare in modo astratto il miniworld di interesse. Si basa su tre concetti principali:

- **entità**: oggetti o concetti del mondo reale che hanno un'identità distinta e possono essere rappresentati nel database (es. studente, corso, libro, ecc.)
- **associazioni**: associazioni tra entità che rappresentano come le entità interagiscono tra di loro (es. uno studente si iscrive a un corso, un libro è scritto da un autore, ecc.)
- **attributi**: proprietà, descrizioni o caratteristiche delle entità o delle associazioni (es. nome, cognome, data di iscrizione, data di scrittura, ecc.)

2.2 Entità

L'entità rappresenta un oggetto o concetto del mondo reale che ha un'identità distinta. Ogni entità è rappresentata da un rettangolo nel modello ER e può avere uno o più attributi associati. Ogni entità è identificata da uno o più attributi univoci detti identificatori. Le entità possono essere:

- **entità forti**: entità che possono essere identificate in modo univoco dai loro attributi
- **entità deboli**: entità che non possono essere identificate in modo univoco dai loro attributi e dipendono dall'esistenza di un'altra entità forte per la loro identificazione (approfondite in seguito)

2.3 Attributi

Gli attributi sono proprietà, descrizioni o caratteristiche delle entità o delle associazioni a cui appartengono. Si indicano con un pallino (pieno o vuoto) collegato all'entità o all'associazione di appartenenza. In base alla loro struttura gli attributi si classificano in:

- **attributi semplici**: attributi che non possono essere suddivisi (es. nome, cognome, badge, ecc.)
- **attributi composti**: attributi che possono essere suddivisi in più componenti (es. l'indirizzo può essere suddiviso in via, città, CAP, ecc.) e si indicano con un pallino vuoto più grande da cui si ramificao i componenti dell'attributo con altri pallini vuoti più piccoli

Ogni attributo ha una propria **cardinalità**, ovvero il numero di valori che può assumere per ogni entità o associazione. La cardinalità si indica con una coppia di numeri (minimo, massimo) accanto al pallino dell'attributo (es. (0,1), (1,1), (0,N), (1,N), ecc.). Se non viene indicata, la cardinalità di default è (1,1). In base alla cardinalità, gli attributi si classificano in:

- **attributi obbligatori**: attributi che devono sempre avere un valore (es. nome, cognome), hanno cardinalità minima pari a 1 (es. (1,1), (1,N))
- **attributi opzionali**: attributi che possono non avere un valore (es. secondo nome, numero di fax), hanno cardinalità minima pari a 0 (es. (0,1), (0,N))
- **attributi singoli**: attributi che possono assumere un solo valore per ogni entità (es. data di nascita) hanno cardinalità massima pari a 1 (es. (0,1), (1,1))
- **attributi multipli**: attributi che possono assumere più valori per ogni entità (es. numeri di telefono) hanno cardinalità massima pari a N (es. (0,N), (1,N))

Gli attributi che identificano l'entità sono detti **identificatori** e si indicano con un pallino pieno. Gli attributi identificatori sono per definizione semplici, obbligatori e singoli, con cardinalità (1,1). Se esistono più identificatori indipendenti per la stessa entità, si indicano con più pallini pieni. Se l'identificazione di una entità dipende da più attributi, tutti gli attibuti che formano l'identificatore hanno pallino pieno e sono collegati tra di loro tramite una linea orizzontale che termina sempre con un pallino pieno. Nota importante: le relazioni non possono avere attibuti identificativi.

Nota: prima di passare allo schema relazionale è necessario ristrutturare lo schema ER **trasformando attributi composti** in attributi semplici, creando una nuova entità oppure spostando tutti gli attributi componenti nell'entità madre come attributi semplici.

2.4 Relazioni

Le relazioni rappresentano associazioni tra entità che mostrano come le entità interagiscono tra di loro. Ogni relazione è rappresentata da un rombo nel modello ER e può avere uno o più attributi associati. Le relazioni possono essere:

- **relazioni unarie o ricorsive**: relazioni che coinvolgono una sola entità (es. un dipendente gestisce un altro dipendente o un sovrano succede a un altro sovrano, approfondite in seguito)
- **relazioni binarie**: relazioni che coinvolgono due entità (es. uno studente si iscrive a un corso)
- **relazioni ternarie**: relazioni che coinvolgono tre entità (es. un medico prescrive un farmaco a un paziente, approfondite in seguito)
- **relazioni n-arie**: relazioni che coinvolgono n entità (poco usate, approfondite in seguito)

Ogni entità coinvolta alla relazione ha una propria **cardinalità**, ovvero il numero di volte che un'entità può partecipare alla relazione. La cardinalità si indica con una coppia di numeri (minimo, massimo) nel collegamento tra l'entità e la relazione, accanto all'entità coinvolta (es. (0,1), (1,1), (0,N), (1,N), ecc.). È sempre necessario indicare la cardinalità e non esiste una cardinalità di default.

È possibile che esistano **più relazioni binarie tra le stesse due entità**, ad esempio città di residenza e città di lavoro tra le entità lavoratore e città. In questo caso è necessario fare attenzione ai vincoli di esistenza delle relazioni, imposti dalle cardinalità minime delle relazioni.

Relazioni unarie o ricorsive

Le relazioni unarie o ricorsive coinvolgono una sola entità. Possono essere:

- **simmetriche**: relazioni in cui il ruolo delle entità coinvolte è identico (es. un amico di un amico è un amico) non è necessario indicare il verso di lettura della relazione e l'ordine delle coppie *(entità; entità)* è indifferente
- **non simmetriche**: relazioni in cui il ruolo delle entità coinvolte è diverso (es. un capo dirige un altro dipendente), è necessario indicare il verso di lettura della relazione con una freccia sul collegamento tra l'entità e la relazione e l'ordine delle coppie *(entità; entità)* è importante

Si osserva che per le relazioni simmetriche è sufficiente salvare solo le coppie "dritte", perché quelle invertite si possono dedurre. Salvare sia le coppie "dritte" che quelle invertite di relazioni simmetriche provocherebbe ridondanza di dati. Tale ridondanza può tornare utile per facilitare alcune query di ricerca, in quanto è sufficiente cercare su una sola colonna.

Relazioni ternarie

Le relazioni ternarie sono relazioni che coinvolgono tre entità che partecipano contemporaneamente e sono sempre coinvolte insieme. Alcuni esempi di relazione ternaria sono la relazione "prescrive" tra le entità medico, paziente e farmaco; la relazione verbale tra le entità vigile, multa e veicolo; la relazione di vendita tra le entità cliente, prodotto e negozio.

Le relazioni ternarie in cui un'entità partecipa con cardinalità (1,1) possono essere scomposte in due relazioni binarie in cui l'entità (1,1) funge da collegamento tra le altre due. Ha senso usare le ternarie se le tre le entità sono sempre coinvolte contemporaneamente, insieme e sono logicamente legate tra di loro. Ad esempio la relazione "esposizione" tra le entità opera d'arte, museo e artista non ha senso che venga modellata come relazione ternaria, in quanto l'artista non è sempre legato al museo e la relazione opera-artista avviene in un istante di tempo separato rispetto alla relazione opera-museo.

2.5 Regole di vincolo

Spesso esistono alcuni aspetti del miniworld che non possono essere rappresentati direttamente nel modello ER, oppure alcune volte si effettuano delle scelte semplificative per facilitare la progettazione del database. Per rappresentare questi aspetti si utilizzano delle regole di vincolo esterne indicate alla fine del modello ER come testo e precedute dalla scritta `RV1`, `RV2`, ...

2.6 Entità deboli

Le entità deboli sono entità che non possono essere identificate in modo univoco dai loro attributi e dipendono dall'esistenza di un'altra entità forte per la loro identificazione. Le entità deboli sono rappresentate come normali entità forti, ma il loro insieme di attributi identificatori include anche la relazione con l'entità forte da cui dipendono. Un esempio di entità debole è la entità "posto" dipendente dall'entità "sala" dipendente dall'entità "cinema". È possibile evitare di creare entità deboli aggiungendo un identificatore ausiliario, ma ciò comporterebbe la necessità di gestire l'assegnazione di tali identificatori ausiliari e si prederebbe l'informazione che l'entità dipende da un'altra entità.

La partecipazione delle entità deboli alla relazione con l'entità forte da cui dipendono ha sempre cardinalità (1,1), come negli attibuti identificatori, e va sempre indicata esplicitamente.

2.7 Pattern di progetto

Attributo a valori finiti

Un attributo a valori finiti è un attributo il cui valore viene scelto da un insieme limitato di valori possibili (es. colore dell'auto). Per rappresentare un attributo a valori finiti si possono utilizzare due strategie:

- modellarlo come entità: si crea una nuova entità che rappresenta l'insieme dei valori possibili dell'attributo e si collega tale entità all'entità originale tramite una relazione; questa strategia è utile quando i valori possono cambiare frequentemente nel tempo
- modellarlo come enumeratore: si crea un enumeratore che rappresenta l'insieme dei valori possibili dell'attributo e si collega tale enumeratore all'entità originale tramite l'attributo; questa strategia è utile quando i valori sono pochi e rimangono molto stabili nel tempo (modificare un enumeratore richiede di modificare lo schema del database)

Un caso particolare è quando l'**attributo a valori finiti** che viene promosso a entità **fa parte di una relazione**: si forma una relazione ternaria tra le due entità originali e il nuovo attributo entità. Ad esempio "musicista" "partecipa" ad una "orchestra" suonando uno "strumento", in cui lo "strumento" diventa entità e la relazione "partecipa" diventa ternaria. In questo caso, però, si perde l'unicità della relazione binaria originale, in quanto la stessa coppia di entità può essere collegata a più valori diversi dell'attributo promosso a entità. Nell'esempio sopra, un musicista può partecipare a una orchestra suonando più strumenti diversi. Per risolvere questo problema, viene modellata la ternaria come entità debole che dipende dalle due entità originali ed ha una relazione (1,1) con l'attributo promosso a entità. Nell'esempio sopra, si crea l'entità debole "partecipazione" che dipende dalle entità "musicista" e "orchestra" ed ha una relazione (1,1) con l'entità "strumento".

Pattern "PARTE DI"

Il pattern "PARTE DI" viene utilizzato per rappresentare entità che sono composte da più parti o componenti modellati a loro volta come entità separate. Ad esempio il cinema è composto da più sale, ogni sala è composta da più posti. Per modellare le varie parti si utilizzano entità deboli identificate dall'entità forte di cui fanno parte.

Pattern "ISTANZA DI"

Il pattern "ISTANZA DI" viene utilizzato per rappresentare entità che sono istanze di una categoria o tipo più generale. Ad esempio "edizione" è un'istanza di "torneo", "volo reale" è un'istanza di "volo". L'entità istanza viene modellata come entità debole identificata dall'entità più generale di cui è istanza.

Relazione temporale

La relazione temporale viene utilizzata per rappresentare relazioni ripetute tra le stesse entità in diversi istanti di tempo. Ad esempio la relazione "esame" tra le entità "studente" e "corso". Per modellare la relazione temporale si promuove la relazione in entità debole identificata dalle entità coinvolte nella relazione e dall'attributo "data".

2.8 Generalizzazioni e ristrutturazioni - Extended ER

Generalizzazioni e specializzazoni

Per rappresentare organizzazioni gerarchiche tra entità si utilizzano le generalizzazioni e specializzazioni previste nel modello esteso dell'ER (Extended ER). La relazione di generalizzazione/specializzazione viene rappresentata con una freccia che parte dalle entità figlie (specializzazioni) e punta verso l'entità madre (generalizzazione). Esistono vari tipi di specializzazioni:

- **specializzazioni totali**: specializzazioni in cui ogni istanza dell'entità madre deve appartenere ad almeno una delle entità figlie; si indica riempiendo/colorando l'interno della freccia di generalizzazione/specializzazione
- **specializzazioni parziali**: specializzazioni in cui alcune istanze dell'entità madre possono non appartenere a nessuna delle entità figlie, ovvero rimangono generali; si indica lasciando la freccia di generalizzazione/specializzazione vuota all'interno
- **specializzazioni disgiunte**: specializzazioni in cui un'istanza dell'entità madre può appartenere a una sola delle entità figlie; si indica con una regola di vincolo
- **specializzazioni sovrapposte**: specializzazioni in cui un'istanza dell'entità madre può appartenere a più entità figlie contemporaneamente; si indica sempre con una regola di vincolo

Le entità specializzate non possiedono identificatori propri, ma ereditano (e condividono) l'identificatore dall'entità madre.

