Riassunto di Basi di Dati

di Arlind Pecmarkaj

Anno Accademico 2022/2023

19 maggio 2023

Indice

1 Relazione e schema

Uno schema di relazione di nome R con attributi $t_1, \ldots, t_n$ avra come schema $R(t_1, \ldots, t_2)$ (es CORSI(Corso, Codice, Docente)).
Con $t[A]$ denotiamo il valore del n-upla t col sull'attributo (es $ist_1[Corso] =$ "$Basi\,di\,Dati$").

2 Vincoli di integrità

Non tutte le istanze di una relazione possono considerarsi lecite. Un vincolo è una funzione che per ogni istanza della relazione associa Vero se essa è lecita, Falso altrimenti.

2.1 Vincoli intra-relazionali e chiavi

Son vincoli su ciascuna relazione.
I **vincoli di n-upla** esprimono condizioni su ciascuna n-upla, considerata singolarmente e possono essere espressi mediante espressioni algebriche o espressioni booleane (es. in una relazione che tiene gli esami possiamo mettere $(voto \geq 18) \wedge (voto \leq 30)$).
I **vincoli di chiave** specificano come devono comportarsi le chiavi di una relazione.

- Una **chiave** è un insieme di attributi che consente di identificare in maniera univoca le n-uple di una relazione.
- Un sottoinsieme K di attributi di una relazione è una **superchiave** se NON contiene due n-uple distinte t_1 e t_2 con $t_1[K] = t_2[K]$.
- Perciò una **chiave** K di una relazione r è una **superchiave minimale** di r (ossia non esiste un'altra superchiave K' che sia contenuta in K). In fase di progettazione, le chiavi dovrebbero essere definite a livello di schema, e non di istanza.
- Una **chiave primaria** è una chiave di una relazione su cui NON sono ammessi valori NULL. Ogni relazione deve disporre di una chiave primaria che può essere composta da più attributi.

2.2 Vincoli inter-relazionali e chiavi esterne

Nel modello relazionale, una base di dati può essere composta da molte relazioni collegate tra loro.
Collegamenti tra relazioni differenti sono espresse mediante valori comuni in attributi replicati.
In molti scenari d'uso, risulta utile imporre un vincolo sulle dipendenze tra relazioni.
Ogni riga della tabella **referenziante** si collega al massimo ad una riga della tabella **referenziata**, sulla base dei valori comuni nell'attributo/negli attributi replicati.
Un **vincolo di integrità referenziale** ("foreign key") fra gli attributi X di una relazione R_1 e un'altra relazione R_2 impone ai valori (diversi da NULL) su X in R_1 di comparire come valori della chiave primaria di R_2.
In pratica: consente di collegare le informazioni tra tabelle diverse attraverso valori comuni.

Il vincolo di integrità referenziale è definito tra gli attributi di una tabella (interna) ed il nome della tabella referenziata. Per definirlo, occorre esplicitare i nomi degli attributi (su cui si applica il vincolo) di entrambe le tabelle: con notazione: SCHEMA.NomeAttributo (es. VOLI.Arrivi → AEROPORTO.IdAeroporto).
Può accadere che un'operazione di aggiornamento su una relazione causi **violazioni di vincoli di integrità** su altre relazioni. Generalmente si procede in 3 modi:

- Non consentire l'operazione.
- Eliminazione a cascata.
- Inserimento di valori NULL.

3 SQL

SQL è un linguaggio per basi di dati basate sul modello relazionale Valgono i concetti generali del modello relazionale, ma con qualche differenza:

- Si parla di tabelle (e non relazioni).
- Il risultato di un'operazione sui dati può restituire una tabella con righe duplicate.
- Il sistema dei vincoli è più espressivo
- Il vincolo di integrità referenziale (chiave esterna) è meno stringente.

Presenta due componenti principali

- DDL (Data Definition Language): Contiene i costrutti necessari per la creazione/modifica dello schema della base di dati.
- DML (Data Manipulation Language): Contiene i costrutti per le interrogazioni e di inserimento/eliminazione/modifica di dati

3.1 DDL (Creazione)

Tramite il costrutto **create database**, è possibile costruire uno schema di una base di dati (ossia il collettore di tabelle/viste/etc)

```
create schema NomeSchema
    [authorization Nome]
```

dove Nome è il nome del proprietario dello schema.
Esempio:

```
create schema DB-PEKKA
    authorization Arlind
```

Tramite il costrutto **create table**, è possibile costruire una tabella all'interno dello schema

```
create table NomeTabella (
    nomeAttributo1 Dominio [ValDefault][Vincoli],
    nomeAttributo2 Dominio [ValDefault][Vincoli],
    ...
);
```

Per ciascun attributo, è possibile specificare, oltre al nome e dominio, un valore di default e i vincoli.
Il dominio **character** consente di rappresentare singoli caratteri o stringhe di lunghezza max fissa

```
character/char [varying][(Lunghezza)]
```

Se la lunghezza non è specificata allora è un solo carattere.
Quando vediamo per esempio

```
character varying (20)
```

ci indica che se ho una stringa più piccola dei 20 caratteri viene memorizzata la stringa più corta, mentre se vediamo il costrutto

```
varchar(20)
```

ci indica che lo spazio rimanente viene riempito con degli spazi.
I tipi **numerici esatti** consentono di rappresentare valori esatti, interi o con una parte decimale di lunghezza prefissata

```
numeric [(Precisione[,Scala])])
decimal [(Precisione[,Scala])])
integer
smallint
```

La precisione indica il numero di cifre nel numero, mentre la scala il numero di cifre a destra della virgola.
La keyword **auto_increment** consente di creare dei campi numerici che si auto-incrementano ad ogni nuovo inserimento nella tabella

```
integer auto_increment
smallint auto_increment
```

I tipi **numerici approssimati** consentono di rappresentare valori reali con rappresentazione in virgola mobile

```
float [(Precisione)]
real
double precision
```

La precisione è la lunghezza della mantissa.
I **domini temporali** consentono di rappresentare informazioni temporali o intervalli di tempo

```
date [(Precisione)]
time [(Precisione)]
timestamp
interval PrimaUnita [to UltimaUnita]
```

Per esempio **interval year** to **month** è un intervallo rappresentante il numero di anni e mesi (resto) fra due date.
Il dominio **boolean** consente di rappresentare valori di verità (true/false).
I domini **blob** e **cblob** consentono di rappresentare oggetti di grandi dimensioni come sequenza di valori binari (blob) o di caratteri (cblob).
Tramite il costrutto **domain**, l'utente può costruire un proprio dominio di dati a partire dai domini elementari

```
create domain NomeDominio as TipoDati
        [Valore di default]
        [Vincolo]
```

Per esempio

```
create domain Voto as smallint
        default NULL
        check (value >=18 and value <= 30)
```

Per ciascun dominio o attributo, è possibile specificare un valore di default attraverso il costrutto default

```
default [valore | user | null]
```

- **valore** indica un valore del dominio.
- **user** è l'id dell'utente che esegue il comando.
- **null** è il valore null.

