I Database e MySQL

Forme evolute di persistenza, creazione e interrogazione di una base di dati

Dott. Ferdinando Primerano - 2023

Introduzione - concetto di database

Un “database” (base di dati) costituisce una forma evoluta di persistenza per i nostri oggetti, salvando il loro stato in strutture pensate per rendere più facili la ricerca e la modifica dei dati.

Vantaggi di un database

Il salvataggio in un database rende più semplice risalire all'oggetto desiderato, cercare per un dato criterio ("cercare tutti i docenti di informatica") o eseguire operazioni di modifica dei dati o della loro struttura ("aumentare lo stipendio di John Smith, aggiungere il campo specializzazione a tutti i dottori").

La maggior parte dei database fornisce un linguaggio specifico per eseguire tutte queste operazioni.

Concetto di DBMS

Un Database Management System (DBMS) è il sistema con cui creiamo e gestiamo le nostre basi di dati. Esempi di rilievo potrebbero essere MySQL, che useremo durante il corso, SQL Server, Oracle o MongoDB.

Un DBMS è un software che andrà scaricato e installato sul PC dello studente.

DBMS SQL e NoSQL

DBMS si dividono in due grandi famiglie:

- i DBMS relazionali, in cui i dati vengono salvati in tabelle potenzialmente collegate fra loro, e il cui linguaggio è SQL (Structured Query Language). Noi ci occuperemo di questa famiglia, di gran lunga la più diffusa.

- i DBMS non relazionali, che non utilizzano SQL e che organizzano i dati secondo altri modelli (ad esempio “collezioni di documenti”).

SQL - Structured Query Language

SQL è uno strumento per definire, gestire e interrogare i database relazioni.

La sigla SQL è usata per indicare il nome di una famiglia di linguaggi. Ogni DBMS implementa un sottoinsieme comune di SQL (“SQL Standard”) e può optare per espanderlo con delle aggiunte proprietarie.

Il SQL di MySQL è diverso da quello di Oracle, che è diverso da quello di SQL Server (detto Transact SQL). In gergo diciamo che ogni DBMS parla il proprio dialetto, ma tutti capiscono almeno una forma standard comune che funge da lingua di scambio.

Un database relazionale è composto di tabelle

Quale che sia il DBMS, un database SQL organizza i dati in **tabelle**, potenzialmente collegate da **relazioni**.

Ogni versione di SQL fornisce gli strumenti per modificare il contenuto delle tabelle (**DML**, Data Manipulation Language, linguaggio per la modifica dei dati, sottoinsieme di SQL), per modificarne o definirne la struttura (**DDL**, Data Definition Language, sottoinsieme di SQL) e per ricercare i dati di nostro interesse fra una o più tabelle (**SQL**, Structured Query Language sensu strictu).

Concetto di tabella - introduzione

Una tabella è un insieme di righe a cui abbiamo dato un nome. Ogni riga ha le stesse colonne delle altre righe della stessa tabella. Le colonne, dette anche campi, sono tipizzate – ogni colonna può contenere solo dati del proprio tipo specifico.

id	name	surname	dateofbirth
1	F	P	1980-02-05
2	S	R	1995-01-01

Rapporto fra classi entity e tabelle

Una tabella può essere vista come isomorfa a una classe entity. In effetti definiamo come entity ciò che tipicamente viene salvato sul database.

Con “isomorfa” intendiamo dire che hanno la stessa forma espressa in due maniere diverse.

Una classe entity in Java serve come stampo per gli oggetti di quel tipo. Una tabella SQL serve per definire le colonne, e le regole, che tutte le righe di quella tabella dovranno rispettare.

Java

```
class Person
{
  int id;
  String name;
  String surname;
  Date dateofbirth;
}
```

SQL

```
create table  Person
(
  id int primary key,
  name varchar(100),
  surname varchar(100),
  dateofbirth date
)
```

Rapporto fra oggetti entity e righe

Possiamo dire che, nel caso più semplice, una riga di una tabella corrisponde allo stato di una entity. La prima riga della tabella Person della tabella di esempio fornisce i dati sufficienti a ricostruire lo stato di un oggetto Person in Java. Viceversa, partendo da un oggetto Person in Java possiamo ricostruire una riga SQL.

Possiamo dire che le colonne, o campi, di una tabella SQL, corrispondono, nel caso più semplice, agli attributi (o proprietà) di un oggetto Java.

Così come per Java, i campi sono tipizzati. Il campo dateofbirth potrà contenere solo date. Il DMBS produrrà un errore nel caso in cui dovessimo provare a inserire dati non corrispondenti nella colonna (type mismatch, come in Java), per quanto questo non sia universale per tutti i DBMS SQL.

Creazione di una tabella in un database già esistente

```
USE test;
CREATE TABLE Person
(
    id int primary key,
    name varchar(100),
    surname varchar(100),
    dateofbirth date
);
```

E’ un esempio di DDL (Data Definition Language).

Selezioniamo un database (test) e creiamo una tabella (Person).

La tabella, una volta creata, è permanente e salvata su HD. Non serve ricrearla ogni volta, e questo vale per ogni comando SQL.

Abbiamo definito la struttura di tutte le future righe che andremo a inserire in questa tabella. Per adesso non badiamo al “primary key” dopo id, ma notiamo che abbiamo specificato le tipologie dei campi, e che varchar(100) è equivalente a una stringa con lunghezza massima = 100.

Creazione di nuove righe

```
INSERT INTO Person

(id, name, surname, dateofbirth)

VALUES

(1, 'F', 'P', '1980-05-02'),

(2, 'S', 'R', '1995-01-01'),

(3, 'A', 'B', '1996-01-01');
```

Questo comando è un esempio di DML, e porta alla creazione di due righe nella tabella Person, corrispondenti a due oggetti Person in Java.

Le righe create sono permanenti, memorizzate su HD. Non serve reinserirle ogni volta – è una forma di scrittura su file.

L'inserimento DEVE avvenire dopo la creazione della tabella (DML dopo DDL).

Modifica e cancellazione

Modificamo la data di nascita della persona con id = 2, cancelliamo la persona con id = 3 (DML)

```
UPDATE Person set dateofbirth='1995-02-01'
where id = 2;
DELETE FROM Person where id=3;
```

Assieme a INSERT, i comandi UPDATE e DELETE costituiscono i comandi DML di base (Data Manipulation Language).

Servono a manipolare i dati all'interno delle tabelle, creando o cancellando righe (insert, delete) o modificando i valori al loro interno (update).

Ricerca nelle tabelle

Vogliamo trovare nome e cognome di tutte le persone nate prima del 1990.

select name, surname from Person where dateofbirth < ‘1990-01-01’;

Con risultato:

name	surname
F	P

In questo caso si parla di SQL sensu strictu (Structured Query Language).

Abbiamo eseguito una operazione di filtro (eliminando le righe, vale a dire gli oggetti, che non ci interessavano, dal risultato) e di proiezione, vale a dire di mappatura, escludendo le colonne che non ci interessavano (id, dateofbirth) dal risultato. Tutto con una singola istruzione.

Nota sulle ricerche

Eseguendo nuovamente lo stesso comando potremmo avere risultati diversi – il contenuto della tabella potrebbe essere cambiato nel frattempo. Ogni nuova esecuzione porta al ricalcolo del risultato.

Tutti i comandi visti fino ad ora possono essere definiti “queries”, interrogazioni, ma solo il comando Select è una query strictu sensu. Query significa “domanda” o “interrogazione”.

Digressione sui DBMS

Perchè così tanti DBMS? Non avrebbe senso averne solo uno?

In generale, ogni DBMS copre una esigenza commerciale e/o tecnica. Grandi aziende possono decidere di produrre una propria soluzione in casa, per ragioni commerciali e d'impresa, mentre altri DBMS possono emergere senza una grande realtà alle spalle ma per ragioni tecniche o di alternativa open source.

DBMS da tenere in considerazione

In ambito relazionale:

- SQLite, leggero e diffuso, ma volutamente limitato e corredato solo di funzioni basilari e per questo installato spesso in ambito mobile
- Oracle, per progetti di grandi dimensioni, soprattutto Java
- SQL Server, la soluzione Microsoft per progetti di grandi dimensioni
- MySQL, DBMS diffusissimo per progetti piccoli e medi e per le web application in generale

Ogni DBMS parla il proprio “dialetto” di SQL, ma tutte le queries di base sono supportate universalmente. In ambito non relazionale, la soluzione più diffusa è sicuramente MongoDB, popolare per fornire buone prestazioni ai progetti Big Data.

Installazione di MySQL Server

Il DBMS MySQL viene fornito sotto forma di un server, vale a dire di un applicativo che fornisce un servizio ad altri applicativi, che prendono il nome di client.

Lo studente dovrà installare MySQL Server (il fornitore del servizio) e MySQL Workbench (il client, o consumatore, del servizio).

MySQL Workbench permetterà allo studente di eseguire i comandi SQL sui suoi database. Anche Java sarà un “client” per MySQL, come vedremo parlando di JDBC.

Le parti dell’installazione di MySQL – lato server

L’installazione di MySQL è divisa in lato server e lato client. Considerando il server abbiamo:

- il **server fisico** su cui viene installato, vale a dire il computer dello studente. Il nome standard è “localhost”, vale a dire, “la mia stessa macchina”
- il **server software**, vale a dire il programma che offre il servizio, MySQL Server. E’ il programma che viene scaricato e installato dal link fornito dal docente
- l’**istanza** di MySQL Server. Una istanza è un server attivo e in ascolto dietro una “porta”. La porta standard è la 3306. Una istanza di MySQL può supportare molti database, così come un database può contenere molte tabelle
- i **database**. Vengono creati su una istanza, e come abbiamo visto sono una collezione di tabelle

Le parti dell’installazione di MySQL – lato client

Considerando il client abbiamo:

- il **client fisico**, vale a dire il computer che accede al servizio. Tipicamente è sempre la macchina dello studente, vale a dire localhost

- il **client software**, vale a dire il programma usato per accedere al servizio. Tipicamente MySQL Workbench, che viene scaricato e installato sulla macchina dello studente assieme al server.

Non mancano le alternative, e successivamente anche Java sarà un client software.

Accesso al server – nome utente e password

La connessione al server MySQL richiede di specificare nome utente e password.

In fase di installazione il wizard di MySQL Server porterà lo studente a definire la password per l'utente **root**, vale a dire l'amministratore di sistema. Si consiglia allo studente di NON usare una password che usa per altro, visto che l'insegnante potrebbe avere bisogno di entrare con la sua utenza.

Nome utente e password verranno richiesti alla connessione di MySQL Workbench all'istanza, ma possono essere salvati. Andranno invece specificati in chiaro in Java.

Il codice riportato di seguito è pensato per essere eseguito nella finestra query editor di MySQL Workbench.

Creazione del database e delle tabelle sull’istanza

Si suppone che lo studente si sia connesso e abbia eseguito il login su una istanza locale del proprio MySQL Server.

Procediamo a creare il database a una prima tabella complessa su cui far pratica, come abbiamo visto parzialmente in precedenza. Per eseguire i comandi, riporteremo quanto scritto di seguito nel query editor e cliccheremo sull’icona col fulmine. Le singole istruzioni vanno separate col punto e virgola, come in Java.

