APPUNTI DI DATABASE

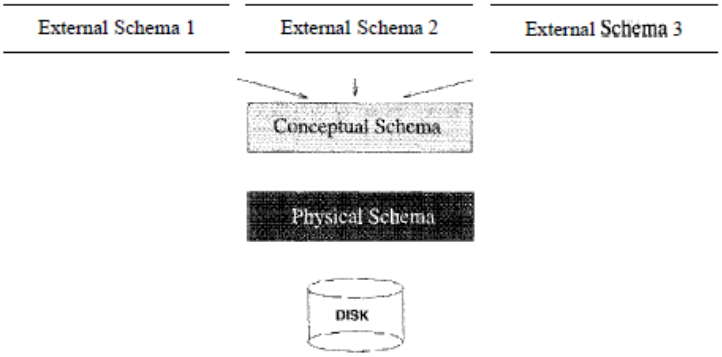
Il database è il luogo dove troviamo posizionate le nostre informazioni. Un database system è un software che serve per gestire questi dati. Di questo software si può servire anche un programma java, ma nella storia recente dell'informatica, i database sono spesso accessibili grazie alla rete. Un Database Management System (DBMS) è un pacchetto di software necessario per gestire le informazioni.

PERCHÉ UN DBMS?

I vantaggi di un sistema di gestione di database sono vari: per prima cosa una efficiente gestione dei dati consente di depositare nel disco una enorme quantità di dati in modo tale che occupino il meno possibile. Questo è reso possibile grazie alle diverse sofisticate tecniche di memorizzazione dati. Inoltre usare un DBMS significa avere maggiore sicurezza poiché un normale file sarebbe protetto per lo più dalla password del sistema operativo; L'integrità dell'informazione consente di accedere ai dati a classi di utenti differenti; condivisione delle stesse informazioni su più programmi, maggiore velocità di accesso alle informazioni. Inoltre con un database è possibile lavorare contemporaneamente da più postazioni (detta concorrenza). L'amministrazione del database è di tipo centralizzato, e questo consente di amministrare le informazioni in maniera più semplice. In caso di crash, il DBMS è in grado di ripristinare la situazione prima dell'evento. Tutti questi vantaggi possono essere ottenuti facilmente utilizzando un software di questo tipo. Anche per un programmatore sarebbe un notevole risparmio di lavoro, in quanto non dovrebbe testare tutte queste complesse funzionalità ogni qual volta progetta un software.
Solo in rare occasioni è sconsigliato usare un DBMS. Tra queste abbiamo la progettazione di particolari software che devono essere ottimizzati per un certo tipo di carico di lavoro. Un altro motivo per non usare un DBMS è quello di manipolare i dati in un modo non supportato dai linguaggi di query. In conclusione, nella maggior parte delle situazioni di gestione dei dati, un DBMS è uno strumento indispensabile.

DATA MODEL E SCHEMA

Un database deve memorizzare tutte le informazioni che rappresentano le entità di un certo sistema, spesso reale, come una banca o un'università. Per questa ragione è fondamentale la struttura di un modello: esso è una rappresentazione di un qualcosa di reale. Allo stesso modo, per i database i modelli sono un insieme di concetti che descrivono come l'informazione deve essere organizzata. Un data model è una collezione ad alto livello di costrutti per descrivere l'organizzazione dei dati. Tra i data model più utilizzati vi sono il "relational model", "l'object-oriented model" e il "relational-object model" che è un ibrido tra i due. Il modello relazionale è un modello di rappresentazione dei dati, che vede come parte centrale dell'informazione un "set di record". Uno schema è il modo in cui il database è costruito, ovvero una descrizione dei dati rappresentato nei termini di un data model. Uno schema per esempio, deve indicare i nomi dei campi e i tipi di dati presenti in essi. Ovviamente la descrizione di un'entità soggetta alla memorizzazione non deve essere completa, ovvero non tutti i campi devono apparire all'interno del database, ma solo quelli necessari per la corretta funzione del sistema. Ogni data model è descritto su tre diversi livelli di astrazione, quindi su tre diversi data schema. Questi livelli sono: schema concettuale (o logico), schema fisico e lo schema esterno (detti anche "view"). Per definire lo schema logico e lo schema esterno è necessario conoscere un "database language", ovvero un linguaggio interpretato dal DBMS, come SQL. Lo schema logico descrive come i dati sono memorizzati, ovvero contiene le informazioni riguardo le entità e le relazioni tra le tabelle che compongono il database. Lo schema fisico riguarda, invece, il modo con cui le informazioni del database vengono



memorizzate, suddivise in più file. Questo schema si occupa di creare degli indici per colonna o per righe per strutturare fisicamente i file e per una gestione veloce dell'informazione. Lo schema esterno permette di creare un particolare punto di vista del database, ovvero permette di personalizzare e di autorizzare la gestione dell'informazione ad un particolare tipo di utenti. Quando si progetta un database ci si occupa a progettare lo scema logico e lo schema fisico e non lo schema esterno. Un database gode di livello di astrazione e questo è garantito proprio dallo schema esterno. Esso consente una certa visibilità delle informazioni a seconda dell'utente che si pone davanti al software. Alcune informazioni infatti possono essere nascoste perché riservare o di inutile rilevanza a seconda dell'utente. In presenza di uno schema esterno, i dati non vengono memorizzati esplicitamente, ma al massimo vengono elaborati e combinati come necessario

L'INDIPENDENZA DEL DATABASE

I tre livelli di astrazione permettono di fare del database un software indipendente e accessibile da aree software distinte. A consentire ciò sono soprattutto lo schema logico e lo schema esterno.

LE QUERY

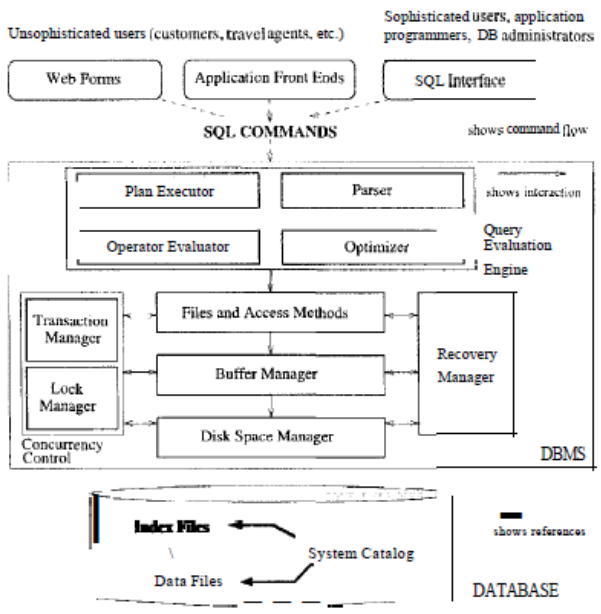
Una query è una stringa, scritta in un linguaggio apposito, che interroga il database. Un DBMS provvede a elaborare la richiesta e a rispondere correttamente restituendo le informazioni consentite.

ACID E LE SUE PROPRIETÀ

Con l'acronimo ACID si intendono i diversi problemi che possono emergere quando si tenta l'accesso al database in maniera concorrente, ovvero da più terminali contemporaneamente. Un buon software deve quindi gestire correttamente questi problemi. Quando una più utenti accedono al database simultaneamente, il DBMS deve ordinare le loro richieste delicatamente, senza generare conflitti. Quando l'utente 1 apporta delle modifiche all'informazione del database, l'utente 2 (in concorrenza) potrebbe non vedere la situazione in tempo reale. Il software deve necessariamente gestire situazione del genere piuttosto che poter eseguire una transizione (per transizione si intende una qualsiasi esecuzione richiesta da un utente, attraverso un applicativo) per volta, in quanto potrebbe degradare le performance del software. Queste operazioni simultanee sono molto delicate, perché generano facilmente dei "fail", ovvero degli errori di sistemi che portano, nelle peggiore delle situazioni, alla perdita di informazioni. Quando un DBMS crasha deve essere in grado di riportare le informazioni nello stato antecedente al drammatico evento. La transizione è l'unità base che costituisce una modifica di un database. Quindi, un DBMS non può effettuare transizioni parziali, e l'effetto di un gruppo di transizione equivale ad una serie di esecuzione delle stesse. Un buon DBMS deve non solo saper gestire una situazione concorrente, ma deve rendere il tutto trasparente all'occhio dell'utente, come se l'applicativo operasse singolarmente e come se il database fosse isolato. Esistono dei protocolli di protezione che stabiliscono delle regole da seguire da ogni transizione. Un modo per vincere il problema della concorrenza è quello di condividere blocchi di oggetti. Con tale condivisione è possibile vedere in tempo reale come l'informazione cambia.

LOG

Per diverse ragioni, il sistema potrebbe crashare e quando questo accade il DBMS deve poter ripristinare la situazione. Per farlo, esiste un file dove tutte le operazioni che vengono eseguite attraverso lo DBMS vengono registrate. Attraverso un file di log possiamo anche osservare quale sia l'errore e magari capire la causa di quest'ultimo. Il file di log è salvato nel disco rigido, dove è salvato il database. Il tempo richiesto per ripristinare il crash può essere ridotto gelando lo stato delle informazioni periodicamente nel disco. Questa operazione è chiamata checkpoint.



LA STRUTTURA DI UN DATABASE

Tutti gli applicativi (locali o web) usano un linguaggio di query per comunicare con il DBMS. Queste query vengono valutate e ottimizzate da una componente chiamata "Query Evaluation Engine". Al suo interno è presente un ottimizzatore che induce la ricerca delle informazioni richieste a seconda di come il database è stato memorizzato e produce un efficiente piano di valutazione della query. Per valutare una query, essa viene rappresentata come un albero di operatori relazionali. Un file che compone il database contiene pagine (o collezioni) di record. Il buffer manager si occupa di portare la "risposta" alla query in memoria per le future elaborazioni. Il livello più basso della struttura rappresenta dove i dati sono fisicamente memorizzati.

L'AMMINISTRATORE DI UN DATABASE

Ad utilizzare un database non ci sono, ovviamente, solo gli utenti finali dei vari applicativi, ma anche gli implementatori del sistema di database stesso. Tra questi ci sono i programmatori degli applicativi collegati al database e gli amministratori del database. I programmatori dei programmi facilitano l'accesso alle informazioni memorizzate nel database attraverso i loro software, quindi progettano gli "external schema" visti in precedenza. L'amministratore (DBA) è responsabile nel progettare lo schema logico e lo schema fisico, nel concedere autorizzazioni nelle modifiche del database e quindi della sua sicurezza, nel ripristino delle informazioni in caso di fail.

ER MODEL

Un *entity-relationship* model è un modello di gestione dei dati che ci permette di descrivere le informazioni presenti nel mondo reale in termini di oggetti e di specificare le relazioni che vi sono fra essi. Questo modello costituisce un punto verso la realizzazione del "design" del database. Il processo che vede la realizzazione del database può essere diviso in sei parti. La **prima** richiede l'analisi dei dati. Ovvero, è fondamentale capire quale tipo di informazione deve essere memorizzata all'interno del database e quali applicazioni devono essere costruire sopra di esse. La **seconda** fase prevede la schematizzazione della parte concettuale, ovvero la realizzazione del "logical schema". Le informazioni scelte devono essere usate per sviluppare una descrizione "ad alto livello", ovvero che nasconde ancora molti dettagli ma che danno l'idea allo sviluppatore del concetto che vi è dietro ogni relazione rappresentata. Il modello ER è uno dei più ad alti livelli usati per la realizzazione di un database in quanto mostra come gli utenti e gli sviluppatori concepiscono i dati. La **terza fase** consiste nello scegliere un DBMS corretto per implementare il nostro "logical model". Le successive fasi verranno trattate in seguito.