I dati all'interno di una base di dati sono corretti se soddisfano un insieme di regole di correttezza, queste regole son dette **vincoli di integrità**.
La verifica della correttezza dello stato di una base di dati può essere effettuata:

- dalle **procedure applicative**, che effettuano tutte le verifiche necessarie: all'interno di ogni applicazione sono previste tutte le verifiche di correttezza necessarie (es. faccio i controlli nel codice PHP)
- mediante la **definizione di vincoli di integrità** sulle tabelle: definiti nelle istruzioni CREATE o ALTER TABLE, memorizzati tra le info di sistema per la gestione del DB e durante l'esecuzione di qualunque operazione di modifica dei dati il DBMS verifica automaticamente che i vincoli siano osservati.
- mediante la definizione di **trigger**, procedure eseguite in modo automatico quando si verificano opportune modifiche dei dati: definiti nell'istruzione CREATE TRIGGER, Quando si verifica un evento di modifica dei dati sotto il controllo del trigger, la procedura viene eseguita automaticamente.

Mediante la clausola **check** è possibile esprimere vincoli di n-upla arbitrari

```
NomeAttributo ... check (Condizione)
```

Il vincolo viene valutato n-upla per n-upla.
Il vincolo **not null** indica che il valore null non è ammesso come valore dell'attributo.
Il vincolo **unique** impone che l'attributo/attributi su cui si applica non presenti valori comuni in righe differenti ossia che l'attributo/i sia una *superchiave della tabella*, presenta due sintassi:

```
Attributo Dominio [ValDefault] unique
```

se la superchiave è un solo attributo o

```
unique(Attributo1, Attributo2, ...)
```

se la superchiave è composta da più attributi.

Il vincolo **primary key** impone che l'attributo/attributi su cui si applica non presenti valori comuni in righe differenti e non assuma valori NULL, ossia che l'attributo/i sia una *chiave primaria*. Come per il vincolo unique, presenta due sintassi in base a se la chiave è composta da un solo attributo o da più

```
Attributo Dominio [ValDefault] primary key
primary key(Attributo1, Attributo2, ...)
```

A differenza di unique e not null che possono essere definiti su più attributi della stessa tabella, il vincolo primary key deve apparire una sola volta per tabella.

I vincoli **references** e **foreign key** consentono di definire dei vincoli di integrità referenziale tra i valori di un attributo nella tabella in cui è definito (tabella

interna) ed i valori di un attributo in una seconda tabella (tabella esterna) a patto che l'attributo/i cui si fa riferimento nella tabella esterna deve/devono essere soggetto/i al vincolo unique.
Il costrutto **foreign key** si utilizza nel caso il vincolo di integrità referenziale riguardi più di un attributo delle tabelle interne/esterne. Per esempio

```
create table Studente {
    Matricola character(20) primary key,
    Nome varhcar(20),
    Cognome varchar(20),
    Datanascita date,
    foreign key(Nome,Cognome,Datanascita) references
    Anagrafica(Nome,Cognome,Data)
);
```

Cosa accade se un valore nella tabella esterna viene cancellato o viene modificato? Il vincolo di integrità referenziale nella tabella interna potrebbe non essere più valido!
È possibile associare **azioni specifiche** da eseguire sulla tabella interna in caso di *violazioni del vincolo di integrità referenziale*.

```
on (delete | update)
    (cascade | set null | set default| no action)
```

- **cascade**: elimina/aggiorna righe (della tabella interna).
- **set null**: setta i valori a null.
- **set default**: ripristina il valore di default
- **no action**: non consente l'azione (sulla tabella esterna). In assenza di comportamenti specifici, "no action" è il valore di default.

È possibile modificare gli schemi di dati precedentemente creati tramite le primitive di **alter** (modifica) e **drop** (cancellazione)

```
drop (schema|domain|table|view) NomeElemento
drop (restrict|cascade) NomeElemento
alter NomeTabella
    alter column NomeAttributo
    add column NomeAttributo
    drop column NomeAttributo
    add constraint DefVincolo
```

3.2 DML (Select, Join)

Le operazioni di interrogazione vengono implementate dal costrutto di **select**

```
select Attributo1, ..., AttributoM
from Tabella1, ..., TabellaN
where Condizione
```

La semantica è effettua il prodotto cartesiano delle Tabella1, ..., TabellaN. Da queste, estrai le righe che rispettano la Condizione. Di quest'ultime, preleva solo le colonne corrispondenti a: Attributo1, ..., AttributoM.

La clausola **where** specifica quali righe delle tabelle devono comparire nel risultato finale.
La condizione della clausola può contenere un'espressione booleana o una combinazione di espressioni mediante gli operatori and, or, not.
Nella clausola where è possibile fare confronti tra stringhe usando l'operatore **like** e l'utilizzo di wildcard:

- _ per un carattere arbitrario.
- % per una sequenza di caratteri arbitraria.

Nella clausola where, l'operatore **between** consente di verificare l'appartenenza ad un certo insieme di valori.

In generale, SQL utilizza una logica a tre valori: true (T), false (F), unknown (U): esistono gli operatori **IS NULL** ed **IS NOT NULL**.