Ristrutturazioni

Prima di passare al modello logico/relazionale è necessario eliminare le strutture gerarchiche create con le generalizzazioni attraverso processi di ristrutturazione. Esistono tre tipi di ristrutturazioni:

- **ristrutturazione verso l'alto**: si fanno collassare le entità figlie nell'entità madre: tutti i loro attributi e relazioni diventano facoltativi e si spostano nell'entità madre; si aggiunge l'attributo "tipo" per distinguere le varie specializzazioni e si aggiungono regole di vincolo per indicare quali attributi e relazioni sono obbligatori in base al tipo; risulta conveniente quando le entità figlie hanno pochi attributi e relazioni propri che le differenziano tra di loro e, collassandoli come opzionali verso l'alto, si ha quindi poca proliferazione di `NULL` values; la presenza di tanti attributi potrebbe generare ritardi nelle query a causa della grandezza dell'unica tabella risultante, ma allo stesso tempo ne semplifica la scrittura in quanto non è necessario fare join tra più tabelle
- **ristrutturazione verso il basso**: si fa collassare l'entità madre verso le entità figlie: si duplicano i suoi attributi e relazioni in tutte le entità figlie; è necessario fare attenzione agli identificatori dell'entità madre che ora diventano condivisi tra le entità figlie; risulta conveniente quando le entità figlie hanno molti attributi e relazioni propri che le differenziano tra di loro e, tenendole separate, si riducono i `NULL` values; la frammentazione dei dati in più tabelle più piccole velocizza le query, ma ne complica la scrittura in quanto è necessario unire i risultati di più tabelle
- **ristrutturazione mista**: si trasformano le entità figlie in entità deboli che dipendono dall'entità madre, senza alterare attributi e relazioni; in questo caso si hanno un po' entrambi i vantaggi e li svantaggi dei casi precedenti come poca proliferazine di `NULL` values, query veloci da eseguire e maggiore semplificazione nella scrittura delle query, ma si ha un grande aumento di spazio occupato perché le entità figlie duplicano al loro interno tutti gli attributi identificatori dell'entità madre

In generale, per facilitare la scrittura delle query all'esame, è fortemente consigliato utilizzare la ristrutturazione verso l'alto in quanto la performance del database e la proliferazione di `NULL` values vengono trascurate e passano in secondo piano.

2.9 Regole e consigli per progettare un buon modello ER

- usare **nomi** significativi e coerenti per entità e attributi, meglio usare nomi di attributi diversi per ogni entità in modo da facilitare la scrittura del modello relazionale e delle query
- le relazioni non hanno verso di lettura, per cui si assume come convenzione di indicare le relazioni con **verbi** alla terza persona singolare in forma attiva (es. "contiene", "iscrive", "possiede", ecc.)

- quando un attributo diventa troppo complesso per essere rappresentato come attributo semplice/-composto, è probabile che debba essere **promosso a entità** separata, viceversa quando un'entità ha pochi attributi e non ha relazioni significative con molte altre entità, è probabile che debba essere **declassata ad attributo**
- la **scelta degli identificatori** deve essere fatta con attenzione, è preferibile usare un attributo della stessa entità come identificatore (es. badge, codice fiscale, numero di serie, ecc.) piuttosto che creare un attributo extra ad hoc (es. ID); l'utilizzo di attributi ID extra può provocare la presenza di più entità che rappresentano lo stesso oggetto reale, ma hanno codici identificativi diversi
- è bene **indicare sempre tutti gli attributi identificatori** dell'entità, in quanto nelle implementazioni software (es. SQL) esistono delle strutture ausiliarie che facilitano l'accesso al database in maniera efficiente attraverso più serie di identificatori (chiavi primarie, chiavi candidate, ecc.)
- allo scopo di superare l'esame nel modo più semplice possibile, è consigliabile di semplificare il più possibile il miniworld di interesse effettuando opportune **scelte semplificative**, trascurando più aspetti e dettagli possibili che esistono nel mondo reale ma che non sono indicati nel testo dell'esame
- è un errore salvare **l'età come attributo** di una entità (persona, animale, ecc.), in quanto tale attributo richiede un aggiornamento continuo nel tempo, mentre è preferibile salvare la data di nascita e calcolare l'età al momento opportuno
- non ha senso memorizzare **dati duplicati** (es. età e data di nascita), in quanto si rischia di avere dati incoerenti se uno dei due viene aggiornato e l'altro no, a meno che non vengano utilizzati per ottimizzare le performance di esecuzione delle query più frequenti
- la scelta di posizionare **attributi nelle relazioni** dipende dal fatto se essi sono riferiti al rapporto tra le entità (descrivono la relazione) e non alle singole entità coinvolte, quando una delle due entità partecipa con cardinalità (1,1) allora l'attributo può essere spostato indifferentemente tra la relazione e l'entità con cardinalità (1,1)

2.10 Analisi della qualità degli ER

Introduzione

L'analisi della qualità del modello ER serve per capire le performance del database sotto l'aspetto dello spazio ocupato e del costo delle query. Per fare ciò si considera una stima/previsione in prospettiva futura del numero di istanze di ogni entità e le query che verranno eseguite più frequentemente sul database.

Analisi dello spazio occupato

Si costruisce una tabella in cui la prima colonna contiene il nome di ogni entità o relazione, la seconda ne indica il tipo (entità o relazione), la terza indica il numero di istanze previste. Per alcune entità/relazioni il numero è fornito dal committente in funzione della destinazione d'uso finale del database, in altri casi è possibile calcolarlo in funzione delle cardinalità delle entità coinvolte.

Analisi del costo delle query

Le query più comuni vengono fornite dal committente in base alla destinazione d'uso finale del database. Le operazioni da svolgere si classificano in:

- **interactive**: se forniscono un valore da cercare all'interno di una tabella
- **batch**: se vengono eseguite su tutti i valori di una certa tabella

Per ogni operazione da analizzare si crea una tabella in cui nella prima colonna si indica il nome dell'operazione, nella seconda il tipo (interactive o batch), nella terza una stima del numero di esecuzioni previste al giorno. Per ogni operazione si calcola il costo in termini di letture e scritture su disco necessarie per eseguire l'operazione; tale numero poi viene moltiplicato per il numero di esecuzioni previste al giorno per ottenere il costo totale giornaliero dell'operazione.

In questi casi è facile notare come la presenza di attributi ridondanti alcune volte riduce notevolmente (anche di 10 volte), il costo delle query più freuenti, a discapito però dello spazio occupato che aumenta leggermente e del rischio di avere dati incoerenti. È necessario quindi valutare se vale la pena prendersi il rischio di incoerenza a vantaggio di un miglioramento delle performance del database.

3 Modello logico - relazionale

3.1 Struttura del modello relazionale

Introduzione

Il modello relazionale è stato proposto da E.F. Codd nel 1970 come modello per rappresentare le relazioni di dipendenza e indipendenza tra i dati in un database. Si basa sul concetto matematico di relazione, rappresentata attraverso l'utilizzo di tabelle bidimensionali. Il **modello si basa sui valori**, ovvero i collegamenti e i riferimenti tra le varie relazioni vengono effettuati attraverso comparazioni tra i valori degli attributi delle tabelle.

Relazione matematica

Dati n insiemi di valori D_1, D_2, ..., D_n, una relazione matematica R è un sottoinsieme del prodotto artesiano ottenuto dalle tuple ordinate $(d_1, d_2, \ldots, d_n)$ dove $d_1 \in D_1$, $d_2 \in D_2$, ..., $d_n \in D_n$.

$$R \subseteq D_1 \times D_2 \times \cdots \times D_n = \{(d_1, d_2, \ldots, d_n) \mid d_1 \in D_1, d_2 \in D_2, \ldots, d_n \in D_n\}$$

Le relazioni matematiche hanno le seguenti proprietà:

- il numero di domini n è detto grado della relazione
- il numero di tuple della relazione $|R|$ è detto cardinalità della relazione.
- siccome la relazione (matematica) è definita come un insieme, dalla definizione di insieme si ottene che non c'è ordine tra le tuple della relazione e non si ammettono tuple duplicate
- le relazioni matematiche si dicono strutture posizionali in quanto l'ordine dei domini D_1, D_2, ..., D_n (e dei valori nelle tuple) è importante per definire il significato della tupla e va sempre rispettato

Relazione del modello relazionale

A differenza delle relazioni matematiche, le relazioni del modello relazionale vengono modellate come tabelle e possiedono alcune proprietà aggiuntive:

- i componenti della relazione si chiamano attributi e ciascuno possiede un nome univoco (indicato nel table header) all'interno della relazione e un insieme di valori detti domini degli attributi (D_1, D_2, ..., D_n), nella rappresentazione tabellare gli attributi corrispondono alle colonne della tabella
- le tuple della relazione sono insiemi ordinati di valori e sono rappresentate come righe della tabella;
- siccome ogni attributo possiede un nome univoco, l'ordine con cui gli attributi sono definiti nella relazione non è importante, per cui le relazioni "tabellari" non sono strutture posizionali; fissato un ordine, però, tale ordine deve comunque essere rispettato da tutte le tuple della relazione per gli stessi motivi sopracitati

Si definisce quindi una relazione su un insieme di attributi $X = \{A_1, A_2, \ldots, A_n\}$ come un insieme di tuple t_j sullo stesso insieme di attributi X per cui esistono:

- la funzione dom che mappa ogni attributo $A_i \in X$ al suo dominio di valori $D_i \in D$
- la funzione t_j che mappa ogni attributo $A_i \in X$ ad un valore $t_j[A_i] \in \text{dom}(A)$

$$R(X) := \{\text{dom}, t_j\} \qquad \text{con} \qquad \begin{array}{ll} \text{dom}\colon X \to D & t_j\colon X \to D_i \\ A_i \mapsto D_i & A_i \mapsto v_{i,j} \in D_i \end{array}$$

Schema relazionale

Lo schema relazionale $R(T)$ di una relazione R formato da un nome della relazione "R" e da un tipo della relazione "T" definito come l'insieme delle associazioni tra gli attributi e i loro domini:

$$R(T) := \{R, T\} \qquad T := \{A_1 : D_1, A_2 : D_2, \ldots, A_n : D_n\}$$

$$R(T) = R(A_1 : D_1,\ A_2 : D_2,\ \ldots,\ A_n : D_n) = R(A_1, A_2, \ldots, A_n)$$

La cardinalità di uno schema relazionale $R(T)$ è il numero di attributi che lo compongono $|T| = n$. Due tipi di relazioni T_1 e T_2 sono detti uguali se e solo se hanno le stesse associazioni tra attributi e domini, indipendentemente dall'ordine con cui sono elencate.

Istanza relazionale

La istanza relazionale $r(R)$ di una relazione $R(T)$ è un insieme di tuple $t_1, t_2, \ldots, t_m$ definite come associazioni tra gli attributi e i loro valori:

$$r(R) := \{t_1, t_2, \ldots, t_m\} \qquad t_j := \{A_1 : v_{1,j}, A_2 : v_{2,j}, \ldots, A_n : v_{n,j}\} \qquad t_j[A_i] = v_{i,j} \in D_i = \text{dom}A_i$$

La cardinalità di una istanza relazionale $r(R)$ è il numero di tuple che la compongono $|r(R)| = m$.