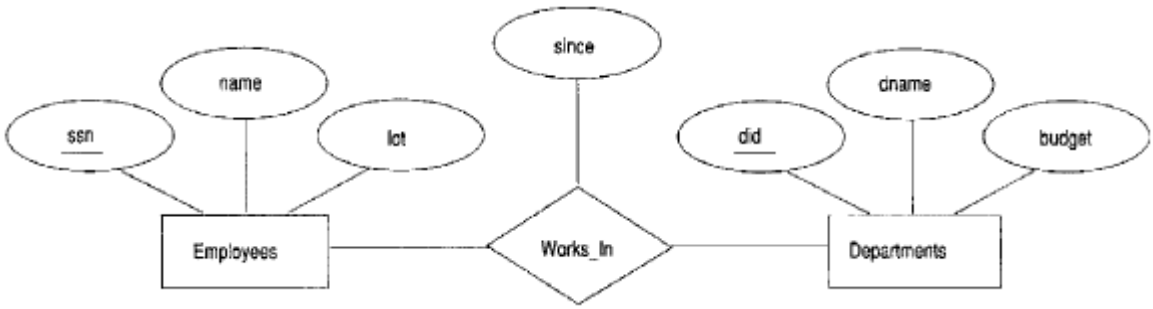
LE ENTITÀ E GLI ATTRIBUTI

Con il termine *entità* si intende indicare ciascun oggetto che nel mondo reale deve essere rappresentato nel database. Una collezione di entità è chiamata "*set*". Un entità è descritta utilizzando un set di *attributi*. La scelta degli attributi da coinvolgere riflette il livello di dettaglio dell'informazione che si vuole raggiungere. Per ciascun attributo che si vuole aggiungere ad un'entità, bisogna specificare il tipo di *dominio* che gli si vuole associare. Per esempio, il nome di un impiegato potrebbe essere impostato come una stringa di 20 caratteri. Per ogni set di entità, possiamo scegliere quale attributo sia una chiave (*key*). Una chiave è un

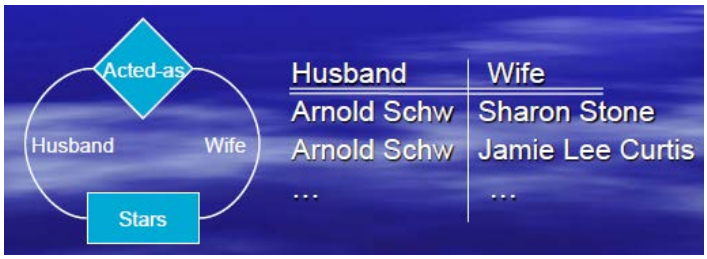
insieme minimo di attributi che identificano univocamente un entità in un set. Ciascun set di entità contiene almeno un set di attributi che permettono di identificare univocamente un'entità al loro interno. All'interno dell'ER model è possibile rappresentare un'entità con un rettangolo contenente il suo nome. Allo stesso modo è possibile rappresentare un attributo con un cerchio. Una chiave è rappresentabile sottolineando il nome dell'attributo scelto per tale ruolo.

LE RELAZIONI

Una *relazione* è un'associazione tra due o più entità. È possibile collezionare relazioni analoghe in un set di relazioni. Anche una relazione può avere degli attributi. Essi servono per memorizzare informazioni riguardanti la relazione a cui è associata. Nell'ER model una relazione è rappresentabile tramite un rombo.

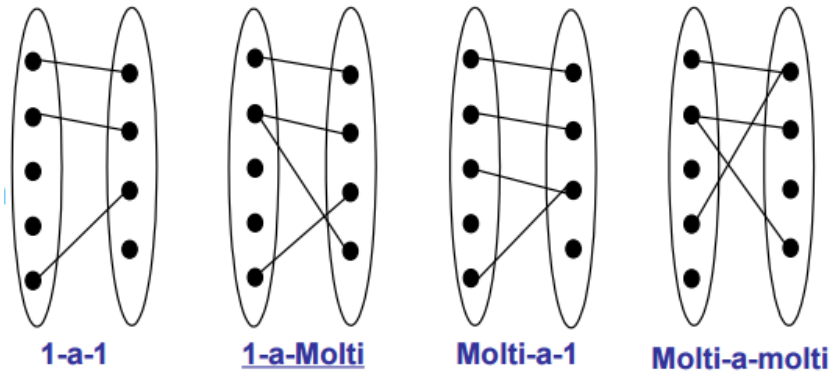


Una relazione può essere identificata dalla co-esistenza di due entità. La combinazioni di due attributi di due entità differenti infatti, possono costituire l'istanza della relazione. Si possono realizzare relazioni binarie o ternarie a seconda se si vuole mettere in relazione relativamente due o tre tipologie di entità. Alcune volte è possibile mettere in relazione due entità dello stesso set di entità, per esempio quando vogliamo distinguere i ruoli di due entità della stessa tipologia.



VINCOLI DI CHIAVI

Una relazione può essere impostata tramite dei vincoli da sottoporre alla chiave. La relazione è detta *one-to-many* se una entità è associata con molte entità. Così esistono le altre restrizioni come rappresentate in figura.



VINCOLI DI PARTECIPAZIONE

I vincoli di partecipazione indicano se obbligano la *partecipazione* di tutti i record alla relazione. Nella figura si può notare che tutte le entità del primo set sono coinvolte e quindi si può dire che la partecipazione è *totale*; in caso contrario si dice che è *parziale*.

COMBINAZIONE DEI VINCOLI

Sostanzialmente si possono combinare i vincoli visti nei paragrafi precedenti realizzando delle configurazioni ben precise che caratterizzano una relazione. Per definire la partecipazione di un set di entità alla relazione si disegna una linea: sottile se non vi è specificato alcun vincolo (relazione 0 o più), spessa se

Figure 2.10 Manages and Works_In

Nettuno Rispondi

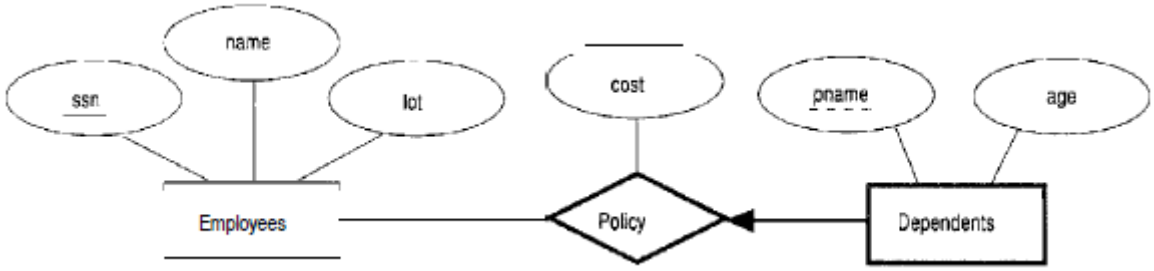
Ciascun dipartimento può essere amministrato da un solo impiegato.
Ciascun impiegato lavora in almeno un dipartimento.
Ciascun dipartimento ha almeno un impiegato.

si vuole una partecipazione totale (ciascuna entità, "each"). La punta della freccia, invece, è rivolta verso il set di entità che subisce il vincolo di chiave e si disegna vicino alla relazione (tra l'entità e la relazione). Qui i significati delle combinazione di questi vincoli:

- "*Thin line*", nessun vincolo (many-to-many);
- "*Thin line with arrow*", massimo 1 (*zero or one*);
- "*Thick line*", almeno 1 (*many-to* or *to-many*);
- "*Thick line with arrow*", esattamente 1 (*one-to or to-one*).

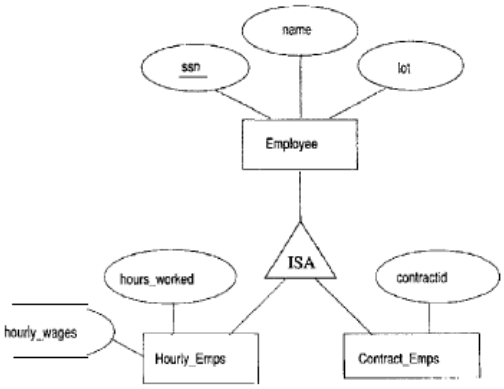
ENTITÀ DEBOLI

Un'entità è *debole* poiché può essere identificata univocamente solo considerando un suo attributo in congiunzione con una chiave primaria di un'altra entità, che funge da proprietario. Per questo motivo un set di entità deboli (che necessitano di una chiave primaria esterna per essere identificate) dispone, tra i suoi attributi, una chiave detta *parziale*. Per rappresentare una chiave parziale si usa sottolineare in maniera tratteggiata il nome dell'attributo desiderato.



Nella figura sopra, "*pname*" è una chiave parziale e ciò indica che è possibile che ci siano due famigliari aventi lo stesso valore per questo attributo. Per identificare univocamente un famigliare infatti, occorre congiungere il valore dell'attributo "*pname*" con quello di "*ssn*".

Per risaltare la debolezza dell'entità, si crea un doppio bordo intorno al rettangolo che identifica l'identità e intorno al rombo che identifica la relazione per la quale l'entità è considerata debole e può essere identificata univocamente.

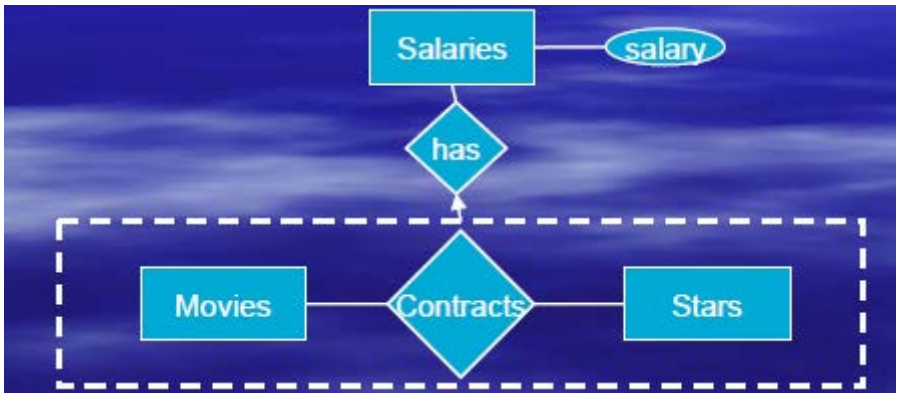


GERARCHIA DELLE CLASSI

Il concetto di gerarchia delle classi è comune per chi viene da un paradigma ad oggetti adottabile in un linguaggio di programmazione. Con il termine "*IS A*" infatti è possibile descrivere un set di entità che eredita le caratteristiche da un set di entità pre-esistente. Logicamente, quando scriviamo una query per chiedere il contenuto di un entità che funge da superclasse occorre domandare i record delle varie sotto-classi. Tipicamente, la superclasse è definita per prima, mentre le sotto-classi vengono definite successivamente e i loro attributi e relazioni sono in aggiunta a quelle ereditate. È possibile specificare due tipologie di **vincoli di gerarchia**. Il primo tipo di vincolo è detto *overlap*. Questa restrizione consiste nel consentire che la stessa entità può essere presente in due sottoclassi differenti. Il secondo tipo di vincolo è detto *covering*. Consiste nell'ereditare tutte le entità della superclasse. L'ereditarietà multipla generalmente non viene ammessa per le classi dei linguaggi orientati agli oggetti.