È possibile ri-denominare le colonne del risultato di una query attraverso il costrutto **as**, per esempio

```
select Nome as Name, Cognome as LastName
from Impiegati
where (Nome = 'Marco')
```

È possibile usare espressioni aritmetiche (semplici) sui valori degli attributi di una select

```
select Nome as Name, Stipendio/12 as SalaryM
from Impiegati
where (Nome = 'Marco')
```

La clausola **from** specifica la lista delle tabelle cui si deve accedere e nel caso le tabelle fossero multiple si fa il prodotto cartesiano di esse.
È possibile specificare degli alias per i nomi delle tabelle, mediante il costrutto **as**, in modo tale da evitare ambiguità tra attributi con nomi uguali di tabelle diverse (in alternativa si può usare la notazione *NomeTabella.NomeAttributo*)

```
select Telefono as Tel
from Impiegati as I, Sedi as S
where (I.Ufficio = S.Ufficio) and (Codice = 145)
```

Il risultato di una query SQL potrebbe avere righe duplicate.
Il costrutto **distinct** (nella select) consente di rimuovere i duplicati nel risultato, invece il costrutto **all** (nella select) NON rimuove i duplicati (comportamento di default).

Il costrutto **order by** consente di ordinare le righe del risultato di un'interrogazione in base al valore di un attributo specificato e va inserito dopo la clausola where

```
order by Attr1 [asc|desc], ..., AttrN [asc|desc]
```

Supponiamo di voler scrivere una query per contare il numero di Impiegati che lavorano nell'ufficio A.
Sorge un problema: la select vista fin qui opera a livello di tuple, e non a livello di colonne. Abbiamo bisogno degli **operatori aggregati**.
Gli operatori aggregati si applicano a gruppi di tuple (e non tupla per tupla), e producono come risultato un solo valore.
Vengono in genere inseriti nella select, e valutati dopo la clausola where e from

```
count (* | [distinct|all] Lista Attributi)
sum (Lista Attributi)
avg (Lista Attributi)
min (Lista Attributi)
max (Lista Attributi)

select OP(Attributi)
from ListaTabelle
where Condizione
```

L'operatore di **raggruppamento** consente di dividere la tabella in gruppi, ognuno caratterizzata da un valore comune dell'attributo specificato nell'operatore

```
select ListaAttributi1
from ListaTabelle
where Condizione
group by ListaAttributi2
```

ListaAttributi1 deve essere un sottoinsieme di ListaAttributi2 e può contenere operatori aggregati.
Vediamo come esempio la query per avere il numero di strutturati in un dipartimento

```
select Dipartimento as Dip, count(*) as numero
from Strutturati
group by Dipartimento
```

È possibile filtrare i gruppi in base a determinate condizioni, attraverso il costrutto **having**, posto dopo il group by.
La clausola viene valutata su ciascun gruppo, contiene operatori aggregati o condizioni sulla lista degli attributi nella group by.
É utile vedere come viene valutata una query con una select generale

```
select ListaAttributi1
from ListaTabelle
where Condizione
group by ListaAttributi2
having Condizione
```

1. Viene fatto il prodotto cartesiano delle tabelle per poi estrarre le righe che rispettano la condizione della clausola where.
2. Viene fatto il partizionamento della tabella in base alla clausola group by.
3. Vengono filtrati i gruppi che rispettano la condizione su having.
4. Viene fatta la selezione dei valori delle colonne o l'esecuzione degli operatori aggregati su ciascuno dei gruppi e viene composta la tabella finale.

Oltre ad i comandi di interrogazione, la parte DML definisce anche le operazioni per la modifica dell'istanza della base di dati.
È possibile inserire una riga esplicitando i valori degli attributi oppure estraendo le righe da altre tabelle del database

```
insert into NomeTabella
[ListaAttributi] values (ListaValori)
```

È possibile cancellare tutte le righe che soddisfano una condizione (cancella tutto se non specificata)

```
delete from Tabella where Condizione
```

È possibile aggiornare il contenuto di uno o più attributi di una tabella che rispettano una certa condizione

```
update NomeTabella
set attributo = expr|select|null|default
[where Condizione]
```

È possibile implementare il join tra tabelle in due modi distinti (ma equivalenti nel risultato).
Il **join implicito** prevede le condizioni del join direttamente nella clausola del where

```
select ListaAttributi
from Tabella1, Tabella2
where Tabella1.AttrX = Tabella2.AttrY
```

Il **join esplicito** prevede l'utilizzo dell'operatore di inner join nella clausola from

```
select ListaAttributi
from Tabella1 join Tabella2 on CondizioneJoin
[where Condizione]
```

Esistono altre tre varianti (poco usate) dell'operatore di join:

- **left join** - risultato dell'inner join + righe della tabella di sinistra che non hanno un corrispettivo a destra (completate con valori null).
- **right join** - risultato dell'inner join + righe della tabella di destra che non hanno un corrispettivo a destra (completate con valori null).
- **full join** - risultato dell'inner join + righe della tabella di sinistra/destra che non hanno un corrispettivo a destra/sinistra (completate con valori null).

3.3 Interrogazioni nidificate

Nella clausola where, oltre ad espressioni semplici, possono comparire espressioni complesse in cui il valore di un attributo viene confrontato con il risultato di un'altra query (query annidate)

```
select ...
from ...
where (Attributo expr select ...
                     from ...
                     where ...
       )
```

Nota: si sta confrontando un singolo valore con il risultato di una query quindi potenzialmente una tabella!
Per esempio se vogliamo il codice dello strutturato con lo stipendio più alto scriveremo

```
select Codice
from Strutturati
where Stipendio = (select max(Stipendio)
                   from Strutturati)
```

Un altro esempio che possiamo fare è trovare il codice dei professori che afferiscono allo stesso dipartimento del professore con Codice 123. La formulazione mediante interrogazioni nidificate consente di separare il problema in due sottoproblemi. Il primo è trovare il dipartimento del professore 123 che sarà la nostra query interna

```
select Dipartimento
from Strutturati
where Codice = 123
```

Il secondo è trovare il codice dei professori che afferiscono allo stesso dipartimento del professore con Codice 123. Per il confronto nel where mettiamo la query del primo problema

```
select Codice
from Strutturati
where Dipartimento = (select Dipartimento
                      from Strutturati
                      where Codice = 123);
```

È possibile utilizzare '=' esclusivamente se è noto a priori che il risultato della select nidificata è sempre un solo valore.
L'esempio precedente è formulabile tramite join

```
select S1.Codice
from Strutturati as S1, Strutturati as S2
where S1.Dipartimento = S2.Dipartimento
      and S2.Codice = 123;
```

Esistono operatori speciali di confronto nel caso di interrogazioni annidate:

- **any**: la riga soddisfa la condizione se è vero il confronto tra il valore dell'attributo ed ALMENO UNO dei valori ritornati dalla query annidata.
- **all**: la riga soddisfa la condizione se è vero il confronto tra il valore dell'attributo e TUTTI i valori ritornati dalla query annidata.