Significato dei valori `NULL`

Quando non è possibile assegnare un valore ad un attributo di una tupla, si utilizza il valore speciale `NULL` per indicare l'assenza di valore. Per cui la definizione di tupla viene modificata come segue:

$$t_j[A_i] = v_{i,j} \in D_i \vee \{\texttt{NULL}\}$$

Il valore `NULL` è diverso da qualsiasi valore presente in qualsiasi dominio ed è diverso da un altro valore `NULL`. Ciò significa che non è possibile effettuare confronti di uguaglianza o disuguaglianza con il valore `NULL`, in quanto non è noto se due valori `NULL` siano uguali o diversi tra di loro. I valori `NULL` possono significare:

- valore indefinito: il valore non esiste ancora per tale attributo (es. data di consegna di un ordine non ancora spedito)
- valore non disponibile: il valore esiste ma non è noto (es. età di una persona di cui non si conosce la data di nascita)
- valore sconosciuto: il valore esiste ma non è noto (es. numero di telefono di una persona che non vuole rivelarlo)

Modello basato sui valori

Il modello relazionale si basa sui valori, ovvero i collegamenti e i riferimenti tra le varie relazioni vengono effettuati attraverso comparazioni tra i valori degli attributi delle tabelle. Ad esempio per collegare due tabelle "`Studente`" e "`Corso`" si utilizza l'attributo "`id_corso`" presente in entrambe le tabelle si uniscono le tuple delle due tabelle che condividono lo stesso valore di "`id_corso`".

I vantaggi di utilizzare un modello basato sui valori sono:

- indipendenza dalla implementazione fisica del database: non è necessario creare collegamenti fisici tra le tabelle, basta confrontare i valori degli attributi
- portabilità dei dati tra sistemi diversi: i dati sono portabili tra due sistemi diversi che implementano il modello relazionale, in quanto i collegamenti sono rappresentati tra valori degli attributi e non dall'implementazione fisica del database
- i collegamenti sono bidirezionali: è possibile navigare tra le tabelle in entrambe le direzioni

Schema del database

Lo schema del database è un insieme di schemi relazionali $R_1(T_1)$, $R_2(T_2)$, ..., $R_n(T_n)$ con nomi diversi

$$\mathbf{R} := \{R_1(T_1), R_2(T_2), \ldots, R_n(T_n)\}$$

Istanza del database

L'istanza del database è un insieme di istanze relazionali $r_1(R_1)$, $r_2(R_2)$, ..., $r_n(R_n)$ corrispondenti agli schemi relazionali dello schema del database:

$$\mathbf{r} := \{r_1(R_1), r_2(R_2), \ldots, r_n(R_n)\}$$

3.2 Superchiavi, chiavi, chiavi primarie, chiavi esterne e chiavi candidate

Superkey - superchiave

Una superkey (o superchiave) è un insieme di attributi SK di una relazione $R(T)$ che può essere utilizzato per identificare in modo univoco ogni tupla t_i all'interno della relazione.

$$\forall\, t_i, t_j \in r \qquad t_i[SK] \neq t_j[SK]$$

Dalla definizione di insieme si ottiene che ogni relazione deve almeno contenere una superkey che identifica in modo univoco ogni tupla.

Key - chiave

Una key (o chiave primaria) è una superkey minimale, ovvero il minimo insieme di attributi K di una relazione $R(T)$ che permette di identificare in modo univoco ogni tupla t_i all'interno della relazione. Ciò significa che se si rimuove anche un solo attributo da K, l'insieme risultante non è più in grado di identificare in modo univoco ogni tupla.

$$K \text{ is superkey and } \exists\, t_i, t_j \in r \;\mid\; t_i[K' \subset K] = t_j[K' \subset K]$$

Una chiave è sempre superchiave, ma non tutte le superchiavi sono chiavi. Non c'è nessun limite sul numero di chiavi che una relazione può avere.

Primary key - chiave primaria

La primary key (o chiave primaria) PK è una chiave scelta tra le possibili chiavi di una relazione $R(T)$ come identificatore principale delle tuple all'interno della relazione. La chiave primaria, oltre ai vincoli dati dalle definizioni di chiave e superchiave, deve anche soddisfare il vincolo che i suoi attributi non possono assumere il valore `NULL`.

Nel modello relazionale, la chiave primaria si indica sottolineando gli attributi che la compongono.

Foreign key - chiave esterna

La foreign key (o chiave esterna) è un insieme di uno o più attributi $FK = \{A_1, A_2, \dots A_n\}$ in una relazione $R_1(T_1)$ che fa riferimento agli attributi della chiave primaria PK di un'altra relazione $R_2(T_2)$. Deve soddisfare:

$$\text{dom}(A_k) = \text{dom}(B_k) \text{ con } k \in [1, n] \;\; \wedge \;\; \forall\, t_i \in r(R_i),\;\; t_i[FK] = \texttt{NULL} \vee \exists\, t_j \in r(R_j) \;\mid\; t_i[FK] = t_j[PK]$$

La chiave esterna viene utilizzata per stabilire e mantenere i collegamenti tra le due relazioni. Si nota che i collegamenti sono basati sui valori degli attributi e non su riferimenti fisici tra le tabelle.

Nel modello relazionale, la chiave esterna si indica con una freccia che parte dagli attributi della chiave esterna e punta agli attributi della chiave primaria a cui fa riferimento. È molto importante notare che l'ordine degli attributi nella chiave esterna deve essere lo stesso dell'ordine degli attributi nella chiave primaria.

Candidate key - chiave candidata

Una chiave candidata è una chiave che potrebbe essere scelta come chiave primaria di una relazione $R(T)$, ma che non è stata effettivamente scelta come tale. Una relazione può avere più chiavi candidate, ma solo una di esse viene selezionata come chiave primaria. Le chiavi candidate sono importanti perché forniscono alternative per l'identificazione univoca delle tuple all'interno della relazione e in SQL possono essere utilizzate per creare indici secondari per migliorare le performance delle query.

3.3 Integrity constraints e vincoli di chiave

Introduzione

Gli integrity constraints sono delle regole o condizioni definite nello schema del database e che devono essere rispettate dalle istanze del database per garantire la coerenza e l'integrità dei dati memorizzati. Un constraint è definito come un predicato logico che restituisce vero o falso a seconda che l'istanza del database soddisfi o meno la condizione specificata. Si considera un database corretto se e solo se tutti i constraints definiti nello schema del database sono soddisfatti.

Constraint intra-relazionali - (domain, tuple, key)

I constraint intra-relazionali sono vincoli che si applicano all'interno di una singola relazione.

- **tuple constraints**: condizioni che i valori dei vari attributi di una singola tupla devono soddisfare; ad esempio il voto dell'esame deve essere 30 per avere la lode
- **domain constraints**: (è un tuple constraint applicato localmente ai valori del singolo attributo) ogni attributo di una relazione deve assumere valori appartenenti al dominio specificato per tale attributo; ad esempio il voto di un esame deve essere un numero intero compreso tra 18 e 30, oppure il nome di una persona non può contenere essere `NULL`
- **key constraints**: specifica quali attributi costituiscono la chiave primaria di una relazione e che questi devono soddisfare la condizione di unicità per ogni tupla, ovvero non possono esistere due tuple con stessi valori per gli attributi della chiave primaria, non c'è vincolo sul numero di chiavi che una relazione può avere
- **primary key constraint**: come key constraint, ma in più specifica che gli attributi della chiave primaria non possono assumere il valore `NULL`

Constraint inter-relazionali o referential integrity constaint

Il referential integrity constraint specifica quali attributi di una relazione sono chiavi esterne che fanno riferimento agli attributi della chiave primaria di un'altra relazione e a quale relazione fanno riferimento. Questo constraint garantisce la coerenza tra i collegamenti tra le varie tabelle/relazioni e previene la presenza di riferimenti a dati inesistenti.

Ad esempio, se una tabella "`Exam`" contiene un attributo "`id_studente`" che fa riferimento alla chiave primaria "`matricola`" nella tabella "`Studente`", allora ogni valore di "`id_studente`" deve essere presente come valore dell'attributo "`matricola`" di una tupla nella tabella "`Studente`".

3.4 Mapping da ER a relazionale

Introduzione

Il mapping da ER a relazionale consiste nel trasformare il modello ER (ristrutturato) in un modello relazionale seguendo una serie di regole di trasformazione. In genere dopo aver mappato il modello ER in relazionale, si procede con la normalizzazione del modello relazionale per eliminare ridondanze e anomalie. Tale passaggio, però, non è necessario se si è già provveduto a progettare un buon modello ER.

Mappatura delle entità

Ogni entità viene mappata in una relazione con lo stesso nome dell'entità. Gli attributi dell'entità diventano gli attributi della relazione. Gli attributi identificatori dell'entità diventano la chiave primaria della relazione e vengono sottolineati.

Mappatura delle entità deboli

Le entità deboli vengono mappate come le entità forti, ma in più si aggiungono alla relazione gli attributi identificatori dell'entità forte da cui dipendono come chiave primaria dell'entità debole.

Mappatura delle relazioni many to many

Le relazioni many to many (con cardinalità (x,N), (y,N)) vengono mappate come una nuova relazione con lo stesso nome della relazione tra le due entità coinvolte. Gli attributi della relazione diventano gli attributi della nuova relazione. In più si aggiungono alla nuova relazione gli attributi identificatori di entrambe le entità coinvolte come chiavi esterne della nuova relazione. La chiave primaria della nuova relazione è formata dalla combinazione delle chiavi esterne delle due entità.

Mappatura delle relazioni one to many con partecipazione obbligatoria (1,1) - (x,N)

La relazione one to many viene condensata tutta dalla parte dell'entità sul lato one (1,1). Gli attributi della relazione (dell'ER) diventano attributi dell'entità sul lato one (1,1) e si aggiungono gli attributi di chiave esterna riferiti alla chiave primaria dell'entità sul lato many (x,N). La chiave primaria dell'entità sul lato one (1,1) rimane invariata.

Mappatura delle relazioni one to many con partecipazione opzionale (0,1) - (x,N)

In questo caso se si mappasse come relazione one to many con partecipazione obbligatoria, si rischierebbe di avere proliferazione di `NULL` values. In alternativa si può mappare la relazione one to many come una relazione many to many in cui la chiave primaria della nuova relazione è soltanto la chiave primaria dell'entità sul lato one (0,1), ma in questo caso si aumentano le complessità delle query (sarà necessario fare più join). La scelta di quale metodo utilizzare dipende da quanto si è disposti ad accettare la proliferazione di `NULL` values rispetto alla complessità delle query.

Per l'esame è sempre consigliato usare il primo metodo (one to many con partecipazione obbligatoria).

Mappatura delle relazioni one to one con partecipazione obbligatoria (1,1) - (1,1)

La relazione one to one viene condensata tutta da una delle due entità coinvolte (si può scegliere arbitrariamente una delle due, magari preferendo quella che più facilita la scrittura delle query). In questo modo una delle due entità avrà anche tutti gli attributi della relazione (dell'ER) e gli attributi di chiave esterna riferiti alla chiave primaria dell'altra entità.

Mappatura delle relazioni one to one con partecipazione "semi-obbligatoria" (1,1) - (0,1)

In questo caso, si utilizza la stessa tecnica della mappatura delle relazioni one to one con partecipazione obbligatoria (illustrata sopra), ma per evitare proliferazione di `NULL` values, si è obbligati a condesare tutta la relazione nell'entità con partecipazione obbligatoria (1,1).

Mappatura delle relazioni one to one con partecipazione opzionale (0,1) - (0,1)

In questo caso si può utilizzare di nuovo la stessa logica della mappatura delle relazioni one to one con partecipazione obbligatoria oppure si può scegliere di mapparle anche come relazioni many to many in cui la chiave primaria della nuova relazione è formata dalla combinazione delle chiavi primarie di una delle due entità coinvolte, in modo da eliminare la proliferazione di `NULL` values. In questo ultimo caso, però, si aumentano le complessità delle query. La scelta di quale metodo utilizzare dipende da quanto si è disposti ad accettare la proliferazione di `NULL` values rispetto alla complessità delle query.

Per l'esame è sempre consigliato usare il primo metodo (one to many con partecipazione obbligatoria).