LE AGGREGAZIONI

Fino ad ora, i set di relazioni sono stati tra due o più entità. Esistono però, anche relazioni tra entità e altre relazioni. Le aggregazioni dunque, ci permettono di indicare questa tipologia di relazioni in cui partecipano altre relazioni. Per rappresentare un'aggregazione si usa congiungere il rombo (che rappresenta l'aggregazione) con una relazione che viene interamente "rettangolarizzata" in modo tratteggiato.



Intuitivamente, usiamo un'aggregazione quando necessitiamo di creare una relazione tra relazioni.

ENTITÀ O ATTRIBUTI?

Quando riconosciamo gli attributi di un set di entità, non è spesso chiaro se considerare una determinata proprietà come un ulteriore attributo o come un set di entità. L'opzione di aggiungere questa proprietà come nuovo attributo è appropriata se necessitiamo di memorizzare un solo valore per entità. Un'alternativa sarebbe creare un'entità differente e collegare i due set di entità con una relazione. Ricorriamo a questa opzione se:

- Dobbiamo memorizzare più di un valore per questa proprietà;
- Vogliamo fornire di un modello ER questa proprietà.

ENTITÀ O RELAZIONE?

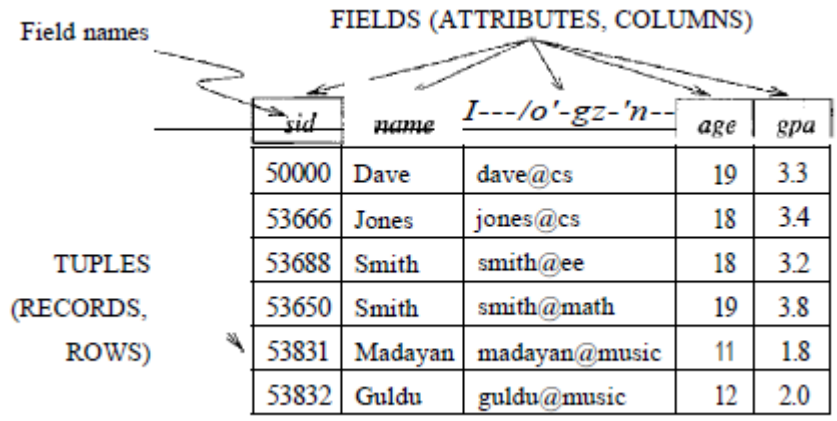
A volte vogliamo aggiungere un attributo alla relazione e ci viene naturale aggiungerlo. Ma bisogna fare molta attenzione. Se ragioniamo con meno precisione, possiamo andare in contro alla memorizzazione di valori ridondanti. Cioè, è consigliabile aggiungere un attributo alla relazione, solo se è realmente associata alla relazione. Altrimenti è consigliabile creare un nuovo set di entità. In questo caso, di solito, si sfrutta la gerarchia delle classi e la nuova entità può essere considerata una "IS A" della precedente.

IL RELATIONAL MODEL

Il "relational model" è un modello molto semplice ed elegante. Esso vede un database come una collezione di una o più relazioni, dove ogni relazione è una tabella composta da righe e colonne. Il miglior vantaggio di un modello relazionale consiste nella sua semplicità di rappresentazione dei dati accompagnati dalla complessità delle query con cui essi possono essere espressi.
Una relazione consiste in un "relation schema" e in delle istanze della relazione. Tradotto, l'istanza di una relazione è la tabella vera e propria con i propri record, mentre il "relation schema" sono le intestazioni della tabella. Lo schema specifica il nome della relazione, il nome di ciascun campo e il dominio di essi.

Students(sid: string, name: string, login: string, age: integer, gpa: real)

Un'istanza di una relazione è un set di tuple, chiamate anche record. Ciascuna istanza può essere vista come una riga della tabella, dove tutte le righe hanno lo stesso numero di campi.



sid	name	login	age	gpa
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

L'ordine con la quale le righe appaiono non è importante. Lo schema di una relazione specifica il dominio di ciascun campo (o colonna) per le istanze della relazione. Il dominio dichiarato nello schema specifica un'importante condizione ce permette a ciascuna istanza di memorizzare valori "sicuri". Per sicuri, si intende che il DBMS deve accertarsi di accettare un dato conforme alla richiesta, in base al campo che si sta compilando. Per farlo, bisogna dichiarare, al momento della creazione della tabella, delle restrizioni, ovvero il tipo di valore che possono apparire nel campo.

CREARE E MODELLARE RELAZIONI USANDO SQL

Il linguaggio SQL standard usa la parola "table" per denotare una relazione. Un linguaggio SQL consente la creazione, l'eliminazione e la modifica di tabelle ed è chiamato DDL (Data Definition Language). SQL però, è anche un linguaggio di manipolazione dei dati, quindi rientra anche tra i DML (Data Manipulation Language).
La creazione di una tabella può essere effettuato attraverso il seguente codice SQL:

```
CREATE TABLE Students ( sid CHAR(20),
                        name CHAR(20),
                        login CHAR(20),
                        age INTEGER,
                        gpa REAL)
```

È possibile aggiungere record, usando il commando "INSERT".

INSERT
INTO Students (sid, name, login, age, gpa)
VALUES (53688, 'Smith', 'smith@ee', 18, 3.2)

È opportune scrivere I valori nell'appropriato ordine con cui i campi sono disposti. È possibile eleminare una tupla usando il comando DELETE, mentre è possibile modificarlo con il comando UPDATE.

LE CHIAVI

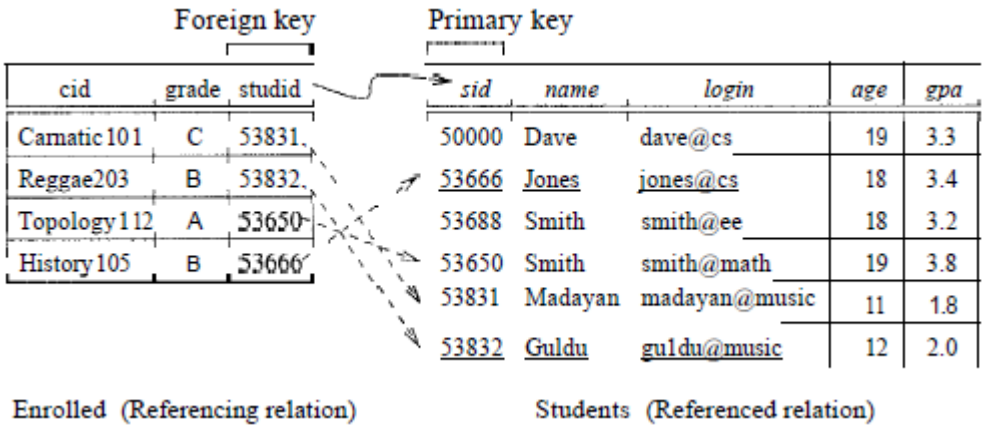
UN DBMS per essere considerato "legale" e quindi gestire correttamente le informazioni, deve prevenire le entrate di informazioni sbagliate. Bisogna dunque specificare tutte le restrizione specificate nel "logical schema" per poter ottenere una corretta istanza delle relazioni.
Una restrizione di chiave per esempio, può essere applicata ad un insieme di campi della relazione per identificare la tupla. Ovviamente, bisogna saper riconoscere quali campi possono fungere da chiave. UN set di campi identificano obbligatoriamente la tupla, e perciò è utile osservare quali campi sono "chiavi candidate". I valori che saranno poi presenti nel campo chiave permetteranno di identificare il record univocamente, e quindi non sarà possibile inserire due valori uguali nel campo chiave. Una chiave primaria può essere anche un set di campi, perché solo insieme possono identificare univocamente una tupla. Quando invece consideriamo un set di campi, in cui uno di essi può essere già considerata una chiave primaria, allora si dice che stiamo considerando una "super chiave". Per ogni relazione, se consideriamo tutti i campi come chiave, stiamo parlando sempre di una super chiave. Ad una relazione, però, può essere assegnata solo una chiave primaria. Questo però non vieta l'esistenza di altri campi, dove i valori non possono essere ripetuti su record differenti. Analizziamo il seguente codice SQL.

```
CREATE TABLE Students ( sid CHAR(20) ,
                        name CHAR (30) ,
                        login CHAR(20) ,
                        age INTEGER,
                        gpa REAL,
                        UNIQUE (name, age),
                        CONSTRAINT StudentsKey PRIMARY KEY (sid) )
```

In questa relazione abbiamo definite una chiave primaria sul campo "sid". Ma abbiamo definito anche un'altra "constraint", ovvero abbiamo definito insieme i campi "age" e "name" come UNIQUE, ovvero che, in coppia, non potranno avere valori ripetuti, anche se essi non sono ovviamente campi chiave. Se una di queste restrizione è violata, verrà ritornato un errore.

FOREIGN KEY

Qualche volta le informazioni da memorizzare in una relazione sono salvate in relazioni già esistenti. In questo caso, si parla di "foreign key", ovvero di chiave esterna, in quanto il valore accettato per un campo che possiede questo tipo di restrizione deve essere presente nel campo a cui la restrizione si riferisce. Una "foreign key" deve essere una chiave primaria per la relazione cui si riferisce.



Se proviamo ad aggiungere una tupla con un valore nel campo con chiave esterna che non esiste nella tabella di riferimento, allora il database system rifiuterà l'inserimento. Analogamente, se vogliamo eliminare una tupla dalla relazione d'origine, dobbiamo dichiarare che azioni far fare al nostro DBMS per le relazioni ad essa collegata.

CREATE TABLE Enrolled (studid CHAR(20),
cid CHAR(20),
grade CHAR(10),
PRIMARY KEY (studid, cid),
FOREIGN KEY (studid) REFERENCES Students)

RESTRIZIONI SU CHIAVE ESTERNA

Il dominio di ciascun campo, la chiave primaria e la chiave esterna sono considerate delle restrizioni fondamentali per un corretto funzionamento del modello relazionale. E per essere tale, un DBMS deve possedere tutte le informazioni necessarie per gestire le situazioni più delicate. Tra queste vi è l'eliminazione di un record nella tabella di riferimento. Cosa succede in un caso come questo?
Le opzioni possono essere:

- Eliminare tutte le righe che si riferiscono al record eliminato.
- Non permettere l'eliminazione delle righe se ci sono record (in altre relazioni) che si riferiscono ad esse.
- Impostare un valore di default da inserire nei campi "chiave esterna" per ciascuna riga che si riferisce al record eliminato.
- Impostare il campo che possiede la restrizione di "chiave esterna" con valore "null".

Quando in SQL definiamo i campi chiave, nella creazione di una tabella, possiamo impostare cosa deve accedere in caso di aggiornamento/eliminazione di un record a cui la relazione che stiamo creando può riferirsi.