Il costrutto **in** restituisce true se un certo valore è contenuto nel risultato di una interrogazione nidificata, false altrimenti (esiste l'analogo **not in** che fa il contrario)

```
select ListaAttributi
from TabellaEsterna
where Valore/i in select ListaAttributi2
                  from TabellaInterna
                  where Condizione
```

Il costrutto exists restituisce true se l'interrogazione nidificata restituisce un risultato non vuoto (almeno un elemento trovato)

```
select ListaAttributi
from TabellaEsterna
where exists select ListaAttributi2
                  from TabellaInterna
                  where Condizione
```

Per esempio vogliamo estrarre nome e cognome degli strutturati del dipartimento di Informatica che guadagnano più di tutti i loro colleghi di Fisica

```
select Nome, Cognome
from Strutturati
where (Dipartimento = 'INFORMATICA') and
      (Stipendio > all(select Stipendio
                       from Strutturati
                       where Dipartimento = 'FISICA'
                      )
      )
```

Le interrogazioni nidificate possono essere:

- **Non correlate**: non c'è passaggio di binding tra un contesto all'altro. Le interrogazioni vengono valutate dalla più interna alla più esterna.
- **Correlate**: c'è passaggio di binding attraverso variabili condivise tra le varie interrogazioni. In questo caso, le interrogazioni più interne vengono valutate su ogni tupla.

Un esempio di interrogazione correlata può essere l'estrazione di nome/cognome degli impiegati di un'aziente che hanno omonimi

```
select Nome, Cognome
from Impiegati AS I
where (I.Nome, I.Cognome) =
        (select Nome, Cognome
         from Impiegati AS I2
         where I.Nome = I2.Nome
         and I.Cognome = I2.Cognome
         and I.Codice <> I2.Codice)
```

Con le **table function** possiamo definire una tabella temporanea che può essere utilizzata per ulteriori operazioni di calcolo.
Una table function ha la struttura di una SELECT, è definita all'interno di una clausola FROM epuò essere referenziata come una normale tabella. Inoltre permette di calcolare più livelli di aggregazione e permette di formulare in modo equivalente le interrogazioni che richiedono la correlazione.
Per esempio dato il seguente schema

Studente (Matricola, AnnoIscrizione)
Esame–superato (Matricola, CodC, Data, Voto)

vogliamo trovare la media massima conseguita da uno studente per ogni anno di iscrizione

```
select ...
from Studente,
     (select Matricola, avg(Voto) AS MediaStudente
      from Esame–superato
      group by Matricola) AS Medie
where Studente.Matricola = Medie.Matricola
```

3.4 Operatori insiemistici

L'operatore insiemistico **union** esegue l'unione delle due espressioni relazionali A e B. Le espressioni relazionali A e B possono essere generate da istruzioni SELECT e richiede la compatibilità di schema tra A e B.
Per esempio la query per trovare il codice dei prodotti di colore rosso o forniti dal fornitore F2 (o entrambe le cose) sarà

```
select CodProd
from Prodotti
where Colore = 'Rosso'
union
select CodProd
From FornitoriProdotti
where CodForn = 'F2';
```

L'operatore insiemistico **intersect** esegue l'intersezione delle due espressioni relazionali A e B. Le espressioni relazionali A e B possono essere generate da istruzioni SELECT e richiede la compatibilità di schema tra A e B.
Per esempio la query per trovare le città che sono sia sede di fornitori, sia magazzino di prodotti sarà

```
select Sede
from Fornitori
intersect
select Magazzino
from Prodotti;
```

L'operazione di intersezione può essere eseguita anche mediante il join o l'operatore IN: nel primo caso la clausola FROM contiene le relazioni interessate dall'intersezione mentre la clausola WHERE contiene condizioni di join tra gli attributi presenti nella clausola SELECT delle espressioni relazionali A e B. Nel secondo caso una delle due espressioni relazionali diviene un'interrogazione nidificata mediante l'operatore IN e gli attributi nella clausola SELECT esterna, uniti da un costruttore di tupla, costituiscono la parte sinistra dell'operatore IN.
L'operatore insiemistico **except** esegue la sottrazione dell'espressione relazionale B dall'espressione A e richiede la compatibilità di schema tra A e B.
Per esemempio la query per trovare le città che sono sede di fornitori, ma non magazzino di prodotti sarà

```
select Sede
from Fornitori
except
select Magazzino
from Prodotti;
```

L'operazione di differenza può essere eseguita anche mediante l'operatore NOT IN dove l'espressione relazionale B è nidificata all'interno dell'operatore NOT IN.

3.5 Viste

Le **viste** rappresentano "tabelle virtuali" ottenute da dati contenute in altre tabelle del database e ogni vista ha associato un nome ed una lista di attributi, e si ottiene dal risultato di una select

```
create view NomeView [ListaAttributi]
as <select query>
[with [local | cascade] check option]
```

I dati delle viste NON sono fisicamente memorizzati a parte, in quanto dipendono da altre tabelle (ad eccezione delle viste materializzate: le viste esistono a livello di schema ma non hanno istanze proprie e le operazioni di aggiornamento di viste potrebbero non essere consentite in alcuni DBMS.
A cosa servono le viste?

- Implementare meccanismi di indipendenza tra il livello logico ed il livello esterno.
- Scrivere interrogazioni complesse, semplificandone la sintassi.
- Garantire la retro-compatibilità con precedenti versioni dello schema del DB, in caso di ristrutturazione dello stesso.

