

DBMS

E' un **SOFTWARE PACKAGE** che facilita la creazione e gestione di un database computerizzato.

DATABASE

Collezione di dati correlati memorizzata su supporto permanente

DDL DATA-DEFINITION LANGUAGE

E' il linguaggio che ci permette di definire i dati nello schema logico. Per lo schema relazionale useremo SQL

ASSUNZIONE ①

DBMS RELAZIONALE:

Modello dei dati relazionale (rappresentato da una tabella)

I dati in tabella sono detti **METADATI**

ASSUNZIONE ②

La base di dati è **CENTRALIZZATA** e in **MULTIUTENZA**
(utenti in parallelo con utilizzo privato)

TRANSAZIONI

Sono sequenze di operazioni elementari da svolgere in maniera atomica (la sequenza viene eseguita per intero).

Se la sequenza viene interrotta verranno annullate le operazioni già svolte.

PROPRIETÀ DELLE TRANSAZIONI A.C.I.D.

• ATOMICITÀ

• CONSISTENZA

Si possono introdurre dei vincoli sui dati inseriti affinché siano coerenti con il database (Es. $1 \leq \text{giorno} \leq 31$)

• ISOLAMENTO

Lavorando sul database l'utente non percepisce le azioni degli altri utenti.

• DURABILITY

Alla fine delle transazioni i dati vanno ricoppiati in memoria e in caso di errore si deve poter ripristinare una versione consistente dei dati.

Queste sono FUNZIONALITÀ della DBMS TRANSIZIONALE

FUNZIONALITA' DBMS TRANSIZIONALE

Tra le altre funzionalità di una dbms transizionale c'è la possibilità di proteggere i dati modificando i permessi degli utenti, limitandone l'accesso.

L'affidabilità di queste funzioni determina la qualità di una BDMS

DATA MODEL

Insieme di concetti che descrivono **STRUTTURA, OPERAZIONI** e **VINCOLI** di un database.

Il modello da noi studiato è quello **RELAZIONALE**
ovvero privo di dati strutturati

SCHEMA LOGICO

È una rappresentazione astratta (tramite strutture dati e vincoli) del problema da rappresentare

Lo schema è progettato dal progettista tramite il DDL.

Esiste anche lo schema **FISICO** che dipende strettamente dal BDMS
e si appoggia al sistema operativo (tramite i file system)

Sullo stesso livello di astrazione dello schema logico si pone lo schema **CONCETUALE** che serve a costruire l'immagine della base di dati vista dall'esterno.

Queste 3 parti sono tra loro separate e l'isolamento di queste ci permette di modificarle indipendentemente l'una dell'altra.

E.s.

Immaginiamo di avere il seguente schema logico:

STUD (MATR, COGNOME, NOME, ESAME, VOTO, DATA, LODE) **METADATI**

100 ESPOSITO CIRO BD 18 ... NO **DATI**

Il seguente schema non va bene poiché presenta la duplicazione dei dati. Possiamo modificarla e **NORMALIZZARE** la tabella:

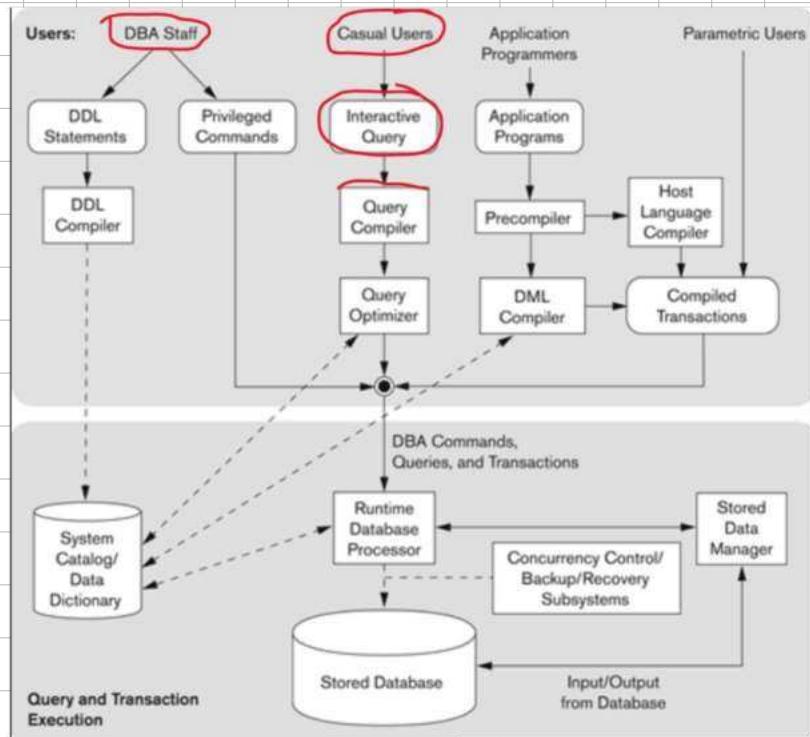
STUD1 (MATR, COGNOME, NOME)

ESAMI (MATR, ESAME, VOTO, DATA, LODE)

A questo punto però dovrà cambiare tutti gli applicativi precedentemente sviluppati per la tabella STUD.

Se i due schemi (concettuale e logico) sono isolati possiamo

creare una tabella virtuale uguale a STUD della VISTA
sulla quale si appoggeranno i miei applicativi.



MODULO DI UNA BDSSM

SERVER - CLIENT

Il SERVER mette a disposizione dei servizi sottosistema di interfacce utilizzabili dai CLIENT detti PROTOCOLLI STANDARD.

All'interno del server troviamo tutto il database e ogni applicativo gestito dalla BDSSM.

Il client anche possiede applicativi di natura diversa.

Le interfacce del BDSSM servono a comunicare con un linguaggio di programmazione

PROGETTARE UN DATA BASE

Si suddivide in 3 fasi:

- PROGETTAZIONE CONCETTUALE
- PROGETTAZIONE LOGICA
- PROGETTAZIONE FISICA

Una volta completata la parte concettuale dovranno codificare il nostro modello dei dati (relazionale)

CLASS DIAGRAM → SCHEMA RELAZIONALE → SQL

PROGETTAZIONE CONCETTUALE

In questa fase individuiamo i dati rilevanti da inserire nel database (in base alla richiesta). Si devono individuare le ENTITÀ (classi) e le ASSOCIAZIONI (legami tra le entità)

Vanno posti i VINCOLI ai dati (Es. campo MATRICOLA del record STUDENTE dev'essere unico); bisogna avere chiaro le INTERROGAZIONI che l'utente può richiedere, la FREQUENZA di queste e il NUMERO di utenti che possono interagire (per ottimizzare le prestazioni).

la caratteristica principale della progettazione concettuale è l'
INDIPENDENZA dal modello di dati e dal BDDM.

DOCUMENTAZIONE DELLA PROG. CONCETTUALE

La descrizione delle entità e delle loro associazioni utilizzeremo i
CLASS DIAGRAM oli UML.

Questi sono accompagnati da **DIZIONARI** (di vincoli, interrogazioni
e entità-relazioni)

E.s.

Si assume di dover progettare una base di dati per la gestione di un cinema multisala.
Ogni sala ha caratteristiche specifiche per le apparecchiature tecniche (schermo, proiezione, impianto
audio), per la capienza, per la dimensione. Ciascuna poltrona di ogni sala deve essere individuata
univocamente ed ha comunque indicazione di fila e posizione nella fila.

Il sistema deve gestire la programmazione dei film nel multisala. In particolare,

- La proiezione di un film in una sala
- La proiezione degli spot pubblicitari che precedono il film
- La proiezione delle anteprime dei film che possono precedere i film

Il sistema deve gestire la interazione con i clienti registrati. In particolare,

- Gestire le prenotazioni e le vendite online di biglietti
- La vendita di biglietti da botteghino

Il sistema deve permettere l'analisi dei dati storici delle vendite dei biglietti

- Incassi per proiezione e film
- Etc.

Individuare le entità

Un' **ENTITÀ** è una descrizione astratta del problema che
rappresenta un insieme d'istanze.

Associando i valori agli attributi determino in modo univoco
le istanze (in questo caso, le persone).

le entità sono rappresentate da **CLASSI**, gli attributi sono i campi di un record

ENTITÀ: CLIENTE

- ATTRIBUTI:
- NOME
 - COGNOME
 - DATA DI NASCITA
 - CODICE FISCALE
 - PASSWORD

Iniziamo a costruire il class diagram:

Creiamo la nostra classe: CLIENTE

Poniamo gli attributi: NOME, COGNOME...

Stabilisiamo i metodi (per il momento ignoriamo questa parte)

E' importante la **CARDINALITÀ** dell' attributo, ovvero quanti valori verranno associati a un attributo di un' istanza. La cardinalità è espressa attraverso una coppia (MIN, MAX):

- $(0,1)$ attributo a valore singolo **PARZIALE** (non obbligatorio)
- $(1,1) \rightarrow 1$ // **TOTALE** (obbligatorio)
- $(0,N)$ attributo a valore multiplo **PARZIALE**
- $(1,N)$ // **TOTALE**
- $(0,*) \rightarrow *$ parziale multiplo illimitato
- $(1,*)$ totale multiplo illimitato

CLIENTE	
Nome	1
Cognome	1
Email	1
DDN	$(0,1)$
Telefono	$(0,2)$
Sesso	1
Eta	$(0,1)$

} ATTRIBUTI MENSORIZZATI
ATTRIBUTO CALCOLATO

Agli attributi viene associato un tipo di dato:

string, integer, float, DATE, TIME, TIMESTAMP, ENUMERAZIONE.

Per l'enumerazione in UML cerchiamo

ENUMERATION	
S	T
H	
F	
X	

Nel class diagram avremo quindi

ETA'	(0,1)	: INTEGER
SESSO	1	: S-T

...

Gli attributi con tipo di dato semplice si chiamano **SEMPLICI**

Gli attributi con tipo di dato strutturato si chiamano **STRUTTURATI**

INDIRIZZO 1	: STRUCT (IND: STRING, NUM: INT, ...)
-------------	---------------------------------------

OSS

Per un modello relazionale i tipi più fastidiosi e che vanno segnalati sono **STRUCT** e **MULTIPLI**

ASSOCIAZIONI TRA ENTITA'

E' possibile associare due classi (entita') tra di loro.

Quest' ASSOCIAZIONE ha:

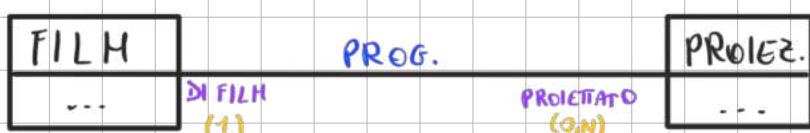
- NOME
- RUOLO (nome che qualifica la partecipazione della classe all' associazione)
- GRADO (numero di entita' coinvolte nell' associazione)
- CARDINALITA' (range di partecipazione di una classe ad una associazione)

OSS

La cardinalita' va indicata in ogni verso dell' associazione

Es.

Immaginiamo di avere le nostre entita' FILM e PROIEZIONE.

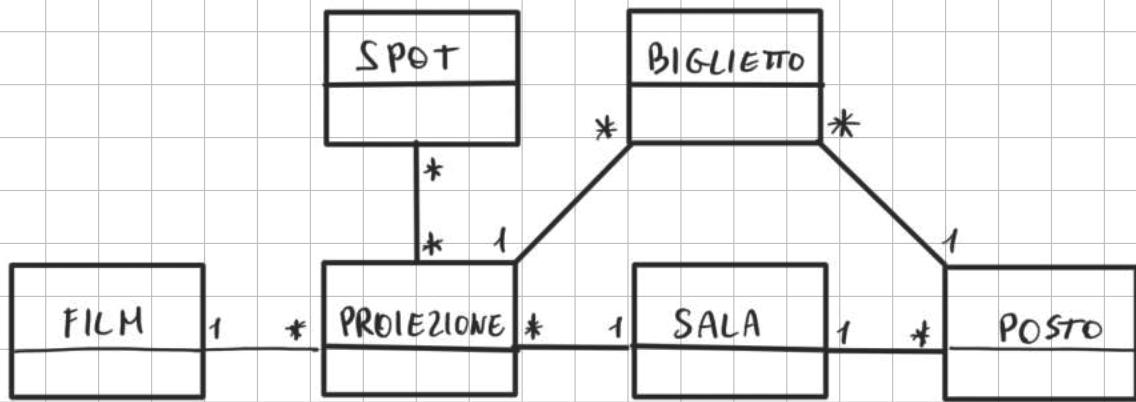


ASSOCIAZIONE BINARIA

In questo caso la cardinalita' indica numero min e max di proiezioni del film e numero di film per proiezione (1)

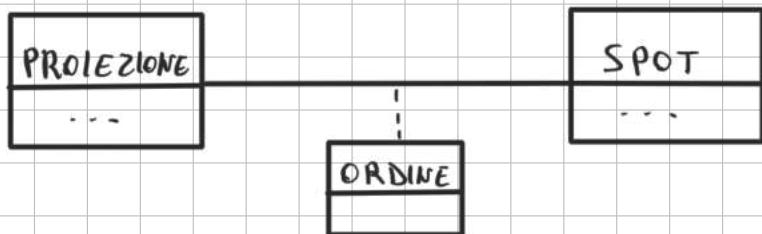
In base alla cardinalita' si definisce il tipo di associazione

Vediamo un caso più complesso:



CLASSE DI ASSOCIAZIONI

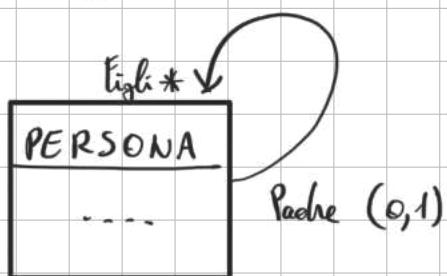
Ritornando sull'esempio precedente, immaginiamo di voler attribuire per ogni film un ordine degli spot. Non c'è un attributo ne della proiezione (perché dipende dallo spot) né dello spot (perché l'ordine dipende dalla proiezione). Sarà un ATTRIBUTO NELL'ASSOCIAZIONE, rappresentato dalla CLASSE DI ASSOCIAZIONE (che prenderà ovviamente il nome dell'associazione)



ASSOCIAZIONE RICORSIVA

E' possibile associare una classe a se stessa:

Vediamo una classe PERSONA che ha delle associazioni di parentela. Le associazioni vengono effettuate da PERSONA a PERSONA



TIPI DI ASSOCIAZIONI

In base alla **CARDINALITÀ** si ci suddivide in

- MOLTI A MOLTI
- 1 A MOLTI
- 1 A 1

Ad esempio una classe PERSONA e una PATENTE.

Nel caso di un' associazione 1-1 potremmo unire le due classi in una sola



Cio' puo' convenire o meno a seconda della richiesta:

- ① Anagrafica
- ② N° di iscritti al corso di laurea in INF.

In questo caso conviene tenere separate le due entità perché la richiesta non richiede per forza l'utilizzo di entrambe le entità.

- ③ N° studenti c.d.l. INF nati a NAPOLI

In questo caso **SEMPRE** verranno richieste informazioni

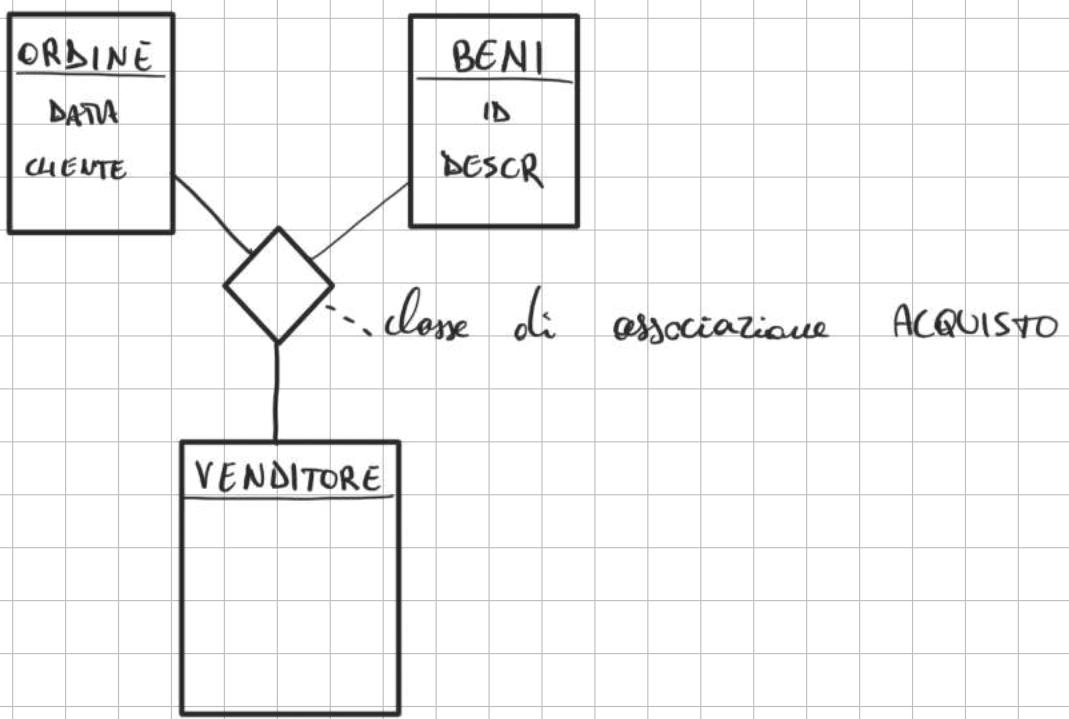
da entrambe le entità

OSS

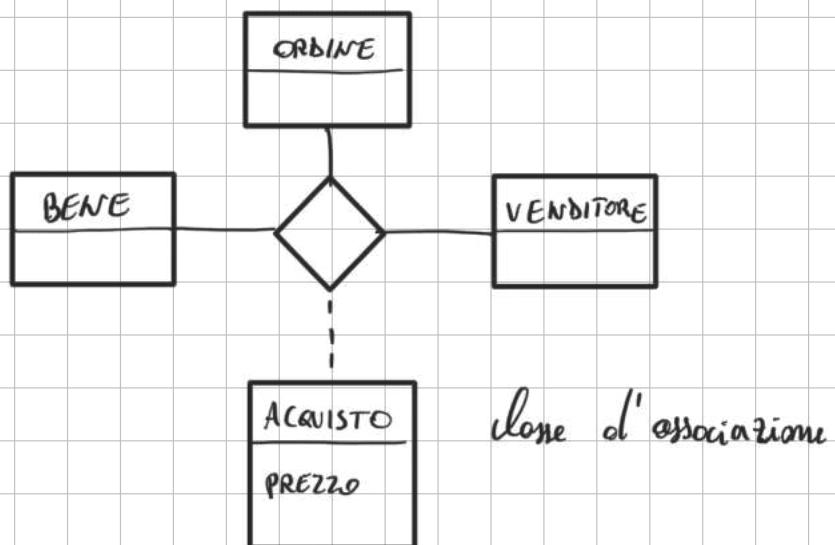
Tabelle più grandi implicano una copia di pagine più grandi
dalla memoria secondaria alla primaria del database

ts.