CREATE TABLE Enrolled (studid CHAR(20),
cid CHAR(20),
grade CHAR(10),
PRIMARY KEY (studid, dd),
FOREIGN KEY (studid) REFERENCES Students
ON DELETE CASCADE
ON UPDATE NO ACTION)

L'opzione di default è "*NO ACTION*", ed essa significa che un'azione che può andar incontro a questa restrizione (DELETE o UPDATE) sarà rifiutata. La parola chiave *CASCADE* invece dice che al momento della rimozione di un record nella relazione di riferimento, verranno eliminati "a cascata" anche tutti i record della tabella che possiede i valori nel campo chiave esterna a cui si riferisce. Possiamo impostare un valore di default usando "*ON DELETE SET DEFAULT*". Il valore chiave è specificato al momento della creazione del campo nella tabella nel seguente modo: *sid CHAR(20) DEFAULT '53666'*. Per ultimo, possiamo permettere di usare il valore "null" come default specificando *ON DELETE SET NULL*.

QUERYING RELATIONAL DATA

Possiamo interrogare il nostro database grazie ad un "query language" specializzato per la scrittura di query.

*SELECT **
FROM Students S
WHERE S.age < 18

Con il simbolo * si intende considerare tutti I campi delle tuple selezionate nel risultato. È possibile anche combinare le informazioni presenti in più tabelle.

SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid = E.studid AND E.grade = 'A'

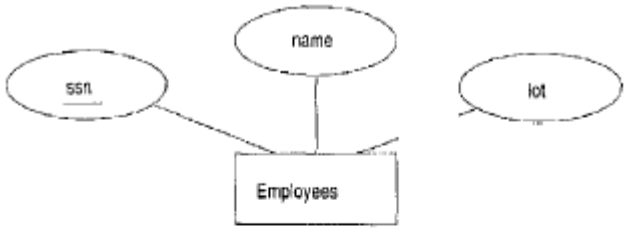


Figure 3.8 The Employees Entity Set

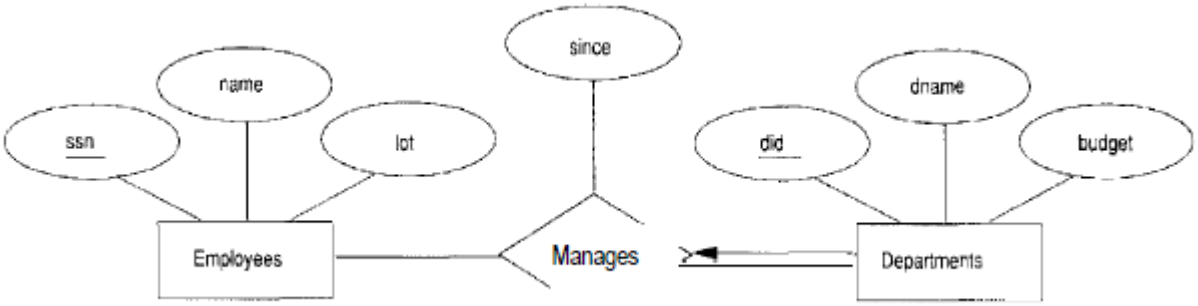
ssn	*name*	*lot*
123-22-3666	Attishoo	48
231-31-5368	Smiley	22
131-24-3650	Smethurst	35

DA ENTITÀ ER A RELAZIONI

Il modello ER, precedentemente trattato, permette di rappresentare un iniziale schema, ad alto-livello, per database. Per generare una database nel modello relazionale dobbiamo usare un linguaggio SQL. Con esso la traduzione risulterà approssimata, perché non è possibile catturare tutte le restrizioni implicite presenti in un "ER model" usando SQL.
Un set di entità è tradotto in una relazione. Ciascuno dei sui attributi in un campo della tabella. Al momento di traduzione, sappiamo già i domini di ciascun attributo e l'esistenza di chiavi primarie per la medesima relazione.

TRADURRE LE RELATIONSHIP SET NEL MODELLO RELAZIONALE

Generalmente, possiamo considerare una relationship del modello ER come una nuova relazione (tabella) tra altre due, specificando gli opportuni campi, le opportune chiavi esterne e la giusta chiave primaria.



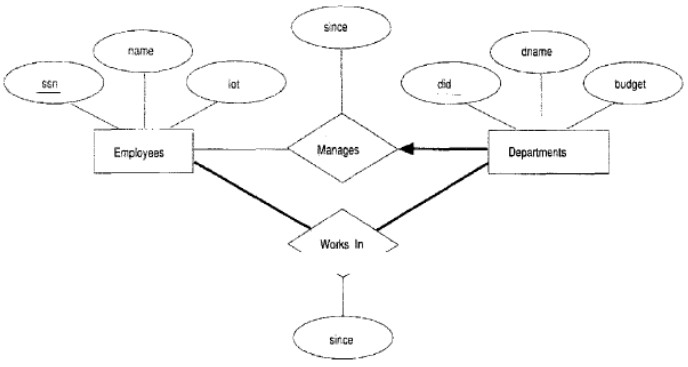
CREATE TABLE Manages (ssn CHAR (11),
did INTEGER,
since DATE,
PRIMARY KEY (did),
FOREIGN KEY (ssn) REFERENCES Employees,
FOREIGN KEY (did) REFERENCES Departments

Questo approccio consente di combinare le informazioni delle due tabelle in una nuova relazione separata dalle altre. A seconda del set di chiavi primarie impostate variamo la nostra "key constraint". Per realizzar una relazione many-to-many, bisogna settare come chiave primaria entrambe le chiavi esterne della relazione. Invece, come abbiamo visto nell'esempio, per settare una relazione del tipo "zero or one", possiamo avere due modalità. La prima prevede la creazione di una nuova tabella impostando come chiave primaria solo la chiave esterna collegata all'entità che riceve il vincolo di chiave. Un secondo approccio per tradurre una relationship set con queste restrizioni nel modello relazionale è quello di estendere la relazione soggetta ad una restrizione. Nell'immagine per esempio, si nota che è presente una restrizione di chiave del tipo "ogni dipartimento deve avere al massimo un dipendente che lo gestisce". Perciò la tabella Departments viene sostituita con una nuova DepLMgr che ha come chiave primaria il campo "did" e il campo ssn sarà una "foreign key" che potrà assumere anche valore NULL (visto che non c'è nessuna restrizione di partecipazione).

CREATE TABLE DepLMgr (did INTEGER)
dname CHAR(20),
budget REAL,
ssn CHAR (11),
since DATE,
PRIMARY KEY (did),
FOREIGN KEY (ssn) REFERENCES Employees)

LA TRADUZIONE DELLA PARTECIPAZIONE TOTALE

Se invece ogni dipartimento richiede un manager (freccia con linea spessa), allora non deve essere permesso di inserire valore nullo. Quindi la SQL sarà la seguente:



CREATE TABLE DepLMgr (did INTEGER,
dname CHAR(20),
budget REAL,
ssn CHAR (11) NOT NULL,
since DATE,
PRIMARY KEY (did),

FOREIGN KEY (ssn) REFERENCES Employees
ON DELETE NO ACTION)

La relazione con partecipazione totale (linea spessa) si realizza garantendo che ogni istanza dell'entità sia in relazione almeno una volta estendendo la tabella con campi (NOT NULL) chiave esterna e aggiungendo una nuova tabella per garantire più relazioni impostando come chiavi primarie i giusti campi, come visto in precedenza, a seconda della constraint che si vuole realizzare.
Per assicurare la relazione one-to-one, infine, dobbiamo garantire che ogni valore di una primary key appaia all'interno di una foreign key della tabella che rappresenta la relationship set. SQL però non ci permette di garantire questa restrizione. Sarebbe opportuno perciò cambiare approccio. La miglior traduzione in questo caso sarebbe inglobare tutte le entità di una relazione in una singola tabella. Quindi basterebbe espandere la relazione che gode di questa restrizione con gli attributi della relationship.

DROP E ALTER
SQL fornisce il comando DROP per rimuovere un'intera tabella, con l'ovvio obbligo di seguire le potenziali restrizioni a cui è predisposta.

DROP TABLE name_table

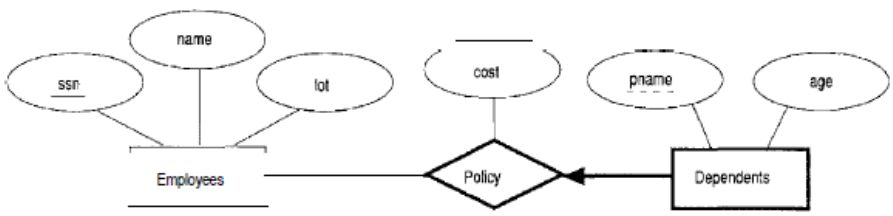
Inoltre se vogliamo estendere, in un secondo momento (successivo alla creazione della tabella), SQL ci fornisce il comando ALTER TABLE per aggiungere un nuovo campo.

ALTER TABLE name_table
ADD COLUMN name_field CHAR(10)

ALGEBRA RELAZIONALE
L'algebra relazionale è una dei due linguaggi di query formali associati con il modello relazione. Le query scritte in algebra sono composte usando una serie di operatori. Ciascun operatore accetta una o più (a seconda se è un operatore unario o binario) istanze di relazioni come argomenti e restituisce una nuova istanza relazionale come risultato. Conoscere approfonditamente l'algebra relazionale permette di descrivere fase per fase le procedure di calcolo che il calcolatore esegue quando noi desideriamo una risposta una volta inviato una "query string". Per valutare correttamente una query dobbiamo conoscere le varie operazioni che si possono fare con una relazione.

WEAK ENTITY IN SQL
Un'entità debole è caratterizzata dalla presenza di una chiave parziale. Perciò un'entità di questo genere è facilmente individuabile poiché tra i campi che formano la chiave primaria possiede una chiave esterna.

CREATE TABLE Dep_Policy (pname CHAR(20) ,
age INTEGER,
cost REAL,
ssn CHAR (11),
PRIMARY KEY (pname, ssn),
FOREIGN KEY (ssn) REFERENCES Employees
ON DELETE CASCADE)



TRADURRE LE GERARCHIE
Una gerarchia ("ISA") di entità è facilmente traducibile in SQL poiché tutte le entità coinvolte possiedono come chiave primaria la chiave primaria della super classe.

SELECTION E PROJECTION
Tra questi operatori abbiamo l'operazione SELECT (σ), che permette di selezionare delle righe da una relazione e PROJECT (π), che permette di selezionare delle colonne.

$\sigma_{rating} > 8(S2)$

Questa stringa per esempio restituisce come risultato una nuova tabella composta dai campi della tabella S2 ma aventi solo i record che hanno valore superiore a 8 nel campo "rating". L'operatore di selezione quindi, specifica le tuple attraverso una selezione condizionata. Una condizione può essere una combinazione di più espressioni condizionali legati tra loro dai connettivi logici, and e or.
L'operatore projection permette invece di estrarre colonne da una relazione.

$\pi_{sname,rafing}(S2)$

Dal risultato di una espressione algebrica si genera sempre una nuova relazione, che possiamo sostituire con l'espressione adatta a individuarla. Ecco una combinazione:

$\pi_{sname,rafing}\left(\sigma_{rating} > 8(S2)\right)$

OPERAZIONI DI SET
Altre operazioni presenti nell'algebra relazionale sono l'unione, l'intersezione, la differenza e il cross-product. *Nota: per gli operatori dell'algebra relazionale non esistono output duplicati*.