Creazione del database e delle tabella sull'istanza: codice

```
CREATE DATABASE CENSUS;
USE CENSUS;
CREATE TABLE PERSON
(
    ID INT PRIMARY KEY AUTO_INCREMENT,
    NAME VARCHAR(100),
    SURNAME VARCHAR(100),
    CITY VARCHAR(100),
    ADDRESS VARCHAR(100),
    DATEOFBIRTH DATE,
    JOB VARCHAR(100),
    SALARY NUMERIC(7,2),
    GENDER VARCHAR(1)
);
```

Abbiamo creato il database census, lo abbiamo impostato come database corrente (use census), e abbiamo proceduto a creare una tabella.

Notiamo come salary sia un numero di sette cifre, di cui due dopo la virgola (quindi fino a 99999.99 euro).

Primary Key

Concentriamo l’attenzione su **ID INT PRIMARY KEY AUTO_INCREMENT**.

Ogni tabella dovrebbe avere (e avrà quasi sempre nella pratica) un campo o un insieme di campi definiti come “primary key”, chiave primaria, o “identificatore”. Due righe della stessa tabella non possono avere la stessa primary key – in questo caso, non potremo avere due persone con lo stesso ID – e a nessuna riga può mancare un valore per la primary key (non può essere nulla).

E’ corretto dire che la primary key identifica la riga all’interno della tabella. Una riga in un’altra tabella potrebbe avere lo stesso ID (potremmo avere una Person con ID = 3 e una Car con ID = 3).

AUTO_INCREMENT

AUTO_INCREMENT è una clausola di MySQL per indicare che il valore di ID viene generato automaticamente in successione. La prima persona avrà ID=1, la seconda ID=2, e poi 3, 4…

Questo rende non necessario specificare l'ID in fase di creazione della persona, ma è ancora possibile farlo (forzare un ID, a patto di non creare un doppione, come visto prima).

Notiamo che questo non impedisce la creazione di "buchi". Se ho quattro persone e cancello la persona con ID = 3, quell'ID sarà semplicemente perso, e la prossima persona avrà ID = 5, non 3. I "buchi" sono normali nei DB con ID AUTO_INCREMENT.

Scelta della chiave primaria

Un ID è una chiave primaria comoda e utilizzata quasi sempre, ma è artificiale e non è l'unica soluzione.

Qualunque campo il cui valore sia univoco per la riga nella tabella può fungere da chiave primaria. Per la persona avremmo potuto usare il codice fiscale, l'email o il numero di carta di identità.

Ogni campo che possa essere usato come chiave prende il nome di chiave candidata, e può essere utilizzato per risalire in maniera alternativa alla riga, ma nella pratica si preferisce quasi sempre avere un ID.

INSERT - sintassi ed uso

INSERT serve per creare nuovi dati in una tabella.
La sintassi può essere duplice:

INSERT INTO Tabella (nomicampi) values(valoricampi)
O
INSERT INTO Tabella VALUES (valoricampi)

Nel secondo caso è necessario specificare il valore per tutti i campi. Nel primo, possiamo specificare quali campi vogliamo inserire nelle nuove righe. I campi non specificati conterranno il valore null, o verranno assegnati automaticamente in casi particolari (ad esempio, nel caso di id int auto_increment).

INSERT - esempio di riferimento

```
INSERT INTO PERSON
(NAME,SURNAME,CITY,ADDRESS,DATEOFBIRTH,JOB,SALARY,GENDER)
VALUES
('JILL', 'RED', 'RACCOON VILLE', '2000-01-02', 'RANGER', 2500.00, 'F'),
('JACK', 'RED', 'RACCOON VILLE', '1998-01-02', 'RESEARCHER', 2700.00, 'M'),
('CHRIS', 'WHITE', 'NEW YORK', '1995-05-02', 'RANGER', 2500.00, 'M'),
('KEITH', 'WHITE', 'LISBON', '1980-06-02', 'RESEARCHER', 3500.00, 'M');
```

Si tratta della seconda forma di insert, avendo specificato i campi. Abbiamo inserito quattro righe nella tabella Person. I campi specificati vanno forniti per tutte le righe create. Le righe inserite vanno separate con una virgola e racchiuse fra parentesi tonde.

UPDATE - sintassi ed uso

INSERT crea nuove righe in una tabella. Update modifica i valori dei campi di righe già esistenti.

La sintassi è

UPDATE tabella SET nomeCampo1 = v1, nomeCampo2 = v2… nomeCampoN = v2 where Condizione

Update può modificare, potenzialmente moltissime righe allo stesso tempo, in base alla condizione. Vediamo un esempio concreto.

UPDATE - esempi di riferimento

UPDATE Person SET city='Seville', dateofbirth = '1982-06-02' WHERE id =4;

Questo comando modificherà i campi city e dateofbirth della persona con id=4 (ammesso che ci sia). Modificherà al massimo una riga. Volendo possiamo lanciare update di massa, in questo modo:

UPDATE Person SET Salary = Salary * 1.2 WHERE City = 'Seville';

Notate come abbiamo inserito una espressione (che funziona esattamente come le espressioni di Java) all'interno di un comando Update. Abbiamo ordinato al database di alzare del 20% lo stipendio a tutte le persone di Siviglia.

Safe Mode e come gestirlo

Non è detto che il DBMS accetti la query precedente – il DBMS può essere impostato per non permettere questa tipologia di modifica, e pretendere che nella condizione di modifica sia presente l'ID. In questo caso parliamo di "safe mode".

Ipotizzando di avere il safe mode attivato, la query di sopra non funzionerà. Possiamo disabilitare il safe mode, o trovare manualmente gli id dei Sivigliani. Ipotizziamo siano le persone 1 e 4, e riscriviamo la query:

UPDATE Person SET Salary = Salary * 1.2 WHERE ID in (1,4);

Funzionerà, anche in Safe mode.

DELETE - sintassi ed uso

Delete serve per rimuovere righe. La sintassi è molto semplice

DELETE FROM Tabella where Condizione

Notiamo che Delete NON cancella i valori dalle righe, ma le righe stesse (e quindi tutto il loro contenuto). Quindi non serve specificare i campi su cui agire, ma solo la condizione.

DELETE - esempio di riferimento

Vale per DELETE il discorso fatto per Update sul SAFE MODE.

DELETE FROM PERSON where City = 'Lisboa'

Potrebbe non funzionare.

DELETE FROM PERSON where id In (2,5)

funzionerà (ipotizzando che gli abitanti di Lisbona siano le persone 2 e 5).

Interrogazioni – il comando SELECT

Il comando SELECT è di gran lunga il più usato e permette di ritrovare i dati memorizzati in una o più tabelle tramite un singolo comando.

L'esecuzione del comando select produce una risposta sotto forma di una o più righe, le cui colonne non devono necessariamente corrispondere a quelle di una tabella ma possono essere mutate o calcolate, come vedremo in seguito. Procediamo a illustrare una serie di esempi di SELECT di complessità crescente, con commento.

SELECT * FROM PERSON;

Seleziona tutti i campi (*, "ALL") dalla tabella Person, prendendo tutte le righe.
Ipotizzando tre persone:

id	name	surname	city	address	dateofbirth	job	salary	gender
1	N1	S1	C1	A1	1980-01-01	J1	1000	M
2	N2	S2	C1	A2	1990-01-01	J1	1000	M
3	N3	S3	C2	A1	2000-01-01	J2	1200	F

SELECT NAME,SURNAME FROM PERSON;

Non ci interessavano tutti i campi stavolta. Non *, ma i campi che ci interessavano separati da virgola. In questo caso si parla di “proiezione” - scegliere i campi, cioè le colonne, di nostro interesse. Notiamo che non abbiamo eliminato righe, ma solo colonne. Coi dati di prima, questo sarebbe il risultato

name	surname
N1	S1
N2	S2
N3	S3

SELECT * FROM PERSON WHERE GENDER = 'F';

In questo caso parliamo di selezione – applichiamo una condizione (detta anche predicato) al risultato. Solo le righe che soddisfano la condizione vengono prese. Le altre non fanno parte del risultato. Stiamo prendendo tutti i campi, ma non tutte le righe.

id	name	surname	city	address	dateofbirth	job	salary	gender
3	N3	S3	C2	A1	2000-01-01	J2	1200	F

SELECT NAME,SURNAME FROM PERSON WHERE GENDER = 'F';

Abbiamo combinato selezione (eliminare righe) e proiezione (eliminare o modificare colonne), prendendo solo i nomi e i cognomi delle persone di sesso femminile.

name	surname
N3	S3

Connettori logici Java-Like nelle Select

Valgono i consueti connettori logici già visti in Java, che in SQL si possono scrivere in maniera più naturale:

SELECT NAME,SURNAME FROM PERSON WHERE GENDER='F' AND CITY='RACCOON VILLE';

SELECT NAME,SURNAME FROM PERSON WHERE GENDER='F' OR CITY='RACCOON VILLE';

SELECT NAME,SURNAME FROM PERSON WHERE GENDER='F' OR NOT CITY='RACCOON VILLE';

Operatori di confronto specializzati: IN

SQL nasce per eseguire ricerche raffinate con pochissimo codice, e dispone di operatori specializzati per semplificare le ricerche più comuni. E’ il caso dell’operatore **IN**:

SELECT NAME FROM PERSON WHERE CITY IN ('SEVILLE', 'LONDON');

È una versione più elegante e concisa di:

SELECT name FROM Person WHERE city = 'Seville' or city = 'London';

Operatori di confronto specializzati: BETWEEN

SQL mette a disposizione un operatore **between** per individuare un valore numerico compreso fra due estremi (minimo e massimo, inclusi):

SELECT NAME FROM PERSON WHERE YEAR(DATEOFBIRTH) BETWEEN 1980 AND 1990

Year è una funzione di MySQL. Estrae l'anno a partire da una data, ed è una delle tante funzioni predefinite di MYSQL. In questo caso si parla di una funzione scalare – non lavora su un vettore di elementi, ma su un numero fisso di valori. In questo caso stiamo estraendo tutti i nati fra il 1980 e il 1990. Coi dati di prova, avremo due righe di risultato.

Operatori di confronto specializzati: LIKE

L'operatore **LIKE** è un operatore specializzato per la ricerca parziale nelle stringhe. In alcuni casi non vogliamo una stringa specifica, ma qualunque riga che contenga in un dato campo una parola o parte di una parola:

SELECT * FROM PERSON WHERE NAME LIKE 'N%';

Questa query restituirà tutti i campi di tutte le Persone il cui nome comincia con la N. Il carattere % è un carattere speciale, e indica "0 o più caratteri, di qualunque genere, in qualunque ordine".

Questa query prenderà gli uomini di nome N, Nino, Nina, Nathaniel, Nostradamus, NonNeHoIdea… NON prenderà Marlon, perché la N deve essere la prima lettera.

LIKE (segue)

SELECT * FROM PERSON WHERE NAME LIKE ‘%N%’;

Questa query prenderebbe anche Marlon. In generale, questa query prenderà qualunque persona il cui nome contenga una N.

Ci sono altri caratteri speciali, ma il loro utilizzo per ora non è necessario.