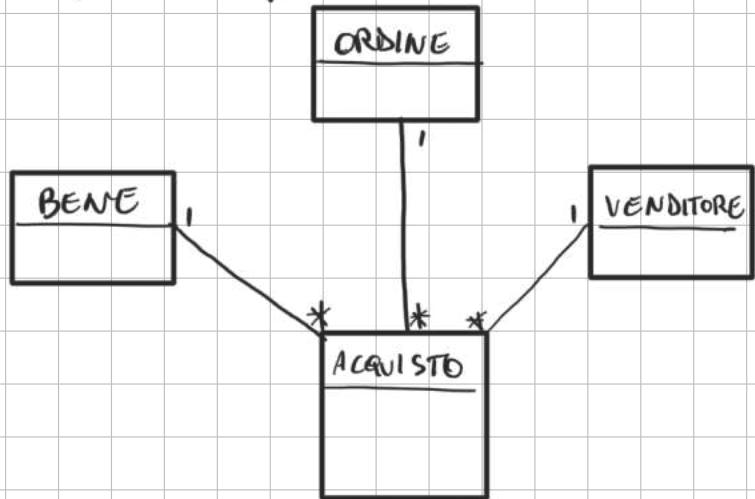
Richiesta: ordini di materiale a fornitori



Se noi vogliamo aggiungere un attributo che definisca il numero di elementi che voglio acquistare non possiamo inserirlo in alcuna classe che non sia proprio ACQUISTO



In alternativa possiamo definire una vera e propria classe

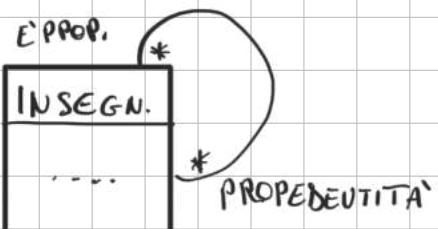


Nel caso di associazioni non binarie (≥ 3) l'espressione della cardinalità risulta inutile perché non mi dà informazioni aggiuntive rilevanti.

In uno schema concettuale si preferisce la prima.

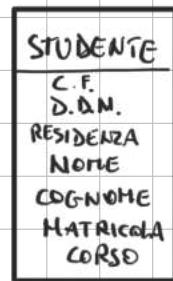
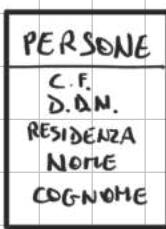
OSS

Ricordiamo un' associazione di tipo ricorsivo



SPECIALIZZAZIONI e GENERALIZZAZIONI

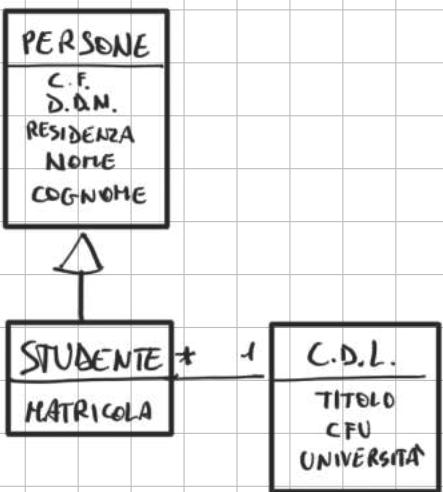
Immaginiamo di avere due classi in cui una e' un caso particolare
dell' altro



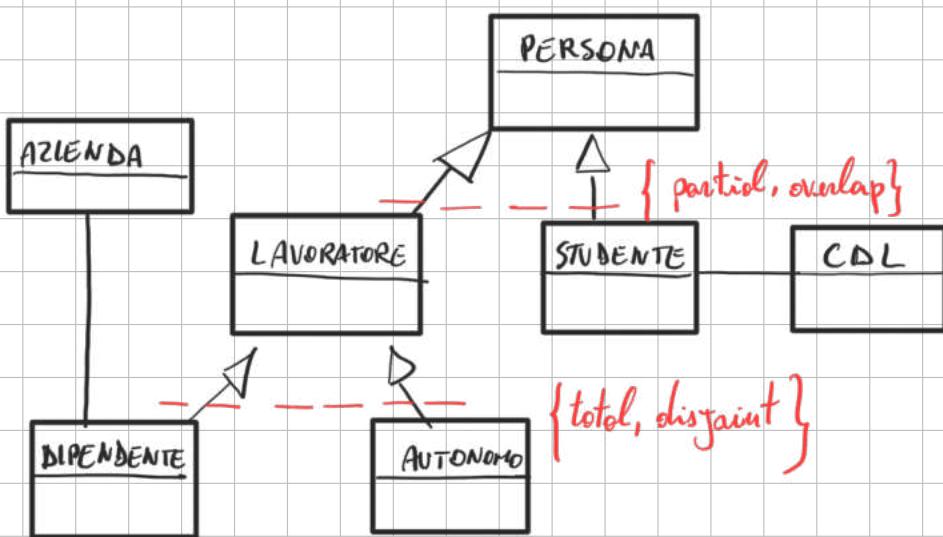
La classe STUDENTE e' una **SPECIALIZZAZIONE** di PERSONA

possedendo tutti i suoi attributi + altri aggiuntivi.

PERSONA e' la **GENERALIZZAZIONE** di STUDENTE



Ogni istanza della SPECIALIZZAZIONE è istanza della GENERALIZZAZIONE. NON vale il contrario.



TIPI DI SPECIALIZZAZIONI

Si suddividono in :

- **TOTALI**

Se per ogni istanza della classe generale esiste una istanza di una specializzazione che lo specializza. Viceversa sarà **PARZIALE**

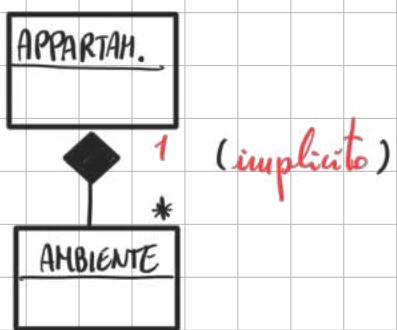
● DISJOINT

Se ogni istanza della generale e' specializzata al piu' da un' specializzazione. Altrimenti si dico' OVERLAP

Nel caso precedente la specializzazione di persona e' PARZIALE-OVERLAP mentre la specializzazione di LAVORATORE e' TOTALE-DISJOINT

COMPOSIZIONE e AGGREGAZIONE

Se abbiamo una classe che puo' essere "divisa in parti" attraverso una COMPOSIZIONE



I vincoli sono:

- Un ambiente e' legato ad un appartamento
- Un ambiente NON puo' esistere senza appartamento

OSS

Si potrebbe definire AMBIENTE come associazione, va bene ma
e' un legame molto meno forte visto che mancano quei vincoli
impliciti che la composizione ha per definizione

Una forma più attenuata della composizione è l'**AGGREGAZIONE**
che non presenta i vincoli

DOCUMENTAZIONE PROGETTAZIONE CONCETTUALE

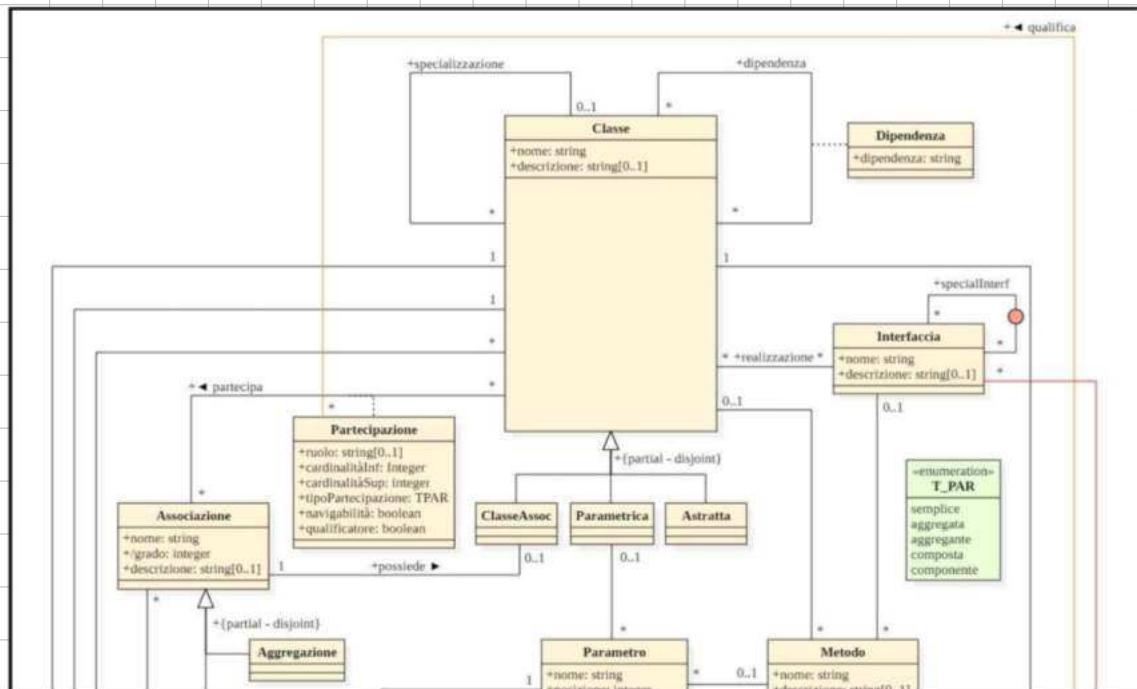
Per le entità e le associazioni usiamo i CLASS DIAGRAM di UML.

Questi vengono utilizzati nel linguaggio di descrizione grafico E-R.

Ai class diagram vengono associati DIZIONARI, in particolare avremo quello delle entità-associazioni, dei ruoli e delle interazioni e della loro frequenza.

Vediamo l'esempio di uno schema concettuale di un class diagram:

CLASS DIAGRAM



DIZIONARIO DELLE CLASSI

2.4.1 | Dizionario delle classi

utente

Classe	Descrizione	Attributi
ClassDiagram	Descrittore di un Class Diagram UML.	ID_ClassDiagram (integer): Chiave tecnica. Identifica univocamente ciascuna istanza di ClassDiagram. Nome (string): Nome associato al Class Diagram. Autore (string): Nome dell'autore del Class Diagram. Data (date): Data della creazione del Class Diagram. Descrizione (string): Descrizione facoltativa del Class Diagram.
Classe	Descrittore di una classe appartenente ad un Class Diagram.	ID_Classe (integer): Chiave tecnica. Identifica univocamente ciascuna istanza di Classe. Nome (string): Nome associato alla classe. Descrizione (string): Descrizione facoltativa della classe. Tipo (<i>T_CLASS</i>): Indica il tipo della classe.
Dipendenza	Descrittore di Dipendenza	Dipendenza (string): Indica il tipo di dipen-

N.B. In quest'esempio mancano informazioni sulla totalità e parzialità.

● NOME DELLA CLASSE

● DESCRIZIONE DELLA CLASSE

● ELENCO DEGLI ATTRIBUTI CON DESCRIZIONE

DIZIONARIO DELLE ASSOCIAZIONI

2.4.2 | Dizionario delle associazioni

Associazione	Descrizione	Classi coinvolte
Specializzazione	Esprime una relazione gerarchica che intercorre tra due classi.	Classe [0..1] ruolo (generale) : Indica la classe generalizzata. Classe [0..*] ruolo (specializzata) : Indica le classi che specializzano una classe.
Dipendenza	Esprime una relazione di dipendenza che intercorre tra due classi.	Classe [0..*] ruolo (superiore) : Indica le classi da cui ne dipendono altre. Classe [0..*] ruolo (dipendente) : Indica le classi che dipendono da altre.
Realizzazione	Esprime l'implementazione di un'interfaccia da parte di una classe.	Classe [0..*] ruolo (implementa) : Indica le classi che implementano le interfacce. Interfaccia [0..*] ruolo (implementata) : Indica le interfacce implementate.
SpecialInterf	Esprime una relazione gerarchica che intercorre tra due interfacce.	Interfaccia [0..*] ruolo (generale) : Indica le interfacce generalizzate. Interfaccia [0..*] ruolo (specializzata) : Indica le interfacce che ne specializzano altre.

● NOME DELL' ASSOCIAZIONE

● DESCRIZIONE DELL' ASSOCIAZIONE

● ELENCO DELLE CLASSI ASSOCIATE E CARDINALITÀ

● RIFERIMENTO AD EVENTUALI CLASSI DI ASSOCIAZIONE E LORO ATTRIBUTI

(manca nell'immagine)

DIZIONARIO DEI VINCOLI

● NOME DEL VINCULO

● DESCRIZIONE DEL VINCULO

N.B.

NON verranno inseriti vincoli già espressi dal class diagram (es. totalità degli attributi, cardinalità delle associazioni, disgiunzione e sovrapposizione delle generalizzazioni, vincoli di obbligo). Verranno indicate inoltre le regole di calcolo degli attributi calcolati.

Vincolo	Tipo	Descrizione
isValidLiteral	Interrelazionale	L'associazione tipo contiene ([Tipo - Literal]) risulta valida solo se il tipo è "Enumerazione".
isValiddRefClass	N-upla	L'associazione riferisce classe ([Classe - Tipo]) risulta valida solo se il tipo è "Classe".
uniqueClassName	Intrarelazionale	Il nome di una classe deve essere univoco nel Class Diagram di appartenenza.
uniqueAssocName	Intrarelazionale	Il nome di un'associazione deve essere univoco nel Class Diagram di appartenenza.
uniqueAttrName	Intrarelazionale	Il nome di un attributo deve essere univoco nella classe di appartenenza.
uniqueCDName	Intrarelazionale	Il nome di un Class Diagram deve essere univoco.
uniqueQualif	Intrarelazionale	Un attributo può qualificare una partecipazione una ed una sola volta.

DIZIONARIO DELLE INTERROGAZIONI

- NOME DELL' INTERROGAZIONE + DESCRIZIONE
- FREQUENZA DELLE INTERROGAZIONI
- DISPONIBILITÀ DEGLI ATTORI DELL' INTERROGAZIONI

MODELLO DEI DATI RELAZIONALI

Il concetto fondamentale per questo modello dei dati è lo **SCHEMA RELAZIONALE**.

$$R(A_1, A_2, \dots, A_N)$$

↑ ↑
NOME ATTRIBUTI

Per ogni attributo A_i indico l'insieme dei valori che A_i può assumere
(DOMINIO)

E.

DATE (GIORNO, MESE, ANNO)

$$\text{dom}(\text{GIORNO}) = \{1, \dots, 31\}$$

$$\text{dom}(\text{MESE}) = \{1, \dots, 12\}$$

$$\text{dom}(\text{ANNO}) = \{1900, \dots, 2100\}$$

RELAZIONE

Fissato uno schema relazionale $R(A_1, \dots, A_m)$ con $\text{dom}(A_i)$ fissati, una RELAZIONE r su $R(A_1, \dots, A_m)$ è un sottoinsieme del prodotto cartesiano:

$$r \subseteq \text{dom}(A_1) \times \text{dom}(A_2) \times \dots \times \text{dom}(A_m)$$

E.s.

Sia lo schema relazionale DATE(GIORNO, MESE, ANNO)

Rappresentiamo graficamente DATE

GIORNO	MESE	ANNO

Per definizione di prodotto cartesiano, r è rappresentato da una TABELLA cui ogni RIGA rappresenta un elemento.

La più piccola relazione è $r = \emptyset$, la più grande è $r_m = \text{dom}A_1 \times \dots \times \text{dom}A_m$
nel nostro caso sarà $r = \{1, \dots, 31\} \times \{1, \dots, 12\} \times \{1900, \dots, 2100\}$

NON c'è molteplicità dell'istanza (gli insiemi sono unici)

PARZIALITA' DEGLI ATTRIBUTI

Come possiamo rappresentare la PARZIALITA' di un attributo?

Introduciamo il valore **NULL** indicando che il dato è inesistente o non è disponibile.

$$\text{dom}(\text{GIORNO}) = \{\text{NULL}, 1, \dots, 31\}$$

OSS

Lo schema relazionale ha due fonti limitazioni:

- Gli oggetti del dominio sono oggetti **NON STRUTTURATI**, ovvero non hanno valore, verranno manipolati come stringhe.
- In un elemento del prodotto cartesiano ad ogni attributo è associato un **UNICO** valore

CODIFICARE LE ASSOCIAZIONI

Prendiamo una classe S con attributi A_1, \dots, A_n



Associando il valore agli attributi individuo in maniera univoca l'istanza. Vediamo i concetti chiave per definire le associazioni.

SUPERCHIAVE per $S(A_1, \dots, A_n)$

E' un sottinsieme di $\{A_1, \dots, A_n\}$ che mi consente di individuare in modo univoco un'istanza.

Ese.

STUDENTE (C.F., NOME, COGNOME, DATA DI NASCITA, RESIDENZA, SESSO, MATRICOLA)

$\{\text{NOME}, \text{COGNOME}\}$ **NON** e' una superchiave

$\{\text{C.F.}, \text{NOME}, \text{COGNOME}\}$ **E'** una superchiave

$\{\text{C.F.}\}$ **E'** una superchiave

$\{\text{C.F.}, \text{NOME}, \text{COGNOME}, \text{DATA DI NASCITA}, \text{RESIDENZA}, \text{SESSO}, \text{MATRICOLA}\}$ **E'** una superchiave

PROPRIETA' DELLE SUPERCHIAVI

- Esiste **SEMPRE** una superchiave (dunque quella totale)
- La superchiave **NON** è unica
- Conviene prendere la **SUPERCHIAVE MINIMALE** (superchiave con il numero minore di attributi)

CHIAVE CANDIDATA

Sono Superchiavi minimali.

CHIAVE PRIMARIA

È una chiave **SCELTA** tra le chiavi primarie.

Non esiste uno specifico criterio di scelta, purché sia giustificata.

La chiave primaria funge da riconoscitivo compatto delle istanze, perciò è detta anche **IDENTIFICATIVO**

CHIAVE ESTERNA

Dato lo schema $S(A_1, \dots, A_m)$ una chiave esterna è un sottoinsieme degli attributi in corrispondenza biunivoca con una chiave primaria di un altro schema ($\circ S$ stesso)

E.

- STUDENTE (CF, NOME, COGNOME, DATA DI NASCITA, RESIDENZA, SESSO, MATRICOLA)

PRIMARY KEY (MATRICOLA)

- ESAMI (DATA , VOTO, CORSO, LODE, MATRICOLA)

PRIMARY KEY (CORSO, MATRICOLA)

FOREIGN KEY (MATRICOLA)

OSS

- La chiave esterna **NON** deve per forza avere lo stesso nome, può invece avere stesso dominio per attributi corrispondenti e stesso numero di attributi
- Gli attributi della chiave primaria devono essere **TOTALI**

DA SCHEMA CONCETTUALE A RELAZIONALE

Per passare dai class diagram al nostro schema relazionale
dobbiamo pone delle restrizioni:

- Non ci sono gerarchie di **GENERALIZZAZIONE-SPECIALIZZAZIONE**
- Non ci sono attributi **MULTIPLI** nelle entità
- Non ci sono attributi **STRUTTURATI** nelle entità

In particolare il passaggio è:



CONIFICA DELLE ENTITA' (CLASSI)

① CLASSE

C
A ₁
⋮
A _m

SCHEMA LOGICO

$$C(A_1, \dots, A_m)$$

$\text{dom}(A_1) \dots \text{dom}(A_m)$ dipende dai tipi
di A_1, \dots, A_m

② Individuare la chiave primaria

Questo passaggio è necessario **SOLO** se l'entità (classe) è coinvolta in un' associazione

CODIFICA DELLE ASSOCIAZIONI

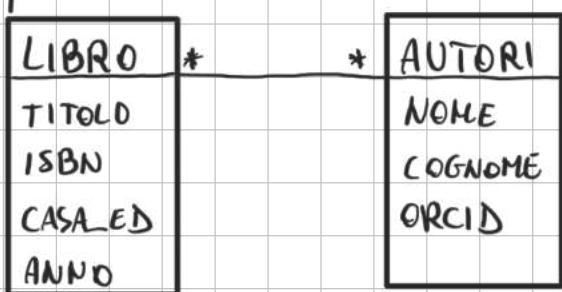
Questa dipende dalla **CARDINALITÀ** delle associazioni:
cardinalità diverse implicano codifiche diverse.