- **Union** ($\boldsymbol{R} \cup \boldsymbol{S}$), restituisce una nuova istanza di relazione che contiene tutte le tuple che risiedono in entrambe le relazioni operandi (le tuple di A e quelle di B insieme). Per eseguire questa operazione correttamente però, le istanze devono essere "union-compatible", ovvero devono avere le seguenti caratteristiche: devono avere lo stesso numero di campi e i domini a loro associati devono essere gli stessi.
- **Intersezione** ($\boldsymbol{R} \cap \boldsymbol{S}$), restituisce una nuova istanza di relazione contenente solo le tuple che occorrono in entrambe le relazioni. Anche per questa operazione vale il discorso della compatibilità trattata nel punto precedente, le relazioni operandi devono perciò essere "union-compatible".
- **Set-difference** ($\boldsymbol{R} - \boldsymbol{S}$), restituisce una nuova istanza di relazione contenente tutte le tuple che occorrono in R, ma non in S. Le relazioni coinvolte nell'operazione devono essere "union-compatible".
- **Cross-product** ($\boldsymbol{RxS}$), restituisce una nuova relazione che ha come schema tutti i campi di R (nello stesso ordine con cui compaiono in R) seguite da tutti i campi di S (sempre nel medesimo ordine). Il risultato di *RxS* contiene tutte le tuple di R in coppia con quelle di S. Questa operazione è detta anche prodotto cartesiano tra le due relazioni. Esiste però il caso dove è possibile andare incontro ad un conflitto di nomi. Quando abbiamo più campi con lo stesso nome si ha una situazione di "naming conflict". Per ovviare a questo problema esiste l'operazione di renaming.

RENAMING ($\boldsymbol{\rho}$)
L'operazione di renaming consente di generare una nuova istanza relazionale avente un nome e che possiede campi rinominati.

$\rho_{C(1\rightarrow sid1,5\rightarrow sid2)}(S1xR1)$

Questa espressione algebrica genera una nuova tabella di nome C, dove i campi 1 e 5 della relazione prodotto S1xR1 sono rinominati rispettivamente “sid1” e “sid2”.

OPERAZIONE JOIN (⋈)

L’operazione di join è una delle più utilizzate operazioni in ambito di algebra relazionale ed è la il modo più comune per combinare relazioni. Una “join operation” può essere definita come un cross-product seguito da una selezione. Il risultato di una join è tipicamente più filtrato di quello di un cross-product. La versione più generale dell’operatore join accetta una “join conditions” ed un paio di istanze di relazioni come argomenti e restituisce una nuova istanza di relazione.

$$R \bowtie_C S = \sigma_C(RxS)$$

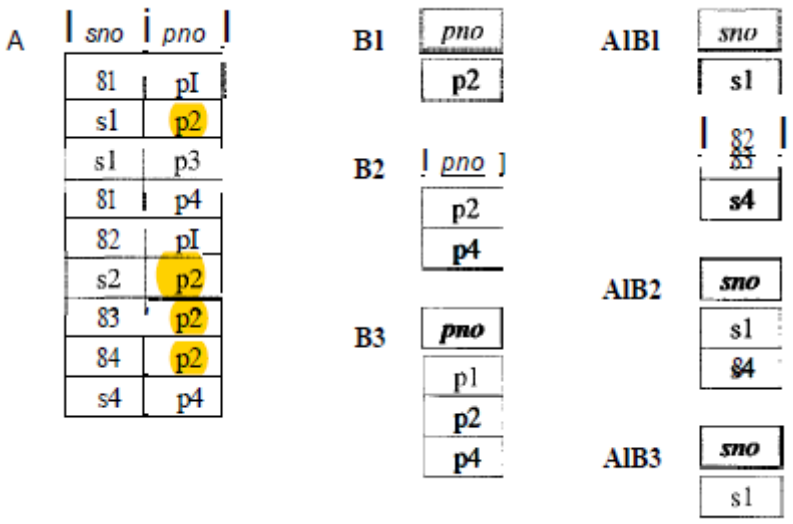
Questo simbolo ⋈ definisce un cross-product seguito da una selezione.
Uno speciale caso dell’operazione join è la “equijoin”. Questo caso prevede che la condizione utilizzata per la join sia composta da uguaglianze. Per esempio,

$$R \bowtie_{R.sid=S.sid} S$$

Lo schema risultante da questa equijoin contiene i campi della relazione R seguita dai campi della relazione S. I record mostrati saranno quelli descritti nella condizione, ovvero quelli con i valori dei due campi specificati uguali. In più una join eredita solo uno dei due campi aventi lo stesso nome.
Un caso speciale dell’operazione join è $R \bowtie S$; la condizione implicita è quella di selezionare solo i record aventi ugual valore nel campi che hanno stesso nome.

LA DIVISIONE

Questa operazione non ha la stessa importanza come gli altri operatori, infatti tutti i database system non provano neanche ad implementarla in quanto si può esprimere attraverso gli altri operatori. Con questo operatore possiamo chiedere al database query del tipo “trovare tutti i nomi dei venditori che possiedono TUTTE le barche”. Questa parola evidenzia l’uso di questa operazione, in quanto è necessario escludere tutti i set (venditori in questo esempio) che non rispettano la clausola. In parole povere, la divisione tra due entità A e B restituisce un set di tutti i valori di *x* che sono accoppiati con il campo *y* (in un’altra entità B) per ogni valore in cui *y* esiste.



Analizzando questa immagine si può capire meglio l’utilità di queste query. $A/B1$ restituisce tutti i valori di *x* (*sno*) con cui è associata ogni istanza di *y* (*pno* in B1). Per far ciò bisogna letteralmente escludere delle istanze. Le istanze che non sono “squalificate” saranno il risultato della nostra query. Per “squalificati” si intendono tutti quei valori di *x* che se concatenati con *y* (da B) otteniamo una tupla che non è in A. Da qui:

$$A/B = \pi_x(A) - \pi_x\big((\pi_x(A)\ x\ B) - A\big)$$

INTRODUZIONE ALLE SQL QUERY

Un DBMS, tipicamente, esegue una query in un differente e più efficiente modo. Lo "scheletro" di una query SQL è il seguente:

SELECT [DISTINCT] select-list
FROM from-list
WHERE qualification

Ogni query deve avere una clausola *SELECT*, con la quale specifichiamo le colonne che vogliamo ottenere nel risultato. In più una query deve avere la clausola *FROM*, con la quale specifichiamo un prodotto cartesiano tra tabelle. La clausola *WHERE* invece è opzionale. Con essa esprimiamo una "selection condition" sulle tabelle specificate dal *FROM*. Verranno visualizzati solo i record a cui questa condizione verrà associato il valore true.
Possiamo affermare che ogni clausola corrisponde ad un operatore dell'algebra relazionale. In una semplice query contenente queste tre clausole si fa riferimento agli operatori "selection", "projection" e "cross-produtct".
La keyword *DISTINCT* nominata tra parentesi quadre è opzionale. Se noi non scriviamo *DISTINCT* vogliamo visualizzare nella risposta della query anche le possibili copie di tuple che possono essere restituiti dal calcolo del sistema. Di default i duplicati non sono eliminati (al contrario della relational algebra).

SELECT S.sid, S.sname, S.rating, S.age
FROM Sailors AS S
WHERE S.rating > 7

In questa query usiamo la keyword AS per introdurre una variabile. Nella query sopra, chiamiamo "S" la tabella "Sailors". Inoltre, possiamo scrivere semplicemente *SELECT ** per indicare l'inclusione di tutte le colonne possibili delle tabelle elencate nella clausola *FROM*.
La risposta finale ad una query è se stessa una nuova relazione la quale contiene un set di righe in SQL.
Nella clausola WHERE possiamo definire condizioni che restituiscono valore true o false. Come in ogni linguaggio per definire condizioni sono ammessi i connettivi logici e operatori relazionali.

SELECT S.sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = 13.bid AND B.color = 'red'

In questa query è stata fatta la join di tre tabelle.
Quando abbiamo a che fare con stringhe spesso vogliamo esaminare delle condizioni particolari. SQL ci fornisce un importante operatore di supporto per un corretto "matching" delle stringhe. L'operatore LIKE ci fornisce la possibilità di definire un raggio ampio di scelta tra le stringhe da comparare. Per esempio, con '_AB%' vogliamo associare tutte le stringhe che hanno come seconda e terza lettera rispettivamente la A e la B e poi qualsiasi altro contenuto. Quindi con il simbolo '_' (blank) è possibili dichiarare che deve esserci un carattere qualsiasi in quella precisa posizione, mentre con il simbolo '%' vogliamo specificare la presenza di diversi caratteri.

SELECT S.age
FROM Sailors S
WHERE S.sname LIKE 'B.%B'

Una possibile risposta a questa query è l'età di Bob.

UNIONE, INTERSEZIONE E EXCEPT IN SQL

SQL supporta queste operazioni sotto il nome di *UNION*, *INTERSECT* e *EXCEPT*. In generale, possiamo esprimere l'operatore *UNION* attraverso il connettivo *OR* inserito nel *WHILE*:

```
SELECT S.sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid
        AND (B.color = 'red' OR B.color = 'green')
```

Con questa query vogliamo ottenere I signori che possiedono almeno una barca di colore rosso insieme ai signori che possiedono almeno una barca di colore verde. È possibile riscrivere questa query nel seguente modo:

```
SELECT S.sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sicl = R.sid AND R.bid = B.bid AND B.color = 'red'
UNION
SELECT S2.sname
FROM Sailors S2, Boats B2, Reserves H2
WHERE S2.sid = H2.sid AND R2.bid = B2.bicl AND B2.color = 'green'
```

Mentre, per quanto riguarda l'operatore di intersezione, possiamo generalmente usare il connettivo *AND* nella clausola *WHERE* per riprodurlo. Vediamo la seguente query:

```
SELECT S.sname
FROM Sailors S, Reserves RI, Boats BI, Reserves R2, Boats B2
WHERE S.sid = Rl.sid AND R1.bid = Bl.bid
        AND S.sid = R2.sid AND R2.bid = B2.bid
        AND B1.color='red' AND B2.color = 'green'
```

Qui vogliamo estrarre i nomi dei signori che possiedono almeno una barca rosso e almeno una barca verde. Possiamo riscriverla utilizzando l'operatore *INTERSECT*.

```
SELECT S.snarne
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
INTERSECT
SELECT S2.sname
FROM Sailors S2, Boats B2, Reserves R2
WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green'
```

Infine possiamo utilizzare l'operatore *EXCEPT* quando vogliamo eseguire una differenza.

```
SELECT S.sid
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
EXCEPT
SELECT S2.sid
FROM Sailors S2, Reserves R2, Boats B2
WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green'
```

Qui vogliamo visualizzare gli id di tutti i signori che possiedono solo barche rosse.
In tutti questi operatori si può notare che è presente una condizione che permette di usarli: le due tabelle

SQL che diventano operandi devono essere "union-compatible", ovvero che devono avere lo stesso numero di colonne e ciascuna colonna deve essere dello stesso tipo, cioè deve avere lo stesso dominio.