Confronti fra date

In SQL le Date sono tipi primitivi. Possono essere utilizzate con i consueti operatori di confronto: maggiore, minore, uguale e loro combinazioni:

SELECT NAME FROM PERSON WHERE DATEOFBIRTH> '1980-01-01';

Questa query restituirà i nomi di tutte le persone nate dopo il 1980. Avremo tre righe di risultato, e una sola colonna, coi dati di prova.

Espressioni in SQL

Le SELECT, le UPDATE e altre istruzioni di SQL consentono il calcolo di espressioni.

Una espressione SQL è equivalente a una espressione in Java – è qualunque cosa restituisca un valore. Uno degli utilizzi più intuitivi è il calcolo di un valore per ogni riga del risultato di una query, creando una colonna derivata che somiglia, nella pratica, al calcolo di un metodo.

SELECT NAME, SURNAME, YEAR(NOW())-YEAR(DATEOFBIRTH) AS AGE FROM PERSON

Questa query produce un risultato composto da tre colonne e, coi dati di prova, tre righe.
Le prime due colonne sono campi proiettati normalmente, letti dalla tabella Person, mentre la terza è il prodotto di una espressione, e merita un'analisi più approfondita.
L'espressione viene calcolata *per ogni riga di risultato*, e per ognuna genera una nuova colonna. Si parla sempre di proiezione – il valore così calcolato prende il nome provvisorio, e non necessario ma piacevole, di age. E' l'equivalente MySQL di un metodo getAge() per un oggetto Person in Java.

Analisi dettagliata dell’espressione

Posta l’espressione YEAR(NOW()) - YEAR(DATEOFBIRTH):

- **year(date)** è una funzione predefinita. Estrae l’anno da una data
- **now()** è una funzione predefinita. Fornisce la data e il tempo attuali in MySQL
- **year(now())** è l’anno corrente. Al momento in cui scriviamo (Agosto 2023) year(date()) = 2023
- **year(dateofbirth)** è la funzione year applicata a ogni singola riga. Ad esempio, per il signor N1, nato il primo Gennaio 1980, year(dateofbirth)=1980- l’anno attuale meno l’anno di nascita restituisce, approssimativamente, l’età della persona.

A questo calcolo viene dato il nome di colonna di “age”, tramite l’operatore as (operatore di aliasing)

I risultati coi dati di prova

name	surname	age
n1	s1	43
n2	s2	33
n3	s3	23

La colonna age NON è memorizzata, ma calcolata. E’ il frutto di una espressione, l’equivalente del calcolo di un metodo in Java.

Ordinare i risultati

Siamo anche in grado di ordinare i dati e di limitare il numero di righe restituite. Vediamo un esempio reale:

SELECT CONCAT(NAME,' ', SURNAME) AS CITIZEN, YEAR(NOW()) - YEAR(DATEOFBIRTH) AS AGE FROM PERSON WHERE CITY = 'RACCOON VILLE' ORDER BY YEAR(NOW()) - YEAR(DATEOFBIRTH) DESC LIMIT 10;

La clausola ORDER permette di ordinare per una o più colonne (in questo caso, per una colonna derivata, l'età), in questo caso in ordine decrescente (opzione DESC), e riporta solo i primi 10 risultati. Abbiamo stampato nominativo ed età dei dieci cittadini più anziani di Raccoon Ville. A scopo dimostrativo, abbiamo anche fuso nome e cognome in un campo unico, separato da uno spazio in mezzo, di nome "citizen", tramite la funzione concat.

Ordinare i risultati - un altro esempio

Un'altra applicazione immediata:

SELECT NAME, SURNAME, NATIONALITY FROM RUNNINGCONTEST WHERE TITLE='MARATHON' ORDER BY TRACKTIME LIMIT 3;

In mancanza dell'opzione DESC l'ordinamento avviene dal basso verso l'alto, quindi prenderemo i tre atleti col tempo più basso ad aver partecipato alla maratona, vale a dire gli atleti sul podio.

In Java useremo quasi sempre la forma SELECT *, per ricavare tutti i campi, ma imposteremo spesso dei filtri per caricare dal database solo i dati che ci interessano, e tipicamente vorremo anche specificare un ordine. Possiamo ordinare e filtrare anche in Java, ma è meno efficiente che ricevere i dati già manipolati dal DB

Lavorare riga per riga

Abbiamo visto come SQL offra funzioni equivalenti a map e filter, ma non ancora a reduce.

Una volta selezionate m righe a partire da n (con m<=n), possiamo mapparle per ottenere righe diverse dalle originali (ad esempio aggiungendo o togliendo colonne, o rinominandole), ma non abbiamo ancora visto come ricavare risultati che riguardino più di una riga.

Diciamo che la nostra visibilità è "sulla singola riga". YEAR(), NOW() e CONCAT(), così come i normali operatori (+, -, ecc..) sono "scalari", vale a dire che prendono in ingresso i campi di una riga. Stiamo lavorando “riga per riga”.

Funzioni di gruppo

Di contro, SQL offre anche funzioni di gruppo che lavorano su insiemi di righe, sintetizzandole (riducendole, reduce) a dei risultati singoli:

```
SELECT SUM(SALARY) FROM PERSON WHERE CITY = 'RACCOON VILLE';
-- SOMMA DEI SALARI DI RACCOON VILLE, IN SOSTANZA IL SUO PIL.
SELECT AVG(SALARY) FROM PERSON WHERE JOB = 'RESEARCHER';
-- SALARIO MEDIO DI UN RICERCATORE.
SELECT COUNT(*) FROM PERSON;
-- NUMERO DELLE PERSONE IN ARCHIVIO.
SELECT MAX(SALARY), MIN(SALARY) FROM PERSON WHERE YEAR(DATEOFBIRTH) = 1980;
-- SALARIO MASSIMO E MINIMO DEI NATI NEL 1980.
```

Risultati delle funzioni di gruppo

Mentre le funzioni di riga, scalari, producono un risultato per ogni riga, le funzioni di gruppo producono un risultato per ogni gruppo, dove con gruppo intendiamo un qualunque insieme di righe. Ad esempio:

SELECT COUNT(*) as n FROM PERSON;

Produrrà i seguenti risultati dai dati di prova:

n
3

Il gruppo su cui abbiamo applicato la riduzione è la tabella Person. La funzione di gruppo è count, conteggio, e produce una sola riga di risultato a fronte di un gruppo con tre righe. Avrebbe prodotto una riga di risultato anche se avessimo avuto 2000 righe.

Funzioni di gruppo ed espressioni

SUM, AVG, COUNT, MAX e MIN sono funzioni di gruppo, lavorando non su una riga ma su gruppi di righe. Entrano insiemi di righe ed escono valori singoli, che è il funzionamento di base del reduce.
Possiamo usare queste funzioni anche come parte di una espressione numerica:

-- SALARY GAP FRA IL PIÙ PAGATO E IL MENO PAGATO DEI NATI NEL 1980.
SELECT MAX(SALARY)-MIN(SALARY) AS DELTA FROM PERSON WHERE YEAR(DATEOFBIRTH) = 1980;

Analisi dell’esempio precedente

Partiamo da:

SELECT MAX(SALARY)-MIN(SALARY) AS DELTA FROM PERSON WHERE YEAR(DATEOFBIRTH) = 1980;

Operiamo una selezione (solo i nati nel 1980), e sulle righe selezionate abbiamo eseguito due funzioni di gruppo (max e min). Questo ci ha restituito una sola riga a partire dalle n righe dei nati nel 1980.

max(salary)	min(salary)
2500	1200

Analisi dell’esempio precedente - segue

Sulla riga vista in precedenza eseguiamo una trasformazione aritmetica scalare (una proiezione sui dati di una sola riga):

max(salary)	min(salary)	delta = max(salary)-min(salary)
2500	1200	1300

In questo caso abbiamo lavorato su un gruppo. Il gruppo era composto da una selezione di righe (tutti i nati nel 1980), ed è poi stato compresso in una sola per fornire alla fine un unico risultato. E' una operazione di riduzione, o matematicamente parlando una funzione che trasforma un insieme di righe in un valore

Creazione di gruppi per segmentazione del gruppo originale : GROUP BY

Supponiamo di volere lo stipendio medio per professione. Una possibilità è la seguente:

SELECT AVG(SALARY) AS AVERAGE FROM PERSON WHERE JOB = 'RESEARCHER';
SELECT AVG(SALARY) AS AVERAGE FROM PERSON WHERE JOB = 'CLERK';
…
…
E così via per ogni professione.

Primo esempio di GROUP BY

Non è una vera soluzione. Dovremmo conoscere a priori tutte le professioni da tracciare, ed eseguire n queries, ciascuna delle quali restituirà una sola riga.

La soluzione corretta è quella di ordinare a SQL di raggruppare in base ai valori presenti nella tabella, e di applicare a ogni gruppo una o più funzioni di gruppo. Questo comportamento si ottiene tramite la clausola GROUP BY. Vediamo la soluzione corretta e spieghiamo cosa avviene:

SELECT JOB, AVG(SALARY) AS AVERAGE FROM PERSON GROUP BY JOB;

Funzionamento di GROUP BY

GROUP BY job significa "dividi la tabella in gruppi di righe con lo stesso valore per la colonna job". Una altra definizione potrebbe essere “segmenta l’insieme originale in sottoinsiemi”, o il gruppo in sottogruppi.

Notiamo che GROUP BY non elimina righe, ma redistribuisce le righe presenti in gruppi separati e disgiunti.

Vediamo il funzionamento a livello grafico, ipotizzando di avere 3 Researcher, 2 Clerk e un Officer nella nostra tabella Person.

Funzionamento di GROUP BY - seconda parte

Ipotizziamo i seguenti dati di partenza, omettendo le colonne non necessarie:

id	job	salary
1	Researcher	3000
2	Clerk	1800
3	Researcher	2000
4	Researcher	2500
5	Officer	2000
6	Clerk	1600

Funzionamento di GROUP BY - terza parte

GROUP BY divide i dati originali in gruppi in base al valore della colonna job. Otteniamo tre gruppi distinti:

id	job	salary
1	Researcher	3000
3	Researcher	2000
4	Researcher	2500
id	**job**	**salary**
2	Clerk	1800
6	Clerk	1600
id	**job**	**salary**
5	Officer	2000

Funzionamento di GROUP BY - quarta parte

Come visto in precedenza, ogni gruppo genera una riga di risultato, che conterrà il risultato delle funzioni di gruppo calcolate e, potenzialmente, i campi per cui abbiamo raggruppato (in questo caso job). Il gruppo “Officer” contava di una sola riga, ma conta comunque come gruppo e quindi produce un risultato.

Job	Average
Researcher	2500
Clerk	1700
Officer	2000

Raggruppamenti multipli

E’ possibile raggruppare (partizionare) per diversi campi o espressioni. Quali potrebbero essere i risultati dalla query : **SELECT JOB, GENDER, AVERAGE(SALARY) AS AVERAGE FROM PERSON GROUP BY JOB, GENDER** ?

job	gender	average
Researcher	M	x1
Researcher	F	x2
Clerk	M	x3
Clerk	F	x4
Officer	F	2000

Raggruppamenti multipli - segue

Il totale delle righe è sempre sei, ma abbiamo partizionato più finemente. I gruppi che prima erano semplice raggruppati per job (Researcher, Clerk, Officer) ora sono stati raggruppati per job e gender, quindi potremmo avere, potenzialmente, sei gruppi (tre professioni moltiplicate per due generi):

Researcher F, Researcher M, Clerk F, Clerk M, Officer M, Officer F

Gruppi omessi

Alcuni gruppi potrebbero mancare. Avevamo un solo Officer, che poteva essere M o F. Nel nostro caso era F – il gruppo Officer M era vuoto ed è stato omesso dal risultato: i gruppi vuoti vengono omessi dal risultato finale di una query con group by.