In generale, l'aggiornamento di una vista è un'operazione molto delicata, ed è consentita solo in un sottoinsieme (limitato) di casi.
In molti DBMS commerciali, non è consentito l'aggiornamento di viste che sono ottenute da più di una tabella.
L'opzione **with check option** consente di definire viste aggiornabili, a condizione che le tuple aggiornate continuino ad appartenere alla vista (in pratica, la tupla aggiornata non deve violare la clausola WHERE).
Una vista può essere costruita a partire da altre viste dello schema (viste derivate).
È possibile l'aggiornamento di viste derivate?

- with **local** check option: Il controllo di validità si limita alla vista corrente.
- with **cascade** check option: Il controllo di validità si estende ricorsivamente a tutte le viste da cui la corrente è derivata.

Le **Common Table Expression** (CTE) rappresentano viste temporanee che possono essere usate in una query come se fossero una vista a tutti gli effetti A differenza di una vista, una CTE non esiste a livello di schema del DB.

```
with Nome(Attributi) as
<SQL Query>
```

La CTE è valida soltanto nella query sottostante.
Per esempio vogliamo estrarre il nome del dipartimento che ha la spesa più alta in stipendi

```
with SpeseDipartimenti (NomeDip, Spesa) as
select Dipartimento, sum(Stipendi)
from Strutturati
group by Dipartimento

select NomeDip
from SpeseDipartimenti
where Spesa = (select max(Stipendi)
               from SpeseDipartimenti
              );
```

Le **asserzioni** (SQL2) sono un costrutto per definire vincoli generici a livello di schema. Il vincolo può essere immediato o differito (ossia verificato al termine di una transazione)

```
create assertion NomeAsserzione check Condizione
```

Esempio mettiamo come stipendio massimo degli impiegati 35000

```
create table Impiegati (
    Nome varchar(20);
    Cognome varchar(20);
    Salario numeric;
    Codice smallint primary key;
);

create assertion salario_controllo
check(not exists(select * from Impiegati
                 where salario > 35000)
     );
```

3.6 Procedure, trigger, permessi

Le **stored procedures** sono frammenti di codice SQL, con la possibilità di specificare un nome, dei parametri in ingresso e dei valori di ritorno. Ogni DBMS offre estensioni procedurali e sintassi differenti.

```
create function <nome procedura>
returns <tipo ritorno>
<Lista di statement di routine SQL>
```

I **trigger** (o regole attive) sono meccanismi di gestione della base di dati basati sul paradigma ECA (Evento/Condizione/Azione)

- Evento: primitive per la manipolazione dei dati (insert, delete, update).
- Condizione: Predicato booleano.
- Azione: sequenza di istruzioni SQL, talvolta procedure SQL specifiche del DBMS.

I trigger servono a garantire il soddisfacimento di vincoli di integrità referenziale, e/o specificare meccanismi di reazione ad hoc in caso di violazione dei vincoli e specificare regole aziendali (business rules), ossia vincoli generici sullo schema della base di dati.

```
Create trigger Nome
<Modo> <Evento> on Tabella
[referencing <Referenza>]
[for each <Livello>]
[when (IstruzioneSQL)]
Istruzione/ProceduraSQL
```

- Modo può essere before o after è indica una modalità di esecuzione immediata o differita.
- Evento definisce il tipo di evento che causa il trigger tra insert, delete o update.
- In Referenza possiamo mettere variabili globali per aumentare l'espressività del trigger.
- In Livello inseriamo row se il triggere deve agire a livello di righe o statement se il trigger deve agire globalmente a livello di tabella.

Vediamo come sempio un trigger che non permette aumenti superiori al 20% dello stipendio di un impiegato

```
create trigger CheckAumento
before update of conto on Impiegato
foreach row
when (new.Stipendio > old.Stipendio * 1.2)
setnew.Stipendio = old.Stipendio * 1.2
```

SQL2/SQL3 prevede meccanismi di controllo di accesso alle risorse dello schema del DB. Di default ogni risorsa appartiene all'utente che l'ha definita. Su ciascuna risorsa sono definiti dei privilegi (grant).
Il comando **grant** consente di assegnare privilegi su una certa risorsa ad utenti specifici. I privilegi possono essere select, delete, update, insert, ecc.

```
grant Privilegio on Risorsa/e to Utente/i
[with grant option]
```

L'opzione with grant option consente di propagare il privilegio ad altri utenti del sistema.
Il comando **revoke** consente di revocare privilegi su una certa risorsa ad utenti specifici

```
revoke Privilegio on Risorsa/e to Utente/i
[cascade|restrict]
```

L'opzione cascade agisce ricorsivamente sui privilegi eventualmente concessi da quell'utente.

4 Progettazione di Basi di Dati

La progettazione di un DB si divide in 3 parti:

- **Progettazione concettuale**: ci si focalizza sul contenuto informativo dei dati ad alto livello di astrazione, senza focalizzarsi sull'implementazione. L'output è il modello concettuale che è indipendente dallo schema logico e dal DBMS in uso.
- **Progettazione logica**: In questa fase, si rappresenta la base di dati nello schema logico del DMBS (nel nostro caso, nello schema relazionale). Comprende la traduzione dello schema concettuale, l'ottimizzazione dello schema logico ottenuto e una volta ottenuto lo schema logico, la rimozione delle ridondanze attraverso la normalizzazione e l'analisi delle prestazioni.
- **Progettazione fisica**: In questa fase, si descrivono le strutture per la memorizzazione dei dati su memoria secondaria, e l'accesso (efficiente) ai dati.

4.1 Progettazione concettuale

La progettazione concettuale avviene modellando la realtà di dinteresse attraverso un modello concettuale. Quello che viene usato per il nostro caso è il *modello E-R*. Il modello è composto da diversi componenti.

4.1.1 Entità

Un'entità è una classe di oggetti (fatti, persone, cose) della realtà di interesse con proprietà comuni e con esistenza autonoma.
Graficamente, viene rappresentata attraverso un rettangolo (con nome dell'entità al centro).
Ad ogni entità è associato un nome, che identifica l'oggetto rappresentato, per convenzione si usano nomi al singolare.
L'istanza di un'entità è uno specifico oggetto appartenente a quell'entità (es. una specifica persona, uno specifico studente, uno specifico professore, etc)

4.1.2 Relazioni

Una relazione è un legame logico fra due o più entità, rilevante nel sistema che si sta modellando.
Graficamente viene rappresentata attraverso un rombo collegato ad entità (anche più di 2).
Ad ogni relazione è associato un nome, che la identifica nello schema e per convenzione, si usano nomi al singolare (non i verbi, se possibile).
L'istanza di una relazione è una combinazione di istanze dell'entità che prendono parte all'associazione, (es. la coppia (c,d) è un'istanza della relazione Lavoro, dove c è un'istanza di Impiegato, e d è un'istanza di Dipartimento).