Suddividiamo le associazioni in

- MOLTI - A - MOLTI
- UNO - A - MOLTI
- UNO - A - UNO

MOLTI - A - MOLTI

Prendiamo un esempio:



PRIMARY KEY (ISBN)

PRIMARY KEY (ORCID)

● CODIFICA DELLE ENTITA'

LIBRO (TITOLO, ISBN, CASA_ED, ANNO)

AUTORI (NOME, COGNOME, ORCID)

● CODIFICA DELL' ASSOCIAZIONE

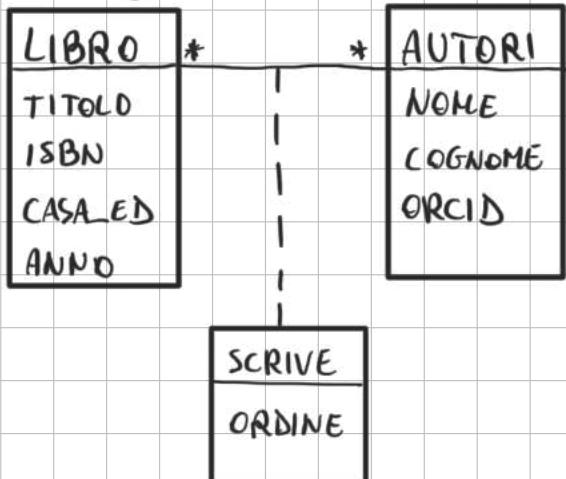
Introduciamo un nuovo schema relazionale per l'associazione

SCRIVE (ISBN, ORCID)

Indichiamo con la doppia sottolineatura le chiavi esterne
e le colleghiamo alle entità della classe alla quale fanno
riferimento.

NON c'è bisogno di individuare una chiave primaria in SCRIVE
(ipoteticamente sarebbe {ISBN, ORCID} ma è inutile)

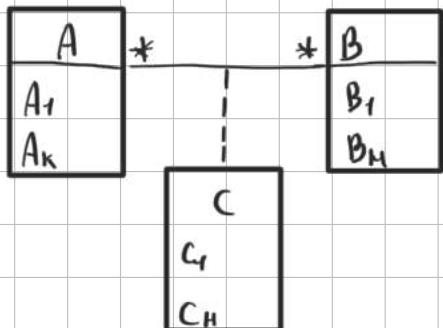
Aggiungiamo una classe di associazione



SCRIVE (ORDINE, ISBN, ORCID)

La classe di associazione
prende le PK delle due classi
più i propri attributi nello
schema relazionale

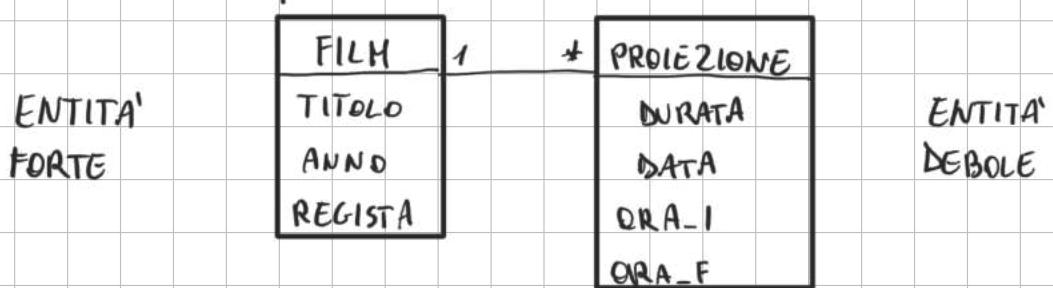
Generalizzando



$A(A_1, \dots, \underline{A_R}, \dots, A_k)$ PK (A_1, \dots, A_R)
 $B(\underline{B_1}, \dots, B_s, \dots, B_m)$ PK (B_1, \dots, B_s)
 $C(\underline{A_1}, \dots, \underline{A_R}, \underline{B_1}, \dots, \underline{B_s}, C_1, \dots, C_k)$

UNO - A - MOLTI

Partiamo da un esempio



Individuiamo le primary key dell'entità che partecipa con molti
 (detta entità forte, l'altra è debole perché dipende strettamente dalla forte)

In questo caso (TITOLO, ANNO)

● CODIFICA DELLE ENTITA'

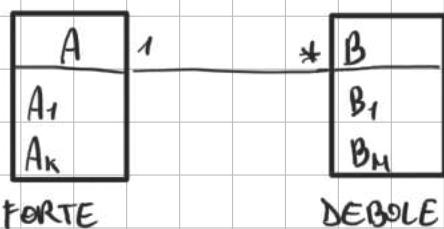
L'entità debole è codificata dai suoi attributi + le chiavi primarie
 dell'entità forte. L'entità forte è codificata come visto prima

FILM (TITOLO, ANNO, REGISTA)

PROIEZIONE (DURATA, DATA, ORA-I, ORA-F, TITOLO, ANNO)

Anionente TITOLO e ANNO saranno chiavi esterne per PROIEZIONE

Generalizzando



$A(A_1, \dots, A_R, \dots, A_k)$

$$A_1, \dots, A_R = PK_A$$

$$B(B_1, \dots, B_{H_1}, \underline{A_1, \dots, A_{R_1}})$$

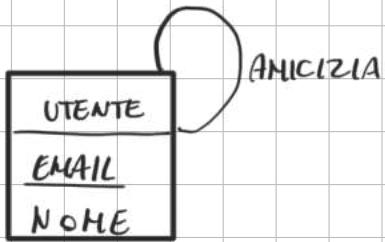
- ① Schema per la forte
 - ② Schema per la debole + PK della forte

Esempio di uno-a-multi ricorsiva



PERSONA (CF, NOME, COGNOME } CF PADRE)

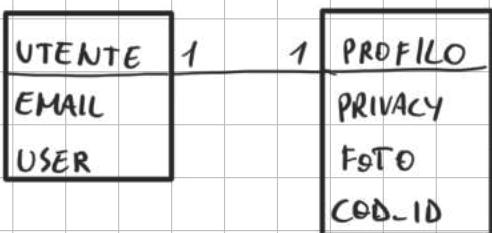
Esempio multi-a-multi ricorsivo:



UTENTE (EMAIL, NOME)

AMICIZIA (AMICO1, AMICO2)

UNO-A-UNO



Abbiamo 4 possibilità:

① UTENTE (EMAIL, USER)

(PROFILO debole)

PROFILO (PRIVACY, FOTO, EMAIL, COD-ID)

② UTENTE (EMAIL, USER, COD-ID)

(UTENTE debole)

PROFILO (PRIVACY, FOTO, COD-ID)

③ UTENTE (EMAIL, USER, COD-ID)

(entrambi deboli)

PROFILO (PRIVACY, FOTO, COD-ID, EMAIL)

④ UTENTE - PROFILO (EMAIL, USER, PRIVACY, FOTO, COD-ID)

Questa soluzione è accettabile **SOLO** nelle uno-a-uno

OSS

Nel caso della molti a molti avremmo un altro modo di esprimere l'associazione



Lo schema relazionale sarà lo stesso

FILM (TITOLO, ANNO, REGISTA)

ATTORE (NOME, COGNOME, DATA-N)

RECITAZIONE (RUOLO, PERSONAGGIO, TITOLO, ANNO, COGNOME, DATA-N)

ADATTARE LO SCHEMA CONCETTUALE

Ricordiamo che per lo schema relazionale non sono ammesse:

- GERARCHIE
- ATTRIBUTI MULTIPLI
- ATTRIBUTI STRUTTURATI

Vediamo come tradurre schemi concettuali che presentano queste caratteristiche (perfettamente ammesse) nel nostro schema logico relazionale

ATTRIBUTI STRUTTURATI

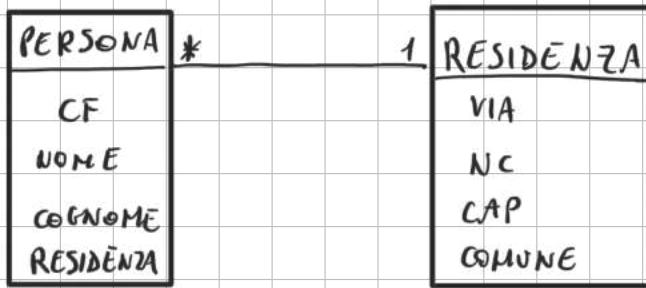
Ci sono 3 alternative:

PERSONA
CF
NOME
COGNOME
RESIDENZA

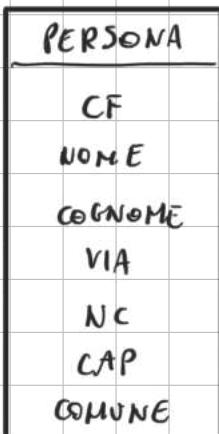
RESIDENZA

- VIA
- NC
- CAP
- COMUNE

- ① Introdurre una classe per l'attributo strutturato



② Eseguire gli attributi della classe con i campi dell'attributo strutturato



③ Trascurare la struttura interna

In questo caso abbiamo due alternative

- Considerare l'attributo come un unico tipo.
- Usare una codifica della struttura

Attenzione nella stesura della codifica, e' compito del DDL

estrapolare i dati da questa.

OSS

La prima soluzione e' molto meno vincolante, la seconda soluzione

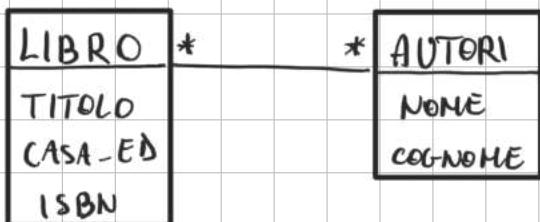
conviene se le interrogazioni richiedono piu' o meno sempre determinati dati

ATTRIBUTI MULTIPLI



Il modello relazionale ammette un solo valore per attributo

- ① Creare un'entità esterna associata



- ② Trattare l'attributo multiplo come singolo

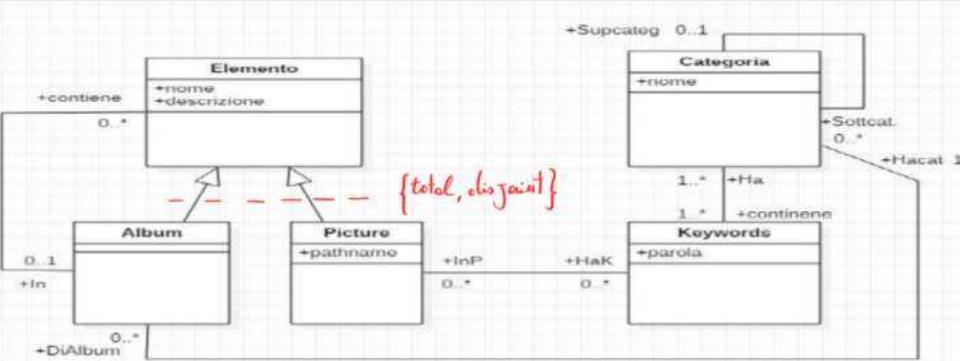
Questa soluzione non è ottimale se dobbiamo effettuare una ricerca all'interno dell'attributo (una ricerca tra gli autori in questo caso)

- ③ Replicare l'attributo nella classe

ci si presenta il problema di dover specificare quanti campi inserire per l'attributo multiplo.

In questo caso sceglieremo un numero ragionevole di campi da inserire, o scartiamo quest'alternativa.

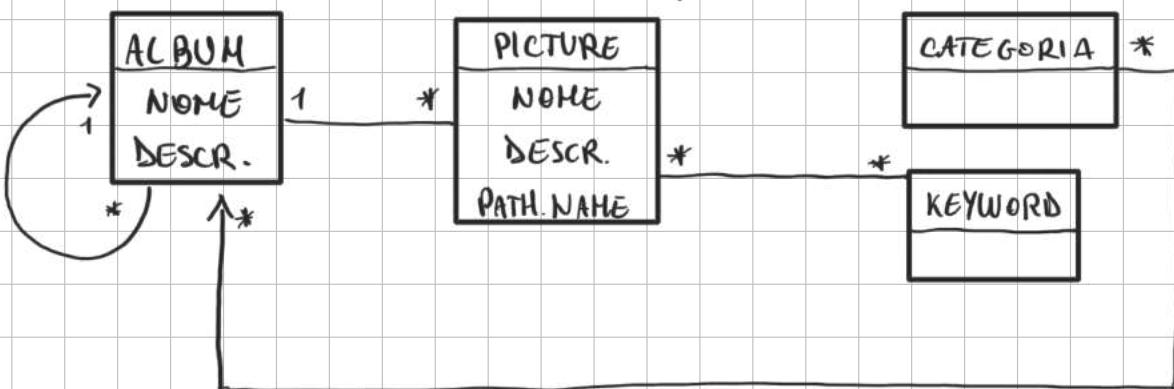
GERARCHIE (RISTRUTTURAZIONE)



Vediamo come rimuovere le gerarchie di specializzazione, le soluzioni dipendono dal tipo di specializzazione

① Compressione della generale sulla specializzazione

Questa soluzione vale **SOLO** per specializzazioni **TOTALI**



Dobbiamo fare attenzione a ricadutare le associazioni.

② Compressione della specializzazione sulla generalizzazione



TIPO è detto **ATTRIBUTO DISCRIMINANTE** e sarà sempre un'enumerazione. Bisogna aggiungere alla generalizzazione gli attributi delle specializzazioni.

Si osservi che vorremo posti dei **VINCOLI DI CONSISTENZA** in quanto gli attributi che prima erano totali possono essere ora parziali (in questo caso PATH_NAME è NULL se e solo se TIPO=PICTURE)

Questa tecnica va bene per **TUTTE** le specializzazioni



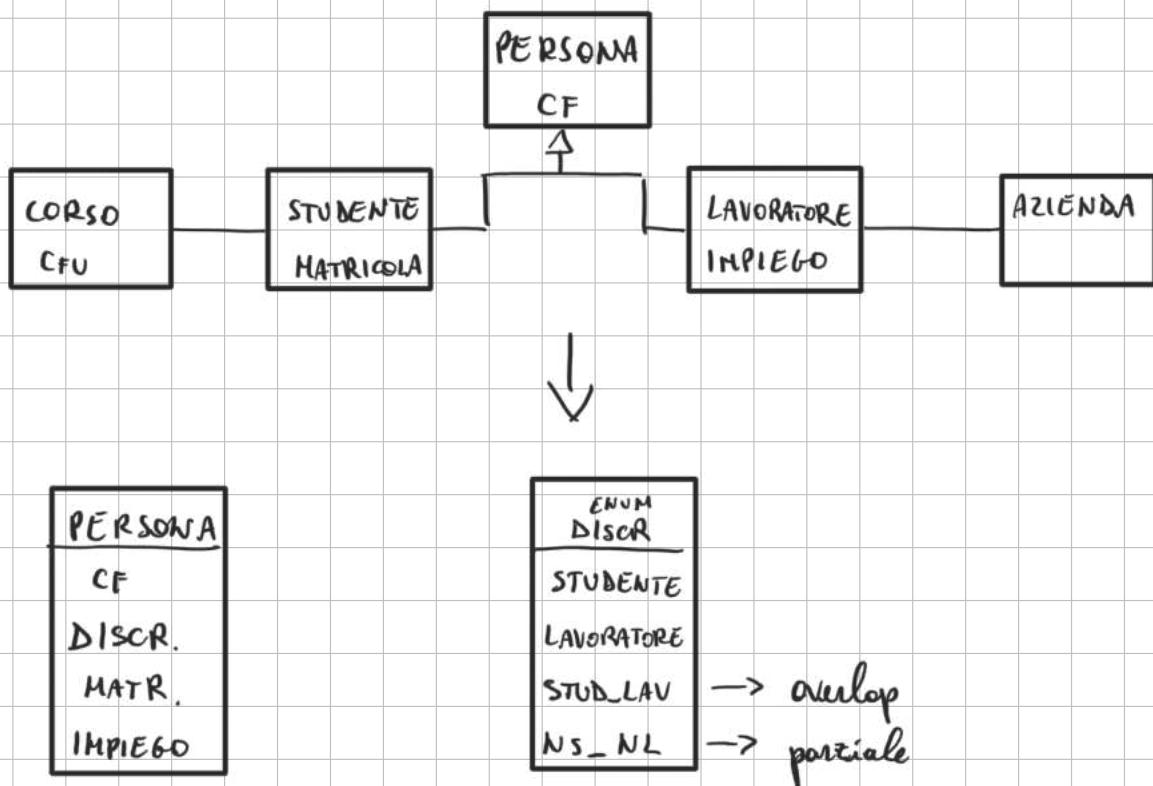
Anche per le associazioni bisogna pone dei vincoli di consistenza perché le associazioni non sono sempre determinanti:

L'associazione ELEMENTO - KEYWORD solo se TIPO=PICTURE e

ELEMENTO-CATEGORIA solo se TIPO=ALBUM

OSS

Analogamente vale per una specializzazione in overlapping

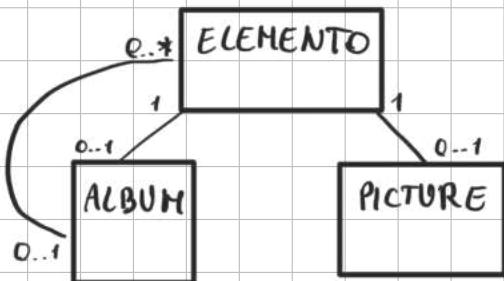


Ovviamente andranno posti numerosi vincoli di consistenza

(se **DISCR**=**NS_NL** allora **HATR=NULL**, **IMPIEGO=NULL**, **CFU=NULL** ...)

sia per gli attributi che per le associazioni

③ Rimpiazzare le specializzazioni con associazioni



Anche questa soluzione si puo' applicare a **TUTTI** i casi

Dobbiamo aggiungere un vincolo che mantenga la **TOTALITA'** della generalizzazione.

- **VINCOLO TOTALITA'**: un ELEMENTO e' associato ad un album o una picture
- **VINCOLO DISGIUNZIONE**: un ELEMENTO puo' essere sia album che picture

OSS

la terza soluzione e' la piu' conservativa (strutturalmente) ma la meno elegante (perche' si perde concettualmente la relazione tra gerarchie e specializzazioni). Viceversa la seconda soluzione.