Avremmo potuto non avere Researcher M, se tutti i ricercatori fossero stati donne, e così via.

Una volta ottenuti i gruppi “superstiti” si producono i risultati applicando a ogni gruppo le funzioni di gruppo desiderate.

Combinare i nostri strumenti

Capiterà spesso di usare assieme raggruppamento, selezione, proiezione e ordinamento:

SELECT JOB, GENDER, AVERAGE(SALARY) AS AVERAGE FROM PERSON WHERE SALARY>=1000 GROUP BY JOB, GENDER ORDER BY JOB

Abbiamo escluso chi guadagnava meno di 1000 euro dal calcolo, e questo potrebbe creare dei gruppi vuoti. Stiamo raggruppando per due campi, ordinando per uno, proiettando due campi e una funzione di gruppo.

Combinare i nostri strumenti - segue

Data: **SELECT JOB, GENDER, AVERAGE(SALARY) AS AVERAGE FROM PERSON WHERE SALARY>=1000 GROUP BY JOB, GENDER**

L'ordine di esecuzione è selezione - partizionamento - riduzione – proiezione - ordinamento.
Prima vengono eliminate le righe non desiderate, successivamente si creano i gruppi, si eseguono i calcoli e si proiettano i risultati desiderati. L'ordinamento viene eseguito per ultimo. Notiamo che si tratta sempre di selezione sulla riga, almeno per ora, ma se rimuoviamo tutte le righe rimuoviamo anche il gruppo ("un gruppo senza righe non produce risultato").

Proiezione e funzioni di gruppo

Se usiamo le funzioni di gruppo, possiamo proiettare solo i campi per cui abbiamo raggruppato e funzioni di gruppo o calcoli su questi.
SELECT C1, C2, AVG(C3), SUM(C4), AVG(C1+C2), MIN(C3)+MAX(C2) FROM P GROUP BY C1, C2 è corretto. Posso prendere MIN(C3) anche senza raggruppare per C3.

Invece **SELECT C1, C2,C3, AVG(C3), SUM(C4), AVG(C1+C2), MIN(C3)+MAX(C2) FROM P GROUP BY C1, C2;** è errato. Non dovrei prendere C3 se non raggruppo per C3.

MySQL lo accetterà ma altri DBMS, più rigorosi, non lo permettono. La ragione è semplice: che senso ha proiettare un campo che potrebbe cambiare da una riga all'altra dello stesso gruppo? Ogni gruppo genera una riga, e quella riga dovrebbe rappresentare il gruppo. Non voglio proiettare un campo che potrebbe cambiare all'interno dello stesso gruppo.

Selezione sui gruppi - HAVING

Supponiamo di avere molti più dati di quelli presi per esempio adesso. Magari tutti i dati nazionali italiani, quindi circa sessanta milioni di righe, o magari anche solo trenta milioni non considerando bambini e persone fuori dal mercato del lavoro.

Ipotizziamo di voler calcolare il salario medio per città. E' abbastanza semplice:

SELECT CITY, AVG(SALARY) AS AVERAGE FROM PERSON GROUP BY CITY;

Esempio precedente e uso di HAVING

Ma l'Italia è la nazione delle tante città. Potremmo non essere interessati a nuclei troppo piccoli, soprattutto considerando che magari le persone indicate lavorano poi in una città vicina. Supponiamo di voler considerare solo le città con almeno 30.000 abitanti (che è già una buona soglia, escludendo anche alcuni capoluoghi di provincia). Notiamo che noi non possiamo desumere dalla singola riga quanti abitanti abbiamo in una data città.

E' una proprietà **del gruppo** (una funzione di gruppo, la funzione count), **non del singolo**. Dovremmo tenere quei gruppi (quelle città) per cui COUNT(*) ("conta le righe") è >=30.000.

La maniera corretta di farlo è tramite l'utilizzo della clausola **HAVING**

Primo esempio di HAVING

SELECT CITY, AVG(SALARY) AS AVERAGE FROM PERSON GROUP BY CITY HAVING COUNT(*)>30000;

In having abbiamo un predicato (condizione) applicato a un gruppo, non alla singola riga, ed è il punto ideale per imporre condizioni (predicati) su funzioni di gruppo. In questo caso, abbiamo escluso i gruppi con meno di 30.000 righe. Ma potremmo fare di meglio:

Secondo esempio di HAVING

SELECT CITY, AVG(SALARY) AS AVERAGE FROM PERSON GROUP BY CITY WHERE SALARY>1000
HAVING COUNT(*)>30000
ORDER BY AVG(SALARY) DESC;

Proiezione, selezione, raggruppamento, ordinamento e selezione di gruppo (HAVING) . Abbiamo eliminato dal calcolo chi aveva un salario troppo basso, trattandosi tipicamente di borse di studio o stage e decidendo che non ci interessavano. Questa query prenderà solo le città con almeno 30.000 lavoratori con stipendi significativi, quindi principalmente le grandi città.

Lavorare su più tabelle

Il nostro lavoro precedente ha determinato il salario medio degli stipendiati senior delle grandi città. Ora vogliamo rapportarlo al potere di acquisto e ai costi delle stesse.

Questo ci porta ad avere bisogno di una seconda tabella, City, coi dati dei costi delle città. Per comodità, supporremo di avere bisogno solo dei dati di Roma e di Milano, useremo il nome della città come chiave primaria (pessima idea) e i nostri dati potrebbero essere i seguenti:

La nostra seconda tabella: City

name	averagerent	basicexpenses
Milano	800	200
Roma	650	150

La tabella city ha il campo name, a indicare il nome della città. La tabella Person ha il campo city, a indicare la città di residenza. Questi due campi indicano una associazione, ma bisognerà spiegare al DB come usarli.

Usare due tabelle nella stessa query

Vogliamo confrontare i salari delle persone col costo della vita nelle rispettive città. I dati sui salari sono in Person, i dati sulle spese in City.

Questo ha senso - non vogliamo memorizzare i dati delle spese per ogni persona, o semplicemente non disponiamo dei dati di dettaglio, e vogliamo confrontare ogni persona con le spese medie della sua zona, in mancanza di meglio. E’ il nostro caso.

Potremmo provare in questo modo:

Primo tentativo di unire dati da due tabelle diverse

La nostra query potrebbe avere questa forma:

SELECT PERSON.SURNAME, PERSON.CITY, CITY.NAME, PERSON.SALARY - CITY.AVERAGERENT - CITY.BASICEXPENSES FROM PERSON, CITY;

Stiamo prendendo i campi surname, city e salary dalla tabella Person, name, averagerent e basicexpenses da city. Il nostro FROM legge i dati da due tabelle diverse, Person e City. Intuitivamente potremmo pensare che ogni persona verrà associata alla sua città ma **non è così**. Cominciamo definendo dati di prova:

I nostri dati di prova

Person

id	surname	city	salary
1	Rossi	Roma	1200
2	Verdi	Milano	1300

City

name	averagerent	basicexpenses
Milano	800	200
Roma	650	150

Il nostro primo risultato, e un problema

Partendo da quanto vista prima:
SELECT PERSON.SURNAME, PERSON.CITY, CITY.NAME, PERSON.SALARY - CITY.AVERAGERENT - CITY.BASICEXPENSES FROM PERSON, CITY;
otteniamo:

surname	city	. city.name	salary-averagerent-basic expenses
Rossi	Roma	Milano	1200-800-200
Rossi	Roma	Roma	1200-650-150
Verdi	Milano	Milano	1300-800-200
Verdi	Milano	Roma	1300-650-150

Cosa è successo?

La sintassi **SELECT … FROM A, B** esegue un **prodotto cartesiano** fra la prima tabella (A) e la seconda (B), vale a dire, combina tutte le righe della prima e della seconda. Il signor Rossi risulta vivere sia a Roma che a Milano, e lo stesso vale per il signor Verdi.

Il prodotto cartesiano restituisce tutte le possibili combinazioni fra le righe di A e di B, in questo ordine. Quindi (A1,B1), (A2,B1),(A2,B1),(A2,B2) nel nostro caso, con A = Person e B = City. Alcune di queste combinazioni hanno un senso - altre no.

Prodotto cartesiano - introduzione formale

Il risultato di **SELECT * FROM A,B** viene detto **prodotto cartesiano fra A e B**, o **CROSS JOIN**. Come visto alla slide precedente, combinerà tutte le righe di A con tutte le righe di B. Ipotizzando di avere n righe in A ed m righe in B, e di indicare come Ai o Bi la i-esima riga di A o di B, il prodotto cartesiano è definito come qualunque possibile combinazione:

(Ai,Bk), con 1<=i<=n e 1<=k<=m.

Il prodotto cartesiano conterrà quindi n x m super-righe, ciascuna composta dall'unione di una riga di A e una riga di B. Ipotizzando di avere A con 3 righe e B con 2, il prodotto cartesiano AxB sarà il seguente:

Esempio di prodotto cartesiano AxB

A	B
a1	b1
a1	b2
a2	b1
a2	b2
a3	b1
a3	b2

a3b1: la terza riga di a associata con la prima riga di b

Prodotto cartesiano - riepilogo e proiezione

SELECT * FROM A, B, il prodotto cartesiano fra A e B, produce super-righe composte, a sinistra, da una riga di A e a destra da una di B. Non essendoci condizioni particolari, il DB ci propone ogni possibile combinazione di righe A con le righe B.

Lo stesso varrebbe utilizzando la proiezione, e prendendo quindi solo alcuni campi da A e da B (come si fa nella pratica). Ipotizzando di prendere solo i campi C1 e C2 da A e il campo C3 da B, avremmo il seguente risultato:

Prodotto cartesiano e proiezione

Tabella A		Tabella B
A.C1	A.C2	B.C3
c1-1	c2-1	c3-1
c1-1	c2-1	c3-2
c1-2	c2-2	c3-1
c1-2	c2-2	c3-2
c1-3	c2-3	c3-1
c1-3	c2-3	c3-2

Select A.c1, A.c2, B.c3 from A,B

c1-1 è il valore del campo c1 per la riga 1 della tabella A.
La combinazione (c1-1, c2-1, c3-1) è la combinazione della prima riga di A (di cui prendiamo solo c1 e c2) con la prima riga di B, di cui prendiamo solo c3.
Resta comunque un prodotto cartesiano: abbiamo combinato tutte le righe con tutte le altre.
E questo non lo vogliamo.