Mappatura delle relazioni ternarie

Le relazioni ternarie si mappano come relazioni many to many in cui la chiave primaria della nuova relazione è formata dalla combinazione delle chiavi primarie di tutte e tre le entità coinvolte. Gli attributi della relazione (dell'ER) diventano gli attributi della nuova relazione.

Nel caso in cui una delle entità coinvolte partecipi con cardinalità (1,1), si può scegliere di condensare tutta la relazione nell'entità con cardinalità (1,1) seguendo le stesse regole di mappatura delle relazioni one to many con partecipazione obbligatoria.

Mappatura delle relazioni ricorsive

Per la mappatura delle relazioni ricorsive è consigliato prima "linearizzare" la relazione ricorsiva come relazione tra due entità separate (ma che in realtà rappresentano la stessa entità) e poi applicare le regole di mappatura viste sopra. In particolare i nomi degli attributi di chiave esterna prendono il nome dai rami della relazione (es. in "succede (al trono)", i due attributi saranno "successore" e "predecessore"). In generale le relazioni ricorsive "linearizzate" possono essere di tre tipi:

- con struttura a lista (es. sovrano succede un altro sovrano) e si mappano come one to one
- con struttura ad albero (es. dipendente dirige di altri dipendenti) e si mappano come one to many
- con struttura multigrafo (es. persona conosce altre persone) e si mappano come many to many

Caso particolare di relazioni ternarie ricorsive

Nel caso di relazioni ternarie ricorsive (es. medico visita medico per una malattia), le due chiavi esterne che puntano alla stessa entità ("medico") devono essere distinte tra di loro, per cui avranno due frecce distinte che puntano alla stessa chiave primaria dell'entità di riferimento.

3.5 Normalizzazione del modello relazionale

Scopi della normalizzazione

La normalizzazione del modello relazionale è un processo che mira migliorare la struttura dello schema relazionale per garantire:

- **rendere chiara la semantica degli attributi**: separare entità che nella realtà sono distinte in relazioni distinte ed esplicitare le relazioni tra di esse attraverso l'uso di chiavi esterne, in questo modo si facilita la comprensione dello schema relazionale anche a chi non lo ha progettato
- **riduzione della ridondanza dei dati**: evitare di memorizzare più volte lo stesso dato o un dato deducibile da altri in quanto un aggiornamento parziale dei dati ridondanti può portare a problemi di incoerenza
- **riduzione della proliferazione di `NULL` values**: evitare di avere molti attributi opzionali in una singola relazione in quanto ciò può portare a spreco di spazio, crea difficoltà nell'interpretazione dei risultati delle query, siccome esistono molteplici interpretazioni di `NULL`, e serve maggiore attenzione nel caso di inner e outer join o nelle funzioni di aggregazione
- **eliminazione della generazione di tuple spurie**: permettere alle tabelle di essere ricomposte tramite join che coinvolgono almeno una chiave primaria o una chiave esterna senza generare tuple spurie che non rappresentano dati reali, ma solo ricombinazioni di valori prive di significato

Processo di normalizzazione

La normalizzazione si basa sull'analisi delle dipendenze funzionali tra gli attributi delle relazioni e sull'applicazione di una serie di regole di normalizzazione per minimizzare ridondanza, anomalie nell'aggiornamento parziale dei dati e tuple spurie. Per raggiungere questo obiettivo, si effettuano alcuni passaggi per portare lo schema ad uno dei tre livelli di normalizzazione (1NF, 2NF, 3NF) desiderati. In alcuni casi è sufficiente fermarsi al secondo livello per questioni di performance, in generale però si cerca di arrivare almeno al terzo livello di normalizzazione.

Il processo di normalizzazione è necessario solo per i modelli relazionali ottenuti senza una buona progettazione ER preliminare. Gli schemi relazionali ottenuti da modelli ER ben progettati non necessitano di normalizzazione.

Prime attribute - attributo primo

Un attributo si dice primo se appartiene a qualche chiave candidata della relazione. In caso contrario se non appartiene a nessuna chiave candidata, si dice non-primo.

Functional dependency - dipendenza funzionale

Due insiemi di attributi X e Y di una relazione $R(T)$ sono in dipendenza funzionale, indicata come $X \to Y$, se per ogni coppia di tuple t_i e t_j si ha che se i valori degli attributi in X sono uguali, allora anche i valori degli attributi in Y sono uguali.

$$\forall\, t_i, t_j \in r(R) \text{ vale } t_i[X] = t_j[X] \implies t_i[Y] = t_j[Y]$$

Si osserva che $X \to Y$ non implica necessariamente che $Y \to X$. Inoltre se X è una chiave candidata, allora X è in dipendenza funzionale con tutti i sottoinsiemi di Y e in generale con tutti gli altri attributi della relazione. Infine la dipendenza funzionale gode delle seguenti proprietà:

- **riflessività**: se $Y \subseteq X$, allora $X \to Y$
- **aumentatività**: se $X \to Y$, allora $XZ \to YZ$ per ogni insieme di attributi Z
- **transitività**: se $X \to Y$ e $Y \to Z$, allora $X \to Z$
- **decomposizione**: se $X \to YZ$, allora $X \to Y$ e $X \to Z$
- **unione**: se $X \to Y$ e $X \to Z$, allora $X \to YZ$
- **pseudo-transitività**: se $X \to Y$ e $YW \to Z$, allora $XW \to Z$

Full functional dependency - dipendenza funzionale completa

Una dipendenza funzionale $X \to Y$ si dice completa se non esiste un sottoinsieme proprio $X' \subset X$ tale che $X' \to Y$. Ovvero se rimuovendo un elemento da X la dipendenza funzionale non è più valida. In caso contrario si dice parziale.

Transitive dependency - dipendenza transitiva

Una dipendenza funzionale $X \to Y$ si dice transitiva se esiste un insieme di attributi Z che non è chiave candidata e nemmeno sottoinsieme di nessuna chiave candidata, tale che valgono $X \to Z$ e $Z \to Y$. In altre parole, un attributo Y dipende transitivamente da X se dipende da un altro attributo Z che a sua volta dipende da X.

Trivial and nontrivial functional dependency - dipendenza funzionale banale e non banale

Una dipendenza funzionale $X \to Y$ si dice banale se $Y \subseteq X$. In caso contrario si dice non banale.

First Normal Form - 1NF

La prima forma normale (1NF) richiede che tutti i domini degli attributi di una relazione siano composti solo da elementi atomici (non divisibili). In altre parole, ogni attributo deve contenere un singolo valore per ogni tupla e non è possibile avere attributi multivalore, attributi composti o relazioni annidate (più righe per tupla). Per eliminare queste situazioni si può procedere in diversi modi:

- **attributi multivalore**:
  1. si sposta l'attributo multivalore in una nuova relazione con anche la chiave primaria della relazione originale (metodo preferito)
  2. si estende la chiave primaria della relazione originale con l'attributo multivalore in modo da duplicare le tuple per ogni valore dell'attributo, si aggiunge, però, ridondanza di dati
  3. si duplica l'attributo multivalore in più attributi atomici (es. telefono1, telefono2, telefono3), se si conosce a priori il numero massimo di valori, si provoca però proliferazione di `NULL` values e complicazione delle query per cui è molto sconsigliato
- **relazioni annidate**: si crea una nuova relazione per la relazione annidata, in cui si aggiunge come chiave esterna la chiave primaria della relazione originale in modo da riunire i dati con una join

Second Normal Form - 2NF

La seconda forma normale (2NF) richiede che tutti gli attributi non-primi di una relazione siano in dipendenza funzionale completa con tutte le chiavi della relazione. In altre parole, gli attributi della relazione devono dipendere dalle chiavi completamente e non da parti delle chiavi. Per portare una relazione in 2NF, si separano gli attributi non-primi che dipendono solo da una parte della chiave primaria in altre relazioni insieme alla parte di chiave da cui dipendono.

Third Normal Form - 3NF

La terza forma normale (3NF) richiede che tutti gli attributi non-primi di una relazione non siano in dipendenza transitiva con qualsiasi chiave della relazione. Ovvero tutti gli attributi devono dipendere direttamente dalle chiavi. Per portare una relazione in 3NF, si separano gli attributi non-primi che dipendono transitivamente da altri attributi in altre relazioni insieme agli attributi da cui dipendono.

Alternative Third Normal Form - 3NF con definizione alternativa

In alternativa alla definizione di 3NF vista sopra, una relazione si dice in 3NF se per ogni dipendenza funzionale $X \to Y$ non banale, X deve esssere superchiave oppure ogni attributo in Y è primo. Questa definizione è equivalente alla precedente in quanto se X non è superchiave, allora gli attributi in Y non possono dipendere transitivamente da X se sono tutti primi.

4 Algebra relazionale

Introduzione

L'algebra relazionale è un linguaggio formale per la manipolazione e l'interrogazione di database relazionali. Si basa su un insieme di operatori che operano su relazioni (tabelle) per produrre nuove relazioni come risultato. Gli operatori dell'algebra relazionale si dividono in due categorie principali: operatori di aggiornamento e operatori di interrogazione (query).

Le operazioni di algebra relazionale possono essere combinate tra di loro per formare espressioni più complesse. Inoltre l'algebra relazionale è un **linguaggio procedurale**, in quanto l'utente deve specificare come ottenere il risultato desiderato attraverso una sequenza di operazioni.

4.1 Operazioni di aggiornamento - inserimento, modifica e rimozione

Le opzioni di aggiornamento consistono nelle operazioni di inserimento, modifica e rimozione di tuple. Sono in funzioni che mappano un'istanza del database in una nuova istanza del database che rispetta sempre tutti i vincoli imposti dallo schema del database (sia a livello di relazione che di integrità referenziale).

Inserimento

L'inserimento di una nuova tupla t_{new} in una relazione $R(T)$ avviene se e solo se la nuova istanza $r'(R)$ ottenuta dopo l'inserimento rispetta tutti i constraints imposti dallo schema (domain constraints, tuple constraints, key constraints, referential integrity). Se ciò non avviene, l'inserimento può essere rifiutato oppure si possono compirere operazioni di correzione dell'inserimento affinché la nuova istanza $r'(R)$ rispetti tutti i constraints.

Modifica

Anche la modifica di una tupla t_i in una relazione $R(T)$ avviene se e solo se la nuova istanza $r'(R)$ ottenuta dopo la modifica rispetta tutti i constraints imposti dallo schema. In analogia all'inserimento, se ciò non avviene, la modifica può essere rifiutata (restrict) oppure vengono compiute operazioni di correzione per propagare la modifica a tutte le tuple coinvolte (es. aggiornamento delle chiavi esterne) affinché la nuova istanza $r'(R)$ rispetti tutti i constraints (cascade).

Rimozione

Anche per la rimozione è necessario che la nuova istanza $r'(R)$ ottenuta dopo la rimozione rispetti tutti i constraints imposti dallo schema. In questo caso, se la rimozione di una tupla t_i comporta la violazione di un vincolo di integrità referenziale (es. la tupla rimossa è referenziata da una chiave esterna in un'altra relazione), si può scegliere di rifiutare la rimozione (restrict) oppure di propagare la rimozione, rimuovendo anche tutte le tuple che fanno riferimento alla tupla rimossa (cascade). È possibile scegliere di impostare le chiavi esterne che fanno riferimento alla tupla rimossa ad un valore di default (set default) oppure a `NULL` (set null) anche se quest'ultima opzione è fortemente sconsigliata in quanto crea tuple "orfane" e senza significato.