Si possono adottare soluzioni diverse su livelli di gerarchia distinti

CHIAVE SURROGATA

Se la chiave primaria di una classe ha troppi attributi possiamo introdurre, in fase di ristrutturazione, una

CHIAVE SURROGATA ovvero una chiave identificativa che non ha valore dal punto di vista reale (es. CodConferenza)

RIASSUNTO RISTRUTTURAZIONE SCHEMA CONCETTUALE

① Analisi delle chiavi primarie

- Introdotte se serve esprimere chiavi esterne
- Se gli attributi sono eccessive si valuti l'introduzione di chiavi surrogate (1 attributo)

② Analisi delle ridondanze

- Attributi calcolati
- Associazioni ridondanti per transitività

E.s.



CARRIERA è una classe ridondante perché tutti i suoi

attributi sono calcolabili tramite l'associazione STUD-ESAMI
(di conseguenza anche CARRIERA-ESAMI è ridondante)

OSS

Associazioni ridondanti difficilmente possono avere un fine,
invece classi ridondanti possono tornare utili a seconda delle
interrogazioni

③ Associazioni 1-1

Può essere utile accoppare le classi, dipende dalle interrogazioni

- Le interrogazioni implementano sempre entrambe le classi (acoppo)
- Le interrogazioni non sempre richiedono entrambe le classi (non acoppo)

④ Trattamento attributi strutturati

⑤ Trattamento attributi multipli

⑥ Rimozione gerarchie

SCHEMA CONCETUALE → SCHEMA RISTRUTTURATO → PROGETTAZIONE
+ VINCOLI LOGICA
(SCHEMA RELAZIONALE)

SQL - DDL

Partendo da uno schema relazionale costruiremo una TABLE

le funzioni collegate ad una Table sono:

- CREATE definizione dei metadati

- ALTER modifica i metadati

- DROP cancella metadati

Per le interrogazioni useremo il costrutto **SELECT**

DEFINIZIONE TABELLA

Dobbiamo avere una struttura + vincoli

STUDENTI (MATRICOLA, CF, DATAN, NOME, COGNOME)

ESAMI (MATRICOLA, CORSO, DATA, VOTE, LODE)

CREATE TABLE < nome.tabella >

(

< definizioni attributi >

↳ < nome_attributo > < tipo_attributo > [vincoli]

)

Per le stringhe a lunghezza fissa utilizziamo CHAR(n) per quelle a lunghezza variabile avremo VARCHAR(max n)

Si osservi che dove possibile si preferiscono quelle a lunghezza fissa per una migliore gestione interna.

Esempio:

CREATE TABLE Studente

(matricola CHAR(8),
cf CHAR(16),
nome VARCHAR(20),
cognome VARCHAR(20),
dataN DATE)

CREATE TABLE Esame

(matricola CHAR(8),
corso VARCHAR(20),
voto INTEGER,
laurea BOOLEAN,
data DATE)

DOMAIN

È possibile creare un **DOMINIO** per un attributo (ad esempio per le enumerazioni)

E.

● CREATE DOMAIN Sesso AS CHAR(1) ← creazione del dominio "sesto"

CHECK Sesso='M' OR Sesso='F' ← restrizione sul dominio

● CREATE DOMAIN voto-e AS SMALLINT

CHECK voto-e >= 0 OR voto-e <= 30

OSS

CHECK introduce un **VINCOLO**.

VINCOLI

Definendo un attributo possiamo porre i seguenti vincoli:

- DI TOTALITÀ
- DI DOMINIO
- VALORE DI DEFAULT
- PRIMARY KEY
- VINCOLO DI UNICITÀ

VINCOLO DI TOTALITA'

Il vincolo di totalità si esprime con la definizione di un default value

<vincolo> NOT NULL (totale)
NULLABLE (NULL è default value, quindi è possibile)

VINCOLO DI DOMINIO

<vincolo>:= CHECK <espressione booleana> (l'espressione nomina solo l'attributo)

Può controllare il vincolo di dominio basta verificare il valore dell'attributo

VINCOLO DI CHIAVE PRIMARIA

Il controllo di questo vincolo è molto più pesante perché dev'essere controllato ogni attributo affinché non vi sia un'altra PK perché è UNICA (è implicito il vincolo di unicità sull'attributo).

Se la chiave primaria è costituita da più attributi NON possono utilizzare questo metodo.

cio' che fra inoltre e' costruire una **STRUTTURA DI INDICE**
(una struttura ad albero utilizzata per agevolare la verifica dell'
unicita della PK) sull' attributo. Non e' visibile ma e'
importantisima, e' grande circa il 20% della Tabella Totale.

<attributo> PRIMARY KEY

VINCOLO DI UNICITA'

<attributo> UNIQUE

OSS

Nei vincoli PRIMARY KEY e UNIQUE e' implicito il vincolo di **TOTALITA'**

CONSTRAINT

Dopo aver definito gli attributi (con i relativi vincoli)
possiamo definire ulteriori vincoli così definiti:

- CHECK <espressione booleana>
- UNIQUE (<lista attributi>)
- PRIMARY KEY (<lista attributi>)

CONSTRAINT <nome_vincolo> <tipo vincolo> <definizione>

E.

STUDENTI (MATR, CF, DATAN, NOME, COGNOME)

ESAMI (MATR, CORSO, DATA, VOTO, LODE)

● CREATE TABLE Studenti:

(matr CHAR (8) PRIMARY KEY,
cf CHAR (16) UNIQUE,
datan DATE NOTNULL,
nome VARCHAR (20) NOTNULL,
cognome VARCHAR (20) NOTNULL)

● CREATE TABLE Esami:

(matr CHAR (8) NOT NULL,
corso CHAR (5) NOT NULL,
voto SMALLINT CHECK voto >= 18 AND voto <= 30 ,
data DATE NOT NULL,
lode BOOLEAN,

CONSTRAINT esameVolta UNIQUE (matr, corso),

CONSTRAINT lodeck CHECK (lode = TRUE AND voto = 30 OR NOT lode = TRUE)

CHIAVI ESTERNA

Nella definizione dei vincoli possiamo definire la CHIAVE ESTERNA

FOREIGN KEY (< lista attributi >) ← chiave esterna

REFERENCES < nome tabella > (< lista attributi pk >) ← pk nella tabella esterna

Possiamo avere diversi modi numero e tipo degli attributi dev'essere lo stesso, c'è una CORRISPONDENZA POSIZIONALE (tra < lista attributi > e < nome tabella >)

Si faccia attenzione che < lista attributi pk > DEVE essere dichiarato come PRIMARY KEY in < nome tabella >

IMPLICAZIONI DEFINIZIONE DELLA CHIAVE ESTERNA

① Imponete un vincolo d'INTEGRITÀ REFERENZIALE

Esempio: Ogni esame è accettato solo se la matricola è presente tra le matricole degli studenti.

② I valori che compaiono nelle chiavi esterne DEVONO compiere nella chiave primaria referenziata.

VIOLAZIONI VINCOLO D'INTEGRITÀ REFERENZIALE

Vediamo le violazioni in base all'azione eseguita:

INSERT

● VIOLAZIONE

Cerco di inserire un esame per uno studente non presente

● AZIONE

I vincoli non sono soddisfatti, l'inserimento non è eseguito

DELETE

● VIOLAZIONE

Cancello uno studente al quale sono associati degli esami

● AZIONE

► L'azione viene bloccata (**NO ACTION**)

► Cancello anche gli esami dello studente (**CASCADE**)

UPDATE

● VIOLAZIONE

Modifica della PK dello studente

● AZIONE

► l'azione viene bloccata (**NO ACTION**)

► Cambia la PK agli esami associati (**CASCADE**)

Quando dichiariamo le chiavi esterne possiamo dichiarare esplicitamente la gestione delle violazioni:

[ON DELETE <azione>]

[ON UPDATE <azione>]

<azione> = NO ACTION - CASCADE - SET DEFAULT - SET NULL

E.s.

CONSTRAINT fk FOREIGN KEY (matr)

REFERENCES Studenti (matr)

ON DELETE NO ACTION

ON UPDATE CASCADE (in oracle questo non e' possibile)

OSS

E' importante la dichiarazione separata dei vincoli per una gestione migliore (si possono attivare e disattivare)

E.

● ALTER TABLE esami (modifica la tabella)

DROP CONSTRAINT esami1volta (cancella il vincolo)

● ALTER TABLE esami

SET CONSTRAINT esami1volta DISABLED

● ALTER TABLE esami

SET CONSTRAINT esami1volta ENABLED

● ALTER TABLE esami

ADD CONSTRAINT nuovo_vincolo <definiz. >

VINCOLO DI ENNUPLA

È un vincolo che riguarda più attributi di uno stesso record (riga).

Dev'essere definito rigorosamente nella sezione dei constraint

Ese.

laude TRUE se e solo se voti = 30

CONSTRAINT <name.vincolo> CHECK expr.bool

OSS

Sia i vincoli di dominio che di esempio sono verificati all'inserimento del record. Per verificare questi vincoli basta controllare i dati inseriti in riga, quindi il controllo è computazionalmente economico

VINCOLO INTRARELAZIONALE

È un vincolo che fa riferimento a più istanze della relazione, per essere verificato richiede un controllo più pesante perché su più righe. I vincoli di esempio ad esempio sono intrarelazionali:

Ese.

CONSTRAINT PK PRIMARY KEY (...)

CONSTRAINT UQ UNIQUE (...)

ESAMI (matri, corso, CFU, data, voto, look)

COSTRAINT <nome>

CHECK SQL_EXPR (ricerca in tabella)

↑
posso richiamare solo la tabella
che sto definendo e **NON** altre

VINCOLO INTERRELAZIONALE

l' espressione del vincolo coinvolge più tabelle. I vincoli FOREIGN KEY sono interrelazionali (integrità relazionale), queste sono gli UNICI interrelazionali che è possibile scrivere nella definizione di una tabella , per gli altri si usa l'**ASSERTIONE** e i **TRIGGER**

QUERY LANGUAGE

Ogni relazione corrisponde a una tabella popolata , il modello fisico corrisponde a una **CREATE TABLE**. In una relazione però possono esserci duplicati e non sono ordinati cosa che non è possibile e verrà gestito con le **QUERY** . Nello schema TEORICO (relazione) useremo l' **ALGEBRA RELAZIONALE** , nel modello implementato usciamo

ha funzione **SELECT** di SQL.

Questa traduzione avviene internamente al DBHS.

ALGEBRA RELAZIONALE

Trasformo le relazioni mediante le operazioni, quindi scriviamo delle espressioni che voluterò, il risultato sarà il valore dell'interrogazione

Guardiamo un esempio e vediamo le applicazioni:

- STUDENTE (matr, nome, cognome, sesso)

100	ciao	Esposto	M
2 =	200	Ciomi	Esposto M
	1300	Rosa	Rossi F
	400	Harta	Rossi F

PROIEZIONE

Data una Tabella $S(A_1, \dots, A_m)$, indichiamo la PROIEZIONE con

Π_{B_1, \dots, B_k} in cui $B_i \in S$ e sono solo gli attributi che ci interessano.

- $\Pi_{\text{matr}, \text{cognome}}(1) = \begin{matrix} 100 \text{ Esposto} \\ 200 \text{ Esposto} \\ 300 \text{ Rossi} \\ 400 \text{ Rossi} \end{matrix} ?(\text{matr}, \text{cognome}) = 1'$

Diverso da $\Pi_{\text{cognome}, \text{nome}}(1)$

Π cognome (1) = Esposto
Rossi NON ci possono essere duplicati

IMPLEMENTAZIONE IN SQL

Usiamo le clausole SELECT e FROM

Π cognome, matricola (1)

```
SELECT S.cognome, S.matricola
FROM Studenti S
```

Con il comando

```
SELECT S.Cognome
FROM Studenti S
```

\rightarrow

Esposto
Esposto
Rossi
Rossi

Per rimuovere i duplicati:

```
SELECT DISTINCT S.Cognome
FROM Studenti S
```

SELEZIONE (righe)

Indichiamo la selezione con $\delta_c (r)$ in cui " c " è la condizione booleana sui nomi degli attributi e ci permette di "filtrare" la tabella secondo le operazioni di confronto.

Le operazioni relazionali sono: uguaglianza =
disuguaglianza \neq $\rightarrow >, <, \leq, \geq$

E.s.

$$\tilde{O}_{SESSO} = \begin{cases} "M" & (1) \\ F & \end{cases} \rightarrow \begin{array}{l} 300 \text{ Rosa} \\ 400 \text{ Marta} \end{array}$$

OSS

Se si controlla un campo NULL sul confronto dovrà come risultato

UNDEFINED che nella riscrittura della tabella verrà escluso (per una condizione ATTRIBUTO = TRUE)

Alcuni confronti di base sono:

- attributo IS NULL
- <attributo> <op. relazionale> <attributo>

Possiamo combinare condizioni attraverso l'uso dei connettivi logici

AND, NOT, OR :

C_1	C_2	$C_1 \wedge C_2$	$C_1 \vee C_2$
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F
<hr/>			
T	U	U	T
U	T	U	T
U	U	U	U
F	U	F	U
U	F	F	F

IMPLEMENTAZIONE SQL

Introduciamo alle clausole obbligatorie SELECT e FROM una nuova clausola **WHERE**:

SELECT *

FROM Esami E

WHERE E.Voto = 30 OR
E.Lode = TRUE

La SELECT proietta e * indica che vogliamo proiettare tutti gli attributi

In SQL l'interrogazione di base è una combinazione di selezione e solo successivamente la proiezione

E.s.

SELECT E.Matricola, E.Corso ← seleziona solo 2 attributi

FROM Esami E

WHERE E.Voto = 30 OR

E.Lode = TRUE

↓

$\Pi_{\text{matricola, corso}} (\delta_{\text{lode} = \text{TRUE}} (i))$

In algebra relazionale viene prima eseguita la proiezione e
dopo la selezione (viceversa di SQL)

$\sigma_{lock=TRUE} (\Pi_{mater, corso})$ NO!

Ripasso

la formula generale in SQL per esprimere un' interrogazione

SELECT [DISTINCT] <lista attributi>

FROM <nome_tabella> [AS <alias>]

WHERE <espressione booleana>

Si ricorda che nell'algebra relazionale la selezione è sempre l'ultima operazione.

OPERAZIONI INSIEMISTICHE

Si suddividono in:

- UNION \cup
- INTERSECT \cap
- EXCEPT \setminus

Per operare con queste operazioni gli insiemi devono avere struttura **COMPATIBILI** avendo stesso numero di attributi, gli attributi posizionalmente corrispondenti devono avere lo stesso dominio (non importa il nome)

E.

A	B	C
a	a	a
a	b	c
a	b	b

D	E	F
a	a	a
a	b	b
a	c	d

$$I \cap S = \begin{array}{|c|c|c|} \hline ? & ? & ? \\ \hline a & a & a \\ \hline a & b & b \\ \hline \end{array}$$
$$I \cup S = \begin{array}{|c|c|c|} \hline ? & ? & ? \\ \hline a & a & a \\ \hline a & b & c \\ \hline a & b & b \\ \hline a & c & d \\ \hline \end{array}$$

(l'ordine è irrelevante)

$$I \setminus S = \begin{array}{|c|c|c|} \hline ? & ? & ? \\ \hline \cdot & \cdot & \cdot \\ \hline a & b & c \\ \hline \end{array}$$

● PARTITE (Data, Stadio, Sq1, Sq2, Gol1, Gol2)

a) Squadra che non ha mai vinto in trasferta

b) Squadra che non ha mai fatto gol

c) Squadra che non ha mai vinto al Maradona

d) Squadre che hanno vinto nella giornata 24-11-2021

d) $\Pi_{Sq1} (\sigma_{goal1 > goal2} (p))$ proietto squadra 1 se goal1 > goal2
AND DATA = "2021-11-24"

sulla relazione p. Uniamo con l'antogo caso Sq2

$\cup \Pi_{Sq2} (\sigma_{goal1 < goal2} (p))$
AND DATA = "2021-11-24"

Possiamo usare la differenza perché le relazioni sono compatibili.

b) Analogia alla c

$$\left(\text{TS}_{q_1}(p) \cup S_{q_2}(p) \right) \setminus \left(\text{TS}_{q_1}(\sigma_{goal_1 > 0}(p)) \cup \text{TS}_{q_2}(\sigma_{goal_2 > 0}(p)) \right)$$

$$a) (\Pi_{S_{q_1}}(p) \cup S_{q_2}(p)) \setminus \Pi_{S_{q_1}}(\sigma_{goal_1 \rightarrow goal_2}(p))$$

IMPLEMENTAZIONE IN SQL

l'equivalente in SQL si applica sulla SELECT

`SELECT <..>` `UNION` `SELECT <..>`
`INTERSECT`
`EXCEPT`

Si ricordi eventualmente di inserire DISTINCT per rimuovere i duplicati

E_s

Le squadre che hanno vinto al Maracanã

(SELECT P. SQ1

FROM PARTITE AS P

WHERE P.STADIO = "Maradona"

AND P.GOAL1 > P.GOAL2)

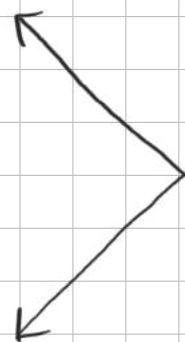
UNION DISTINCT

(SELECT P. SQ2

FROM PARTITE AS P

WHERE P.STADIO = "Maradona"

AND P.GOAL2 > P.GOAL1)



Le due interrogazioni
sono indipendenti fra loro

PRODOTTO CARTESIANO

I prodotti consentono di combinare tra due tavole.

$$r \leftarrow R(A_1, \dots, A_k)$$

$$s \leftarrow S(B_1, \dots, B_m)$$

$$r \times s = (A_1 B_1, \dots, A_1 B_m, A_2 B_1, \dots, A_2 B_m, \dots, A_k B_m)$$

Ese.

$$r = \begin{array}{|c|c|} \hline A_1 & A_2 \\ \hline a & a \\ a & b \\ \hline \end{array}$$

$$s = \begin{array}{|c|c|} \hline B_1 & B_2 \\ \hline c & c \\ c & d \\ \hline \end{array}$$

$$r \times s = \begin{array}{|c|c|c|c|} \hline A_1 & A_2 & B_1 & B_2 \\ \hline a & a & c & c \\ a & b & c & d \\ \hline \end{array}$$

Se r ha m elementi e s ne ha n , $r \times s$ ha $m \cdot n$ elementi

IMPLEMENTAZIONE IN SQL

L'implementazione si effettua nella clausola FROM con ","

STUDENTI (matr, nome, cognome)

ESAMI (matr, voto, corso)

SELECT *

FROM STUDENTI AS S,

ESAMI AS E

S = 100 Lino Esposito
200 Paolo Esposito
300 Maria Rossi

E = 100 BD1 27
100 BD1 30
300 BD1 30

$S \times E$	Matr.	Nome	Cognome	Matr.	Esame	Voto
100	Carlo	Esposito		100	BDI	27
				100	BDI	30
				300	BDI	30
200	Paolo	Esposito		100	BDI	27
				100	BDI	30
				300	BDI	30
300	Maria	Rossi		100	BDI	27
				100	BDI	30
				300	BDI	30

GIUNZIONE

Se vogliamo associare alla matricola di studente la matricola
in esame effettuiamo una **GIUNZIONE** ovvero una connessione
speciale sulla selezione. Di solito la giunzione avviene tra la
primary key di una Tabella e la chiave esterna (che riferisce
la PK) dell'associata.