NESTED QUERY

```
SELECT S.sname
FROM Sailors S
WHERE S.sid IN ( SELECT R.sid
                 FROM Reserves R
                 WHERE R.bid = 103 )
```

In questa query è possibile osservare la presenza di una query annidata. Proprio quest'ultima viene calcolata per prima e grazie all'operatore *IN* viene testato se un valore è presente all'interno di questo set di elementi. È possibile rimpiazzare l'operatore *IN* con *NOT IN*. Non esiste un limite di annidamento tra query, ma bisogna fare attenzione che il campo selezionato (Projections) sia lo stesso di quello verificato nella query soprastante (con la clausola *WHERE*).
Una "nested query" è completamente indipendente dalle altre. In generale, la query soprastante può dipendere dal set di righe restituite dalle altre query.
Oltre a *IN* esiste l'operatore *EXITS* che permette di verificare se un set è vuoto.

```
SELECT S.sname
FROM Sailors S
WHERE EXISTS ( SELECT *
               FROM Reserves R
               WHERE R.bid = 103
                     AND R.sid = S.sid )
```

Un altro operatore tra "nested query" è *UNIQUE*. Quando applichiamo questo operatore ad una sotto query, esso restituisce true se non esiste una riga che appare due volte (se non sono presenti duplicati) nella risposta della sotto query.

COMPARARE NESTED QUERY

Quando usiamo query annidate, si fa maggiormente uso di operatori di confronto. SQL perciò mette a disposizione due operatori essenziali per effettuare un confronto, ovvero *ANY* e *ALL*.

```
SELECT S.sid
FROM Sailors S
WHERE S.rating > ANY ( SELECT S2.rating
                       FROM Sailors S2
                       WHERE S2.sname = 'Horatio' )
```

In questo esempio, *S.rating* è confrontato con ciascun valore restituito dalla sotto query. Intuitivamente, la sotto query deve restituire almeno una riga che rende il confronto true. Quindi, l'operatore *ANY* restituisce true se almeno una riga restituisce true. L'operatore *ALL*, invece, restituisce true solo se il confronto con tutte le righe uscenti dalla sotto query restituiscono true. Si guardi questo esempio.

```
SELECT S.sid
FROM Sailors S
WHERE S.rating >= ALL ( SELECT S2.rating
                        FROM Sailors S2 )
```

L'OPERAZIONE DI DIVISIONE IN SQL

```
SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS (( SELECT B.bid
                    FROM Boats B )
                    EXCEPT
                    (SELECT R. bid
                    FROM Reserves R
                    WHERE R.sid = S.sid ))
```

OPERATORI DI AGGREGAZIONE

SQL permette anche l'uso di espressioni aritmetiche. Esistono infatti, in aggiunta ai normali operatori di querying, funzioni che permettono di calcolare operazioni tipiche di analisi di dati.

1. *COUNT ([DISTINCT] A)*: restituisce il numero (non duplicati) dei valori presenti nella colonna A.
2. *SUM ([DISTINCT] A)*: restituisce la somma di tutti i valori (non duplicati) presenti nella colonna A.
3. *AVG ([DISTINCT] A)*: restituisce la media di tutti I valori (non duplicati) presenti nella colonna A.
4. *MAX (A)*: restituisce il massimo valore presente nella colonna A.
5. *MIN (A)*: restituisce il minimo valore presente nella colonna A.

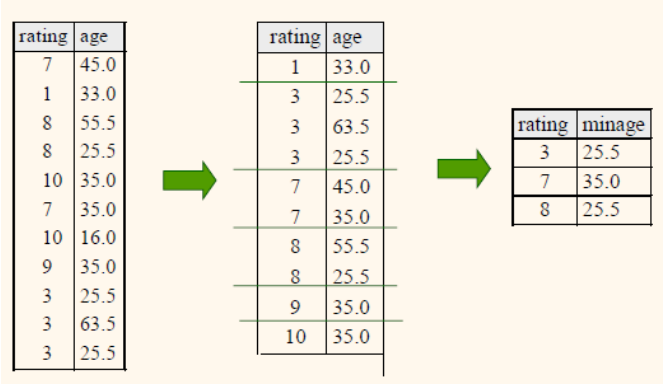
Quando si fa uso di operatori di aggregazione nella clausola *SELECT*, si deve usare al massimo un operatore senza uso della clausola *GROUP BY*.

```
SELECT S.sname
FROM Sailors S
WHERE S.age > ( SELECT MAX ( S2.age )
                FROM Sailors S2
                WHERE S2.rating = 10 )
```

LE CLAUSOLE GROUP BY E HAVING

Spesso noi vogliamo applicare un operatore di aggregazione per ciascun numero di gruppi di righe presenti in una relazione. Per scrivere query del genere, necessitiamo di una maggiore estensione delle query SQL. È compito della clausola *GROUP BY* estendere queste funzionalità. L'estensione include anche un'opzionale clausola chiamata *HAVING* che può essere usata per specificare condizioni di formazione di ciascun gruppo. Le espressioni che appaiono nella clausola *HAVING* devono avere un singolo valore per gruppo.

```
SELECT S.rating, MIN (S.age) AS minage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT (*) > 1
```



rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5

rating	age
1	33.0
3	25.5
3	63.5
3	25.5
7	45.0
7	35.0
8	55.5
8	25.5
9	35.0
10	35.0

rating	minage
3	25.5
7	35.0
8	25.5

Notare: la clausola HAVING rappresenta un vincolo sui dati risultanti dall'operazione di raggruppamento. Il suo funzionamento è molto simile a quello della clausola WHERE ma anziché operare sui campi del database opera sui raggruppamenti; i campi richiesti possono essere sia quelli delle funzioni di aggregazione sia quelli indicati nelle clausole GROUP BY.

Nella clausola HAVING è possibile introdurre due possibili funzioni, le solite ANY e **EVERY** che considerano il risultato solo in alcuni casi. La keyword EVERY richiede che ogni riga deve soddisfare la condizione per

essere considerata nel raggruppamento. Al contrario, con **ANY** si vuole considerare il record se possiede almeno un true nella condizione.

SELECT F.fname, COUNT(*) AS CourseCount
FROM Faculty F, Class C
WHERE F.fid = C.fid
GROUP BY F.fid, F.fname
HAVING EVERY (C.room = 'R128')

COMPARARE VALORI NULLI

Se noi compariamo due valori nulli usando operatori relazionali del tipo <, >, = su attributi che non hanno alcun valore, allora il risultato restituito sarà sempre "*unknow*". SQL possiede uno speciale operatore di confronto dedicato a situazioni del genere. Con ***IS NULL*** possiamo testare se il valore di una colonna è nullo. Esso restituisce *true* in tal caso.

SELECT LastName, FirstName, Address
FROM Persons
WHERE Address IS NULL

Anche per I connettivi logici il valore "*unknow*" si propaga. L'operatore "*NOT unknow*" è definito come "*unknow*". L'operatore *OR* valuta vera solo l'espressione che ha argomenti entrambi veri, se uno di questi argomenti risulta "*unknow*" l'esito dell'espressione è "*unknow*". Stessa cosa per l'operatore *AND* che valuta falso solo se entrambi sono falsi. Se intercetta un "*unknow*" restituisce "*unknow*".

Anche l'operazione = tra due *NULL* restituisce "*unknow*". E una qualsiasi operazione aritmetica restituisce *NULL* se uno degli argomenti è *NULL*.

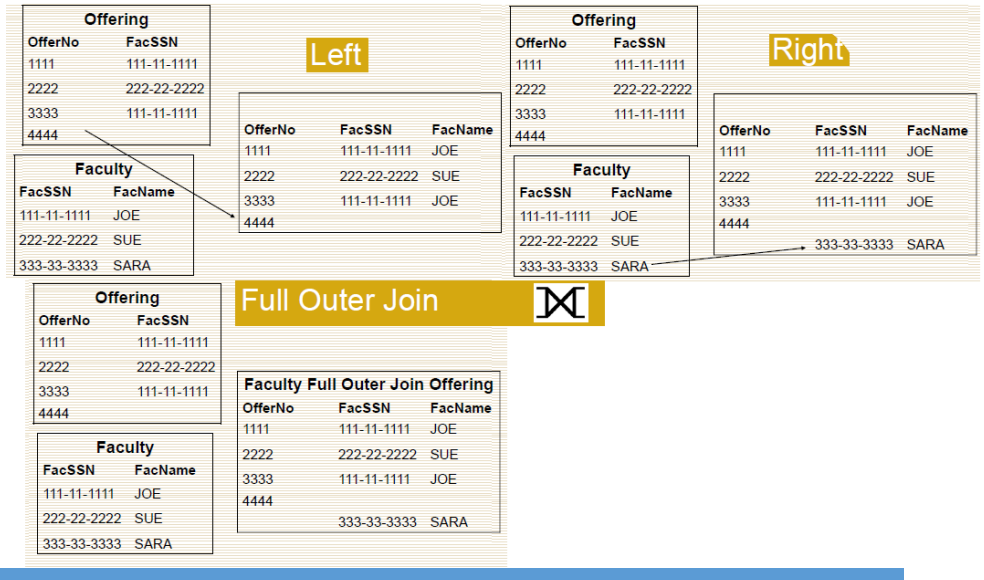
OUTER JOINS

Esiste una operazione alternativa di *JOIN* per trattare i valori *NULL*, chiamata *outer join*. Si sa che le tuple della prima tabella che vengono escluse dalla *joind condition* non appaiono nel risultato. Ma in un *outer join*, i record che non sono stati matchati dalla *join* in quanto possiedono valore *NULL* compaiono lo stesso nella tabella risultante, assegnando ovviamente un valore *NULL*.

Esistono tre tipi di *outer join*. In una *left outer join* appaiono anche i set della prima tabella che hanno valore nullo. Al contrario, in una *right outer join* appaiono anche i set della seconda tabella che hanno valore nullo. In una *full outer join*, compaiono i record che provengono da entrambe le tabelle.

SELECT S.sid, R.bid
FROM Sailors S NATURAL LEFT OUTER JOIN Reserves R

La parola chiave *NATURAL* specifica che la *join condition* è quella di uguaglianza sugli attributi comuni.



IMPEDIRE VALORI NULLI

Possiamo impedire il propagarsi dei valori nulli specificando *NOT NULL* come parte della definizione del campo. Comunque, c'è un implicito "*NOT NULL constraint*" per ogni campo impostato come *PRIMARY KEY*.

SETTARE CONSTRAINT IN UNA TABELLA

Possiamo impostare una restrizione sui valori che memorizzeremo in una tabella grazie alla parola chiave *CHECK*. Così facendo siamo sicuri di trovarci record che rispettano un certo dominio. Per esempio:

```
CREATE TABLE Sailors ( sid INTEGER,
                       sname CHAR(10),
                       rating INTEGER,
                       age REAL,
                       PRIMARY KEY (sid),
                       CHECK (rating >= 1 AND rating <= 10 ))
```

Quando una riga è inserita nella tabella viene valutata la condizione che è nella clausola *CHECK*. Se è valutata "*false*", il comando è rifiutato.