4.2 Query e interrogazioni di base (set operators - selection, projection)

Operazioni fondamentali

Le operazioni fondamentali dell'algebra relazionale si dividono in set operators e relational operators:

Set operators

- union ($\cup$)
- intersection ($\cap$)
- difference ($-$)
- cartesian product ($\times$)

Relational operators

- rename (ρ)
- selection (σ)
- projection (π)
- natural join ($\bowtie$)
- theta join ($\bowtie_\Theta$)
- outer theta join (⟗_Θ)

Compatibilità all'unione

Due relazioni $R_1(X_1, X_2, \ldots, X_n)$ e $R_2(Y_1, Y_2, \ldots, Y_n)$ sono dette compatibili all'unione se e solo se hanno lo stesso grado n (stesso numero degli attributi) e se ogni coppia ordinata di attributi corrispondenti ha lo stesso dominio e lo stesso nome. Formalmente:

$$R_1 \equiv R_2 \quad \Leftrightarrow \quad \forall i \in \{1, \ldots, n\} \text{ vale } X_i = Y_i \ \wedge \ \text{dom}(X_i) = \text{dom}(Y_i)$$

Unione

Date due relazioni R_1 e R_2 compatibili all'unione, l'unione tra le due relazioni $R_1 \cup R_2$ è la relazione che contiene tutte le tuple che sono presenti in R_1 o in R_2 (o in entrambe).

$$R_1 \cup R_2 = \{t \mid t \in R_1 \vee t \in R_2\}$$

Intersezione

Date due relazioni R_1 e R_2 compatibili all'unione, l'intersezione tra le due relazioni $R_1 \cap R_2$ è la relazione che contiene tutte le tuple che sono presenti sia in R_1 che in R_2.

$$R_1 \cap R_2 = \{t \mid t \in R_1 \wedge t \in R_2\}$$

Differenza

Date due relazioni R_1 e R_2 compatibili all'unione, la differenza tra le due relazioni $R_1 - R_2$ è la relazione che contiene tutte le tuple che sono presenti in R_1 ma non in R_2. Viceversa per $R_2 - R_1$. Si osserva che l'operazione di differenza non è commutativa.

$$R_1 - R_2 = \{t \mid t \in R_1 \wedge t \notin R_2\} \qquad R_2 - R_1 = \{t \mid t \in R_2 \wedge t \notin R_1\}$$

Prodotto cartesiano

Date due relazioni $R_1(X_1, X_2, \ldots, X_n)$ e $R_2(Y_1, Y_2, \ldots, Y_m)$, non necessariamente compatibili all'unione, il prodotto cartesiano tra le due relazioni $R_1 \times R_2$ è la relazione che contiene tutte le possibili combinazioni di tuple di R_1 e R_2. La relazione risultante ha grado $n + m$ e ha la cardinalità pari al prodotto delle cardinalità delle due relazioni originali.

$$R_1 \times R_2 = \{t_1 t_2 \mid t_1 \in R_1, \ t_2 \in R_2\}$$

Rename - rinominazioni

Data una relazione $R(X)$ con un set di attributi $A_1, A_2, \ldots, A_n \in X$ ed un nuovo set di attributi $B_1, B_2, \ldots, B_n \notin X$, la rinominazione consiste nel creare una nuova relazione con gli stessi dati iniziali, ma con un nuovo nome per gli attributi richiesti. I domini degli attirbuti e i dati contenuti non vengono alterati. Formalmente:

$$\rho_{B_1, B_2, \ldots, B_n \leftarrow A_1, A_2, \ldots, A_n}(R) = \left\{ t \mid \exists x \in R, \text{ t.c. } \begin{cases} t[B_1, \ldots, B_n] = x[A_1, \ldots, A_n] \\ t[C] = x[C] \text{ con } C \neq A_1, \ldots, A_n \end{cases} \right\}$$

La rinominazione non altera né il grado, né la cardinalità, né i dati della relazione originale.

Esempio: si vogliono rinominare gli attributi della relazione "Studente(id, name, surname)" in "Studente(student_id, first_name, last_name)":

$$\rho_{\text{student\_id, first\_name, last\_name} \leftarrow \text{id, name, surname}}(\text{Studente})$$

Selection - selezioni - horizontal decomposition

Data una relazione $R(X)$ e una condizione booleana θ che coinvolge gli attributi di R, la selezione $\sigma_\theta(R)$ è la relazione che contiene tutte le tuple di R che soddisfano la condizione θ.

$$\sigma_\theta(R) = \{t \mid t \in R \ \wedge \ \theta(t) = \text{true}\}$$

La proposizione θ può essere una combinazione di condizioni semplici (es. uguaglianze, disuguaglianze, confronti) oppure una composizione di esse tramite operatori logici (AND, OR, NOT). La selezione non altera il grado della relazione originale, ma può ridurne la cardinalità.

Esempio: si vuole selezionare tutti gli studenti con voto maggiore o uguale a 28 dalla relazione "Esame(student_id, course_id, grade)":

$$\sigma_{\text{grade} \geq 28}(\text{Esame})$$

Projection - proiezioni - vertical decomposition

Data una relazione $R(X)$ e un sottoinsieme di attributi $\{A_1, \ldots, A_m\} \subseteq X$, la proiezione $\pi_Y(R)$ è la relazione che contiene tutte le tuple di R limitate agli attributi in Y.

$$\pi_{A_1,\ldots,A_m}(R) = \{t[A_1, \ldots, A_m] \mid t \in R\}$$

La proiezione riduce il grado della relazione originale al numero m di attributi su cui viene fatta la proiezione, mentre la cardinalità può rimanere invariata (se gli attributi sono superchiave) o ridursi nel caso in cui la proiezione generi tuple duplicate (che vengono eliminate).

Quando si combinano più projection e selection, è preferibile eseguire prima la selection e poi la projection in quanto se si effettua prima la projection non è possibile utilizzare gli attributi eliminati per la selezione. Per questo motivo, le operazioni di projection e selection non sono commutative.

Esempio: si vuole ottenere la lista dei nomi e cognomi degli studenti dalla relazione "Studente(id, name, surname)":

$$\pi_{\text{name, surname}}(\text{Studente})$$

Gestione dei `NULL` values nelle selection e nelle projection

Siccome i `NULL` values non sono uguali a nessun altro valore (inclusi altri `NULL` values), nelle selection le condizioni che coinvolgono attributi con `NULL` values devono utilizzare gli operatori `IS NULL` e `IS NOT NULL` per verificare la presenza o l'assenza di `NULL` values.

Nelle projection, i `NULL` values vengono considerati come elementi uguali tra di loro per cui tuple che differiscono solo per la presenza di `NULL` values in attributi proiettati, vengono considerate duplicate e quindi eliminate.

4.3 Join operations

Theta join

Date due relazioni $R_1(X)$ e $R_2(Y)$ e una proposizione Θ, la theta join $R_1 \bowtie_\Theta R_2$ è la relazione che contiene tutte le combinazioni di tuple $t_1 \in R_1$ e $t_2 \in R_2$ che soddisfano la condizione Θ.

$$\text{Theta join}: \quad R_1 \bowtie_\Theta R_2 = \{t \mid \exists x \in R_1, \exists y \in R_2 \text{ t.c. } t[X] = x \ \wedge \ t[Y] = y \ \wedge \ \Theta(t) = \text{true}\}$$

La theta join ha lo stesso risultato del prodotto cartesiano con una selezione basata sulla condizione Θ, ma nei DBMS le operazioni compiute dalla theta join sono più efficienti:

$$R_1 \bowtie_\Theta R_2 = \sigma_\Theta(R_1 \times R_2)$$

La relazione risultante ha grado pari alla somma dei gradi delle due relazioni originali e la cardinalità minore o uguale al prodotto delle cardinalità.

Esempio: si vuole ottenere tutte le coppie manager-impiegato in cui il manager è più giovane dell'impiegato dalle relazioni "Manager(manager_id, manager_name, manager_age)" e "Impiegato(empl_id, empl_name, empl_age)":

$$\text{Manager} \bowtie_{\text{manager\_age} < \text{empl\_age}} \text{Impiegato}$$

Equi-join

Le equi-join sono un caso particolare di theta join in cui la condizione Θ è composta esclusivamente da uguaglianze tra attributi di due relazioni.

$$\text{Equi join}: \quad R_1 \bowtie_{A_1=B_1, A_2=B_2, \ldots, A_k=B_k} R_2$$

Anche in questo caso, la relazione risultante ha grado pari alla somma dei gradi delle due relazioni originali e la cardinalità minore o uguale al prodotto delle cardinalità.

Natural join

La natural join è un caso particolare di equi-join in cui gli attributi su cui si effettua l'uguaglianza hanno lo stesso nome. In questo caso, nella relazione risultante, gli attributi comuni alle due relazioni originali compaiono una sola volta. Formalmente:

$$\text{Natural join}: \quad R_1 \bowtie R_2 = R_1 \bowtie_{A_1=A_1, A_2=A_2, \ldots, A_k=A_k} R_2$$

In questo caso, la relazione risultante ha grado pari alla somma dei gradi delle due relazioni originali meno il numero degli attributi comuni, e la cardinalità minore o uguale al prodotto delle cardinalità. Più specificatamente:

- se gli attributi comuni non sono chiave primaria in nessuna delle due relazioni, la cardinalità è minore o uguale al prodotto delle cardinalità
- se gli attributi comuni sono chiave primaria in una delle due relazioni, la cardinalità è minore o uguale alla cardinalità della relazione in cui sono chiave primaria
- se gli attributi comuni sono chiave primaria in una delle due e chiave esterna nell'altra, la cardinalità è minore o uguale alla cardinalità della relazione in cui è chiave primaria

Dangling tuples - tuple penzolanti

Le dangling tuples sono tuple che non trovano corrispondenza nell'operazione di join. Nelle inner join (le theta join, equi-join e natural join), le dangling tuples vengono eliminate dalla relazione risultante. Per evitare la perdita di informazioni dovuta all'eliminazione delle dangling tuples, si utilizzano le outer join.

Full outer join

La full outer join tra due relazioni $R_1(XW)$ e $R_2(YZ)$ restituisce tutte le tuple ottenute dalla natural join tra le due relazioni, più tutte le dangling tuples di entrambe le relazioni, riempiendo con `NULL` values gli attributi mancanti. Formalmente:

$$\text{Full outer join}: \quad R_1 \,\overline{\bowtie}_{W\Theta Z}\, R_2 = \left(\begin{array}{c} R_1 \bowtie_{W\Theta Z} R_2 \\ \cup \\ (R_1 - \pi_{X,W}(R_1 \bowtie_{W\Theta Z} R_2)) \times \{Y = \texttt{NULL}, Z = \texttt{NULL}\} \\ \cup \\ (R_2 - \pi_{Y,Z}(R_1 \bowtie_{W\Theta Z} R_2)) \times \{X = \texttt{NULL}, W = \texttt{NULL}\} \end{array} \right)$$

Il grado della relazione risultante è dato dalla somma dei gradi delle due relazioni originali meno il numero di attributi comuni, mentre la cardinalità è data dalla somma del numero di tuple ottenute dalla natural join più il numero di dangling tuples di entrambe le relazioni originali.

Left outer join

La left outer join tra due relazioni $R_1(XW)$ e $R_2(YZ)$ restituisce tutte le tuple ottenute dalla natural join tra le due relazioni, più tutte le dangling tuples della prima relazione R_1, riempiendo con `NULL` values gli attributi mancanti della seconda relazione R_2. Formalmente:

$$\text{Left outer join}: \quad R_1 \,\overline{\bowtie}_{W\Theta Z}\, R_2 = \left(\begin{array}{c} R_1 \bowtie_{W\Theta Z} R_2 \\ \cup \\ (R_1 - \pi_{X,W}(R_1 \bowtie_{W\Theta Z} R_2)) \times \{Y = \texttt{NULL}, Z = \texttt{NULL}\} \end{array} \right)$$

Il grado della relazione risultante è dato dalla somma dei gradi delle due relazioni originali meno il numero di attributi comuni, mentre la cardinalità è data dalla somma del numero di tuple ottenute dalla natural join più il numero di dangling tuples della prima relazione originale R_1.