$\Pi_{\text{nome, cognome}} (\sigma_{\substack{S \times E \\ \text{matr} = \text{matricola}}})$

SELECT S.nome, S.cognome
FROM STUDENTI AS S,
ESAMI AS E
WHERE S.matr = E.matricola
condizione di giunzione

SELECT S.nome, S.cognome
FROM STUDENTI AS S, JOIN
ESAMI AS E ON
S.matr = E.matricola
condizione di giunzione

le due espressioni sono analoghe ma la seconda è più leggibile.

D-JOIN

Il primo tipo di join che abbiamo visto è la θ-join che ci permette di aggiungere una condizione sugli attributi di join delle nostre relazioni. $r_1 \bowtie r_2$ useremo l'espressione

$r_1 \neq r_2$ dove \neq è la condizione.

OSS

le espressioni $S \otimes e$ e $\mathcal{O}(S \otimes e)$ sono equivalenti.

NATURAL JOIN

Useremo il **NATURAL JOIN** se per tutti gli attributi di r_1 e r_2 omonimi vogliamo una giunzione d'ugualanza.

Se non ci sono attributi con lo stesso nome questo agirà come prodotto cartesiano.

r_1 , Δ , r_2

E_s.

$S \leftarrow STUDENTI(\text{matr, nome, cognome})$

$e \leftarrow \text{ESAMI}(\text{math}, \text{covo}, \text{veto}, \text{look})$

matr	nome	cognome	corso	voto	lode
S&e =

I due attributi matr si comprimono

OSS

Bisogna far attenzione che i nomi di chiave primaria e chiave esterna devono essere gli stessi mentre gli altri attributi devono avere nome diverso.

E.s.

- Vogliamo nome e cognome degli studenti che hanno fatto esami

$S \leftarrow \text{STUDENTI} (\text{matr}, \text{nome}, \text{cognome})$

$e \leftarrow \text{ESAMI} (\text{matricola}, \text{corso}, \text{voto}, \text{lode}, \text{data})$

Potremmo usare una θ-join :

$\Pi_{\text{nome}, \text{cognome}} (S \& e \text{ matr} = \text{matricola})$

Proiettiamo nome e cognome degli studenti che "sopravviveranno" alla giunzione ovvero quelli cui matricola figura tra gli esami

In SQL avremo

```
SELECT DISTINCT S.nome, S.cognome  
FROM STUDENTI AS S JOIN  
ESAMI AS e ON S.matr=e.matricola
```

- Vogliamo nome e cognome degli studenti che **NON** hanno fatto esami

$\pi(s) \setminus \pi(sme)$
nome, cognome nome, cognome matr=matricola

```
(SELECT DISTINCT S.nome, S.cognome  
FROM STUDENTI AS S)
```

| EXCEPT

```
(SELECT DISTINCT S.nome, S.cognome  
FROM STUDENTI AS S JOIN  
ESAMI AS e ON S.matr=e.matricola )
```

OSS

In SQL il nome di un attributo è dato da
`<nome-tabella>. <nome-attributo>`.

Il solo <nome-attributo> puo' essere usato solo se non c'e' ambiguita', quindi nessun'altra tabella ha un attributo omomico, altrimenti usiamo appunto l'alias.

EXTERNAL JOIN

Questo tipo di giunzione tiene conto delle righe che non realizzano la giunzione mantenendone traccia.

Abbiamo tre tipi diversi LEFT / RIGHT / FULL OUTER JOIN.

Prendendo in esame il left outer join, eseguiamo il Θ -join e completo con le righe di s che non hanno verificato la giunzione

$S \bowtie_{\theta} t$

Ese.

matr	name	cognome
100	Lino	Esposito
200	Pasquale	Esposito
300	Maria	Rossi

matrula	corso	voto
100	BD1	27
100	BD1	30
300	BD1	30

matr	name	cognome	matrula	corso	voto
100	Lino	Esposito	100	BD1	27
100	Lino	Esposito	100	BD1	30
300	Maria	Rossi	300	BD1	30
200	Pasquale	Esposito	NULL	NULL	NULL

L'espressione associata in algebra relazionale è

$\Pi_{\text{name, cognome}} (\sigma_{\text{matricola IS NULL}} (\text{size}))$

IMPLEMENTAZIONE IN SQL

```
SELECT DISTINCT S.name, S.cognome  
FROM STUDENTI AS S LEFT OUTER JOIN  
ESAMI AS E ON S.matr = E.matricola  
WHERE E.matricola IS NULL
```

OPERAZIONI DI RINOMINA

Sono operazioni che ci permettono di cambiare o attribuire un nome a uno schema o ai suoi attributi.

● $\rho_{S(A_1, \dots, A_m)}^{(s)}$ lo schema della relazione S è ora $S(\bar{A}_1, \dots, \bar{A}_m)$

● $\rho_{A_3 \leftarrow \bar{A}}^{(s)}$ la colonna A_3 della relazione S è ora \bar{A}

OSS

Possiamo sempre applicare la giunzione naturale rinominando un attributo

E.s.

- $S \leftarrow \text{STUDENTI} (\text{matr, nome, cognome})$
 $e \leftarrow \text{ESAMI} (\text{matricola, corso, voto, lode, data})$

$p(s) \bowtie e$ natural joint
matr \leftarrow matricola

$\Pi(e)$ $\rightarrow ?(\text{matricola, corso})$
matricola, corso

Assegniamo un nome a questa tabella

$p_{es2}(\text{matricola, corso}) \xrightarrow{(\Pi(e))} es2(\text{matricola, corso})$

- $S \leftarrow \text{STUDENTI} (\text{matr, nome, cognome})$

Vogliamo le coppie di studenti omomici

La scelta migliore è il prodotto cartesiano, tuttavia non possiamo avere colonne omomiche (che così facendo accade inevitabilmente).

Effettuiamo una rimozione nel prodotto cartesiano

$\prod \sum_{\text{COND}} (S \times_P S)$
matricola, $S(m_1, m_1, c_1)$
nome,
cognome, m_1

con $\text{COND} = \text{matricola} \neq m_1 \text{ AND } \text{name} = m_1 \text{ AND } \text{cognome} = c_1$

IMPLEMENTAZIONE IN SQL

Semplicemente usiamo l'**ALIASING (AS)**

SELECT $S_1.\text{matricola}, S_2.\text{matricola} \dots$

FROM Studente AS $S_1,$

Studente AS S_2

WHERE $S_1.\text{matricola} \neq S_2.\text{matricola}$

AND $S_1.\text{name} = S_2.\text{cognome}$

AND $S_1.\text{cognome} = S_2.\text{cognome}$

E.s.

Cerchiamo gli studenti che hanno fatto tutti gli esami

$t \leftarrow \text{TUTTI ESAMI (corso)}$

$$\pi_{\text{matr}}(s) \setminus \pi_{\text{matr}} \left[\left(\pi_{\text{studenti}}^{(s)} \times \pi_{\text{tutti esami}}^{(t)} \right) \setminus \left(\pi_{\text{matr, corso}}^{(e)} \right) \right]$$

E.s.

Cerchiamo le coppie di studenti che hanno dato gli stessi esami

$$\pi_{\text{matr}, \text{nome}, \text{cognome}} \circ_{\text{COND}} (s \times_p (s))$$

$s(m_1, m_2, c_1)$

$S \leftarrow \text{STUDENTE} (\text{nome}, \text{cognome}, \text{matr})$

$e \leftarrow \text{ESAMI} (\text{matr}, \text{corso}, \text{voto}, \text{lode}, \text{data})$

① Studenti che hanno dato gli esami (compresi nessuno) + rimozione

$$\rho_{es} \left(\begin{array}{l} \text{nome, cognome} \\ \text{matr}, \text{corso} \end{array} \right) \left(\begin{array}{l} \pi_{\text{studenti}}^{(s)} (\sigma_{\text{matr}}^{(s)}) \\ \text{nome, cognome} \\ \text{matr}, \text{corso} \end{array} \right)$$

$\text{matr} \text{ IS NULL}$

② Proiezione delle coppie di studenti (con rimozione nel prodotto)

$$\pi_{\text{matr}, \text{nome}, \text{cognome}} \circ_{\text{COND}} (es \times_p (es))$$

$s(m_1, m_2, c_1, \text{corso}_1)$

COND = matr \leftrightarrow m₁ AND
 corso = corso₁

OPERAZIONI DI CONTEGGIO

Le operazioni di conteggio ci permettono di eseguire operazioni analitiche su attributi. Alcune di queste possono essere effettuate solo tra valori numerici.

Le operazioni sono:

- SUM (applicato sui dati di un attributo)
 - AVG (calcolato su un attributo)
 - MIN
 - MAX
 - COUNTING
- } SOLO VALORI NUMERICI
- } VALORI CON DOMINIO ORDINATO
- } SU OGNI VALORE

La procedura standard è:

- ① Scegliamo una relazione
- ② Determino il criterio di raggruppamento (lista di attributi)
- ③ Ricaviamo i GRUPPI ovvero le righe che hanno gli stessi valori su tutti gli attributi scelti per il raggruppamento.
- ④ Determino il criterio di conteggio (lista di operazioni)

L'espressione generale è $\langle \text{criterio_raggruppamento} \rangle \hat{\wedge} \langle \text{criterio_conteggio} \rangle (s)$

OSS

Il criterio di raggruppamento può essere vuoto, di conseguenza avremo un solo gruppo ovvero quello totale.

Il risultato sarà dato da uno schema che ha una colonna per ogni attributo del criterio di raggruppamento e una colonna per ogni operazione del conteggio.

E.s.

matricola	corso	voto	booke
100	BD1	27	NULL
100	ALG	27	NULL
s = 200	BD2	28	NULL
200	BD1	30	SI
200	ALG	30	NULL
300	BD1	30	NULL

matricola	N-E	H-E	N-L
100	2	28,5	0
200	3	29,5	1
C = 300	1	30	0

- $C \leftarrow p (\text{matricola} \hat{\wedge} \text{COUNT}(\text{matricola}), \text{AVG}(\text{voto}), \text{COUNT}(\text{booke}) (s))$
 $N-E, H-E, N-L$

Avremo un elemento nel risultato per ogni gruppo.

● Cerchiamo su c lo studente con la media più alta

Lavoriamo su **TUTTA** la relazione quindi il criterio di raggruppamento è vuoto:

$$\varphi_{\text{MM}(\text{valmax})} \left(\exists \text{ MAX } (\text{H-E}) (c) \right) \rightarrow \text{MH}$$

Il risultato è una relazione (non un valore) che ha una riga e una colonna (che nominiamo valmax)

● Se vogliamo vedere le informazioni dello studente con la media più alta possiamo fare una giunzione tra (c) e (MH) e proiettare la matricola

$$\pi_{\text{matricola}} (c \bowtie \text{MH}) \bowtie \text{Studenti}$$

$\text{H-E} = \text{Valmax}$

OSS

Le operazioni di conteggio **NON** possono essere ridificate

$$\text{MAX}(\text{AVG}(\text{vuoto})) \text{ No!}$$

Questo perché mancano i gruppi per le operazioni

IMPLEMENTAZIONE IN SQL

Aggiungiamo la nuova clausola GROUP BY per esprimere il raggruppamento.

GROUP BY <lista di attributi>

Se la lista è vuota, la clausola viene omessa.

Le operazioni di conteggio vengono implementate nella SELECT.

Es.

Vediamo l'esempio precedente

SELECT S.matr, COUNT(e.corso) AS N_E

AVG(e.voto) AS M_E

COUNT(e.lode) AS N_L

FROM Studenti AS S LEFT OUTER JOIN

Esami AS E ON

S.matr = E.matricola

GROUP BY S.matr.

OSS

Se vogliamo trovare anche nome e cognome degli studenti
NON possiamo banalmente aggiungerlo alla SELECT perché in
SQL tutti gli attributi che compaiono nella SELECT devono comparire
nella GROUP BY.

SELECT S.matr, nome, cognome COUNT (e.corso) AS N_E

...

GROUP BY S.matr, nome, cognome

Questa modifica sintattica non modifica i gruppi perché la
matricola è chiave primaria e in quanto tale definisce univocamente
tutti gli attributi e il compilatore interpreta correttamente
questa condizione

CLAUSOLA HAVING

C'è una clausola che non ha un corrispettivo diretto in
algebra relazionale ovvero la clausola **HAVING** che ci permette di
filtrare i risultati di un conteggio

HAVING <operazione di conteggio> <condizione>

CLAUSOLA ORDER BY

Per ordinare una tabella abbiamo la clausola **ORDER BY** che assume la forma

ORDER BY <attributi> [ASCENDENT/DESCENDENT]

Se non viene esplicitato il criterio si intende ASCENDENT

QUERY SQL

In SQL quindi una query assume la seguente forma e ordine:

- ⑥ **SELECT <espressioni sugli attributi>** (proiezioni + conteggi)
- ① **FROM prodotti** di tabella
- ② **WHERE** Selezione delle righe dei prodotti
- ③ **GROUP BY <attributi di raggruppamento>**
- ④ **HAVING** Selezioni sui gruppi
- ⑤ **ORDER BY <attributo> [ASCENDENT/DESCENDENT]**

OSS

- Si faccia attenzione a non confondere le condizioni della WHERE con quelle della GROUP BY che agiscono su entità diverse.

● Per la COUNT abbiamo due varianti ovvero:

► COUNT (DISTINCT attributo)

Conta le righe ignorando i duplicati

► COUNT (*)

Conta tutte le righe indistintamente

● Lo SELECT puo' adoperare delle espressioni per elaborare gli attributi che andremo a proiettare.

Queste espressioni sono diverse a seconda del DBMS che fornisce le funzionalita' sul tipo che stiamo adoperando.

E.s.

● Se vogliamo la media calcolata in 110 imi anziché 30 imi possiamo scrivere

SELECT AVG (e.voto) / 30 * 110 AS M_110

● Se vogliamo riportare in UPPERCASE la proiezione di un attributo stringa possiamo scrivere

SELECT S.matri, UPPER(S.cognome) || UPPER (S.nome) AS NC

VISTA

Il risultato di una query è una tabella senza nome.

Per definire questo nome (in SQL) creiamo una **VISTA**

CREATE VIEW <nome-vista> (nomi attributi) AS

SELECT expr₁, ..., expr_n

Il numero di attributi della vista è uguale al numero di espressioni. Questa tabella virtuale è **CALCOLATA** ma non **SALVATA**

E.s.

Vediamo l'esercizio 5 del 02/11

CREATE VIEW Stati_Finali (Cod A, N-SF)

AS SELECT S.Cod A, COUNT(*)

FROM Stati AS S

WHERE S.Finale = 'True'

Group BY S.Cod A

Attraverso la COUNT(*) c'è un'implicita riconmima a N-S-F

Contiamo le parole distinte

CREATE VIEW Parole (Cod A, N_P)

AS SELECT R.CodA COUNT (DISTINCT R.Parole)

FROM Raggiungibile AS R

GROUP BY R.CodA

Contiamo adesso il numero di stati finali raggiungibili

SELECT S.Cod A, S.N_S_F, P.N_P

FROM Stati_Finali AS S

JOIN Parole AS P ON

S.Cod A = P.Cod A

le viste vengono usate come vere e proprie tabelle che però

NON possono essere aggiornate.

CONDIZIONI ATOMICHE AGGIUNTIVE

Nella clausola WHERE viene verificata la condizione booleana a partire da condizioni atomiche sugli attributi della forma $\langle \text{attributo} \text{ op. re. valore} \rangle$ oppure $\langle \text{attributo} \text{ op. re. attributo} \rangle$.

Vediamo nuove forme di condizioni atomiche:

- nome-attributo IS [NOT] NULL

Ci permette di valutare l'associazione di un valore a un attributo

- nome-attributo IN (valore₁, ..., valore_n)

E' l'abbreviazione di

nome-attributo = valore₁ OR ... OR nome-attributo = valore_n

Ese.

SESSO IN ('M', 'F')

- nome-attributo BETWEEN valore₁ AND valore₂

E' una condizione per intervallo di solito su attributi numerici

E.s.

VOTO BETWEEN 0 AND 30

- nome-attributo LIKE "pattern"

E' una condizione su stringhe (**PATTERN MATCHING**) per tanto

vale **SOLO** per attributi di tipo CHAR / VARCHAR.

Per "pattern" intendiamo un criterio di uguaglianza.

Usiamo "-" per un solo carattere arbitrario e "%" per una sequenza di 0 o più caratteri arbitrari all'interno del pattern.

E.s.

nome LIKE "Mari"

uguaglianza stretta

"--"

nome ha esattamente 3 caratteri

"Mari%"

nome deve contenere "Mari" all'inizio

"%Mari%"

nome deve contenere "Mari" nella stringa

INTERROGAZIONI INNESTATE

In SQL possiamo inserire delle **SOTTO-INTERROGAZIONI** ad una principale.

Queste vengono inserite nelle clausole FROM, WHERE ed HAVING.

Gia' le viste rappresentano un innesto di interrogazione, perciò il risultato finale e' analogo, cambia solo la forma come vedremo.

Generalmente le viste aiutano la leggibilità, tuttavia se la sottointerrogazione non e' particolarmente articolata puo' risultare più comodo l'innesto "esteso".

INNESTO NELLA CLAUSOLA FROM

Il modo più comodo di inserire una sottointerrogazione è nello clausola FROM.

E.s.

Vediamo l'esempio della scorsa lezione delle viste, implementando le interrogazioni nella FROM

① CREATE VIEW Stati_Eivali (CodA, N_S_F)

AS SELECT S.CodA, COUNT(*)

FROM Stati AS S

WHERE S.Eivali = 'True'

GROUP BY S.CodA

② CREATE VIEW Parole (CodA, N_P)

AS SELECT R.CodA, COUNT(DISTINCT R.par)

FROM Raggiungibilita' AS R

WHERE S.Eivali = 'True'

GROUP BY R.CodA

①+② SELECT S.CodA, S.N_S_F, P.N_P

FROM S.StatiEivali AS S

JOIN Parole AS P ON

S.CodA = P.CodA

Senza usare le viste avremo una forma più comoda:

SELECT

FROM (SELECT S.CodA, COUNT(*) AS N_S_F

FROM Stati AS S

WHERE S.Finale = TRUE

GROUP BY S.CodA) AS C₁

JOIN

(SELECT R.CodA, COUNT(DISTINCT) AS R.par

FROM Raggiungibilita' AS R

WHERE S.Finale = 'True'

GROUP BY R.CodA) AS C₂

ON C₁.CodA = C₂.CodA

INNESTO NELLA CLAUSOLA WHERE

Nella clausola WHERE possiamo usare solo espressioni booleane, per sfruttare le relazioni useremo l'appartenenza di un elemento alla nostra Tabella (e quindi sottointerrogazione) oppure che la tabella sia vuota. Questo viene implementato con le clausole [NOT] IN e [NOT] EXIST

[NOT] IN

Questo "metodo" prende la forma

(attributo₁, ..., attributo_n) IN/[NOT] IN (SELECT expr₁, ..., expr_m
FROM ...)

Dove attributo_i e expr_i hanno la stessa struttura

E.

STUDENTI (Matr, Nome, Cognome)

ESAMI (Matr, data, voto, lode, corso)

Se vogliamo trovare gli studenti che non hanno dato esami possiamo cercare gli studenti che non appartengono all'insieme degli studenti che hanno dato esami.