In generale, una relazione può coinvolgere un numero arbitrario di entità (relazioni n-arie) o può coinvolgere più istanze della stessa entità (e si dice che la relazione è ricorsiva) e in tal caso il modello E-R consente di definire un ruolo per ciascun ramo della relazione.

4.1.3 Attributi

Un attributo è una proprietà elementare di un'entità o di una relazione del modello e ogni attributo è definito su un dominio specifico.
È possibile definire attributi composti come unione di attributi affini di una certa entità/relazione: sono rappresentati da un cerchio.

4.1.4 Cardinalità delle relazioni

La cardinalità delle relazioni è una coppia di valori (min, max) che specificano il numero minimo/massimo di occorrenze delle relazioni in cui ogni occorrenza di entità può partecipare.
Nella pratica si usano solo due valori per il minimo: 0 per indicare la partecipazione facoltativa, 1 per indicare la partecipazione obbligatoria.
Analogamente si usano solo due valori per il massimo: 1 per indicare al massimo una entità coinvolta, N per indicare che non c'è un valore massimo.
In base al valore della cardinalità massima delle entità E1 ed E2 (cardMax(E1), cardMax(E2)) coinvolte in una relazione R, si distinguono tre casi:

1. Relazioni uno-ad-uno: cardMax(E1)=1, cardMax(E2)=1.
2. Relazioni uno-a-molti: cardMax(E1)=1, cardMax(E2)=N oppure cardMax(E1)=N, cardMax(E2)=1
3. Relazioni molti-a-molti: cardMax(E1)=N, cardMax(E2)=N

La cardinalità può essere specificata anche in presenza di relazioni ricorsive con ruoli.
Se una o più entità partecipano con cardinalità massima 1 a un'associazione ternaria siamo in presenza di una "falsa ternaria" che può essere sempre modellata attraverso due binarie.

4.1.5 Cardinalità degli attributi

Come per le relazioni, anche per gli attributi è possibile definire una cardinalità minima e massima

4.1.6 Identificatori

Un identificatore è uno strumento per identificare in maniera univoca le istanze di una entità. Corrisponde al concetto di chiave nel modello relazionale (quindi deve godere del requisito di minimalità!).

Ogni entità deve avere un identificatore che può essere interno (composto da attributi dell'entità) o esterno (composto da attributi dell'entità più entità esterne).
Nell'identificatore esterno l'entità esterna deve essere in relazione (1,1) con l'entità corrente: In pratica, gli identificatori esterni servono a modellare le situazioni in cui l'istanza di un'entità ha valori univoci solo all'interno di un certo contesto, definito dalle relazioni cui partecipa l'entità.
Le relazioni sono identificate dagli identificatori delle entità che raccordano e dunque non hanno identificatore.

4.1.7 Generalizzazioni

In generale, un'entità E è una generalizzazione di E1, E2, ..., En se ogni istanza di E1, E2, ..., En lo è anche di E.
Le generalizzazioni si dividono in parziali, ossia esistono occorrenze dell'entità padre che non sono occorrenze delle entità figlie, e si indicano con una freccia vuota, e totali, ossia ogni occorrenza dell'entità padre è occorrenza di almeno una delle due figlie, e si indicano con una freccia piena.
Le gerarchie di generalizzazione non sono tipicamente utilizzate per modellare aspetti dinamici della realtà di interesse.
Attenzione a non confondere entità con entità di istanze di entità, tentando di modellare attraverso gerarchie la conoscenza di specifiche istanze. Es: ... un campeggio è diviso in tre aree (spiaggia, centrale, ingresso), ognuna delle quali è caratterizzata da una certa tariffa... non creiamo una generalizzazione Area e creiamo entità figlie che hanno tariffe dierse, ma creiamo direttamente solo l'entità Area.

4.1.8 Dizionario dei dati

Il dizionario dei dati è una tabella contenente la descrizione delle entità/relazioni del modello E-R. Comprende nome dell'entità/relazione, descrizione, attributi, identificatore.

4.1.9 Business Rules

Il diagramma E-R è uno strumento di modellazione molto potente e completo, ma non tutti i vincoli sono esprimibili nel modello, per esprimere questi vincoli, si utilizzano delle business rules (regole aziendali).
I vincoli di integrità possono essere espressi mediante asserzioni, ossia affermazioni che devono essere sempre verificate sulla base di dati, nella forma: <concetto> deve/non deve <concetti>
Una regola di derivazione specifica le operazioni (aritmetiche, logiche, etc) che consentono di ottenere un concetto derivato ed è della forma: <concetto> si ottiene <operazioni>.
Le business rules possono essere raccolte in tabelle, e devono essere allegate al diagramma E-R

4.2 Progettazione logica

L'obiettivo della progettazione logica è la realizzazione del modello logico (es. relazionale) a partire dalle informazioni del modello E-R.
La progettazione logica include due step:

- Ristrutturazione del modello concettuale: modificare lo schema E-R per semplificare la traduzione ed ottimizzare il progetto.
- Traduzione nel modello logico: traduzione dei costrutti del modello E-R nei costrutti del modello relazionale

4.2.1 Eliminazione delle generalizzazioni

Per eliminare le generealizzazioni sono presenti tre alternative:

1. Accorpamento delle entità figlie nell'entità genitore (con relativi attributi/relazione): introduce valori nulli ed un attributo aggiuntivo, ma è conveniente quando non ci sono troppe distinzioni tra entità padre ed entità figlie.
2. Accorpamento delle entità genitore nelle entità figlie (con relativi attributi/relazione): è possibile solo se la generalizzazione è totale, introduce valori nulli, ma è conveniente quando ci sono operazioni che coinvolgono solo le entità figlie, ma non l'entità genitore.
3. Sostituzione delle generalizzazione con relazioni tra entità genitore ed entità: non introduce valori nulli, ed è utile quando ci sono operazioni che si riferiscono solo ad entità figlie e del padre, ma si presenta la necessità di introdurre dei vincoli:
    - Un'occorrenza del padre non può partecipare in contemporanea a quelle figlie
    - Se la generalizzazione è totale, ogni occorrenza padre deve appartenere ad almeno una figlia.