Right outer join

La right outer join tra due relazioni $R_1(XW)$ e $R_2(YZ)$ restituisce tutte le tuple ottenute dalla natural join tra le due relazioni, più tutte le dangling tuples della seconda relazione R_2, riempiendo con `NULL` values gli attributi mancanti della prima relazione R_1. Formalmente:

$$\text{Right outer join}: \quad R_1 \bowtie\!\!\subset_{W\Theta Z} R_2 = \begin{pmatrix} R_1 \bowtie_{W\Theta Z} R_2 \\ \cup \\ (R_2 - \pi_{Y,Z}(R_1 \bowtie_{W\Theta Z} R_2)) \times \{X = \texttt{NULL}, W = \texttt{NULL}\} \end{pmatrix}$$

Il grado della relazione risultante è dato dalla somma dei gradi delle due relazioni originali meno il numero di attribut comuni, mentre la cardinalità è data dalla somma del numero di tuple ottenute dalla natural join più il numero di dangling tuples della seconda relazione originale R_2.

Gestione dei `NULL` values nelle join operations

Le join operations ignorano i `NULL` values durante il confronto degli attributi. Di conseguenza, le tuple che contengono `NULL` values negli attributi coinvolti nella condizione di join non vengono abbinate e possono diventare dangling tuples.

4.4 Extended relational algebra - generalized projection, aggregation functions, grouping

Le operazioni di algebra relazionale estesa permettono di effettuare operazioni che coinvolgono più tuple contemporaneamente, come il raggruppamento e l'aggregazione dei dati.

Generalized projection

Data una relazione $R(X)$ e un insieme di funzioni $F_1, F_2, \ldots, F_n$ che agiscono sugli attributi di R, la generalized projection $\pi_{F_1,F_2,\ldots,F_n}(R)$ è la relazione che contiene i risultati delle funzioni applicate alle tuple di R. Formalmente:

$$\pi_{F_1,F_2,\ldots,F_n}(R) = \{t \mid \exists x \in R, \text{ t.c. } t[F_1, F_2, \ldots, F_n] = (F_1(x), F_2(x), \ldots, F_n(x))\}$$

Il grado della relazione risultante è pari al numero di funzioni applicate, mentre la cardinalità può variare in base al risultato delle funzioni applicate (si cancellano eventuali tuple duplicate), in assenza di cancellazioni la cardinalità rimane invariata.

Esempio: si vuole l'incasso per ordine dalla relazione "Ordine(order_id, product, quantity, price)":

$$\pi_{\text{order\_id, quantity}\times\text{price}}(\text{Ordine})$$

Siccome la nuova funzione non ha un nome, è necessario rinominare l'attributo risultante:

$$\rho_{\text{total\_amount}\leftarrow\text{quantity}\times\text{price}}\left(\pi_{\text{order\_id, quantity}\times\text{price}}(\text{Ordine})\right)$$

Aggregation functions e aggregation operations

Le aggregation functions sono funzioni che permettono di calcolare statistiche sui dati di una relazione. Le funzioni sono applicate ad un insieme di tuple e restituiscono un singolo valore. Le principali funzioni di aggregazione sono `SUM` (somma i valori), `AVG` (calcola la media), `COUNT` (conta il numero di tuple), `MIN` (trova il valore minimo) e `MAX` (trova il valore massimo).

Le aggregation operations permettono di applicare le funzioni di aggregazione su un insieme di tuple. La sintassi generale per un'operazione di aggregazione è:

$$\mathcal{F}_{f_1(A_i),\ldots,f_n(A_j)}(R(X))$$

Il risultato è una relazione con grado pari al numero di funzioni di aggregazione applicate e cardinalità pari a 1, in quanto l'aggregazione restituisce un singolo valore per ogni funzione applicata.

Esempio: si vuole calcolare il numero totale degli ordini effettuati, attraverso la seguente relazione "Ordine(order_id, product, quantity, price)":

$$\mathcal{F}_{COUNT(*)}(\text{Ordine})$$

Se si vuole calcolare la somma totale degli incassi, si utilizza:

$$\mathcal{F}_{SUM(\text{total\_amount})}\left(\rho_{\text{total\_amount}\leftarrow\text{quantity}\times\text{price}}\left(\pi_{\text{order\_id, quantity}\times\text{price}}(\text{Ordine})\right)\right)$$

Grouping

Il grouping (raggruppamento) permette di suddividere le tuple di una relazione in gruppi basati sui valori di uno o più attributi in modo da poter applicare le funzioni di aggregazione su ciascun gruppo separatamente. La sintassi generale per un'operazione di raggruppamento è:

$$_{(A_1,A_2,\ldots,A_k)}\mathcal{F}_{f_1(A_i),\ldots,f_n(A_j)}(R(X))$$

Il grado della relazione risultante è pari al numero di attributi di raggruppamento più il numero di funzioni di aggregazione applicate, mentre la cardinalità dipende dal numero di gruppi distinti formati in base agli attributi di raggruppamento.

Esempio: si vuole calcolare la quantità totale venduta per ogni prodotto dalla seguente relazione "Ordine(order_id, product, quantity, price)":

$$_{(\text{product})}\mathcal{F}_{SUM(\text{quantity})}(\text{Ordine})$$

Gestione dei `NULL` values nelle funzioni di aggregazione e nei raggruppamenti

Le funzioni di aggregazione `SUM`, `AVG`, `MIN` e `MAX` ignorano i `NULL` values durante il calcolo dei risultati. La funzione `COUNT`, se applicata ad un singolo attributo, ignora i `NULL` values, mentre se è applicata a più attributi, i `NULL` values vengono considerati come valori validi e uguali tra di loro.

Nei raggruppamenti, i valori `NULL` negli attributi di raggruppamento vengono considerati come valori validi e uguali tra di loro, formando un gruppo separato con tutti i `NULL` values.

Esempio: per seguente tabella sono mostrati i risultati alle funzioni di aggregazione di raggruppamento in presenza di `NULL` values:

A	B	C
1	10	30
2	`NULL`	40
3	20	`NULL`
4	`NULL`	`NULL`

$\longrightarrow$

Function	Result
`SUM(B)`	30
`SUM(C)`	70
`COUNT(B)`	2
`COUNT(A, B)`	4

Group by	Groups
B	10, 20, `NULL`
C	30, 40, `NULL`

5 Linguaggio SQL

5.1 Elementi fondamentali al linguaggio SQL

Tabella

Una tabella è una collezione di zero o più colonne ordinate e zero o più righe non ordinate. Non è possibile avere tabelle con lo stesso nome e non è possibile avere righe duplicate. Ogni colonna può memorizzare solo uno specifico tipo di dato. Le tabelle vengono utilizzate nei database relazionali per memorizzare e organizzare i dati in modo strutturato. Esistono due tipi di tabelle in SQL:

- **tabelle base**: tabelle che rispettano i vincoli e contengono i dati memorizzati nel database
- **tabelle derivate**: tabelle ottenute da una query e non necessariamente rispettano i vincoli

Query

La query sono operazioni compiute su una o più tabelle per recuperare, inserire, aggiornare o eliminare dati. In genere possono restituire una tabella derivata come risultato. Le query terminano con un "`;`".

Schema e istanza

- **lo schema** definisce come i dati sono organizzati nel database (tabelle, colonne, tipi di dato delle tabelle, relazioni tra tabelle, vincoli, ecc.). Uno schema non contiene dati e molto difficilmente cambia nel tempo. Gli schemi in SQL sono raggiuppati in "catalog" che in SQL sono anche detti "database"
- **l'istanza** è il contenuto effettivo del database in un dato momento, si riferisce ai dati memorizzati e cambia frequentemente

5.2 Linguaggio SQL

Introduzione

Il linguaggio SQL (Structured Query Language) è un linguaggio standard per la gestione e manipolazione di database relazionali attraverso i DBMS. SQL è un linguaggio dichiarativo, ovvero l'utente specifica cosa vuole ottenere senza dover specificare come esattamente eseguire l'operazione. Implementa lo schema relazionale e l'algebra relazionale. È composto da due sotto-linguaggi:

- **DDL (Data Definition Language)**: per definire e modificare la struttura del database
- **DML (Data Manipulation Language)**: per manipolare i dati all'interno delle tabelle

Esistono vari livelli di implementazione di SQL in base alle funzioni offerte:

- **entry SQL**: simile a SQL-89, include le funzionalità di base per la gestione dei dati
- **intermediate SQL**: con funzionalità più complesse implementate nella maggior parte dei DBMS
- **full SQL**: include tutte le funzionalità definite dallo standard SQL che non è detto siano implementate di base in qualsiasi DBMS

Sintassi di SQL

- **alfabeto**: i simboli validi per scrivere query SQL sono lettere (A-Z, a-z), numeri (0-9) e alcuni caratteri speciali; per scrivere le lettere accentate si usano sequenze di escape (`U&'d\0061 = a`)
- **token**: sono le unità lessicali delle query in SQL, si dividono in:
  - **identificatori**: nomi di oggetti del database (tabelle, colonne, vincoli, ecc.), sono case-insensitive e possono essere regular (iniziano con una lettera e contengono lettere, numeri e _) o delimited (racchiusi tra doppi apici, diventano case-sensitive e possono contenere qualsiasi carattere, es. `"Column-1&2"`)
  - **keywords**: parole riservate di SQL (`SELECT`, `FROM`, `WHERE`, ecc.), sono case-insensitive
  - **literals**: valori costanti (numeri, stringhe, date, ecc.), sono case-sensitive e possono essere scritti racchiusi tra apici singoli (`'Mario Rossi'`)
- **separatori**: caratteri che separano i token (white space) o commenti (`--` o `/* ... */`)

5.3 Data types in SQL

Dati built-in

- `CHARACTER(n)`, `CHAR(n)`: stringa di lunghezza fissa di esattamente n caratteri (n > 0)
- `CHARACTER VARYING(n)`, `VARCHAR(n)`: stringa di lunghezza variabile fino a n caratteri (n > 0)
- `BINARY(n)`: stringa di lunghezza fissa di esattamente n byte (n > 0)
- `BINARY VARYING(n)`, `VARBINARY(n)`: stringa di lunghezza variabile fino a n byte (n > 0).
- `NUMERIC(p [, s])`: numero a precisione arbitraria con p cifre totali e s cifre nella parte frazionaria. È particolarmente raccomandato per memorizzare importi monetari e altre quantità dove è richiesta esattezza, ad esempio denaro.
- `SMALLSERIAL`, `SERIAL`, `BIGSERIAL` [PostgreSQL only]: intero auto-incrementante
- `INTEGER`: intero con segno (in PostgreSQL: 4 bytes integer $\in [-2147483648, +2147483647]$)
- `SMALLINT`: piccolo intero con segno, (in PostgreSQL: 2 bytes integer $\in [-32768, +32767]$)
- `BIGINT`: grande intero con segno (in PostgreSQL: 8 bytes integer $\in [-9.22 \cdot 10^{18}, +9.22 \cdot 10^{18}]$)
- `REAL`: decimale con segno, (in PostgreSQL, 4 bytes floating point con 6 cifre di precisione)
- `DOUBLE PRECISION`: decimale con segno a doppia precisione (in PostgreSQL: 8 bytes floating point con 15 cifre di precisione)
- `BOOLEAN`: valore logico booleano
- `DATE`: data (giorno, mese, anno)
- `TIME [WITH TIMEZONE | WITHOUT TIMEZONE]`: orario del giorno, con o senza fuso orario
- `TIMESTAMP [WITH TIMEZONE | WITHOUT TIMEZONE]`: data e orario, con o senza fuso orario
- `INTERVAL x [TO y]`: intervallo di tempo

UUID types

Gli UUID o Universally Unique Identifier sono identificatori univoci universali a 128 bit generati utilizzando uno dei diversi algoritmi standard nel modulo "uuid-ossp". Hanno la proprietà di avere bassissima probabilità di collisione. Sono spesso usati come identificatori in database distribuiti dove risulterebbe troppo costoso mantenere un contatore centralizzato per generare chiavi primarie univoche.