Decidere quali combinazioni tenere

surname	city	. city.name	salary-averagerent-basic expenses
Rossi	Roma	Milano	1200-800-200
Rossi	Roma	Roma	1200-650-150
Verdi	Milano	Milano	1300-800-200
Verdi	Milano	Roma	1300-650-150

Torniamo al nostro primo risultato.
Si tratta di una proiezione su prodotto cartesiano. Abbiamo combinato tutte le persone (Rossi e Verdi, P1 e P2) con tutte le città (Milano e Roma, C1 e C2). Il nostro risultato è P1 C1, P1 C2, P2 C1, P2 C2.
Alcune di queste super-righe hanno senso, altre no. Decidiamo quali tenere.

Introduzione formale al concetto di JOIN

Un **JOIN** è un prodotto cartesiano a cui viene imposta una condizione, vale a dire una forma di selezione. Essendo il prodotto cartesiano formato da super righe provenienti da più origini (ad esempio, P1 C1, Persona e Città, Rossi e Milano), il JOIN è la condizione che specifica quali combinazioni tenere e quali buttare.

Lo stesso prodotto cartesiano può essere visto come un **JOIN** (Cross Join), ipotizzando una condizione sempre vera (“WHERE TRUE”).

Introduzione formale al concetto di INNER JOIN

Un **INNER JOIN** è un caso particolare di JOIN, per cui vengono mantenute solo le super righe le cui componenti (Ai, Bk) sono in relazione fra loro. La relazione è specificata tramite una condizione particolare che prende in esame campi di entrambe le tabelle.

Se la condizione è vera per una data super-riga, le sue parti sono in relazione e la super-riga viene mantenuta. In caso contrario non sono in relazione, e la super-riga viene scartata.

INNER JOIN in pratica

Partiamo dal prodotto cartesiano:
SELECT PERSON.SURNAME, PERSON.CITY, CITY.NAME, PERSON.SALARY - CITY.AVERAGERENT - CITY.BASICEXPENSES FROM PERSON, CITY;
ma imponiamo una condizione aggiuntiva. Decidiamo che le righe di Person sono collegate alle righe di City se il campo City di Person contiene lo stesso valore del campo Name di City.
Esponendolo sotto forma di condizione avremmo:

Person.City = City.Name

“il campo City della Person è uguale al campo Name della City per quella super riga”.
Ci basta aggiungerla in un WHERE (selezione) alla fine della query.

Il predicato di JOIN

La condizione **Person.City = City.Name** viene detta predicato di join, o condizione di unione. Come possiamo vedere è una condizione che coinvolge campi di tutte e due le tabelle, e non è un caso. Serve per decidere se la riga i-esima della prima tabella è in relazione con la riga k-esima della seconda. La incorporiamo nella query in questo modo:

```
SELECT PERSON.SURNAME, PERSON.CITY, CITY.NAME, PERSON.SALARY - CITY.AVERAGERENT - CITY.BASICEXPENSES FROM PERSON, CITY WHERE PERSON.CITY = CITY.NAME;
```

Il predicato di JOIN in azione

surname	city	. city.name	salary-averag erent-basicex penses	person.city = city.name
Rossi	Roma	Milano	1200-800-200	false SCARTATA
Rossi	Roma	Roma	1200-650-150	true TENUTA
Verdi	Milano	Milano	1300-800-200	true TENUTA
Verdi	Milano	Roma	1300-650-150	false SCARTATA

Il risultato finale - INNER JOIN di Person e City per Person.City = City.Name

SELECT PERSON.SURNAME, PERSON.CITY, CITY.NAME, PERSON.SALARY - CITY.AVERAGERENT - CITY.BASICEXPENSES FROM PERSON, CITY WHERE PERSON.CITY = CITY.NAME;

surname	city	. city.name	salary-averagerent-basicexpenses
Rossi	Roma	Roma	1200-650-150
Verdi	Milano	Milano	1300-800-200

Definizione formale di relazione fra tabelle

Non è strettamente necessario che due tabelle abbiano una relazione predefinita per poterle usare assieme, ma è di gran lunga il caso più comune.

Una relazione determina quali **righe** della tabella A devono essere associate alla tabella B, vale a dire, quali righe dovrebbero essere accettate da una operazione di JOIN operante su quella relazione e quali rifiutate. Quando diciamo che A e B hanno una relazione vuol dire che alcune righe di A potrebbero essere associate logicamente ad alcune righe di B.

In SQL le relazioni sono espresse tramite condizioni sui campi, vale a dire tramite predicati di join. Nel nostro esempio precedente, **Person.City = City.Name**

Analisi del rapporto fra City e Person

La relazione fra City e Person in Java prende il nome di associazione, e sappiamo darle una cardinalità: una città - n persone.

In SQL questo concetto, come del resto tutte le relazioni in Java, viene reso come relazione tra tabelle, e il predicato di join ci dirà per quali righe vale la relazione "vive a" fra Person e City, vale a dire, quali oggetti di tipo Person sono collegati a quali oggetti di tipo City. Un oggetto (riga) di tipo City potrà essere collegato a n oggetti (righe) di tipo Person.

Analisi del rapporto fra City e Person - segue

Ricapitolando:

- fra Person e City c'è una relazione, per ora implicita, che potremmo leggere "vive in" (andando da Person a City) oppure "ospita", andando da City a Person.
- potremmo anche scriverla come "una persona vive in una città" o "una città ospita n persone"
- La relazione è 1-n. A una persona assoceremo una città, mentre a una città più persone. Per ora la relazione è stata ottenuta implicitamente dalla presenza dei campi Person.city e City.name, ma in seguito la potremo dichiarare per ottenere diversi vantaggi operativi.

Migliorare la relazione fra City e Person

Usare il nome della città per indicare la relazione non è una buona idea. Diverse città al mondo portano lo stesso nome, in stati diversi. Ipotizzando un'anagrafe globale, rischiamo di non saper distinguere fra una Paris e un'altra, o fra le varie Springfield degli stati Uniti.

Nella pratica, torniamo al problema della chiave primaria. Le città non vengono identificate univocamente dal loro nome, e questo rende difficile anche collegarle correttamente ai loro abitanti.

Primo passo - aggiungere un ID alla tabella City

Il nome non era una chiave primaria adatta. Rifattorizziamo la struttura di City aggiungendo un campo id che farà da chiave primaria (quindi da identificatore della riga). **City.id diventa chiave primaria in City**.

id	name	averagerent	basicexpenses
1	Milano	800	200
2	Roma	650	150

Secondo passo - passare da city a cityid in Person

Modifichiamo la struttura di Person per avere non il nome della città (che potrebbe essere ambiguo, ma il suo id). Abbiamo la chiave primaria di una tabella (l'id della città) all'interno di un'altra (la tabella Person).

In questo caso, ma non solo, parliamo di **chiave esterna**. Una chiave esterna è un id (o altro campo, ma quasi sempre un id) che indica una riga a cui la riga in cui siamo è collegata.

La nostra tabella Person, una volta modificata, diventa come segue:

Secondo passo - passare da city a cityid in Person (segue)

id	surname	cityid	salary
1	Rossi	2	1200
2	Verdi	1	1300
3	Blu	1	1500

Il campo id in city non può ripetersi. E' chiave primaria. Di contro, il campo cityid in Person può ripetersi eccome. E' chiave esterna e non ci sono vincoli di non ripetizione. Per l'occasione abbiamo aggiunto il signor Blu, milanese. Come vedete cityid si può ripetere: sia la prima che la terza riga di Person sono collegate alla prima riga di City.

Terzo passo - cambiare il predicato di join

Non abbiamo più city in Person, e name non è più chiave primaria in city. La nostra condizione di unione, il nostro predicato di join, diventa City.ID = Person.CityID, e la query diventa

SELECT PERSON.SURNAME, PERSON.CITY, CITY.NAME, PERSON.SALARY - CITY.AVERAGERENT - CITY.BASICEXPENSES FROM PERSON, CITY WHERE CITY.ID = PERSON.CITYID

Il risultato è lo stesso di prima, ma abbiamo prevenuto qualunque tipo di ambiguità.

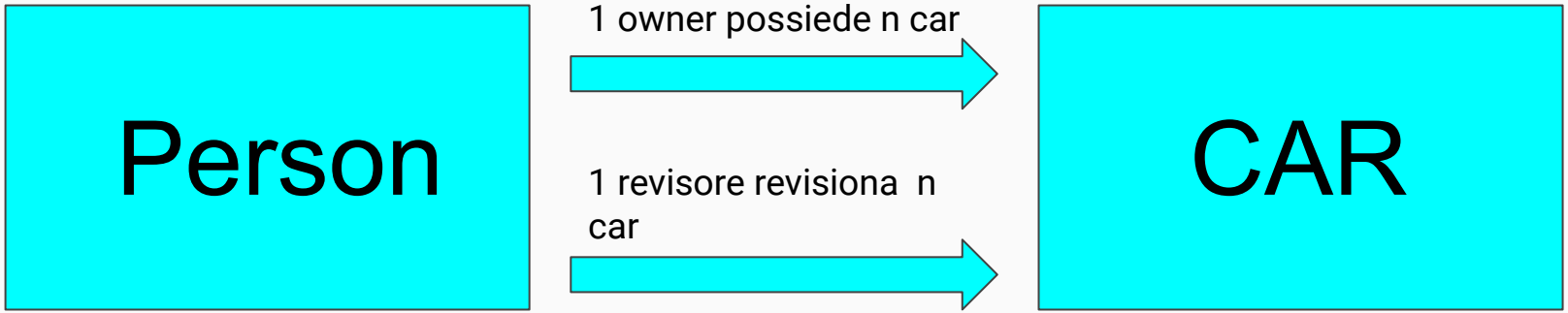
Relazioni multiple tra le stesse tabelle

Due tabelle possono presentare, fra di loro, più di una relazione. La relazione fra City e Person ci è venuta naturale - “una città ospita molte persone”, ma studiamo un esempio diverso.

Partiamo da una tabella Car (auto). Un’auto deve avere un proprietario, e per semplicità diciamo solo uno. Un’auto deve anche passare il processo di revisione periodico, e deve esserci un revisore.

Tanto il proprietario che il revisore sono persone. Una auto potrebbe voler memorizzare sia l’id del proprio proprietario sia l’id dell’ultimo revisore. *Ogni relazione richiederà una chiave esterna diversa.*

Relazioni multiple - l’esempio delle auto



Tra Person e Car ci sono due relazioni 1-n. Una persona può possedere molte auto e revisionare molte auto. Un auto può avere un solo proprietario e un solo revisionatore (consideriamo l’ultimo).

La tabella Car

La tabella Car conterrà il proprio id (sempre presente), il modello, la targa, l'id del proprietario (l'id di una persona) e l'id del proprio revisore (un'altra persona, o potenzialmente anche la stessa). Gli ultimi due campi sono *chiavi esterne verso la tabella Person*, vale a dire, collegamenti a righe di Person. Una riga della tabella Car potrebbe essere la seguente:

id (pk)	model	plate	ownerid (fk)	revisorid (fk)
1	Fintus	666	1	2

Ricavare il proprietario, ricavare il revisore

La riga con targa=666 di Car è collegata al signor rossi come owner, al signor Verdi come revisore. Per prendere il nome del proprietario possiamo scrivere:
SELECT PERSON.NAME, PERSON.SURNAME FROM PERSON, CAR WHERE PERSON.ID = CAR.OWNERID AND CAR.PLATE='666';

Mentre per prendere il nome dell'ultimo revisore:
SELECT PERSON.NAME, PERSON.SURNAME FROM PERSON, CAR WHERE PERSON.ID = CAR.REVISORID AND CAR.PLATE='666';
Ogni chiave esterna indica una relazione.