I TRIGGER

Un trigger è una procedura che è automaticamente invocata dal DBMS in risosta ad uno specifico cambiamento al database. La composizione di un trigger contiene tre parti: l'evento, il cambiamento sul database che scatena l'attivazione del trigger. Una condizione è una query che viene valutata e se restituisce *true* allora viene eseguito il trigger. L'azione vera è propria invece è la procedura da eseguire quando il trigger è attivato.

```
CREATE TRIGGER incLcount AFTER INSERT ON Students
    WHEN (new.age < 18)
    FOR EACH ROW
    BEGIN
        count := count + 1;
    END
```

NORMALIZZAZIONE

Quando una relazione non soddisfa una forma normale, allora presenta ridondanze e si presta a comportamenti poco desiderabili durante le operazioni dì aggiornamento. Questo concetto può essere n un utile strumento dì analisi nell'ambito dell'attività di progettazione di una base di dati. È possibile applicare un procedimento, detto di *normalizzazione*, che consente di trasformare questi schemi non normalizzati in nuovi schemi per i quali il soddisfacimento di una forma normale è garantito. Va detto innanzitutto che le metodologie di progettazione viste nei capitoti precedenti permettono di solito di ottenere schemi che soddisfano una forma normale. In questo contesto, la teoria della normalizzazione costituisce un ulteriore strumento di verifica.

Nome	Età	Professione
Alberto	30	Impiegato
Gianni	24	Studente
Alberto	30	Impiegato
Giulia	50	Insegnante

La ***prima forma normale*** definita per un database esprime un concetto semplice ma fondamentale: ogni riga di ciascuna tabella deve poter essere identificata in modo univoco. Nell'immagine non è possibile, infatti, distinguere il dato inserito nella prima riga da quello inserito nella terza: le due righe sono infatti identiche. Il problema potrebbe essere risolto inserendo un altro campo nella tabella, con valore diverso per ogni riga, la chiave primaria della tabella.

Un ulteriore miglioramento è possibile passando alla ***seconda forma normale***. Per esserlo, tutti i campi non chiave dipendano dall'intera chiave primaria (e non solo da una parte di essa).
Quando ci troviamo nella 1NF, possiamo individuare dipendenze tra diversi attributi generando ridondanza ed essere vulnerabili a delle anomalie. Si consideri per esempio la seguente tabella.

Impiegato	Stipendio	Progetto	Bilancio	Funzione
Rossi	20 000	Marte	2000	tecnico
Verdi	35 000	Giove	15 000	progettista
Verdi	35 000	Venere	15 000	progettista
Neri	55 000	Venere	15 000	direttore
Neri	55 000	Giove	15 000	consulente
Neri	55 000	Marte	2000	consulente
Mori	48 000	Marte	2000	direttore
Mori	48 000	Venere	15 000	progettista
Bianchi	48 000	Venere	15 000	progettista
Bianchi	48 000	Giove	15 000	direttore

- *Anomalia 1:* **RIDONDANZA**. Il valore dello stipendio di ciascun impiegato è ripetuto in tutte le tuple relative a esso;
- *Anomalia 2*: **ANOMALIA DI AGGIORNAMENTO**. Se lo stipendio di un impiegato varia, è necessario andarne a modificare il valore in tutte le tuple corrispondenti affinché la dipendenza continui a valere;
- *Anomalia 3*: **ANOMALIA DI CANCELLAZIONE**. Se un impiegato interrompe la partecipazione a tutti i progetti senza lasciare l'azienda, tutte le corrispondenti tuple vengono eliminate e non è possibile conservare traccia del suo nome;
- *Anomalia 4*: **ANOMALIA DI INSERIMENTO**. Se si hanno informazioni su un nuovo impiegato, non è possibile inserirle finché questi non viene assegnato a un progetto.

Una motivazione intuitiva della presenza di questi inconvenienti può essere la seguente: abbiamo usato un'unica relazione per rappresentare informazioni eterogenee.
Esiste, infine, un ulteriore miglioramento che conduce il nostro database alla ***terza forma normale***. Una base dati è in 3NF se è in 2NF e tutti gli attributi non-chiave dipendono soltanto dalla chiave, ossia non esistono attributi che dipendono da altri attributi non-chiave.

DIPENDENZA FUNZIONALE
Per capire come scomporre un database nelle giuste entità relazionali è necessario far uso dì uno specifico strumento di lavoro: la *dipendenza funzionale*. Si tratta di un particolare vincolo di integrità per il modello relazionale che, come ci suggerisce il nome, descrive legami eli tipo funzionale tra gli attributi di una relazione. Una dipendenza funzionale tra gli attributi X e Y viene generalmente indicata con la notazione $X \rightarrow Y$. In una dipendenza funzionale, esiste una funzione che associa a ogni elemento del dominio dell'attributo X che compare nella relazione un solo elemento del dominio dell'attributo Y.
Se prendiamo una chiave K di una relazione r, si può facilmente verificare che esiste una dipendenza funzionale tra K e ogni altro attributo dello schema di r. Questo perché, per definizione stessa di vincolo di chiave, non possono esistere due tuple con gli stessi valori. Perciò possiamo concludere dicendo che se tutti gli attributi di una relazione dipendono da un attributo X, allora X è una chiave. Attenzione: X, chiave primaria, può essere composto anche da più campi.

PROBLEMI RELATIVI ALLA DECOMPOSIZIONE

Abbiamo visto che per passare alla 2NF occorre decomporre. La decomposizione può essere effettuata producendo tante relazioni quante sono le dipendenze funzionali definite. In generale, purtroppo, le dipendenze possono avere una struttura complessa: può non essere necessario (o possibile) basare la decomposizione su tutte le dipendenze e può essere difficile individuare quelle su cui si deve basare la decomposizione.

La decomposizione però può portare ad una perdita di dati se fatta male. Si pensi alla tabella precedente. Potremmo decomporla in corrispondenza alle dipendenze $Impiegato \rightarrow Sede$ e $Progetto \rightarrow Sede$ ottenendo le seguenti tabelle sulla destra.

Impiegato	Sede
Rossi	Roma
Verdi	Milano
Neri	Milano

Progetto	Sede
Marte	Roma
Giove	Milano
Saturno	Milano
Venere	Milano

Impiegato	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Venere	Milano
Neri	Saturno	Milano
Neri	Venere	Milano
Verdi	Saturno	Milano
Neri	Giove	Milano

Possiamo osservare che possiamo ritrovare la tabella originale per mezzo di una operazione di *join naturale* delle due proiezioni. Purtroppo, il *join naturale* delle due relazioni produce una relazione diversa dall'originale. Abbiamo ottenuto una relazione che contiene tutte le tuple di r, più eventualmente altre, che possiamo chiamare "'spurie". Diciamo che r si decompone senza perdita se il *join* delle due proiezioni è uguale a r stessa.

Per raggiungere un risultato del genere, possiamo dire che r si decompone senza perdita su due relazioni se l'insieme degli attributi comuni alle due relazioni è chiave per almeno una delle relazioni decomposte. Nell'esempio, possiamo vedere che l'intersezione degli insiemi di attributi su cui abbiamo effettuato le due proiezioni è costituita dall'attributo Sede, che non è il primo membro di alcuna dipendenza funzionale.

FORMA NORMALE DI BOYCE E CODD (BCNF)

Una relazione è in forma normale di *Boyce-Codd* (*BCNF, Boyce-Codd Normal Form*) quando è nella forma 1FN e in essa ogni determinante è una chiave candidata, cioè ogni attributo dal quale dipendono altri attributi può svolgere la funzione di chiave. Da questo fatto discende immediatamente che una relazione che soddisfa la BCNF è anche in seconda e in terza forma normale, in quanto la BCNF esclude che un determinante possa essere composto solo da una parte della chiave, come avviene per le violazioni alla 2FN, o che possa essere esterno alla chiave, come avviene per le violazioni alla 3FN.

LA CHIUSURA

Per procedere nella decomposizione dobbiamo definire il concetto di chiusura. Una chiusura è un'implicazione di dipendenze funzionali, cioè l'insieme degli attributi che dipendono funzionalmente da X, esplicitamente o implicitamente.

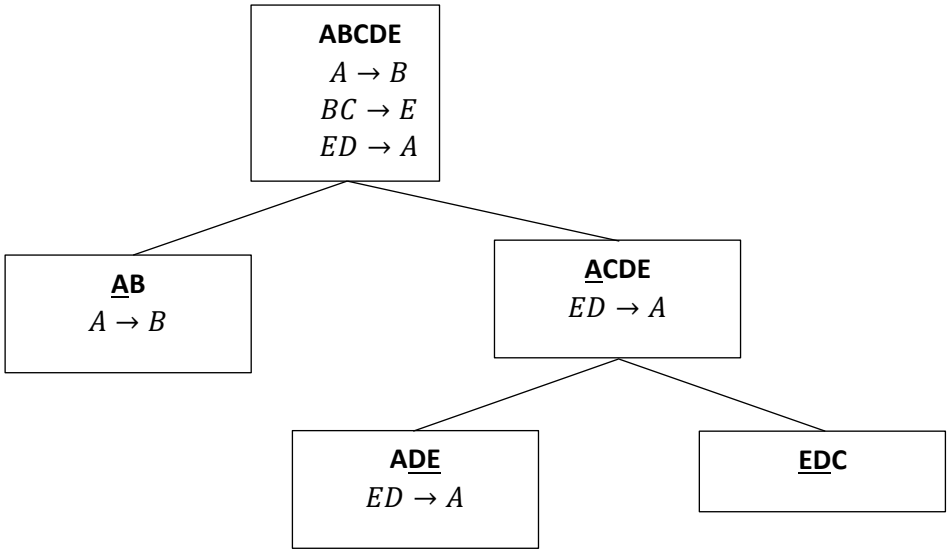
$A^+ \rightarrow AB$

In questo esempio con la chiusura A si vogliono esplicitare tutti gli attributi che dipendono da A. Il concetto di chiusura è utile anche per formalizzare il legame che c'è fra il concetto di dipendenza funzionale e quello di chiave. Una chiusura infatti, può indicarci se l'attributo è tale da poter essere considerato chiave. Per definizione, quando da una chiusura dipende ogni campo della relazione, allora esso può essere considerato chiave.

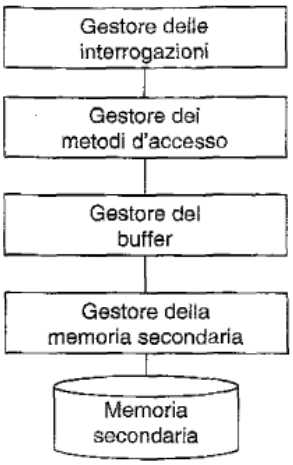
ALGORITMO PER LA DECOMPOSIZIONE

La decomposizione avviene seguendo una procedura ad albero, in cui ciascun nodo è composto dall'elenco degli attributi della tabella e dall'elenco delle dipendenze funzionali presenti tra essi. Considerando una dipendenza funzionale alla volta, si aggiunge la dipendenza come figlio sinistro, mentre come figlio destro vengono elencati tutti gli altri attributi che sono indipendenti, compreso l'attributo che origina la

dipendenza. Dopodiché si individuano le restanti dipendenze funzionali presenti in quest'ultima relazione e si ripete l'algoritmo. Vediamo subito un esempio riassuntivo.