Range types

I Range types rappresentano un intervallo di valori di un tipo di dato specifico. In PostgreSQL sono disponibili i seguenti range types:

- `int4range`: range of `integer`
- `int8range`: range of `bigint`
- `numrange`: range of numeric
- `tsrange`: range of `timestamp without time zone`
- `tstzrange`: range of `timestamp with time zone`
- `daterange`: range of `date`

JSON types

I JSON types sono utilizzati per salvare dati in formato JSON (non relazionale). Sono spesso usati in database non relazionali chiamati NoSQL database. In PostgreSQL esistono due tipi di dati JSON:

- `json`: memorizza i dati in formato testo esattamente come sono stati inseriti, sono più veloci da inserire, ma più lenti da processare
- `jsonb`: memorizza i dati in un formato binario decomposto che li rende più lenti da inserire a causa dell'overhead di conversione, ma significativamente più veloci da processare, poiché non è necessaria una nuova analisi

Enumerated Types

```
-- crea un enumerated type con nome e valori specificati
CREATE TYPE <enum_name> AS ENUM ('<value1>', '<value2>', ...)
-- es. crea un enum "mood" con i valori 'happy', 'sad' e 'neutral'
CREATE TYPE mood AS ENUM ('happy', 'sad', 'neutral');

-- elimina un enumerated type con il nome specificato
DROP TYPE <enum_name> [CASCADE | RESTRICT];
-- es. elimina l'enum "mood"
DROP TYPE mood;
```

L'opzione `CASCADE` elimina anche gli oggetti che dipendono dal tipo, mentre `RESTRICT` impedisce l'eliminazione se il tipo è referenziato da altri oggetti. Il comportamento di default è `RESTRICT`.

Domain Types

```
-- crea un domain type basato su un tipo di dato esistente con vincoli opzionali
CREATE DOMAIN <domain_name> AS <data_type> [<constraint_name> <constraint_definition>];
-- es. crea un domain type basato su INTEGER con vincolo di essere positivo
CREATE DOMAIN positive_int AS INTEGER CHECK (VALUE > 0);
```

I vincoli possono essere:

- `[DEFAULT NOT NULL | NULL]`: per specificare se il valore di default è `NULL` o se è necessario sempre specificarne il valore
- `CHECK (expression)`: per specificare una condizione che i valori devono soddisfare

```
-- elimina un domain type con il nome specificato
DROP DOMAIN <domain_name> [CASCADE | RESTRICT];
-- es. elimina il domain type "positive_int"
DROP DOMAIN positive_int; -- elimina il domain type "positive_int"
```

L'opzione `CASCADE` elimina anche gli oggetti che dipendono dal tipo, mentre `RESTRICT` impedisce l'eliminazione se il tipo è referenziato da altri oggetti. Il comportamento di default è `RESTRICT`.

Null values

I valori nulli possono rappresentare tre situazini diverse:

- un valore indefinito
- un valore non disponibile (non è stato ancora assegnato)
- un valore sconosciuto (non è noto)

Gli attributi che costituiscono la chiave primaria o altri attributi obbligatori non possono assumere valori nulli ed è necessario indicare il vincolo `NOT NULL` durante la creazione della tabella.

In generale i valori `NULL` non sono uguali tra di loro e non sono uguali a nessun altro valore perché in assenza di informazione possono essere tutto e niente.

```
-- errato: non si possono confrontare i valori NULL con l'operatore di uguaglianza
SELECT * FROM table WHERE column = NULL;

-- corretto: si usano gli operatori IS NULL e IS NOT NULL
... WHERE column IS NULL;
... WHERE column IS NOT NULL;
```

5.4 Data definition Language (DDL)

Creazione ed eliminazione di database

```
-- crea un nuovo database con il nome specificato
CREATE DATABASE <database_name> [OWNER <username>] [ENCODING <encoding_name>];
-- es. crea il database "Example" con codifica UTF-8
CREATE DATABASE Example ENCODING 'UTF-8';

-- elimina un database con il nome specificato
DROP DATABASE <database_name>;
-- es. elimina il database "Example"
DROP DATABASE Example;
```

È possibile, in fase di creazione, specificare opzionalmente il proprietario del database e la codifica dei caratteri da utilizzare.

Creazione ed eliminazione di schema

```
-- crea uno schema con il nome specificato
CREATE SCHEMA <schema_name> [AUTHORIZATION <username>];
-- es. crea uno schema chiamato "my_schema"
CREATE SCHEMA my_schema;

-- elimina uno schema con il nome specificato
DROP SCHEMA <schema_name> [CASCADE | RESTRICT];
-- es. elimina lo schema "my_schema"
DROP SCHEMA my_schema;
```

L'opzione `CASCADE` elimina anche tutti gli oggetti all'interno dello schema, mentre `RESTRICT` impedisce l'eliminazione se lo schema contiene oggetti. Il comportamento di default è `RESTRICT`.

Creazione di tabelle

```
-- crea una nuova tabella
CREATE TABLE <schema_name>.<table_name> (
    <column_name> <data_type> [<default_value>] [<column_constraint>],
    [<column_name> <data_type> [<default_value>] [<column_constraint>], ...]
    [, <table_constraint>, ...]
);
```

Se non viene specificato lo schema in cui inserirla, viene usato lo schema `public` di default. Le colonne devono avere i rispettivi tipi di dato e possono essere indicati altri vincoli o constraint sulle colonne (valore di default, vincoli, ecc.) o sulla tabella. I constraint di colonna possono essere:

- `NOT NULL`: la colonna non può contenere valori nulli
- `CHECK (expression)`: la colonna deve soddisfare una condizione specifica
- `DEFAULT <constant> | niladic-function | NULL`: valore di default che viene utilizzato se non ne viene specificato uno durante l'inserimento. Le `niladic-function` sono ad esempio `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`, `LOCALTIME`, ...

I constraint di tabella possono essere:

- `PRIMARY KEY (<column_name>, ...)`: specifica una chiave primaria su una o più colonne
- `UNIQUE (<column_name>, ...)`: specifica una chiave candidata su una o più colonne
- `FOREIGN KEY (<column_name>, ...) REFERENCES <ref_table>(<ref_column>, ...)`: definisce una chiave esterna che fa riferimento a un'altra tabella, è possibile indicare azioni opzionali per la cancellazione `ON DELETE` e l'aggiornamento `ON UPDATE` dei dati referenziati (ad esempio `CASCADE`, `SET NULL`, `SET DEFAULT`, `RESTRICT`, `NO ACTION`)

Se i constraint di tabella coinvolgono una singola colonna (ad esempio chiave primaria di una singola colonna), è possibile definirli direttamente nella definizione della colonna come constraint di tabella. Se invcene coinvolgono più colonne, devono necessariamente essere definiti come constraint di tabella.

```sql
-- es. crea la tabella "students" nello schema "my_schema"
CREATE TABLE my_schema.students (
    badge INTEGER PRIMARY KEY,      -- badge deello studente, chiave primaria
    name VARCHAR(100) NOT NULL,     -- nome dello studente, colonna non nulla
    surname VARCHAR(100) NOT NULL, -- cognome dello studente, colonna non nulla
    dob DATE DEFAULT NULL,          -- data di nascita, valore di default NULL
    degree VARCHAR(50)              -- corso di laurea
    school FOREIGN KEY REFERENCE my_schema.schools(school_id) -- chiave esterna che fa
    riferimento alla colonna "school_id" della tabella "schools"
);

-- oppure si definiscono i constraint di colonna separatamente in constraint di tabella
CREATE TABLE my_schema.students (
    badge INTEGER,      -- chiave primaria non indicata come constraint di colonna
    school VARCHAR(50), -- chiave esterna non indicata come constraint di colonna
    ...
    PRIMARY KEY (badge) -- chiave primaria definita come constraint di tabella
    FOREIGN KEY (school) REFERENCES my_schema.schools(school_id) -- chiave esterna
    definita come constraint di tabella
);
```

Eliminazione di tabelle

```sql
-- elimina una tabella con il nome specificato
DROP TABLE <table_name> [CASCADE | RESTRICT];
-- es. elimina la tabella "students"
DROP TABLE students; -- elimina la tabella "students"
```

L'opzione `CASCADE` elimina anche gli oggetti che dipendono dalla tabella, mentre `RESTRICT` impedisce l'eliminazione se la tabella è referenziata da altri oggetti. Il comportamento di default è `RESTRICT`.

Modifica di tabelle

```sql
-- modifica una tabella esistente
ALTER TABLE <table_name> <
    ADD COLUMN <column_definition> |                 -- aggiunge una nuova colonna
    DROP COLUMN <column_name> [RESTRICT | CASCADE] | -- rimuove una colonna
    ALTER COLUMN <column_name> <SET DEFAULT <new_default>> | -- cambia il valore di def.
    ALTER COLUMN <column_name> <DROP DEFAULT> |      -- rimuove il valore di default
    ADD CONSTRAINT <constraint_definition> |         -- aggiunge un nuovo vincolo
    DROP CONSTRAINT <constraint_name>                -- rimuove un vincolo esistente
>;
```

5.5 Data Manipulation Language - DML

Inserimento di dati

```
-- inserisce una nuova riga in una tabella, se non vengono specificate le colonne, si
    assume che i valori siano forniti per tutte le colonne e nell'ordine in cui le
    colonne sono state definite
INSERT INTO <table_name> [(<column_name> ...)] VALUES (<value1>, ...);
-- es. inserimento di una nuova riga nella tabella "employees" esplicitando le colonne
INSERT INTO employees (name, dob, degree, salary)
    VALUES ('John Doe', '1990-01-01', 'Computer Science', 50000);
-- es. inserimento senza specificare le colonne
INSERT INTO employees
    VALUES ('Jane Smith', '1985-05-15', 'Mathematics', 60000);
```

Eliminazione di dati

```
-- elimina righe da una tabella, opzionalmente filtrate da una condizione
DELETE FROM <table_name> [WHERE <condition>];
-- es. elimina tutte le righe della tabella "employees"
DELETE FROM employees;
-- es. elimina le righe dalla tabella "employees" dove il salario e' inferiore a 30000
DELETE FROM employees WHERE salary < 30000;
```

Aggiornamento o modifica dei dati

```
-- aggiorna i valori delle colonne in una tabella
UPDATE <table_name> SET <column_name> = <expr.> | NULL | DEFAULT, ... [WHERE <expr.>];
-- aggiorna il salario dei dipendenti nella tabella "employees" aumentando del 10% per
    quelli con salario inferiore a 60000
UPDATE employees SET salary = salary * 1.1 WHERE salary < 60000;
```

5.6 Query

Struttura generale

```
SELECT [DISTINCT | ALL] <column_list> | *
    FROM <table_name> [AS <alias>]
    [JOIN <table_name> [AS <alias>] ON <join_condition>]
    [WHERE <condition>]
    [GROUP BY <column_list> HAVING <condition>]
    [ORDER BY <column_name> [ASC | DESC], ...];
```

- `SELECT`: specifica le colonne da recuperare, può essere usato `DISTINCT` per eliminare i duplicati o `ALL` per includerli tutti (default), se si vuole recuperare tutte le colonne si usa l'asterisco `*`, eventualmente si possono usare funzioni di aggregazione (opportunamente rinominate)
- `FROM`: specifica la tabella da cui recuperare i dati, può essere usato un alias per riferirsi alla tabella attraverso un altro nome (utile nelle join)
- `JOIN`: unisce le tuple di una o più tabelle basandosi su una condizione di join, per facilitare la scrittura delle condizioni di join si utilizzano gli alias
- `WHERE`: filtra le righe in base a una condizione
- `GROUP BY`: raggruppa le righe in gruppi in base ai valori di una o più colonne, eventualmente si possono applicare funzioni di aggregazione sui gruppi ed è possibile filtrare i gruppi attraverso la clausola `HAVING`
- `ORDER BY`: ordina i risultati in base ai valori di una o più colonne, se specificati più ordinamenti su più colonne, vanno indicati in ordine di priorità decrescente

Ordine di valutazione delle clausole

```
FROM --> JOIN --> WHERE --> GROUP BY --> HAVING --> SELECT --> ORDER BY
```

Condizioni ed espressioni delle clausole

Le condizioni nelle varie clausole possono utilizzare operatori di confronto (`=`, `<>`, `<`, `>`, `<=`, `>=`) e di appartenenza (`IN`, `NOT IN`). È possibile combinare più condizioni utilizzando operatori logici (`AND`, `OR`, `NOT`). Inoltre, esiste l'operatore `LIKE` per confrontare stringhe con pattern che possono includere i caratteri jolly: il `%` rappresenta una sequenza di zero o più caratteri, mentre il `_` rappresenta un singolo carattere

```
-- 'J_n%' <=> 'J' + 1 carattere + 'n' + qualsiasi sequenza di caratteri
SELECT * FROM employees WHERE name LIKE 'J_n%';
```

Selection in SQL - clausola `WHERE`

Le selection selezionano le righe di una tabella che soddisfano una certa condizione e vengono implementate in SQL attraverso la clausola `WHERE`. Ad esempio:

```
SELECT * FROM employees WHERE age > 30L, oppure il nome di
una persona deve essere una stringa di caratteri ;
```

Projection in SQL - clausola `SELECT`

Le projection selezionano le colonne di una tabella e vengono implementate in SQL attraverso la clausola `SELECT`. Ad esempio:

```
SELECT name, age FROM employees;
```

Si osserva che, siccome in SQL le tabelle derivate possono contenere righe duplicate, le projection in SQL non corrispondono a quelle dell'algebra relazionale, a meno che non si usi la clausola `DISTINCT`.