L’integrità referenziale e il problema degli orfani

E se avessimo righe di questo tipo in Person?

id	surname	cityid	salary
10	A	-30	1200
11	B	NULL	1000

Definizione informale di orfano

Visto che gli id tipicamente partono da 1, direi che nè A1 nè A2 sanno dove vivono. I due cityid dovrebbero riferirsi a una riga di City, ma probabilmente quelle righe non esisteranno.

In questo caso diremo che le righe 1 e 2 di Person sono **orfane** rispetto alla relazione "vive in" con City. "Orfane" intese come "senza padre", o "senza un elemento di riferimento". Possiamo usare le informazioni di quella riga, ma non possiamo collegarla in maniera sensata a nessuna città. Lo stesso accadrebbe se avessimo cityid = 1, e la riga corrispondente mancasse in city, o fosse stata modificata cambiando id.

Gestione degli orfani

Questo ci pone di fronte al problema della politica di gestione degli orfani. Vogliamo accettarli o meno? E possiamo impedirli?

La risposta alla prima domanda è "dipende". Dobbiamo decidere se la relazione fra le tabelle è semanticamente **obbligatoria** per il nostro lavoro (e quindi renderebbe inutile il programma) o se è **opzionale**. In questo caso dobbiamo decidere se vogliamo sapere per forza la città di appartenenza di una persona.

La risposta alla seconda domanda è "sì, ma c'è un prezzo".

Prassi comune di gestione degli orfani

La prassi comune è di evitare gli orfani. Questo si ottiene con due step:

- si enunciano in maniera esplicita le relazioni, in fase di creazione delle tabelle
- si impostano delle politiche di gestione degli orfani quando la tabella "padre" o "riferimento" cambia

Studiamo il problema tornando all'esempio dell'auto, e decidendo che vogliamo avere sempre le informazioni relative a proprietario e revisore.

Dichiarazione esplicita delle relazioni e della politica di gestione dei cambiamenti

```
CREATE TABLE CAR
(
  ID INT PRIMARY KEY AUTO_INCREMENT,
  PLATENUMBER VARCHAR(500),
  MODEL VARCHAR(500),
  COST NUMERIC(10,2),
  OWNERID INT,
  REVISORID INT,
  FOREIGN KEY(OWNERID) REFERENCES PERSON(ID)
  ON UPDATE CASCADE
  ON DELETE RESTRICT,
  FOREIGN KEY(REVISORID) REFERENCES PERSON(ID)
  ON UPDATE CASCADE
  ON DELETE RESTRICT
);
```

La parte evidenziata dichiara in maniera esplicita le relazioni che prima erano solo de facto.

Adesso il DB è al corrente del legame fra i campi ownerid e revisorid di Car e il campo id di Person, e si pone la domanda: cosa succederà se cancelliamo una Person collegata alla Car, o se ne modifichiamo l'id? Analizziamo gli effetti di questa DDL.

Analisi della clausola FOREIGN KEY

FOREIGN KEY dichiara in maniera esplicita una relazione, vale a dire un legame fra due o più campi. Nel nostro caso, la tabella Car sta dichiarando una relazione prima fra il proprio campo ownerid e il campo id di person e poi fra il proprio campo revisorid e il campo id di person. La riga di Car è legata a due righe diverse di Person, come abbiamo visto prima, con due rapporti diversi.

Questo basta a ottenere un primo risultato: non potremmo inserire valori nulli in ownerid e revisorid, in quanto le chiavi esterne (come del resto le primarie) non possono essere nulle. In fase di creazione la riga dovrà sapere di chi è l'auto e chi l'ha revisionata.

ON UPDATE CASCADE

Le due righe successive (on update cascade e on delete restrict) specificano una politica di gestione del rapporto, e indirettamente degli orfani. Cosa succede se qualcuno cambia l'id di una persona che possiede tre macchine, portandolo da 3 a 6?

Si tratta di una operazione di update. Sto aggiornando la tabella padre, e per gli aggiornamenti abbiamo optato di effettuare un **cascade**, vale a dire una propagazione, verso l'entità figlia. Le auto della persona 3, che è diventata 6, adesso hanno ownerid = 6 anche loro. Il cambiamento è stato eseguito automaticamente dal DBMS.

ON DELETE RESTRICT

E se provassi a cancellare la persona 3? In questo caso la politica che abbiamo scelto era **restrict**, come accade tipicamente in questi casi. Restrict significa che non posso cancellare la persona finchè ci saranno auto di sua proprietà, o anche auto che ha revisionato. Se voglio uccidere il "padre" devo prima cancellare tutti i suoi "figli". Se voglio eliminare una riga della tabella lato 1 devo cancellare tutte le righe delle tabelle lato 2 a cui è legata.

In generale, con restrict, non ho l'autorizzazione a cancellare una riga finché esistono righe legate a lei da una qualunque relazione con questa clausola

Politiche alternative

E se avessimo scelto on delete cascade per la chiave esterna ownerid? Cancellando la persona, avremmo cancellato tutte le auto a lei collegate - la cancellazione si sarebbe propagata ai "figli".

Potrebbe avere senso per un cliente che desidera cancellare il proprio storico. Invece potrebbe non volerlo e mantenere restrict per il revisore (in generale, il revisore non dovrebbe poter cancellare i propri dati o i dati del proprio lavoro o dei propri errori, quindi è più corretto restrict).

Integrità referenziale

L'unione della clausola FOREIGN KEY, che mi obbliga ad avere delle righe collegate, e delle sottoclausole ON UPDATE e ON DELETE forniscono quella che viene detta "integrità referenziale", vale a dire la sicurezza di avere le righe a cui ci riferiamo, e indirettamente l'assenza di orfani. Un modo rapido di ricordarne la funzione è appunto **"niente orfani"**.

Non è una funzione supportata da tutti i DBMS, e può rendere più complicato l'inserimento dei dati, non essendo possibile inserire figli senza padri in presenza di integrità referenziale, ma è una funzione praticamente onnipresente, e necessaria, in ambiente enterprise.

Da Java a SQL - modellizzare le relazioni fra oggetti

Le relazioni fra tabelle rispecchiano quelle fra entità (in effetti, le tabelle rappresentano lo stato di entità), ma mentre in UML abbiamo rapporti

specifici e in Java abbiamo oggetti collegati in un DBMS SQL abbiamo "solo" una tipologia di rapporto: il collegamento fra righe. Vediamo come tradurre quello a cui siamo abituati in tabelle, partendo dal rapporto di ereditarietà.

Il rapporto di ereditarietà portato in SQL

Il rapporto di ereditarietà è in realtà un rapporto di specializzazione. Un oggetto di un tipo A può anche essere di un tipo B, dove B è una sottocategoria di A. Supponiamo di voler tracciare, solo per gli insegnanti, la scuola di riferimento ("scuola polo"), gli anni di anzianità e il titolo di studio, oltre alle materie per cui è abilitato.

Una prima soluzione sarebbe modificare Person, aggiungendo le colonne relative (anzianità, titolo di studio, scuola polo), ma *resterebbero vuote* per tutte le righe che non sono relative a degli insegnanti. Quello che vogliamo fare invece è simulare l'ereditarietà (che non esiste nei DBMS SQL standard) con un rapporto 1-1 **opzionale**. Creeremo una tabella “figlio” che conterrà i dati del sottotipo, separati da quelli del supertipo, che farà capo a una tabella “padre”.

La tabella Teacher (“sottotipo” di Person)

La categoria più generale, la classe "padre", viene "mappata" sulla tabella Person che abbiamo visto in precedenza. A questa aggiungiamo una tabella Teacher con questa forma:

```
CREATE TABLE TEACHER
(
ID INT PRIMARY KEY, SCHOOL VARCHAR(100),
DEGREE VARCHAR(100), SUBJECTS VARCHAR(100), YEARS INT,
FOREIGN KEY(ID) REFERENCES PERSON(ID) ON CASCADE UPDATE ON DELETE RESTRICT
);
```

Analisi del rapporto fra Teacher e Person

La colonna id in Teacher ha una duplice funzione. Internamente a Teacher è chiave primaria ma anche *chiave esterna verso Person*. Questo vuol dire che una riga di Teacher deve avere una riga di Person di riferimento per completarsi. Teacher non conosce il proprio nome, la propria città, la propria data di nascita.

Di contro, Person non è costretto ad avere chiavi esterne, perché per Person questa relazione non è obbligatoria, ma opzionale. Una riga di Person potrebbe essere collegata a una riga di Teacher, ma potrebbe anche fare un altro lavoro, nel quel caso non troveremo l'id di quella Person nella tabella Teacher. Il rapporto è obbligatorio per Teacher (tutti gli insegnanti sono persone) ma opzionale per Person (non tutte le persone sono insegnanti).

Esempio concreto di Teacher e Person

Person

id	name
1	A
2	B
3	C

Teacher

id	school
1	S1
3	S2

Abbiamo omesso i campi non rilevanti per amor di brevità. Ma perché manca l'id 2 in Teacher? E' la stessa ragione per cui l'id di Teacher non è auto_increment. Potremmo dover saltare dei numeri. L'ID di Teacher non ha una esistenza autonoma, ma **deve** essere uguale a un id in Person.

I signori A e C sono insegnanti (le righe 1 e 3 di Person sono collegate con righe di Teacher) e lavorano in S1 ed S2 rispettivamente. Il signor B si occupa di altro, e da questi dati non sappiamo di cosa. Potrebbe essere uno studente, ma non abbiamo informazioni.

Il predicato di INNER JOIN per una relazione 1-1

Poste due tabelle con un rapporto 1 a 1 (che è sempre opzionale per la tabella “padre”, che rappresenta la superclasse), il nostro predicato di INNER JOIN si ottiene semplicemente uguagliando le chiavi primarie:

TabellaPadre.PK = TabellaFiglio.PK o, più correttamente, **TabellaSuperClasse.PK = TabellaSottoClasse.PK**

Solo in questo caso, quindi, è sufficiente uguagliare gli id: **Person.ID = Teacher.ID**

Esempio di INNER JOIN 1-1 - Person e Teacher

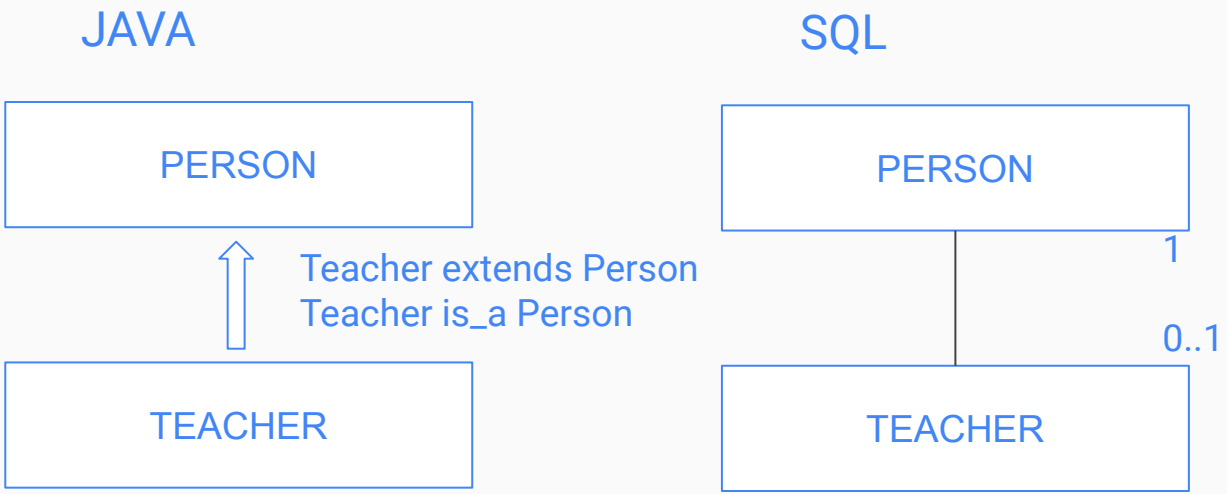
Il nostro INNER JOIN quindi si può scrivere:

SELECT * FROM PERSON, TEACHER WHERE PERSON.ID = TEACHER.ID;

e produrrà:

Person.id	Person.name	Teacher.id	Teacher.School
1	A	1	S1
3	C	3	S2

Il nostro lavoro in grafica - da Java a SQL



Una riga di Teacher deve essere collegata a una riga di Person, ma una riga di Person potrebbe essere collegata a 0 righe di Teacher - un Teacher è una Person, ma non tutte le Person sono Teacher.