SELECT *

FROM Studenti AS S

WHERE S.matr NOT IN (SELECT E.Matr

FROM Esami AS E)

↑
tabella degli studenti che hanno dato esami

[NOT] EXIST

Con questo metodo possiamo verificare che una interrogazione dia risultato nullo oppure ci sia almeno un elemento

E.s.

Gli studenti che non hanno fatto esami sono gli students che hanno ricevuto degli esami sostenuti pari a 0

SELECT *

FROM Studenti AS S

WHERE NOT EXISTS (SELECT *

FROM Esami E

WHERE E.matr = S.matr)

La sottointerrogazione è **CORRELATA** tramite S.matr e **NON** può essere quindi eseguita singolarmente.

INTERROGAZIONI CORRELATE

Se una sottointerrogazione utilizza un oggetto definito in un'interrogazione a livello più alto parleremo di interrogazione **CORRELATA**

Un'integrazione correlata **NON** puo' essere eseguita da sola,
mentre una **SCORRELATA** si.

OSS

- Quando si vuole controllare che manchi qualcosa non conviene un **Outer JOIN**, bensì conviene usare una sottointegrazione con **EXIST / NOT EXIST**.
- **EXIST** e **NOT EXIST** prevedono quasi sempre integrazioni correlate

Es.

Trovare gli studenti che hanno fatto tutti gli esami dello studente di matricola '100'.

Cerco gli studenti per cui non esiste un esame sostenuto dallo studente 100 che non sia stato sostenuto da loro (l'insieme degli esami sostenuti da '100' è non dello studente x e' vuoto)

SELECT *

FROM Studenti AS S

WHERE NOT EXISTS (SELECT *

```

    FROM Esami AS E
    WHERE E.Matri = '100' AND
        E.Corsu NOT IN ( SELECT E2.Corsu
                            FROM Esami AS E2
                            WHERE E2.matri = S.matri )

```

- Corsi sostenuti da S.matri
- Insieme dei corsi sostenuti da '100' ma non sostenuti da S.matri

Le due interrogazioni sono entrambe conelate (la prima è
conelata secondo la sua sottointerrogazione).

OSS

Si faccia attenzione alla visibilità:

Tutti i nomi dichiarati a livello più alto possono essere
richiamati a livello più basso. **NON** vale il viceversa.

VINCOLI INTERRELAZIONALI

Sono vincoli che coinvolgono attributi di più tabella.

In SQL avremo il costrutto **ASSERTION**

CREATE ASSERTION <name, vincolo> expr booleana

Ricordiamo che un' interrogazione viene "adattata" ad un' espressione booleana con le clausole EXIST / [NOT] EXIST (in particolare viene quasi sempre usato [NOT] EXIST)

Sfrutteremo il [NOT] EXIST per introdurre la violazione del vincolo

Scriviamo i vincoli:

- SRG e TRG (sorgente e target) sono diversi.

E' un vincolo di esempla (intrarelazionale)

ALTER TABLE ARC

ADD CONSTRAINT C₁

CHECK SRG <> TRG

- Nodi con la stessa sorgente non sono collegati da un arco

E' un vincolo intrarelazionale (CHECK NOT EXIST...)

- L'identificativo della radice è un nodo dell'albero

E' un vincolo interrelazionale.

CREATE ASSERTION A₁

CHECK [NOT] EXIST (SELECT *

FROM TREE AS T

WHERE ID_ROOT NOT IN

(SELECT N.ID_N

FROM NODE AS N

WHERE N.ID_T = T.ID_T)

② Sia SCAMBIO \rightarrow S

$G_1 \leftarrow \rho \left(\text{FILE-ID}, \text{YEAR}(\text{DATA}) \tilde{\sqcap} \text{COUNT}(\text{COD-B}) \mid \begin{array}{l} \text{FILE-ID,} \\ \text{DATA, COD-B,} \\ \text{USER_UP} \end{array} \right)$

FILE-ID,
 DATA, COD-B,
 USER_UP

FILE-ID,
 DATA, COD-B,
 USER_UP

$G_2 \leftarrow \rho \left(\text{FILE-ID}, \text{YEAR}(\text{DATA}) \tilde{\sqcap} \text{COUNT}(\text{COD-B}) \mid \begin{array}{l} \text{FILE-ID,} \\ \text{DATA, COD-B,} \\ \text{USER_DOWN} \end{array} \right)$

FILE-ID,
 DATA, COD-B,
 USER_DOWN

$R \leftarrow G_1 \bowtie G_2$

- Calcolo del numero di file scambiati (salvo un blocco) per ogni utente

$C_1 \leftarrow \rho \left((\text{USER_UP}) \tilde{\sqcap} \text{COUNT}(\text{COD-FILE}) \mid \begin{array}{l} \text{USER_UP,} \\ \text{COD-FILE} \end{array} \right)$

USER_UP,
 COD-FILE

↑
rimuove duplicati

$C_2 \leftarrow \rho \left((\text{USER_UP}) \tilde{\sqcap} \text{COUNT}(\text{COD-FILE}) \mid \begin{array}{l} \text{USER_UP,} \\ \text{COD-FILE} \end{array} \right)$

USER_UP,
 COD-FILE

③ Prendo coppie (utente, file) e **NON** c'è un blocco del file che non sia stato scaricato nel 2020 dall'utente

SELECT U.COD_U, F.COD_F

FROM UTENTE AS U, FILE AS F (prodotto cartesiano)

WHERE NOT EXIST

blocchi del file F (SELECT *

non scaricati

nel 2020 da U

FROM BLOCCHI AS B

WHERE B.COD_FILE = F.COD_FILE AND

B.COD_B NOT IN

(SELECT

FROM SCARICATO AS S,

WHERE YEAR(S.ANNO) = '2020' AND

S.USER_DOWNLOAD = U.COD_U)))

④ Scriviamo i seguenti vincoli:

c) Un utente possiede un file se possiede tutti i suoi blocchi
(in data precedente all' UPLOAD)

b) Un utente non scarica due se stessi

a) La data di posso di un utente dev'essere univoca

a) ALTER TABLE Possiede Blocchi

ADD CONSTRAINT U1 UNIQUE (COD_U, COD_B)

Vincolo intralazionale

b) Vincolo d' esempio

ALTER TABLE SCAMBIO

ADD CONSTRAINT U2

CHECK USER_UP <> USER_DOWN

c) Vincolo interazionale (ASSESSIONE)

CREATE ASSERTION A1 CHECK

NOT EXIST (



violazione: c'è un blocco
 del file che non è posseduto
 in data precedente

```

    ( SELECT
      FROM BLOCCHI AS B, POSSIEDEFILE AS P
      WHERE B.COD_F = P.COD_F AND
      (B.COD_B, P.COD_U) NOT IN
      (SELECT P2.COD_B, P2.COD_U
      FROM POSSIEDEBLOCCHI AS P2
      WHERE P2.DATA <= P.DATA)))
  
```

RIASSUNTO INTERROGAZIONI NIDIFICATE

le introduciamo nella clausola WHERE:

- $(A_1, \dots, A_m) \text{ IN } (\text{SELECT } \text{exp}_1, \dots, \text{exp}_m$
 $\text{FROM } \dots)$

- WHERE [NOT] EXIST $(\text{SELECT } *$
 $\text{FROM } \dots)$

- Attributo op.re. $(\text{SELECT } \text{unico-attributo}$
 $\dots)$

above op.re = $\{ =, <, >, <=, >= \}$

- Attributo op.re. [ANY, ALL] (SELECT m-attributi
---)

In questo caso l'interrogazione restituisce più valori e l'attributo deve verificare l'op.re. su elemento uno (ANY) o tutti (ALL) i valori di intorno dell'interrogazione

OSS

Un'interrogazione che restituisce un solo valore è detta **SCALARE** e in SQL può essere usato come valore di confronto, ad esempio nel caso di due conteggi per vedere se coincidono

OSS

Gli attributi non oggetto di conteggio nella clausola SELECT devono comparsire nella GROUP BY

DML Data Manipulation language

I comandi di modifica delle tabelle sono:

- INSERT popolamento delle tabelle
- DELETE rimozione di righe
- UPDATE cambio dei valori in riga

DELETE

Per eliminare una RIGA (non un attributo) useremo

DELETE

FROM nome-tabella

[WHERE condizione]

Il DBMS cancella tutte le righe che soddisfano la condizione.

Nonostante ciò la clausola WHERE NON è obbligatoria, se omessa implica l'eliminazione di TUTTE le righe

OSS

La `DELETE` rimuove righe, non tabelle. Se vogliamo eliminare la tabella useremo la funzione `DROP <nome tabella>`

INSERT

Vediamo come inserire una o più righe in tabella.

Possiamo inserire dati già forniti come valori oppure calcolati.

INSEGNAMENTO DI VALORI

Il comando è:

`INSERT INTO nome_tabella`

`[(elenco attributi)]`

`VALUES (lista valori), [(lista valori)] ...`

Se l'elenco attributi viene omesso e' implicita la totalità e verranno inseriti nell'ordine di dichiarazione.

Se si esplcitano gli attributi da inserire, l'elenco deve includere tutti gli attributi **TOTALI** (quelli che hanno vincolo `NOT NULL` nella dichiarazione), l'ordine di

inserimento e' quello dell' elenco.

Gli attributi (parziali) non presenti avranno valore NULL oppure il valore di DEFAULT impostato

Es.

ESAMI (Matricola, data, corso, voto, lode)

lode e' parziale

INSERT INTO ESAMI

VALUES ("N86003613", DATE "2021-11-11", "BD1", 26) lode sarà NULL

INSERT INTO ESAMI (data, matricola, corso, voto)

VALUES (DATE "2021-11-11", "N86003613", "BD1", 26)

INSERT INTO ESAMI (lode, data, matricola, corso, voto)

VALUES (NULL, "N86003613", DATE "2021-11-11", "BD1", 26)

VALUES (NULL, "N86003612", DATE "2021-11-11", "BD2", 27)

INSEGNAMENTO DI VALORI CALCOLATI

Il comando e'

INSERT INTO nome_tabella

[(elenco attributi)]

(SELECT lista espressioni

)

Inseriremo tante righe quante quelle recuperate dalla select

OSS

Se vogliamo inserire alcuni attributi fissi e altri calcolati
usiamo questo formato inserendo gli attributi fissi nella select

UPDATE

UPDATE nome-tabella

SET nome_attributo = espressione,
nome_attributo = espressione ---

WHERE condizione

Dobbiamo sempre indicare quali attributi vogliamo aggiornare.

L'attributo da associare a nome_attributo puo' essere il risultato di una sottointervagazione.

I passaggi sono:

- ① Si lavora su nome-tabella
- ② Si selezionano le righe con la WHERE
- ③ Per ogni riga si aggiornano gli attributi della clausola SET

Se la WHERE viene omessa si modificano tutte le righe

E.s.

MAGAZZINO (CodA, Descrizione, Quantita')

ORDINI (CodO, Data, PIVA, InviaTo) inviatO = data d'invio

COTR-ORDINE (CodO, CodA, Quantita', Prezzo)

CARRELLO (CodC, Data, PIVA, Completo)

COMP-CARRELLO (CodC, CodA, Quantita, Prezzo)

① Quando il carrello è completo convertirlo in Ordine

- Creare un nuovo ordine con la struttura del carrello
- Rimuovere il carrello

② Quando un ordine viene effettuato dobbiamo aggiornare il magazzino

Attenzione all'ordine delle operazioni, specialmente per il vincolo di
INTEGRITA' REFERENZIALE (le chiavi esterne)

Lavoriamo con il carrello xyz e supponiamo sia completo

① L'attributo inviato a differenza degli altri non è calcolato.

Infatti va settato a NULL.

● INSERT INTO ORDINI (CodO, Data, PIVA, InviaTo)

(SELECT C.CodC, DATA, PIVA, NULL

FROM CARRELLO AS C

WHERE C.CodC = "xy")

Possiamo nella SELECT inserire valori calcolati o non (come NULL)

Se non avessimo scritto NULL sarebbe andato automaticamente a NULL (vale ovviamente solo per attributi particolari)

INSERT INTO COMP_ORDINI

(SELECT *

FROM COMP_CARRELLO AS C

WHERE C.CodC = xyz)

● DELETE

FROM COMP_CARRELLO AS C

WHERE C.CodC = xyz

DELETE

FROM CARRELLO AS C

WHERE C.CodC = xyz

NON possiamo eliminare prima carrello per la chiave
esterna di COMP_CARRELLO (integrità referenziale).

Se avessimo gestito la violazione sarebbe stato possibile
(ON DELETE CASCADE ...)

② UPDATE MAGAZZINO AS M

```
SET M.QUANTITA' = M.QUANTITA' - (SELECT CO.QUANTITA'  
FROM COMP.ORDINE AS CO  
WHERE CO.CodO = xyz  
AND CO.CodA = M.CodA )  
  
WHERE M.CodA IN (SELECT C.CodA  
FROM COMP.ORDINE AS C  
WHERE C.CodO = xyz)
```

Modifichiamo gli articoli del magazzino oggetto dell'
ordine xyz tramite Cod A.

La nuova quantità dell' articolo va ridotta in funzione
della quantità dell' ordine , per questo va presa da un'
interrogazione.

TRIGGER

Sono sequenze di operazioni attivate sull'avvenimento di un evento. Gli eventi che andremo a studiare più approfonditamente sono quelli che alterano la base di dati (INSERT, UPDATE, DELETE).

Dobbiamo specificare:

- ① Con quale evento si attiva il trigger
- ② Quando si attiva il trigger (prima o dopo l'evento)
- ③ Condizioni di filtraggio
- ④ Reazioni

La struttura è la seguente:

CREATE TRIGGER <name-trigger>

[AFTER / BEFORE / INSTEAD OF] INSERT INTO Tabella

DELETE ON Tabella

UPDATE ON Tabella OF attributo

[FOR EACH ROW]

WHEN condizione

se bisogna agire su tutte le righe

BEGIN

--- corpo del problema
END

E.

CREATE TRIGGER Crea_Ordine

AFTER UPDATE ON carrello OF Completo

FOR EACH ROW

WHEN OLD.Completo = 'Incompleto'

AND NEW.Completo = 'Completo'

BEGIN

INSERT INTO Ordini (CodO, Data, PIVA, Inviato)

(SELECT C.CodC, Data, PIVA, NULL

FROM CARRELLO AS C

WHERE C.CodC = NEW.CodC)

END

Questo trigger si attiva all' UPDATE della classe carrello sull' attributo Completo , in particolare agisce quando il vecchio valore (**OLD**) era "Incompleto" e ora (**NEW**) e' "Completo". OLD e NEW fanno riferimento alle righe modificate e si usano **Solo** con la clausola FOR EACH ROW (che prende tutte le righe)

E.

● EX(NUM, ID)

CREATE TRIGGER

AFTER UPDATE ON EX OF NUM

FOR EACH ROW

WHEN NEW.NUM > OLD.NUM

BEGIN

UPDATE EX

SET NUM = NUM + 1

WHERE NEW.ID = E.ID

END

UPDATE EX

SET NUM = NUM + 1

WHERE EX.ID = "100"

}

Incrementa di 1 EX.NUM

In questo caso il trigger genera un loop infinito in quanto si attiva all'UPDATE di EX.NUM ma lui stesso esegue un UPDATE di EX.NUM (generando un loop)

• TREE (CodT , root)

Nodo (CodT , Etichetta, CodN)

ARCO (CodT , N-IN, N-OUT)

Quando incremento un nodo devo incrementare della stessa quantità a tutti i nodi discendenti:

CREATE TRIGGER Propaga

AFTER UPDATE ON Nodo OF Etichetta

FOR EACH ROW

WHEN NEW.Etichetta - OLD.Etichetta > 0 incremento

BEGIN

UPDATE Nodo N

SET N.Etichetta = N.Etichetta + (NEW.Etichetta - OLD.Etichetta)

WHERE N.Nodo IN (SELECT A.N-IN

FROM ARCO A

WHERE A.CodT = NEW.CodT AND

A.N-OUT = NEW.CodN

Nella WHERE prendiamo i nodi figli

BLOCCARE UN TRIGGER

Esistono vari modi per fermare un trigger:

- ELIMINARE IL TRIGGER

DROP TRIGGER Propaga

- DISABILITARE IL TRIGGER

ALTER TRIGGER Propaga DISABLED

SEQUENZE

In SQL è possibile creare una **SEQUENZA** di valori generati automaticamente, comoda per creare **CHIAVI SINTETICHE** in corrispondenza degli inserimenti.

Vediamo la struttura direttamente con un esempio

CREATE SEQUENCE mio-ID

START WITH 1000

INCREMENT BY 1

MAX VALUE 10000

} Se omessa questa parte supponiamo START=1
INCREMENT BY illimitato.

PROVA (AUTOID, NOME, COGNOME)

INSERT INTO Prova

VALUES (NULL, 'Ciro', 'Esposito')

CREATE TRIGGER correggo_K

BEFORE INSERT ON Prova

FOR EACH ROW

BEGIN

:NEW.autoID := miold.NEXTVAL

END

BEFORE perché AUTOID non può essere NULL

OSS

Dalla versione 12C in poi di ORACLE possiamo usare il seguente costrutto per la sequenza

CREATE TABLE Prova

AutoID GENERATED ALWAYS IDENTITY

START WITH 1 INCREMENT BY 1

Se usiamo ALWAYS NON dobbiamo usare valori per autoID,

se usiamo GENERATED BY DEFAULT

NORMALIZZARE UN INSERIMENTO

Se vogliamo normalizzare i dati prima di inserirli possiamo usare un trigger in particolare con la clausola BEFORE

CREATE TRIGGER consegno-K

BEFORE INSERT ON Prova

FOR EACH ROW

BEGIN

:NEW.autoID := muID.NEXTVAL

:NEW.nome := UPPER

:NEW.cognome := LOWER

END

MANIPOLAZIONE DELLE VISTE

Le VISTE valgono bene per operazioni di lettura (SELECT) ma non per manipolare dati essendo tabella virtuale e non calcolate, tuttavia sotto alcune condizioni possiamo manipolare i dati:

- ① la vista contiene la chiave primaria
- ② la vista contiene gli attributi totali
- ③ la vista si basa su una sola tabella

Vediamo le 3 operazioni di manipolazione come agiscono su una vista che rispetta le proprietà.

CREATE VIEW Studenti_2

```
AS SELECT Matr, CF, nome, cognome  
FROM Studenti AS S
```

DELETE

Per la DELETE bastano ① + ③

DELETE FROM Studenti_2

```
WHERE matricola = "N86000001"
```

INSERT

Sono necessarie le 3 condizioni (la seconda implica la prima)

INSERT INTO Studenti_2

```
VALUES ('N86001000', 'Ciro', 'Esposito', ... )
```

Analogamente vale con l' UPDATE

Queste operazioni vengono tradotte in operazioni su Studenti

CLAUSOLA INSTEAD OF

La clausola INSTEAD OF puo' essere utilizzata **SOLO** su viste
e permette di inserire un trigger che agisca sulla vista quando
vengono violate le condizioni minime per la manipolazioni dei
dati su viste sostituendosi alla violazione.

Vediamo un applicazione:

STUDENTE (Matr, Nome, Cognome)

ANAGRAFE (Matr, CF, Residenza, DataN, ...)