ORGANIZZAZIONE FISICA DEL DATABASE



Le operazioni che vengono specificate in SQL vengono affidate a un modulo detto *gestore delle interrogazioni*, che traduce le interrogazioni in una forma interna, le trasforma al line dì renderle più efficienti e le realizza in termini di operazioni di livello più basso, che fanno riferimento alla struttura fisica dei file e sono eseguite da un modulo sottostante, chiamato *gestore dei metodi d'accesso*. Quest'ultimo modulo, trasforma le richieste in operazioni di lettura e scrittura di dati in memoria secondaria, che vengono però mediate da un modulo, detto gestore del buffer, che ha la responsabilità di mantenere temporaneamente porzioni della base di dati in memoria centrale, per favorirne l'efficienza.
Le basi di dati hanno la necessità di gestire dati in memoria secondaria in quanto la memoria principale non risulta di solito sufficiente per contenere per intero una base di dati. Un altro motivo risiede nelle caratteristiche fondamentali delle basi di dati, ovvero la persistenza: esse hanno un tempo di vita che non è limitato alle singole esecuzioni dei programmi. Ricordiamo che essa non è direttamente utilizzabile dai programmi: i dati, per poter essere utilizzati, debbono prima essere trasferiti in memoria principale. Nella memoria secondaria, i dati sono organizzati in blocchi di dimensione di solito fissa.
Le uniche operazioni possibili su un disco sono la lettura o la scrittura di un intero blocco: questo ha come conseguenza il fatto che l'accesso a un singolo bit ha lo stesso costo dell'accesso a un intero blocco. Inoltre, poiché il tempo necessario per la lettura o la scrittura di un blocco è maggiore del tempo necessario per accedere ai dati in memoria centrale, nelle applicazioni che coinvolgono basi di dati è spesso possibile trascurare i costi di tutte le operazioni che effettuano accessi a memoria. Nel caso di letture o scritture massicce il costo complessivo può essere notevolmente ridotto se i blocchi sono adiacenti, cioè se l'allocazione è contigua. Per riassumere:

$\#tuple\ per\ pagina = \left\lfloor \frac{P_{size}}{T_{size}} \right\rfloor$ Il numero di tuple memorizzabili all'interno di una pagina è dato dal rapporto tra la dimensione di una pagina e quella di una tupla.

$\#pagine\ per\ relazione\ R = \left\lceil \frac{\#tuple}{\#tuple\ per\ pagina} \right\rceil$ Il numero di pagine richieste per memorizzare un'intera relazione R è dato dal rapporto tra il numero di tuple che si intende salvare e il numero di tuple che entrano in una pagina.

IL BUFFER

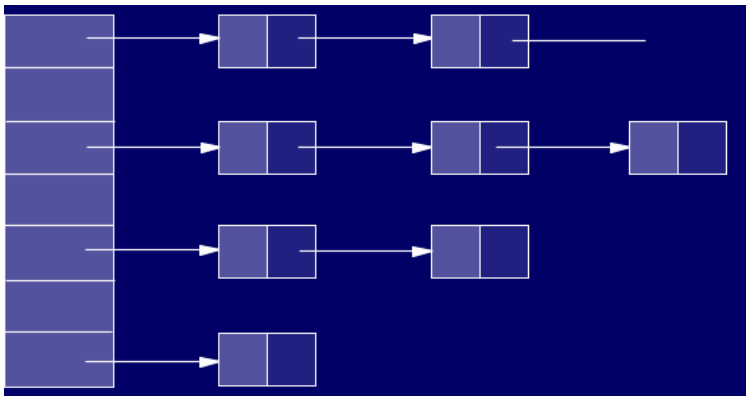
L'interazione fra memoria centrale e memoria secondaria è realizzata nei DBMS attraverso l'utilizzo di un'apposita, grande zona di memoria centrale detta *buffer*. Il buffer è organizzato in pagine. Il gestore del buffer si occupa del caricamento e dello scaricamento (salvataggio) delle pagine dalla memoria centrale alla memoria di massa. Rapidamente, in ciascuna pagina sono presenti sia informazione utile sia i riformazione di controllo; l'informazione utile coincide con Ì dati veri e propri, l'informazione di controllo consente di accedere all'informazione utile. Dunque, il criterio secondo il quale sono disposte le tuple nell'ambito del file è molto importante in termini di efficienza. Le strutture utilizzate possono essere divise in "ad accesso calcolato" (o *hash*) e ad *albero* (o *B⁺ Tree*). In sostanza, le *tabelle hash* collocano le tuple in posizioni determinate sulla base del risultato dell'esecuzione di un algoritmo. Mentre, le strutture ad albero sono utilizzate anche e soprattutto come strutture secondarie (cioè aggiuntive rispetto a quelle primarie).

INDICIZZAZIONE PER TABELLA HASH

Per velocizzare la ricerca di una determinata tupla nel buffer, occorre indicizzare i record per un attributo (magari una chiave). L'indicizzazione può essere implementata tramite funzione hash. La funzione *hash* consente di trasformare un valore nell'indice di un array, e quindi associare ogni record a una posizione specifica all'interno del buffer, evitando così di scorrerlo tutto alla ricerca del valore desiderato. Per fare un esempio, potremmo utilizzare un set di 50 elementi e una funzione hash molto semplice che calcola il resto della divisione per 50. Poiché l'insieme delle chiavi è molto più grande dell'insieme dei possibili valori dell'indice, la funzione hash non può essere iniettava e quindi è sempre possibile che sì verifichino collisioni, cioè valori diversi della chiave che portano allo stesso valore dell'indice.

M	M mod 50
60600	0
86301	1
205751	1
205802	2
200902	2
116202	2
200604	4
66005	5
116455	5
200205	5
201159	9
205610	10
201260	10
102360	10
205460	10
205912	12
205762	12
200464	14
205617	17
205667	17

M	M mod 50
200268	18
205619	19
210522	22
205724	24
205977	27
205478	28
200430	30
210533	33
205887	37
200138	38
102338	38
102690	40
115541	41
206092	42
205693	43
205845	45
200296	46
205796	46
200498	48
206049	49



Ogni possibile chiave presente nella tabella contiene la lista di tutti i valori che hanno la medesima chiave associatagli dalla funzione hash. In questa lista sono presenti i puntatori per effettuare la lettura su disco nel punto indicato. Se l'indicizzazione è di tipo "*clustered*" allora le tuple con ugual valore sono posizionate in modo contiguo su disco. In media il costo di lettura grazie ad un'indicizzazione effettuata tramite tabella hash è di 1.2 operazioni di *I/O*. Il costo per indicizzare tramite hash invece è di $1.2 + \left\lceil SF \cdot \frac{\#tuple}{\#tuple\ per\ pagina} \right\rceil$, dove la sigla *SF* è il "*selectivity factor*", ovvero il rapporto tra tutte le tuple della relazione e le tuple che soddisfano la condizione di ricerca.

INDICIZZAZIONE PER B⁺ TREE

Ogni albero è caratterizzato da un nodo radice, vari nodi intermedi e vari nodi foglia; ogni nodo coincide

con una pagina o blocco a livello di file System. I legami tra nodi vengono stabiliti da puntatori che collegano fra loro le pagine. I nodi foglia contengono i puntatori alle varie tuple. Inoltre è importante per il buon funzionamento di queste strutture dati è che gli alberi siano bilanciati. Negli alberi B^+, i nodi foglia sono collegati da una catena. Tale catena consente di svolgere in modo efficiente anche interrogazioni. Il costo di *lookup* è pari a $\log n$.
Indicizzare con $B^+\ tree$ significa poter rispondere a query aventi un range; per range si intendono valori di comparazione come minore e maggiore. Dunque, conviene usare un $B^+\ tree$ solo in questi casi, per confronti di uguaglianza è più veloce un indice realizzato tramite tabella Hash.

ORDINAMENTI
La necessità di operazioni di ordinamento emerge sia ai fini delle applicazioni, perché si desiderano risultati ordinati, sia per una corretta realizzazione delle proiezioni, con eliminazione dei duplicati. Inoltre, essa può essere utile per operazioni di *join*, oppure di raggruppamento. Il *join* è l'operazione più gravosa per un DBMS, in quanto è presente il rischio di un'esplosione dei numero di tuple del risultato.
Se vogliamo ordinare una relazione in base ad uno specifico attributo, allora dobbiamo ricorrere a due principali algoritmi che possono farlo. Il primo assume che il database ha solo 3 buffer nella sua memoria. Se *N* è il numero di pagine che la relazione occupa, questo algoritmo ha costo: $2 \cdot N \cdot (\lfloor \log_2 N \rfloor + 1)$.
Ma, tipicamente un database ha più di tre buffer in memoria disponibili per l'ordinamento. Perciò viene chiamato in causa un secondo algoritmo più ottimizzato per questa circostanza. Se abbiamo *B* buffer disponibili, allora il costo di ordinamento per una relazione composta da *N* pagine è:
$2 \cdot N \cdot \left(\left\lceil \log_{B-1} \left\lceil \frac{N}{B} \right\rceil \right\rceil + 1\right)$.

JOIN SENZA ORDINAMENTO (SIMPLE NESTED LOOP)
Nel *nested loop* una tabella viene definita come esterna e l'altra come interna alla memoria secondaria. Si esegue una scansione sulla tabella esterna; per ogni tupla ritrovata dalla scansione, si preleva il valore dell'attributo di *join* e si cercano le tuple della tabella interna che hanno lo stesso valore. Per questa seconda parte dell'algoritmo è utile una struttura hash o un indice sull'attributo.

$Costo\ R \bowtie S = N + N \cdot M$, dove *N* è il numero di pagine che costituiscono la relazione *R* e *M* è il numero di pagine che compongono la relazione *S*.

Osservazione: visto il costo di *join*, conviene effettuare il join impostando come primo fattore la relazione avente minor numero di tuple.

JOIN CON ORDINAMENTO (MERGE SCAN)
Questa tecnica richiede di esaminare le tabelle secondo l'ordine degli attributi di join ed è quindi particolarmente efficiente quando le tabelle sono già ordinate oppure quando sono definiti su di esse indici adeguati. Essa viene eseguita per mezzo di scansioni parallele sulle due tabelle. Le scansioni possono così ritrovare nelle tuple valori ordinati degli attributi di join; quando coincidono, vengono generate ordinatamente tuple del risultato.

$Costo\ R \bowtie S = CostoOrdinamentoR + CostoOrdinamentoS + N + M$

ESECUZIONE DI UNA QUERY
Quando una query è data al database per l'esecuzione, il database la converte in una "*query plan*". Una query plan è un albero di operatori di algebra relazionale che implementano la query. Chiaramente ci possono essere diversi query plan (che significa diversi alberi) per la stessa query SQL. Per ogni nodo dobbiamo poter calcolare il costo, il numero di tuple che le operatore genera. Alcuni operatori del query plan possono essere eseguiti appena le tuple arrivano (significa che non c'è bisogno di aspettare per tutti i risultati per essere eseguiti). Ciò significa che il costo è 0 poiché l'esecuzione avviene direttamente in memoria. Questi sono chiamati operatori "*on-the-fly*". Per esempio, immaginate la proiezione sugli attributi

A e B. Non abbiamo bisogno di aspettare tutte le tuple, ma come viene data una tupla, la A e la B sono tenute e il resto viene eliminato. Si noti che se vi è la necessità di fare eliminazione dei duplicati, inevitabilmente l'operatore deve attendere dapprima l'intero insieme di tuple ricevute.