Forma esplicita per INNER JOIN

SQL dispone di una forma esplicita con un operatore apposito, INNER JOIN appunto, per imporre il predicato di join fuori dal where. Questo permette di separare adeguatamente i predicati di join (condizioni strutturali) dalle condizioni di ricerca dei dati (condizioni di dominio).
La forma esplicita per il nostro INNER JOIN, che è da preferire a quella implicita vista fino ad ora, è la seguente: **SELECT * FROM PERSON INNER JOIN TEACHER ON PERSON.ID = TEACHER.ID**.

La sua forma generale invece è la seguente, ed è quella che useremo sempre d'ora in avanti:

SELECT * FROM T1 INNER JOIN T2 ON (PREDICATO DI JOIN)

Notiamo che il risultato è il medesimo calcolato in precedenza. E' solo una forma più elegante.

Il problema di INNER

Ma cosa vuol dire INNER? INNER è una parola chiave che serve a indicare che dobbiamo prendere solo le righe per cui quella condizione è vera, vale a dire per cui vale la relazione, come abbiamo visto prima. Lo possiamo ricordare come "prendi le righe collegate e solo quelle".

Applicando lo INNER JOIN ai nostri dati di esempio abbiamo perso i dati relativi al signor B, perché la seconda riga di Person (id=2, name=B) non ha corrispettivi nella tabella Teacher - **non partecipa alla relazione**, e quindi viene esclusa dal nostro INNER JOIN.

Quando INNER non basta - LEFT JOIN

Ma se volessimo vedere anche B? Chiaramente non potremmo associargli una scuola, non essendo lui un insegnante, ma magari vorremo vederlo nella lista comunque. Come fare a prenderlo, e che valore avrà il campo School per lui?

Possiamo prendere anche B utilizzando un join diverso, usato tipicamente per queste eventualità, che prende il nome di LEFT JOIN. La sintassi generale è la seguente:

SELECT * FROM T1 LEFT JOIN T2 ON (PREDICATO DI JOIN)

Spiegazione del LEFT JOIN

SELECT * FROM T1 LEFT JOIN T2 ON (PREDICATO DI JOIN)

Si legge "prendi le righe di t1 e collegale alle righe di t2 per cui vale la relazione espressa dal predicato di join. Se non trovi righe di t2, presenta comunque la riga di t1 e combinala con una riga di valori null per i campi di t2". La possiamo ricordare come "preserva tutte le righe di t1" o "preserva tutte le righe a sinistra", che partecipino o meno alla relazione.

Vediamone un caso concreto:

Esempio concreto di left join - Person left join Teacher on Person.id=Teacher.id

SELECT * FROM PERSON LEFTJOIN TEACHER ON PERSON.ID = TEACHER;

produrrà:

Person.id	Person.name	Teacher.id	Teacher.School
1	A	1	S1
2	B	NULL	NULL
3	C	3	S2

Considerazioni finali sul LEFT JOIN

La riga 2, il signor B, che non partecipa alla relazione con Teacher, è comunque presente, ed è stato associato a una riga composta di valori null. Il left join non escluderà mai le righe a sinistra, che possono essere quelle della tabella che rappresenta la superclasse, ma possono anche essere quelle della tabella lato 1 in un rapporto 1-n, come vedremo fra poco.

Notiamo che l'ordine conta. E' la tabella che compare a sinistra dell'operatore LEFT JOIN che viene preservata. Non ci sono garanzie di preservare quella a destra.

Condizioni di dominio

Niente vieta che una riga venga esclusa da una condizione di dominio. Supponiamo di voler trovare tutte le persone con un lavoro della città con id =1. Per gli insegnanti vogliamo visualizzare anche la scuola. La soluzione sarebbe la seguente:

SELECT PERSON.NAME, PERSON.SURNAME, PERSON.JOB, TEACHER.SCHOOL FROM PERSON LEFT JOIN TEACHER ON PERSON.ID = TEACHER.ID WHERE PERSON.CITYID = 1;

Anche se il left join mantiene tutte le righe della tabella Person, molte di quelle righe verranno eliminate dalla condizione di dominio, che pretende che cityid sia uguale a 1.

Esempio di combinazione di inner e left join - lavoro su tre tabelle

Abbiamo usato cityid di proposito. Usando solo le tabelle person e teacher non abbiamo accesso al nome della città, ma se avessimo voluto lo stesso risultato per gli abitanti di Milano? Ci sarebbe servita una terza tabella, City:

SELECT PERSON.NAME, PERSON.SURNAME, PERSON.JOB, TEACHER.SCHOOL FROM PERSON INNER JOIN CITY ON CITY.ID = PERSON.CITYID LEFT JOIN TEACHER ON PERSON.ID = TEACHER.ID WHERE CITY.NAME = 'MILANO';

Si traduce in questo modo: "A ogni persona associa la sua città, e pretendi di avere la città o scarta la persona (inner join, solo le righe collegate). A questa combinazione aggiungi, se c'è, una riga di Teacher collegata a Person, altrimenti una riga di valori nulli".

Analisi dell’esempio precedente e regole generali

Il risultato di questa query, prima della proiezione di nome, cognome, lavoro e scuola, è una lista di righe del tipo **{P(i), C(k), T(i)}**, dove P(i) è la riga di person con id i, C(k) è la riga di City con id k collegata a P(i) e T(i) è l'eventuale Teacher collegato a P(i). Notiamo che Person e Teacher hanno lo stesso id (i), essendo il join eseguito sulla chiave primaria. T(i) sarà nullo nel caso in cui la persona non sia un insegnante (campi vuoti).

Come regola generale, se abbiamo **n** tabelle nel nostro from avremo **n-1 join** (left o inner, opzionali o obbligatori) il nostro risultato sarà composto da super-righe divise in n parti, prese una da ogni tabella. Su quella super-riga eseguiremo la proiezione per ricavare quello che ci interessa. Vale per ogni tipologia di relazione, non solo per le 1-1.

Relazioni 1 - n in SQL

I rapporti 1-n sono quelli che in Java abbiamo chiamato variamente uso, associazione, aggregazione o composizione, e che abbiamo già visto applicare in pratica da MySQL nella prima sezione sul join. Si tratta di rapporti *has_many*, non *is_a*.

In termini generici, a **una entità (riga)** di una tabella t1, che chiameremo **lato 1**, corrispondono **n entità (righe)** di una tabella t2, che chiameremo **lato n**. Queste vengono legate tipicamente tramite l'uso di una chiave esterna, per quanto questo non sia sempre il caso, e come abbiamo visto prima due tabelle possono essere legate da più di una relazione (o rapporto che dir si voglia).

Uso / aggregazione / composizione

Nel caso in cui vogliamo esprimere in SQL un rapporto di associazione o aggregazione, la maniera naturale di operare è tramite integrità referenziale con le clausole on update cascade e on delete restrict.

Nel caso in cui si tratti di composizione, la clausola più naturale potrebbe essere on delete cascade (gli oggetti dipendenti nel rapporto di composizione non dispongono di solito di una esistenza indipendente rispetto al contenitore).

Uso / aggregazione / composizione (segue)

Il rapporto fra una squadra e i suoi membri potrebbe essere un rapporto 1-n di associazione / uso / aggregazione in Java (le differenze sono flebili), ma cancellare una squadra non dovrebbe portare alla cancellazione dei giocatori.

Il rapporto fra un prodotto e le sue recensioni dovrebbe essere un rapporto 1-n di composizione - cancellare il prodotto rende insensate le righe delle recensioni, e di conseguenza potrebbe essere il caso di cancellarle contestualmente.
Dal punto di vista delle interrogazioni, però, tutte queste relazioni sono rese allo stesso modo.

Struttura generale per due tabelle collegate da un rapporto 1-n

Poste due tabelle A e B e avendo una relazione 1 A - n B (una riga di A è associata a n righe di B), si procede in questo modo:

- A viene creata per prima
- B viene creata per seconda
- B contiene, *oltre ai propri campi specifici fra cui l'ID*, la chiave primaria di A, che in B diventa chiave esterna verso A, oltre al proprio ID. A contiene il proprio ID (A.ID), B contiene il proprio ID e l'id di una riga di A (B.ID, chiave primaria, B.AID chiave esterna verso A)

Esempio formale di struttura 1-n

A - lato 1

ID
1
2
3

B - lato n

ID	aID
1	1
2	1
3	3

La prima e la seconda riga di B sono collegate alla prima riga di A. La terza riga di B è collegata alla terza riga di A. La seconda riga di A non è collegata a nessuna riga di B, ma non è un problema - una riga lato 1 non è obbligata ad avere delle righe corrispondenti nella tabella lato n.

Predicato di join per una INNER JOIN 1-n

Il predicato per l'INNER JOIN di una relazione 1-N è A.ID = B.aID, dove A è la tabella lato 1 e B è la tabella lato N. In linguaggio naturale, eseguire l'inner join fra A e B significa uguagliare la chiave primaria di A e la chiave esterna di B verso A:

A.PK = B.FK

Esempio concreto di INNER JOIN 1-N

Poste le tabelle Product e Review: un prodotto può ricevere molte recensioni, una recensione può riguardare un solo prodotto. Il rapporto è 1-n, con Product lato 1 e Review lato n.
Per le regole viste in precedenza, Review contiene productID che farà da chiave esterna verso Product. L'inner join fra le due tabelle si scrive come:

SELECT PRODUCT.*, REVIEW.* FROM PRODUCT INNER JOIN REVIEW ON PRODUCT.ID = REVIEW.PRODUCTID

Calcolo dei risultati

Product - lato 1

ID	Name
1	A
2	B
3	C

Review - lato n

ID	Score	Product ID
1	5	1
2	4	1
3	2	3

SELECT PRODUCT.*, REVIEW.* FROM PRODUCT INNER JOIN REVIEW ON PRODUCT.ID = REVIEW.PRODUCTID

Product. ID	Product. Name	Review. ID	Review. Score	Review. Product ID
1	A	1	5	1
1	A	2	4	1
3	C	3	2	3

PK

FK

Uso del predicato di join per la LEFT JOIN

Notiamo come il prodotto B non compaia nei risultati. E’ corretto - la inner join per definizione prende solo le righe collegate fra di loro, e il prodotto B non è collegato a niente. Potremmo voler ovviare al problema, utilizzando lo stesso predicato di join visto in precedenza, ma specificando un operatore di join differente - il LEFT JOIN visto in precedenza.