Vogliamo unire le informazioni:

CREATE VIEW Completo

AS SELECT

FROM STUDENTE S INNER JOIN

ON ANAGRAFE AS A

Questa vista e' basata su 2 tabelle (condizione ③), gestiamola con
un trigger:

CREATE TRIGGER in-studenti

INSTEAD OF INSERT ON completa OR

UPDATE ON completa OR

DELETE ON completa

FOR EACH ROW

BEGIN

IF INSERTING

INSERT INTO Studente (Matr, Nome, Cognome)

VALUES (:NEW.Matr, :NEW.Nome, :NEW.Cognome);

INSERT INTO Anagrafe (Matr, CF, Nome, Cognome, ---)

VALUES (:NEW.Matr, :NEW.CF, :NEW.Nome, :NEW.Cognome, ---);

ENDIF

IF DELETING

DELETE FROM Anagrafe

WHERE Matr = :OLD.Matr (NEW non esiste)

DELETE FROM Studente

WHERE Matr = :OLD.Matr

ENDIF

DSS

L'unico modo per annullare un'operazione è inserire uno trigger
sollevando un'eccezione, quindi con la clausola BEFORE
(si ricordi che il trigger **NON** si sostituisce all'operazione, solo
l'instead of lo fa ma con le viste)

Ese.

Fissato un nodo xxx scrivere tutti i suoi nodi discendenti
TREE (CodT, root)

NODO ($\text{CodT}, \text{Etichetta}, \text{CodN}$)

ARCO ($\text{CodT}, \text{N-IN}, \text{N-OUT}$)

Fissato xxx possiamo fare una query su arco per trovare i figli
e fare un'altra query con arco per trovare i figli dei figli e
così via ma dobbiamo sapere quante volte eseguirlo.

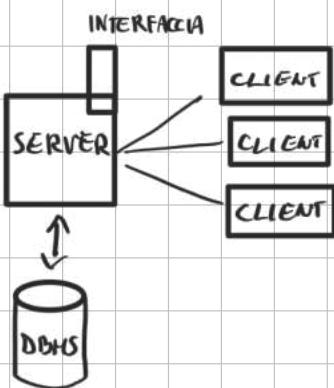
Quindi non si può scrivere una sola interrogazione per ogni
istanza, in questo ci aiuteranno i linguaggi utilizzati nei
trigger (java, PLSQL,...)

ARCHITETTURE DI UNA BASI DI DATI

Essendo il DBMS un deposito di dati, la logica dell'applicazione è demandata ai CLIENT.

Abbiamo due Tipi di architettura:

① SERVER-CLIENT



I servizi messi a disposizione dal DBMS sono di **QUERY** e **DATA MANIPULATION** (**INSERT, DELETE...**), inoltre ci fornisce un **API** (application programming interface) avendo delle interfacce attraverso le quali un linguaggio di programmazione si interfaccia col server. Le API sono generalmente standard, noi vedremo JDBC (Java Data Base Connectivity).

② SQL + strutture di controllo simili a quelle dei linguaggi di programmazione

In questo caso la responsabilità della logica è ancora del DBMS che associa con strutture di controllo alle istruzioni di base di SQL. Queste strutture consistono in:

- Sequenze di istruzioni
- Costrutti condizionali
- Costrutti iterativi

Queste funzionalità vengono implementati attraverso linguaggi di programmazione **PROPRIETARI** (del DBMS) che sono diversi a seconda del DBMS. Per Oracle abbiamo PLSQL, per PostgreSQL abbiamo PSQL.

LINGUAGGI PROPRIETARI

È fondamentale chiarire delle proprietà dei linguaggi di programmazione **PROPRIETARI**:

- Sono proprietari del SERVER

Questo implica che la logica degli applicativi non viene implementato

● Esecuzione sul SERVER

L'esecuzione non avviene sul client, questo significa che gli applicativi (scritti in java ad esempio) funzionano assieme a questi linguaggi tramite gli API.

Noi ci concentreremo sui linguaggi **PROCEDURALI** e non orientati a oggetti, in particolare useremo PLSQL.

TIPI DI VARIABILI IN PLSQL

Sono gli stessi che si possono usare nelle tabelle

Questo ci permette un **ALLINEAMENTO PERFETTO** tra i tipi delle variabili e i contenuti delle colonne. Questo **NON** avviene con comuni linguaggi di programmazione, dove i tipi sono diversi e diversamente definiti.

Questo rappresenta un grande punto di forza dei linguaggi proprietari. La dichiarazione di una variabile ha lo schema

<nome-variabile> <tipo> [:= <valore>]

E.

```
CREATE TABLE Studente (Matricola, Nome, Cognome, ISEE)  
(Matricola VARCHAR2 (10)  
Nome VARCHAR2 (10)  
Cognome VARCHAR2 (10)  
ISEE INTEGER )
```

DECLARE

T_Mat Studente.Matricola % TYPE

T_ISEE INTEGER

BEGIN

Possiamo dichiarare il tipo (come per T.ISEE) che sarà lo stesso
dell'attributo, oppure possiamo **REFERENZIARE UN TIPO** in Tabella
così all'occasione se cambierà il tipo in Tabella manteniamo
l'allineamento (come per T_Mat)

Struttura del blocco

```
[DECLARE
    declaration_statements
]
BEGIN
    executable_statements
[EXCEPTION
    exception_handling_statements
]
END;
/
```

Semplice esempio di blocco

```
DECLARE
    v_width  INTEGER;
    v_height INTEGER := 2;
    v_area   INTEGER := 6;
BEGIN
    -- set the width equal to the area divided by the height
    v_width := v_area / v_height;
    DBMS_OUTPUT.PUT_LINE('v_width = ' || v_width);
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.PUT_LINE('Division by zero');
END;
/
```

L'eccezione ZERO_DIVIDE già gestisce l'errore, possiamo però riprogrammare quest'eccezione a nostro piacimento, come in questo caso mostrando a schermo "Division by zero"

TRASFERIRE DATI DAL DB ALLE VARIABILI

L'operazione SELECT restituisce le righe di una tabella.

I risultati possibili sono 3:

- 0 righe

Ammo una tabella vuota con codice ResultSet

- >1 righe

Non abbiamo una struttura dati per contenere l'intero risultato

- 1 riga

Il costrutto sarà:

SELECT expr₁, ..., expr_n INTO var₁, ..., var_m

FROM ..

WHERE ..

Le variabili devono essere precedentemente dichiarate e devono essere dello stesso tipo dei valori della SELECT

OSS

All'interno del corpo procedurale questo è l'**UNICO** modo di usare la SELECT, non c'è accettata da sola (senza variabili)

CASO RIGHE >1

Per gestire il caso critico ci aiutiamo con un **BUFFER** che conterrà la tabella risultante che verrà poi letta riga per riga.

Il buffer è scandito da un **CURSOR**

CURSORI

Possiamo inserire le righe in un CURSORI.

Dichiarare il cursori NON significa eseguire l'interrogazione, questo avviene con il comando OPEN <nome_cursor> che aggiorna il buffer inserendo il risultato dell'interrogazione e pone la posizione corrente del buffer alla prima riga.

La chiusura del cursori si effettua con CLOSE <nome_cursor>.

Il trasferimento della riga corrente nella variabile si effettua con FETCH <nome_cursor> INTO var₁, ..., var_k

E.s.

DECLARE

Esp. Mat Studenti . Matricola % TYPE

Esp. Nome Studenti . Nome % TYPE

CURSOR Scansione_Esposito IS

SELECT S.Matricola, S.Nome

FROM Studenti S

WHERE S.Cognome = "Esposito"

BEGIN

OPEN (Scansione Esposito)

FETCH ScansioneEsposito INTO Esp. Mat, Esp. Nome

CLOSE (Scansione Esposito)

OSS

Nella DECLARE possiamo dichiarare il cursore nel seguente modo:
riguardante ScansioneEsposito % ROWTYPE

Questa espressione crea un record di tipo compatibile con la riga
della SELECT associata al cursore, senza dover dichiarare molteplici
variabili

E.s.

FETCH ScansioneEsposito INTO riguardante

Per accedere ai campi del record accederemo semplicemente con
riguardante.campo

RICORDA

NON si usa la SELECT nel corpo procedurale.

Useremo SELECT..INTO o un CURSOR, **NON** possiamo usare
la SELECT INTO per SELECT che restituiscano più di una riga

COSTRUTTI ITERATIVI

Si suddividono in:

- LOOP
- WHILE
- FOR
- FOR per cursore

LOOP

la struttura del loop e':

LOOP

<istruzione₁>

--> EXIT WHEN <condizione d'uscita>

--> <istruzione_n>

ENDLOOP

La funzione EXIT WHEN puo' essere posta (e quindi controllata) in qualsiasi parte del blocco di codice.

Ad esempio, per simulare un REPEAT-UNTIL (che non e' disponibile come struttura) faremo il controllo come ultima operazione.

Si puo' inserire all'interno del corpo del codice un'istruzione
CONTINUE [WHEN] che non esegue tutta la parte successiva all'
istruzione fino all'ENDLOOP, riaggiungendo il LOOP da capo

ts.

Loop

istruzione 1

CONTINUE

EXIT WHEN c

istruzione 2

ENDLOOP

Questo e' un loop infinito

WHILE

WHILE <condizione>

Loop

ENDLOOP

CICLO FOR

FOR <variabile> IN <expr₁> .. <expr₂>

Loop

ENDLOOP

L'incremento è sempre unitario, il valore di variabile va da expr₁ a expr₂.

Si può utilizzare un for decrescente usandolo FOR <variabile> REVERSE
in cui expr₁ indica la fine e non l'inizio.

La variabile può essere dichiarata implicitamente nel FOR

E' possibile effettuare una scissione completa di un unsare
ponendo al posto di <expr₁..<expr₂> il unsare che
vogliamo scissione.

E.s.

TREE (CodT, ROOT)

NODO (CodT, CodN, Label)

ARCO (CodT, CodA, Sorg, Dest)

- Scriviamo una funzione che riceve in ingresso "albero" e "nodo" e

restituisce la somma delle labels dei nodi da nodo a radice

Scriviamo una **STORED FUNCTION** avendo una funzione salvata all'interno del database e che **NON** modifica il contenuto delle tabelle.

CREATE [OR REPLACE] FUNCTION

SommaNodi UP (albero TREE.CodT%TYPE, modo TREE.CodN%TYPE)

RETURN INTEGER IS

Res INTEGER := 0



nella procedura il **DECLARE**

e' sottinteso

Curn NODO.CodN%TYPE

Padre NODO.CodN%TYPE

BEGIN

Curn := modo

LOOP

SELECT N.label + Res INTO Res

FROM NODO N

WHERE N.CodT := albero

AND N.CodN = Curn;



Assegnamento alla variabile

SELECT A.Song INTO Padre ← Verifico che ci sia il padre

FROM ARCO A

WHERE A.CodT = albero AND

A.Dest = cum

← Risolviamo l'albero

EXIT WHEN Padre IS NULL

Cum := Padre

END LOOP

RETURN Res

END

Siccome la nostra funzione non altera la base di dati,

possiamo riutilizzarla a nostro piacimento, ed esempio:

SELECT SommaNodi ('1000', N.CodN)

FROM Nodo N

WHERE N.CodT = '1000'

- Vogliamo trovare la somma di tutte le labels di tutti i nodi.

Dobbiamo effettuare una visita in ampiezza perché con la visita in profondità non potremmo usare la SELECT INTO (ogni nodo ha

più figli). Aintiamoci tenendo traccia del livello di ogni nodo
e inserendo etichetta e livello in una struttura TMP (CodN, level)

CREATE [OR REPLACE] FUNCTION

SommaNodi DOWN (albero TREE.CodT%TYPE, modo TREE.CodN%TYPE)

RETURN INTEGER IS

level SMALLINT := 0

BEGIN

DELETE FROM TMP;

INSERT INTO TMP

VALUES (:modo, 0);

LOOP

INSERT INTO TMP

(SELECT A.Dest, level+1

FROM ARCHI A JOIN

TMP T ON

A.Sorg = T.CodN

WHERE A.CodT = 'albero' AND

T.livello = level)

SELECT COUNT(*) INTO Check
FROM TMP T

← Controllo che abbia inserito
qualche nodo

WHERE T.livello = level + 1

EXIT WHEN Check = 0

← Se non ho inserito nulla
ho finito l'albero

level := level + 1

← Altrimenti passa al livello
successivo

ENDLOOP

Abbiamo riempito TMP

SELECT SUM(N.labell) INTO Res

Sommo i nodi solvuti in TMP

FROM TMP T JOIN

NODO N ON

T.Cod/N = N.Cod/N

RETURN Res

END

• Cerchiamo il percorso di somma massima dalle foglie alla radice
usando la funzione SommaNodi UP

CREATE FUNCTION MaxPath (albero TREE.CodT%TYPE)

RETURN INTEGER

DECLARE

Result INTEGER := 0;

foglia_c NODO.CodN%TYPE;

CURSOR Foglie IS

(SELECT

FROM NODO N

WHERE N.CodT = albero AND

NOT EXIST (SELECT

FROM ARCO A

WHERE A.CodT = albero

AND A.Sorgente = N.CodN))

BEGIN

OPEN (Foglie);

Nel cursore ora abbiamo le foglie

LOOP

EXIT WHEN Foglie % NOT FOUND *Eci quando il cursore e' vuoto*

FETCH Foglie INTO foglia_c; *Inseriamo la riga del cursore nelle variabili*

Curr := SommaNodi_UP(albero, foglia_c)

IF current > Result

THEN result := current

ENDIF

ENDLoop

CLOSE (Foglie)

RETURN Result

END

OSS

Per i cursori c'e' una funzione *cursore % COUNT* che ci dice
quante righe sono state lette fino ad ora

Se dobbiamo effettuare una scansione totale c'e' un metodo

piu' caoso:

BEGIN

FOR I IN Pgfile

loop

current := SommaNodi(OP(albero, I.CdN))

IF result < current THEN

result := current

ENDIF

ENDLOOP

Questo far dichiara esplicitamente una variabile compatibile con la struttura del cursore.

Il cursore viene aperto e chiuso automaticamente.

Questo puo' essere usato solo per scansioni complete, e' l' analogo del "For each" in java.

Possiamo inoltre definire il cursore esplicitamente nel for:

FOR I IN (SELECT...)

② 2) ALTER TABLE PROPER

ADD CONSTRAINT C1

FOREIGN KEY (CodeSame)

REFERENCES INSEGNAM(CodI)

ADD CONSTRAINT C2

FOREIGN KEY (CodeSameProper)

REFERENCES INSEGNAM(CodI)

METADATI

Il DBMS memorizza i METADATI e le strutture.

I metadati sono a loro volta salvati in Tabelle del DBMS, questo ci permette di interrogarli e consultarli come dati (SELECT...) le Tabelle relazionali messe a disposizione dipendono dal DBMS. Oracle le fornisce come viste.

Vediamo un esempio

Column	Datatype	NULL	Description
OWNER	VARCHAR2 (30)	NOT NULL	Owner of the external table location
TABLE_NAME	VARCHAR2 (30)	NOT NULL	Name of the corresponding external table
LOCATION	VARCHAR2 (4000)		External table location clause
DIRECTORY_OWNER	CHAR (3)		Owner of the directory containing the external table location
DIRECTORY_NAME	VARCHAR2 (30)		Name of the directory containing the external table location

Ogni Tabella messa a disposizione inizia con OWNER

Grazie a queste informazioni possiamo ottenere molto più facilmente, ad esempio nella Tabella TABLES_COLUMNS abbiamo a disposizione righe che ci dicono quanti valori distinti abbiamo o quanti valori NULL

DICTIONARY

I METADATI si trovano nel DIZIONARIO dove sono definiti.

In particolare il dictionary dipende dal dbms, ed è rappresentato in modo relazionale.

La consultazione dei metadati avviene attraverso viste, quindi non è possibile modificare i metadati della vista.

Vediamo alcune tabelle messe a disposizione.

USER_TABLE

Dispone le tabelle create da USER. Esiste anche la ALL_TABLES che si svilupola dello user.

Alcuni degli attributi (colonne) di questa tabella sono OWNER

(indica il proprietario di una tabella), TABLE_NAME, BLOCKS

(numero di blocchi occupati dalla Tabella, i blocchi sono l'unità di memoria delle tabelle) e NUM_ROWS (numero di righe).

USERTAB_COLUMNS

È una tabella che ci dà informazioni sulle colonne.

Gli attributi sono: TABLE_NAME (chiave esterna che riferisce

TABLE_NAME della tabella USER_TABLE), COLUMN_NAME, COLUMN_ID (di solito indica anche la posizione della colonna all'interno della Tabella), DATA_TYPE, NULLABLE (indica se puo' essere NULL o no, ha valori "Y" o "N"), NUM_NULLS (numero di righe NULL), NUM_DISTINCT (numero di righe con valore diverso) etc.

VINCOLI SUI METADATI

I vincoli che scriviamo su una tabella sono contenuti in una tabella USER_CONSTRAINTS.

Tra gli attributi abbiamo: OWNER, CONSTRAINT_NAME, TABLE_NAME. L'attributo CONSTRAINT_TYPE e' costituito da un solo carattere e indica nel seguente modo il tipo di vincolo: P= Primary key, U= unique, R= foreign key . Per il vincolo CHECK abbiamo un attributo SEARCH_CONDITION e ha come valore la stringa del vincolo stesso.

VINCOLO FOREIGN KEY

L'integrità referenziale e' gestita sempre in USER_CONSTRAINTS

attraverso gli attributi:

R_OWNER, R_CONSTRAINTNAME (e' il riferimento al vincolo di insieme o PK di riferimento), DELETE_RULE (esplicito cosa fare ON DELETE).

STRUTTURA DEI VINCOLI

I vincoli R,U,P sono definiti strutturalmente nella colonna

USER-CONS-COLUMNS.

Alcuni degli attributi sono TABLE_NAME, CONSTRAINT_NAME, POSITION e COLUMN_NAME.

Ese.

① Data una tabella TABELLA, si scriva una funzione che restituisce la definizione della chiave primaria di TABELLA

CREATE FUNCTION PK_NUM(tabella IN USER-TABLE%TABLE-NAME) RETURN VARCHAR2(1000)

(Risultato: VARCHAR2(300) := 'CONSTRAINT'

l'espressione inizia con
CONSTRAINT...

Salva il nome del vincolo

Supporto: VARCHAR2(300) := ''

ci serve solo per verificare ci siano effettivamente PK.

CHECK: INTEGER

BEGIN

```
SELECT COUNT * INTO CHECK  
FROM USER_CONSTRAINTS U
```

restituisc 0 o 1 per questo
caso la SELECT INTO

```
WHERE U.TABLE_NAME=Tabella AND  
U.CONSTRAINT_TYPE="P"
```

```
IF CHECK>0 THEN
```

Se c'e' una PK

```
SELECT U.constraint_name INTO Supporto  
FROM USER_TABLE U
```

```
WHERE U.TABLE_NAME=Tabella AND  
U.CONSTRAINT_TYPE="P"
```

risultato := risultato || ' ' || supporto || PRIMARY KEY ()

Salviamo in supporto il nome del vincolo che ci servirà dopo per definire
gli attributi che dobbiamo aggiungere in risultato.

risultato è la stringa finale che manipoliamo con la concatenazione ||.

Per concatenare l'elenco degli attributi ci aiutiamo con un cursore che
inizializziamo stesso nel for dove lo usiamo.