4.2.2 Eliminazione degli attributi multi-valore

Gli attributi multivalore non sono presenti nel modello logico, ma possono essere sostituiti introducendo una relazione uno-a-molti

4.2.3 Partizionamento/accorpamento di concetti

Per un dato modello E-R, è possibile ridurre il numero di accessi:

- Separando attributi di un concetto che vengono acceduti separatamente ossia i *partizionamenti*.
- Raggruppando attributi di concetti diversi acceduti ossia gli *accorpamenti*.

È necessario avere indicazioni sul volume dei dati per effettuare partizionamenti/accorpamenti.
Gli accorpamenti di entità riguardano in genere associazioni uno-ad-uno (es. relazione Proprietà tra Persona ed Abitazione, notiamo che gli accessi all'entità persona riguardano i dati dell'abitazione e dunque accorpiamo di dati dell'abitazione in Persona).
Il partizionamento verticale di un'entità viene fatto sulla base dei suoi attributi (Per uno studente, le operazioni che riguardano i dati anagrafici non riguardano i dati universitari, dunque creiamo un'entità che raccoglie i dati universitari e li mettiamo in relazione con i studenti).

4.2.4 Scelta degli identificatori

Nei casi di entità con più identificatori, è necessario sceglierne uno:

- Evitare attributi con valori nulli.
- Scegliere l'identificatore minimale.
- Preferire identificatori interni ad identificatori esterni che coinvolgono molte entità.
- Preferire identificatori utilizzati da molte operazioni per l'accesso all'entità

4.2.5 Analisi delle ridondanze

Nel modello E-R, potrebbero essere presenti ridondanze sui dati, ossia informazioni significative ma derivabili da altre già presenti nel modello E-R

1. Sia S lo schema E-R senza ridondanze.
2. Sia S_{rid} lo schema E-R con ridondanze.
3. Si calcolano il costo e l'occupazione di memoria di entrambi gli schemi: $< c(S), m(S) >$ e $< c(S_{rid}), m(S_{rid}) >$.
4. Si confrontano $c(S)/c(S_{rid})$ e $|m(s) - m(S_{rid})|$.

Per effettuare l'analisi del modello E-R, è necessario disporre delle tavole dei volumi e delle operazioni:

- La tavola delle operazioni contiene le operazioni che si andranno ad effettuare e per ogni operazione abbiamo il tipo (interattivo o batch) e la frequenza.
- La tavola dei volumi contiene i concetti usati per le operazioni e per ogni concetto abbiamo il tipo (entità o relazione) e il volume (numero di istanze). Si assume che le informazioni sui volumi siano nelle specifiche dei dati.

- Per ogni operazione abbiamo la tavola degli accessi che indica quali concetti bisogna usare, per ogni concetto indica il tipo, il numero di accessi a quel concetto per quella determinata operazione e il tipo di accesso (lettura e scrittura, ci serve in quanto hanno costi diversi).

Il costo di un'operazione è dato da $c(op) = Freq * (n_W * \alpha_W + n_R * \alpha_R)$ dove n_W è la somma degli accessi in scrittura, α_W è il costo degli accessi in scrittura, n_R e la somma degli accessi in lettura, α_R è il costo degli accessi in lettura.
Per la memoria invece $m(S_{Rid}) = X + Vol * b$ dove X è lo spazio dello schema senza ridondanze, Vol è il volume del concetto in cui c'è ridondanza e b è l'occupazione di memoria in byte del campo ridondante.

4.2.6 Traduzione di entità con identificatore interno

Le entità del modello E-R si traducono in tabelle del modello relazionale e l'identificatore del modello E-R diventa la chiave primaria della tabella.

4.2.7 Traduzione di entità con identificatore esterno

Le entità con identificatore esterno si traducono in una tabella che include tra le chiavi gli identificatori dell'entità esterna.

4.2.8 Traduzione di relazioni molti-a-molti

Ogni entità diventa una tabella con lo stesso nome, stessi attributi e per chiave il suo identificatore.
Ogni relazione diventa una tabella, con gli stessi attributi e come chiave gli identificatori delle entità coinvolte

4.2.9 Traduzione di relazioni uno-a-molti

Sono possibili due traduzioni: traduco la relazione come una tabella separata (come nel caso delle relazioni molti-a-molti) oppure inglobo o la relazione nell'entità con cardinalità massima 1.

4.2.10 Traduzione di relazioni uno-a-uno

Sono possibili 3 diverse alternative, in base alla cardinalità minima delle due entità in gioco:

1. Cardinalità obbligatorie per entrambe le entità (cardMin pari ad 1 per entrambe): si traduce il modello inglobando la relazione in una delle due entità (traduzioni simmetriche).

2. Partecipazione obbligatoria per una delle entità (cardMax=1 per una delle due): si traduce il modello inglobando la relazione nell'entità che ha partecipazione obbligatoria.

3. Partecipazione facoltativa per entrambe le entità (cardMin pari a 0 per entrambe): si traduce il modello traducendo la relazione come una tabella a sé stante (analogo del caso uno-a-molti).

5 Normalizzazione

5.1 Dipendenze funzionali

Data una tabella su uno schema $R(X)$ e due attributi Y e Z di X.
Esiste la dipendenza funzionale $Y \to Z$ se per ogni coppia di tuple t_1 e t_2 di r con $t_1[Y] = t_2[Y]$, si ha anche che $t_1[Z] = t_2[Z]$. Si dice che Y determina Z in R.
Le dipendenze funzionali sono una generalizzazione del vincolo di chiave (e di superchiave) in quanto Data una tabella con schema R(X), con superchiave K, esiste un vincolo di dipendenza funzionale tra K e qualsiasi attributo dello schema R, i.e. $K \to X_1, X_1 \subseteq X$

5.2 Forma Normale di Boyce-Codd

Uno schema $R(X)$ si dice in forma normale di Boyce-Codd se per ogni dipendenza funzionale (non ovvia) $Y \to Z$ definita su di esso, Y è una superchiave di $R(X)$.
Come normalizzare una tabelle che non è in FNBC?
Creare tabelle separate per ogni dipendenza funzionale, ognuna in FNBC.
Le nuove relazioni sono ottenute mediante proiezioni sugli insiemi di attributi corrispondenti alle dipendenze funzionali e le chiavi delle nuove relazioni sono le parti sinistre delle dipendenze funzionali.
Non tutte le decomposizioni vanno bene: si dice che la decomposizione è con perdita di informazione se combinando le due tabelle della decomposizione tramite operatore di join, non ottengo la tabella di partenza.
Non tutte le tabelle possono essere normalizzate in FNBC.