Rename in SQL - clausola `AS`

Le rename in SQL vengono implementate attraverso la clausola `AS` che permette di assegnare un alias a una tabella o a una colonna nella query. Ad esempio:

```
SELECT e.name AS employee_name, e.dob AS date_of_birth FROM employees AS e;
```

Set Operators

I set operators combinano i risultati di due o più query. In SQL non è necessario che le tabelle siano compatibili all'unione (come in algebra relazionale), ma è sufficiente che le colonne siano dello stesso numero e che abbiano tipi di dato compatibili. Inoltre in SQL le tabelle derivate da set operators non contengono righe duplicate, a meno che non venga specificata l'opzione `ALL`. In SQL sono disponibili i seguenti operatori di insieme tra query:

- `query1 UNION query2`: unisce i risultati di due query
- `query1 INTERSECT query2`: restituisce le righe comuni a due query
- `query1 EXCEPT query2`: restituisce le righe presenti nella prima query ma non nella seconda

Aggregate Functions

Le aggregate functions permettono di eseguire funzioni su un insieme di valori di determinati attributi. Le funzioni di aggregazione più comuni in SQL sono:

- `COUNT(<column_name>)`: conta il numero di righe in un gruppo
- `SUM(<column_name>)`: calcola la somma dei valori in un gruppo
- `AVG(<column_name>)`: calcola la media dei valori in un gruppo
- `MIN(<column_name>)`: trova il valore minimo in un gruppo
- `MAX(<column_name>)`: trova il valore massimo in un gruppo

In alternativa al `<column_name>` è possibile usare l'asterisco `*` per indicare di agire su tutte le colonne.

Grouping

Il raggruppamento in SQL viene implementato attraverso la clausola `GROUP BY` che permette di raggruppare le righe in base ai valori di una o più colonne. È possibile inoltre filtrare i gruppi utilizzando la clausola `HAVING` che specifica una condizione che i gruppi devono soddisfare. Inoltre è possibile utilizzare funzioni di aggregazione per calcolare valori sui gruppi. Ad esempio, per calcolare il salario medio per ogni grado di istruzione:

```
SELECT degree, AVG(salary) AS average_salary FROM employees GROUP BY degree;
```

Le colonne elencate nella clausola `GROUP BY` devono essere presenti anche nella clausola `SELECT`. Inoltre è sempre consigliato rinominare le colonne derivate dalle funzioni di aggregazione per facilitare la lettura dei risultati. Si nota che la clausola `HAVING` viene valutata prima della clausola `SELECT` e di conseguenza prima della rename, per cui se si vuole filtrare in base ad una colonna generata con funzione di aggregazione, bisogna usare l'espressione originale e non l'alias. Ad esempio, per trovare i gradi di istruzione con salario medio superiore a 60000:

```
SELECT degree, AVG(salary) AS average_salary FROM employees
    GROUP BY degree HAVING AVG(salary) > 60000;
```

Query con `JOIN`

Le join in SQL permettono di creare tabelle derivate che hanno come righe la combinazione di righe di due o più tabelle basate su una condizione di join. Le condizioni di join sono uguaglianze tra attributi, ovvero le nuove righe risultanti saranno date degli attributi della tabella di sinistra, altri attributi dati dalla tabella di destra e un attributo in comune tra le due. Esitono 4 tipi principali di join:

- `INNER JOIN`: restituisce solo le righe che hanno corrispondenza in entrambe le tabelle
- `LEFT JOIN`: restituisce tutte le righe della tabella di sinistra a cui vengono associate le righe corrispondenti della tabella di destra, se non ci sono corrispondenze, i valori della tabella di destra saranno `NULL`
- `RIGHT JOIN`: restituisce tutte le righe della tabella di destra a cui vengono associate le righe corrispondenti della tabella di sinistra, se non ci sono corrispondenze, i valori della tabella di sinistra saranno `NULL`
- `FULL JOIN`: restituisce tutte le righe di entrambe le tabelle, se non ci sono corrispondenze, i valori della tabella senza corrispondenza saranno `NULL`

È utile effettuare i rename delle tabelle coinvolte nella join per facilitare la scrittura delle condizioni di join. Inoltre è possibile avere più join concatenate. Ad esempio, per unire le tabelle "employees", "departments" e "projects" basandosi sull'attributo "department_id" e "project_id":

```
SELECT e.name, d.department_name
    FROM employees AS e
    -- inner join tra employees e departments
    INNER JOIN departments AS d ON e.department_id = d.department_id
    -- inner join tra il risultato delle join precedenti e projects
    INNER JOIN projects AS p ON d.project_id = p.project_id;
```

Generalized projection

La generalized projection in SQL permette di creare nuove colonne derivate da espressioni o funzioni. Ad esempio, per calcolare l'età dei dipendenti basandosi sulla loro data di nascita:

```
SELECT name, EXTRACT(YEAR FROM AGE(CURRENT_DATE, dob)) AS age FROM employees;
```

Nested Queries

È possibile annidare query all'interno di altre query per creare condizioni più complesse o per calcolare valori intermedi. Le query annidate si indicano tra parentesi tonde.Ad esempio, per trovare i dipendenti con un salario superiore alla media:

```
SELECT name, salary FROM employees WHERE salary > (SELECT AVG(salary) FROM employees);
```

Un esempio comune di utilizzo delle nested query è trovare il valore di un certo attributo associato al valore massimo di un altro attributo, ad esempio per trovare lo studente con il voto più alto:

```
SELECT name, grade FROM student WHERE grade = (SELECT MAX(grade) FROM student);
```

Viste

Le viste in SQL sono tabelle virtuali derivate da altre tabelle attraverso una query. Non sono memorizzate fisicamente nel database, ma vengono generate al momento dell'accesso. Le viste permettono di semplificare query complesse spezzandole in più fasi. Il fatto di essere sempre ricalcolate porta al vantaggio di avere sempre dati aggiornati, ma lo svantaggio di avere prestazioni inferiori rispetto ad utilizzare una query completa e unica (il cui calcolo è ottimizzato dal DBMS). Le viste possono essere:

- **online views**: vengono create e utilizzate direttamente dal database
- **materialized views**: vengono create e memorizzate fisicamente nel database, migliorando le prestazioni a scapito di possibile incoerenza dei dati se le tabelle sottostanti vengono aggiornate

Di seguito sono riportati i comandi per creare ed eliminare viste in SQL:

```
-- crea una vista basata su una query selezionata
CREATE [MATERIALIZED] VIEW <view_name> AS <select_query>;

-- crea una vista chiamata "high_salary_employees" che mostra i dipendenti con salario
    superiore a 70000
CREATE VIEW high_salary_employees AS
    SELECT name, salary FROM employees WHERE salary > 70000;

-- elimina una vista con il nome specificato
DROP VIEW <view_name> [CASCADE | RESTRICT];

-- elimina la vista "high_salary_employees"
DROP VIEW high_salary_employees;
```

L'opzione `CASCADE` elimina anche gli oggetti che dipendono dalla vista, mentre `RESTRICT` impedisce l'eliminazione se la vista è referenziata da altri oggetti. Il comportamento di default è `RESTRICT`.

6 Alcune note per i quiz

- le regole di derivazione nel modello ER indicano come si calcolano gli attributi derivati a partire dagli altri attributi, si indicano in caso di attributi derivati introdotti per ridondanza (in genere) per facilitare le query
- diversamente da quanto succede con le query, i set operator di SQL eliminano i duplicati di default, infatti si chiamano set operator che lavorano su insiemi (set) che non ammettono duplicati
- il modello è l'insieme di costrutti e regole che definiscono la struttura di uno schema, non descrive nulla di specifico e non modella nessun miniworld, uno schema è l'insieme di istruzioni/costrutti per descrivere un miniworld specifico, mentre l'istanza è l'implementazione concreta dello schema con dati specifici; ad esempio il modello relazionale è l'insieme di regole (relazioni, chiavi, vincoli, ecc.), lo schema relazionale è l'insieme di tabelle e vincoli che descrivono un miniworld specifico, mentre l'istanza relazionale è l'insieme di tabelle con i dati effettivi
- un DBMS è organizzato in tre livelli, ciascuno indipendente da quelli superiori: livello esterno (vista utente), livello logico (tabelle, relazioni, vincoli) e livello fisico (memorizzazione su disco); in questo modo è possibile modificare il livello fisico senza influenzare il livello logico e modificare il livello logico senza influenzare il livello esterno
- la forma normale di Boyce-Codd (BCNF) è una forma normale più restrittiva della terza forma normale (3NF), richiede che per ogni dipendenza funzionale non banale $X \to Y$, X sia una superchiave della relazione; ovvero ogni attributo di Y deve necessariamente dipendere da una superchiave, non ci possono essere attributi in Y che dipendono transitivamente da X e nemmeno attributi primi in Y (chiavi candidate) che non dipendono da una superchiave; in altre parole serve per eliminare le chiavi candidate ed avere solo una chiave primaria
- la logica triple-state (3VL) con i valori `TRUE`, `FALSE` e `UNKNOWN` viene utilizzata in SQL per gestire quando si effettuano confronti con valori `NULL`; in questo caso il risultato del confronto è `UNKNOWN`; per calcolare il risultato di espressioni logiche con 3VL si usano le tabelle di verità specifiche supponendo che `UNKNOWN` possa essere sia `TRUE` che `FALSE`:

  - `TRUE AND UNKNOWN = UNKNOWN`, `FALSE AND UNKNOWN = FALSE`
  - `TRUE OR UNKNOWN = TRUE`, `FALSE OR UNKNOWN = UNKNOWN`
  - `NOT UNKNOWN = UNKNOWN`

  la clausola `WHERE` tiene solo le righe per cui la condizione è `TRUE`, scartando quelle con condizione `FALSE` o `UNKNOWN` (ignora i valori `NULL`), la clausola `CHECK` invece ignora solo le righe con condizione `FALSE`, accettando quelle con condizione `TRUE` o `UNKNOWN` (accetta i valori `NULL`)
- in SQL è possibile inserire valori `NULL` negli attributi che fanno parte di una chiave candidata (`UNIQUE`), anche se in teoria non dovrebbe essere possibile (si perde la funzione di chiave candidata), inoltre si possono sempre inserire valori `NULL` negli attributi che fanno parte di superchiavi e chiavi esterne, ma non in chiavi primarie (che assolutamente non possono avere valori nulli)
- il processo di ristrutturazione di uno schema ER comprende: analisi delle ridondanze, eliminazione delle generalizzazioni, partizionamento e accorpamento di entità e relazioni (attributi opzionali su entità secondarie), scelta degli identificatori principali, eliminazione degli attributi multivalore