La variabile I sarà del tipo di ritorno della query

```
FOR I IN (SELECT C.ColumnName  
          FROM USER_CONS_COLUMNS C  
         WHERE C.TABLE_NAME = TABELLA AND  
               C.CONSTRAINT_NAME = Supporto  
         ORDER BY C.POSITION) le colonne sono ordinate
```

Loop

```
risultato := risultato || I.ColumnName || ',';
```

ENDLOOP

```
risultato := RTRIM (risultato, ',') || ')';
```

```
RETURN risultato
```

taglia l'ultima occorrenza di
'', che ci dà fastidio, poi aggiungo
la parentesi per chiudere

Se CHECK=0 non facciamo nulla

ELSE

```
RETURN ''
```

ENDIF

END

② Eseguiamo lo stesso esercizio ma con i vincoli di unicità (che sono più di uno a differenza della PK).

Ci serviranno due cursori, uno per i vincoli e uno per gli attributi del vincolo.

```
CREATE FUNCTION U-DUMP( tabella IN USER-TABLE%TABLE-NAME) RETURN VARCHAR(300)  
(Risultato :VARCHAR2(300) := ''
```

```
CHECK : INTEGER
```

```
BEGIN
```

```
SELECT COUNT * INTO CHECK  
FROM USER_CONSTRAINTS U
```

Controlliamo ci sono vincoli di unicità

```
WHERE U.TABLE_NAME=Tabella AND  
U.CONSTRAINT_TYPE='U'
```

```
IF CHECK >0
```

```
FOR U IN (SELECT C.CONSTRAINT_NAME
```

Scansione dei vincoli

```
FROM USER_CONSTRAINTS C
```

```
WHERE C.TABLE_NAME=Tabella AND  
C.CONSTRAINT_TYPE='U' )
```

LOOP

definizioni precedenti

risultato := risultato || 'CONSTRAINT' || U.Constraint_name || 'UNIQUE ('

FOR A IN (SELECT H.Column_name aggiungiamo gli attributi

FROM USER_CONS_COLUMNS H

WHERE H.Table_name = TABELLA AND

H.Constraint_name = U.Constraint_name

ORDER BY H.Position)

Loop

risultato := risultato || A.Column_name || ','; complettiamo la stringa

risultato := RTRIM (risultato, ',') || ','; coneggiamo ogni riga

END Loop

END Loop

risultato := RTRIM (risultato, ',')

rimuoviamo l'ultima virgola.

RETURN Risultato

ELSE

Return ''

ENDIF

END

Si noti che la query di A e' **PARAMETRICA** perché dipende dal valore del parametro U.

OSS

Per il dump delle foreign key prendiamo il nome del vincolo in USER_CONSTRAINTS, gli attributi che compongono la fk li troviamo in USER_CONS_COLUMNS come prima, le PK referenziate le prendiamo da R_CONSTRAINT_NAME per ricavarci poi gli attributi (associando R_CONSTRAINT_NAME a USER_CONSTRAINTS).

CREATE FUNCTION R-DUMP(tabella IN USER-TABLE%TABLE-NAME) RETURN VARCHAR(300)

(Risultato : VARCHAR2(300) := ''

SupportoConstraint : VARCHAR2(300) := ''

SupportoTabella : VARCHAR2(300) := ''

CHECK: INTEGER

BEGIN

```
SELECT COUNT * INTO CHECK  
FROM USER_CONSTRAINTS U  
WHERE U.TABLE_NAME=Tabella AND  
U.CONSTRAINT_TYPE='R'
```

IF CHECK > 0

```
FOR U IN (SELECT C.CONSTRAINT_NAME  
FROM USER_CONSTRAINTS C  
WHERE C.TABLE_NAME=Tabella AND  
C.CONSTRAINT_TYPE='U')
```

Loop

risultato := risultato || 'CONSTRAINT' || U.Constraint_name || 'FOREIGN KEY('

FOR A IN (SELECT H.Column_name aggiungiamo gli attributi
FROM USER_CONS_COLUMNS H
WHERE H.TABLE_NAME=TABELLA AND
H.CONSTRAINT_NAME = U.Constraint_name
ORDER BY H.POSITION)

Loop

risultato := risultato || A.Column_name || ',';

ENDLOOP

risultato := RTRIM (risultato, ',') || ')' || 'REFERENCES';

SELECT UC.CONSTRAINT_NAME INTO SupportoConstraint

FROM USER_CONSTRAINTS UC

WHERE UC.CONSTRAINT_NAME = U.R_CONSTRAINT_NAME

risultato := risultato || SupportoConstraint || '(';

SELECT UC.TABLE_NAME INTO SupportoTabella

FROM USER_CONSTRAINT UC

WHERE UC.CONSTRAINT_NAME = SupportoConstraint

FOR A IN (SELECT H.Column_name

FROM USER_CONS_COLUMNS H

WHERE H.TABLE_NAME = SupportoTabella

H.CONSTRAINT_NAME = SupportoConstraint

ORDER BY H.POSITION)

Loop

risultato := risultato || A.Column_name || ',';

ENDLOOP

risultato := RTRIM (risultato, ',') || ',';

ENDLOOP

RETURN risultato

SQL DINAMICO

Siamo in ambito **PROCEDURALE** (lato del linguaggio di programmazione) fino ad ora abbiamo visto il lato statico di SQL, ovvero in cui è il programmatore che scrive l'intera procedura.

Nel caso di SQL **DINAMICO** la procedura è completata ed eseguita a run-time.

La caratteristica principale è che i comandi SQL sono dati come **STRINGHE** dati. Questo implica che la composizione dell'istruzione avviene attraverso una variabile **VARCHAR**.

PREPARAZIONE DEI COMANDI DINAMICI

Per eseguire il comando usiamo il seguente costrutto

PREPARE [nome_comando]

EXECUTE [nome_comando]

Il comando **PREPARE** precompila la procedura senza eseguirlo, questo avviene solo una volta.

Il comando **EXECUTE** esegue la procedura solo dopo aver eseguito il prepare e può essere chiamato numerose volte.

E' possibile eseguire un'esecuzione fissa con il comando
EXECUTE IMMEDIATE [nome-comando] , rischiando però di ottenere errori.

COMANDI CON PARAMETRI

Se il comando usa parametri possiamo passare alla sequenza
queste variabili grazie alla formula

EXECUTE [nome-comando] **USING** [lista parametri attuale]

IMPLEMENTAZIONE PLSQL

Vediamo come implementare SQL dinamico in PLSQL con un
esempio

Scriviamo una procedura che riceve il nome di una tabella e
una stringa con nomi di attributi separati da ';' e costruisce
un vincolo di chiave primaria (dobbiamo verificare che gli attributi
siano presenti in tabella)

```
CREATE PROCEDURE AddPK( Tabella IN ALL.TABLES.NAME%TYPE,  
                        Stringa IN VARCHAR(1000) )
```

(Attributo ALL_TAB_COLUMNS.COLUMN_NAME % TYPE

errore - vuoto EXCEPTION

comando VARCHAR(1000)

check INTEGER := 0

concatente VARCHAR(100)

BEGIN

SELECT COUNT(*) INTO check

FROM ALL_TABLES T

WHERE T.TABLE_NAME = tabella

IF check > 0

comando := 'ALTER TABLE' || tabella || 'ADD CONSTRAINT pk' || tabella ||
'PRIMARY KEY (' ||

concatente := Stringa ;

LOOP

EXIT WHEN LENGTH(concatente) = 0

IF INSTR(concatente, ',') = 0 THEN

Attributo := concatente

concatente := "

ELSE

Attributo := SUBSTR (comando, 1, INSTR (comando, ',') - 1)

comando := SUBSTR (comando, INSTR (comando, ',') + 1)

ENDIF

SELECT COUNT (*) INTO check

FROM ALL_TABLES T

WHERE T.TABLE_NAME = Tabella AND

T.COLUMN_NAME = Attributo

IF check = 0

RAISE eccezione_mete

ELSE

comando := comando || Attributo || ',';

ENDIF

ENDLOOP

comando = RTRIM (comando, ',') || ')';

EXECUTE IMMEDIATE comando

EXCEPTION

WHEN errore_mete THEN

DBMS_OUTPUT.PUTLINE ("ERRORE NEI METADATI: TABELLA O ATTRIBUTO")

FUNZIONI DI MANIPOLAZIONE DELLE STRINGHE

Alcune delle funzioni più utili per la manipolazione delle stringhe sono:

- LENGTH (String)

- INSTR (Stringa, caratteri, [posizione, [iterazione]])

Restituisce la posizione della prima occorrenza di 'caratteri' in 'Stringa'. La variabile 'posizione' indica il punto di inizio della ricerca, il parametro 'iterazione' indica quale occorrenza prendere in considerazione (1^a, 2^a, 3^a...). Queste ultime due sono opzionali.

- SUBSTR (Stringa, inizio, [lunghezza])

Restituisce la sottostringa di lunghezza 'lunghezza' (se non esplicitata è totale) che parte dalla posizione 'inizio' della stringa 'Stringa'.

- LOWER / UPPER (String)

Normalizza la stringa in upper o lower case.

- TO_CHAR (Var)

- LTRIM / RTRIM (String, setcaratteri)



● LPAD / RPAD

Aggiunge ad



Queste funzioni sono tra loro concatenabili.

Si ricordi che le stringhe iniziano dalla posizione 1.

E.s.

A = "par1,par2,par3"

Vogliamo estrarre par2 da A. Questa è presente tra la 1^a e la 2^a occorrenza di ','. Immaginiamo di avere due variabili prima e seconda di tipo INTEGER.

prima := INSTR(A, ',') → 5

seconda := INSTR(A, ',', 1, 2) → 10

par2 := SUBSTR(A, prima + 1, seconda - prima)

Una forma più coesa è

SUBSTR(A, INSTR(A, ',') + 1, INSTR(A, ',', 1, 2) - INSTR(A, ',', 1, 2))

PANORAMICA SQL DINAMICO

Ricapitolando, grazie all' SQL dinamico possiamo comporre le istruzioni nel corpo procedurale, ricordando che queste sono stringhe. L'esecuzione è avviata tramite i comandi **PREPARE** (che effettua un controllo sintattico e semantico dell'istruzione) ed **EXECUTE**. In alternativa abbiamo l'**EXECUTE IMMEDIATE**.

SQL DINAMICO PER SELECT

Il contenuto dello **SELECT** va salvato in un cursore.

Vogliamo creare il nome per un cursore. Utilizziamo un tipo di dato che riferisce un cursore, questo è **SYS_REF_CURSOR**.

Possiamo esplicitare il "tipo" del cursore (**STRONGLY TYPED**) oppure ometterlo (**WEAKLY TYPED**), per l'esplicitazione aggiungiamo l'espressione **RETURN tipo** dove **Tipo** sarà un **datatype** della tabella.

La definizione del cursore avviene con il comando **OPEN** ed una query, noi componiamo la query salvandola in una variabile.

① **name_riferimento SYS_REF_CURSOR [RETURN tipo]**

② **OPEN name_riferimento FOR comando.**

E.

⑦ Lavoriamo su una tabella fissata con colonne A,B,C,D tutte VARCHAR

CREATE PROCEDURE Interoga_TAB (colonna ALL_TAB_COLUMNS, column-name%TYPE

(mocursore SYS-REF-CURSOR RETURN TAB%ROWTYPE

riga TAB%ROWTYPE

comando VARCHAR(200)

BEGIN

comando := 'SELECT * FROM TAB WHERE'||colonna||'='||valore;

OPEN mocursore FOR comando;

Loop

EXIT WHEN mocursore%NOTFOUND

FETCH mocursore INTO riga

DBMS_OUTPUT.PUTLINE (riga.All riga.B||riga.C||riga.D);

END loop

END

COSTO DI OPERAZIONI IN BD

I file sono suddivisi in **BLOCCI**. Ogni blocco contiene n record.

L'unità di costo per un'operazione in una base di dati è data

dal numero di blocchi scambiati tra memoria principale e secondaria per eseguire la suddetta operazione.

I record possono essere memorizzati in modo **ORDINATO** (fisicamente) o meno.

- Se ad esempio abbiamo una tabella T con un attributo A unico **NON** di ordinamento fisico (T non è ordinato o se lo è, non lo è rispetto A) e vogliamo eseguire la seguente operazione $\sum_{A=v}^S(T)$ dobbiamo scandire **TUTTO** il file.

Il tempo medio di scansione è $\lceil \frac{\text{NUM_BLOCK}}{2} \rceil$

- Supponendo che A inceca sia di ordinamento, il costo medio sarà $\log_2(\text{NUM_BLOCK}(T))$.

Ovviamente l'attributo d'ordine fisico è **UNICO** (non posso ordinare rispetto più attributi), quindi per rendere più efficienti le interrogazioni dobbiamo creare degli **ORDINAMENTI LOGICI**

INDICI

Per creare un ordinamento logico utilizzeremo delle strutture aggiuntive che mantengono ordine alla base di dati e dipendono dall'interrogazione che vogliamo effettuare, queste strutture sono dette **STRUTTURE DI INDICIZZAZIONE**.

Gli **INDICI** di solito sono associati alle **PRIMARY KEY**; quando dichiariamo una PK implicitamente viene associato un indice. La struttura d'indizzazione è un **B⁺ TREE**.

Se come l'allocazione di questa struttura è dispendiosa, dichiariamo una PK **SOLO** se ci serve un vincolo di unicità o un vincolo di **FOREIGN KEY**.

Il **B⁺ TREE** assicura una verifica efficiente di questi vincoli grazie all'indizzazione, al costo di verificare l'inserimento dei dati e di modificare la struttura quando vengono modificate le tabelle.

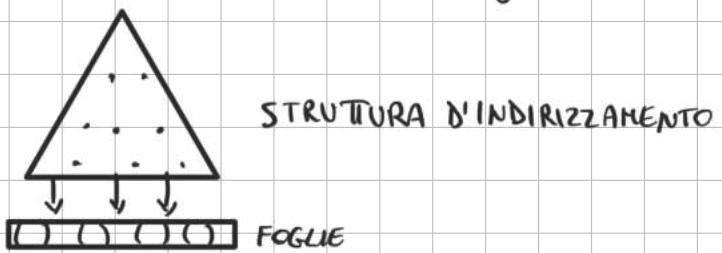
Il dominio della pk ovviamente dev'essere ordinato.

STRUTURA DEGLI INDICI

Supponiamo per comodità che la pk sia un **INTEGER**.

Un indice e' un record che contiene il valore dell'attributo (ordinativo) e un codice **RID** (record identifier).

Queste coppie sono ordinate nel **B⁺ TREE** nel seguente modo:



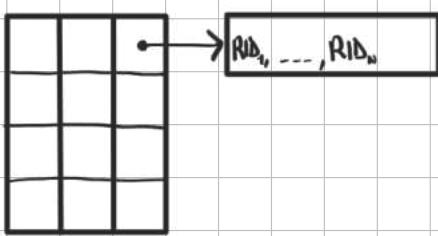
Il **B⁺ TREE** e' un albero **BILANCIATO**, le foglie contengono le PK e i RID. Ogni inserimento viene fatto nelle foglie e modifica tutta la **STRUTTURA D'INDIRIZZAMENTO**, questa e' un bordello l'unica cosa che c'interessa e' che l'inserimento e' logaritmico sull'altezza anziché sul numero di nodi.

INDICI DI ATTRIBUTI NON UNICI

Supponiamo una tabella senza attributi unici.

Si puo' creare un indice su un attributo scelto.

Tra le strutture d'individuazione possibili vediamo l'**INDICE A LISTE INVERTITE**. La struttura e' così composta:



Il record contiene: una lista ordinata dei valori delle primary key, il numero di record che hanno quel valore e infine un riferimento a una lista dei RID associati.

Essendo la lista delle PK unica e ordinata potrei voler costruire un B⁺ per accedere a questi campi.

DICHIARAZIONE DEGLI INDICI

In questo secondo caso la dichiarazione dell' indice dev' essere esplicita. La sintassi e'

CREATE INDEX indice ON tabella (attributo)

Se vogliamo un indice di unicita' su un attributo che non e' PK :

CREATE UNIQUE INDEX indice ON tabella (attributo)

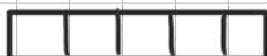
OSS

- Per il costo che implica, conviene istanziare indici **SOLO** se vengono effettuate molte operazioni su quell' attributo e soprattutto se e' fortemente **DISCRIMINANTE** (ovvero indica in modo selettivo un' istanza) Ad esempio su una tabella ESAMI, la matricola o la data sono discriminanti, mentre voto e lode no.
- Le modalità d'esecuzione sulle query dipendono dalla struttura di indicizzazione sottostante, troverà il modo più efficace. Per questo motivo è utile creare indici sugli attributi di giunzione, così l'ugualanza viene verificata più rapidamente.

INDICI BITMAP

Tra le strutture di indicizzazione per attributi non discriminanti.

la più comune è la **BITMAP INDEX**

VALORE 

↑ i esimo record

Ad ogni valore è assegnato un array, ogni cella corrisponde a un record. La cella conterrà 1 se quel record ha quel valore, altrimenti 0.

In questo caso il costo dipende dal dominio dell'attributo, in quanto avremo un array per ogni valore del dominio

Un ottimo esempio di indice bitmap e' ad esempio il sesso, per il quale avremo solo due array. Viceversa il codice fiscale e' un pessimo esempio

OSS

In definitiva, conviene dichiarare un indice quando un attributo viene richiamato spesso nelle where

TRANSAZIONI

ci sono casi in cui alcune operazioni devono essere eseguite in maniera atomica (vengono svolte tutte assieme oppure non vengono svolte). Per soddisfare questa necessità introduciamo il concetto di **TRANSAZIONE**.

Quando una transazione viene eseguita va espresso sempre nel campo se eseguire il **COMMIT** (conferma) o l'**ABORT**.

La transazione gestisce quindi lo scambio di informazioni tra memoria principale e secondaria, permettendo di salvare quindi l'operazione nella memoria secondaria in modo stabile.

Esiste un file detto **LOG** che indica qualsiasi cambio nella base di dati, garantendone la persistenza siccome tiene traccia del valore corrente (modificato) e quello precedente per eventualmente recuperare salvataggi meno recenti.

Es.

③ CREATE TRIGGER T

APFTER INSERT INTO LOG

FOR EACH ROW

WHEN NEW.OPERAZIONE = ABORT

BEGIN

BEGIN

FOR I IN (SELECT L.CODRISORSA , L.VALORE PRIMA
FROM LOG L

WHERE L.CODTRANSAZIONE = NEW.CODTRANSAZIONE AND
L.OPERAZIONE = 'MODIFICA'

ORDER BY DESC L.TEMPO)

LOOP

UPDATE RISORSE R

SET R.VALORE = I.VALORE PRIMA

R.STATO = 'UNLOCK'

WHERE R.CODRISORSA = I.CODRISORSA

ENDLOOP

④ CREATE FUNCTION F (TO INTERVAL) RETURN VARCHAR (1000) IS

(CHECK INTEGER := 0

Risultato VARCHAR (1000)

BEGIN

FOR I IN (SELECT R.CODTRANSAZIONE, R.CODRISORSA, R.

FROM RICHIESTE R NATURAL JOIN

ASSEGNAZIONE A

WHERE SYSTIME_R.TEMPO > TO)

Loop

SELECT * INTO CHECK

FROM ASSEGNAZIONE A NATURAL JOIN

WHERE A.CODRISORSA = I.CODRISORSA AND

Risultato = Risultato || I.CODTRANS || ','