5.3 Terza Forma Normale

Una tabella R è in terza forma normale se per ogni dipendenza funzionale $X \to A$ dello schema, almeno una delle seguenti condizioni è verificata:

- X contiene una chiave K di R;
- ogni attributo in A è contenuto in almeno una chiave K di R;

5.4 Algoritmo di normalizzazione (3FN)

5.4.1 Implicazione funzionale

Dato un insieme di dipendenze funzionali F, ed una dipendenza funzionale f, diremo che F implica f se ogni tabella che soddisfa F soddisfa anche f. Per esempio se prendiamo $F : \{Impiegato \to Livello, Livello \to Stipendio\}$ e $f : impiegato \to Stipendio$, F implica f in quanto in F c'è una catena di dipendenze funzionali che da Impiegato porta a Stipendio.

5.4.2 Chiusura di una dipendenza funzionale

Dato uno schema $R(U)$, con un insieme di dipendenze F.
Sia X un insieme di attributi contenuti in U. Si definisce la chiusura di rispetto ad F (X_F^+) come l'insieme degli attributi che dipendono funzionalmente da X, i.e.

$$X_F^+ = \{A \mid A \in U \text{ e } F \text{ implica } X \to A\}$$

La chiusura di un insieme di attributi indica tutti gli attributi raggiungibile da quell'insieme.
L'algoritmo per trovare la chiusura di X rispetto alle dipendenze F è il seguente

1. Poni $X_F^+ = X$;
2. Per ogni dipendenza funzionale $Y \to A$ in F, se $Y \subseteq X_F^+$ e $A \notin X_F^+$, allora $X_F^+ = X_F^+ \cup \{A\}$;
3. Ripeti il punto 2 fino a che non è possibile aggiungere elementi a X_F^+;

Tramite la chiusura si può verificare se F implica $f : X \to Y$: si calcola la chiusura X_F^+ e se $Y \in X_F^+$ allora F implica f.
L'algoritmo di chiusura può essere usato anche per verificare se un insieme di attributi è superchiave di una relazione: Dato uno schema $R(U)$, con un insieme F di dipendenze funzionali, allora un insieme di attributi K è una (super)chiave di $R(U)$ se F implica $K \to U$.

5.4.3 Insiemi di dipendenze equivalenti

Dati due insiemi di dipendenze funzionali F_1 ed F_2, essi si dicono equivalenti se F_1 implica ciascuna dipendenza di F_2 e viceversa.

5.4.4 Insiemi di dipendenze non ridondanti

Dato un insieme di dipendenze funzionali F definito su uno schema $R(U)$, esso si dice non ridondante se non esiste una dipendenza f di F tale che $F - \{f\}$ implica f.
In parole povere un insieme di dipendenze non è ridondante se ogni dipendenza funzionale è necessaria per raggiungere tutti gli attributi.

5.4.5 Insieme di dipendenze ridotto

Dato un insieme di dipendenze funzionali F definito su uno schema $R(U)$, esso si dice ridotto se

1. non è ridondante,
2. non è possibile ottenere un insieme F' equivalente eliminando attributi dai primi membri di una o più dipendenze di F.

Per esempio $F = \{A \to B, AB \to C\}$ non è ridotto perché si può eliminare B da $AB \to C$ e ottenere un insieme $F' = \{A \to B, A \to C\}$ equivalente a F.

5.4.6 Copertura Ridotta

Dato uno schema $R(U)$ con insieme di dipendenze F, per trovare una copertura ridotta di F si procede in tre passi:

1. Sostituire F con F_1, che ha tutti i secondi membri composti da un singolo attributo.
2. Eliminare gli attributi estranei ("inutili").
   - In generale, se ho una dipendenza funzionale del tipo $AX \to B$, per stabilire se l'attributo A può essere eliminato preservando l'uguaglianza, si calcola X^+ e si verifica se esso include B, se ciò fosse eliminiamo A dalla dipendenza.
3. Eliminare le ridondanze non necessarie
   - Per stabilire se una dipendenza $X \to A$ è ridondante, si calcola $X^+_{F-\{X \to A\}}$ e se contiene A, allora la dipendenza è ridondante.

5.4.7 Algoritmo di normalizzazione

Dati $R(U)$, ed un insieme di dipendenze F, l'algoritmo di normalizzazione in terza forma normale procede come segue:

1. Costruire una copertura ridotta F_1 di F.
2. Decomporre F_1 nei sottoinsiemi $F_1^{(1)}, F_1^{(2)}, ..., F_1^{(n)}$.
3. Se due o più lati sinistri delle dipendenze si implicano a vicenda, si fondono i relativi insiemi.
4. Trasformare ciascun $F_1^{(i)}$ in una relazione $R^{(i)}$ con gli attributi contenuti in ciascuna dipendenza. Il lato sinistro diventa la chiave della relazione.
5. Se nessuna relazione $R^{(i)}$ così ottenuta contiene una chiave K di $R(U)$, inserire una nuova relazione $R^{(n+1)}$ contenente gli attributi della chiave.

5.5 Prima Forma Normale

Una relazione r con schema $R(U)$ è in Prima Forma Normale (1FN) quando rispetta i requisiti del modello relazionale, ossia ogni attributo è elementare, e non ci sono righe/colonne ripetute.

5.6 Seconda Forma Normale

Una tabella con schema $R(U)$ è in Seconda Forma Normale (2FN) quando non presenta dipendenze parziali della forma $Y \to A$, dove:

- Y è un sottoinsieme proprio della chiave.
- A è un qualsiasi sottoinsieme di $R(U)$.