In generale, il predicato di join rimane lo stesso per qualunque tipo di join, ma cambia il modo in cui viene interpretato. Nel caso della INNER JOIN, il predicato assume un valore assoluto ed escludiamo le righe non collegate. Nel caso della LEFT JOIN prendiamo le righe collegate ove esistono, ma in mancanza di queste “creiamo” una riga di NULL.

Calcolo dei risultati con la LEFT JOIN

Product - lato 1

ID	Name
1	A
2	B
3	C

Review - lato n

ID	Score	Product ID
1	5	1
2	4	1
3	2	3

SELECT PRODUCT.*, REVIEW.* FROM PRODUCT **LEFT** JOIN REVIEW ON PRODUCT.ID = REVIEW.PRODUCTID

Product.ID	Product.Name	Review.ID	Review.Score	Review.Product ID
1	A	1	5	1
1	A	2	4	1
3	C	3	2	3
2	B	NULL	NULL	NULL

PK FK

Raggruppamento e join - procurarsi i dati

Possiamo applicare le funzioni di gruppo anche al risultato delle join, e questo è spesso utile nei rapporti 1-n. Immaginiamo di dover calcolare la votazione media per ogni prodotto. Il punteggio (score) sarà contenuto nella recensione, mentre il nome del prodotto e il suo id saranno nella tabella prodotti.

Procediamo per gradi, mettendo assieme i dati necessari:

```
SELECT PRODUCT.ID, PRODUCT.NAME, REVIEW.SCORE FROM PRODUCT
INNER JOIN REVIEW ON PRODUCT.ID = REVIEW.PRODUCTID;
```

Raggruppamento e funzioni di gruppo su JOIN- la media dei punteggi

Abbiamo i dati che ci servono. Ora dobbiamo *ridurli* a un risultato di nostro gusto:

SELECT PRODUCT.ID, PRODUCT.NAME, AVG(REVIEW.SCORE) AS AVERAGE FROM PRODUCT INNER JOIN REVIEW ON PRODUCT.ID = REVIEW.PRODUCTID GROUP BY PRODUCT.ID, PRODUCT.NAME;

Volevamo mantenere product.id e product.name, di conseguenza abbiamo dovuto raggruppare (product.id e product.name sono campi liberi), mentre non abbiamo raggruppato per score. Di score ci *interessava la media*.

Raggruppamento e funzioni di gruppo su JOIN - il miglior punteggio e il peggiore

Possiamo anche trovare il miglior punteggio e il peggiore:

SELECT PRODUCT.ID, PRODUCT.NAME, AVG(REVIEW.SCORE) AS AVERAGE, MIN(SCORE) AS WORST, MAX(SCORE) AS BEST FROM PRODUCT INNER JOIN REVIEW ON PRODUCT.ID = REVIEW.PRODUCTID GROUP BY PRODUCT.ID, PRODUCT.NAME;

Raggruppamento e funzioni di gruppo su JOIN - condizioni di dominio

Potremmo voler escludere alcuni elementi, ad esempio le recensioni troppo vecchie e ormai poco significative. Questo significa imporre condizioni di dominio, non altri join. Le condizioni di dominio tipicamente riguardano le colonne di una sola tabella di quelle usate nel join.

SELECT PRODUCT.ID, PRODUCT.NAME, AVG(REVIEW.SCORE) AS AVERAGE, MIN(SCORE) AS WORST, MAX(SCORE) AS BEST FROM PRODUCT INNER JOIN REVIEW ON PRODUCT.ID = REVIEW.PRODUCTID GROUP BY PRODUCT.ID, PRODUCT.NAME WHERE YEAR(REVIEW.REVIEWDATE)>=2020;

Raggruppamento e funzioni di gruppo - e le righe non collegate?

E così via, ma resta un "problema". E' sempre un inner join, quindi non vedremo i prodotti senza recensioni. Ovviamo usando un LEFT JOIN:

SELECT PRODUCT.ID, PRODUCT.NAME, AVG(REVIEW.SCORE) AS AVERAGE, MIN(SCORE) AS WORST, MAX(SCORE) AS BEST FROM PRODUCT LEFT JOIN REVIEW ON PRODUCT.ID = REVIEW.PRODUCTID GROUP BY PRODUCT.ID, PRODUCT.NAME WHERE YEAR(REVIEW.REVIEWDATE)>=2020;

Una nota finale sui JOIN

Per terminare, non tutti i rapporti sono dichiarati e non tutti i join richiedono chiavi esterne. Supponiamo di avere una tabella buyer con una colonna budget, rappresentante quanto vuole spendere per un regalo, e di avere una tabella present (regali). Supponiamo di voler associare a ogni buyer tutti i regali che può permettersi:

SELECT BUYER.NAME, PRESENT.NAME FROM BUYER, PRESENT WHERE BUYER.BUDGET>= PRESENT.COST;

Anche questa è una relazione uno-a-molti, espressa non tramite una uguaglianza ma tramite una disuguaglianza, su cui non possiamo ovviamente applicare l'integrità referenziale. Il predicato di join non è per forza un uguaglianza, per quanto sia il caso più comune.

Rapporti fra righe della stessa tabella, self join e alias di tabella

In alcuni casi vorremo accoppiare i dati di una tabella con righe provenienti dalla stessa tabella. Si parla in questo caso di self-relationship per la tabella in questione. Nella pratica prenderemo la tabella due volte, con due ruoli diversi.

Vediamo un esempio pratico. Supponiamo di avere una tabella **employee** con un rapporto 1-n fra impiegati, quindi fra righe della stessa tabella: un **responsabile** coordina n **junior**. In questo caso una riga della tabella impiegati avrà il ruolo di responsabile e sarà accoppiata a n righe col ruolo di junior.

Struttura della tabella Employee

Il tracciato della tabella potrebbe essere il seguente:

```
CREATE TABLE EMPLOYEE
(
  ID INT PRIMARY KEY,
  NAME VARCHAR(100),
  SURNAME VARCHAR(100),
  COORDINATORID INT
);
```

coordinatorid è chiave esterna di employee verso employee, ed esprime il rapporto fra la riga e il suo coordinatore.

Non esprimo integrità referenziale in quanto mi aspetto che ci siano impiegati senza responsabili, per cui coordinatorid dovrà essere NULL.

Uso di una self-relationship con alias di tabella

Supponiamo di dover riportare il cognome della risorsa assieme al cognome del suo coordinatore. Cominciamo col prodotto cartesiano:

SELECT * FROM EMPLOYEE, EMPLOYEE;

Questa query darà errore! MySQL ha bisogno di distinguere le due tabelle per poter capire da quale delle due sta prendendo i campi, o meglio, da quale riga. La soluzione è fornire un *alias di tabella*:

SELECT * FROM EMPLOYEE COORDINATOR, EMPLOYEE JUNIOR

INNER JOIN per una self-relationship

La tabella employee viene presa una volta come coordinator, una volta come junior. Tutte le righe vengono combinate con tutte le altre, come abbiamo visto in precedenza. Noi lo eviteremo tramite l'uso del predicato di join:

SELECT * FROM EMPLOYEE COORDINATOR, EMPLOYEE JUNIOR WHERE COORDINATOR.ID = JUNIOR.COORDINATORID;

MySQL "crede" di avere due tabelle, coordinator e junior, che in realtà esistono solo in questa query, ma le vede come tabelle separate e possono essere usate esattamente come se lo fossero, quindi col consueto join 1-n.

Ricavare i dati che ci interessano da più righe di Employee

Adesso possiamo proiettare i campi di nostri interesse:

SELECT JUNIOR.SURNAME AS JUNIORSURNAME, COORDINATOR.SURNAME AS COORDINATORSURNAME FROM EMPLOYEE COORDINATOR, EMPLOYEE JUNIOR WHERE COORDINATOR.ID = JUNIOR.COORDINATORID;

La sintassi tabella.campo è obbligatoria in questo caso: le due tabelle sono identiche e hanno gli stessi campi. MySQL non saprebbe distinguere altrimenti. Notiamo che abbiamo proiettato un campo dalla riga di Employee che funge da junior (junior.surname) e uno dalla riga di Employee che funge da coordinatore (coordinator.surname), ma sempre usando la tabella Employee.

Self relationship e join atipici

Poste queste differenze (gli alias di tabella obbligatori, la sintassi bella.campo) si possono applicare tutti gli altri join. Ad esempio, possiamo scrivere una query pensata per identificare i possibili genitori e figli a partire dalle età nella tabella Person:

SELECT PARENT.ID AS PARENTID, CHILD.ID AS CHILID FROM PERSON PARENT, PERSON CHILD WHERE PARENT.SURNAME =CHILD.SURNAME AND YEAR(PARENT.DATEOFBIRTH) - YEAR(CHILD.DATEOFBIRTH) BETWEEN 20 AND 50

Troviamo tutte le persone con lo stesso cognome e per cui ci sia una differenza fra i 20 e i 50 anni fra la possibile riga genitore e la possibile riga figlio. In questo caso il predicato di join è costituito da tutte le condizioni dopo il where, e si tratta di un inner join implicito.

Il rapporto n-n - introduzione informale

Il rapporto n a n non è supportato nativamente da MySQL, pur essendo molto comune. E' il tipo di rapporto per cui **una riga di t1 è correlata con n righe di t2, e una riga di t2 con n righe di t1**.

In questo caso si parla di rapporto molti-a-molti fra le tabelle A e B, ricordandoci sempre che si parla di rapporti fra righe, e non è possibile definire un lato 1 un lato n, ma bisogna ricorrere a una soluzione più raffinata che richiede l'utilizzo di un'altra tabella, come vedremo a breve, dopo una introduzione a un caso pratico.

Esempio pratico di n-n - Persone e Patenti

Prendiamo un esempio concreto. Per molti arriva il momento in cui si torna a casa dicendo "ho preso la patente". E' una espressione imprecisa, come molte del linguaggio naturale. Non esiste una sola patente (potrebbero essere la B per gli autoveicoli, la A per le moto, una delle varie C per i veicoli pesanti), e soprattutto **non c'è una sola patente A per tutta la popolazione mondiale**.

Il signor James, tornando a casa con la sua patente in tasca, **non ha l'unica patente A esistente**. E' **uno dei tanti** a disporre di quella specifica certificazione. Non solo: il signor James potrebbe anche avere preso la patente B in precedenza, e quindi **non è detto che James "abbia" una sola patente**.

Il rapporto fra una persona e una patente è di tipo **n-n se intendiamo la patente come "tipologia", non come oggetto fisico**. La maggior parte di quelli che mi leggeranno avranno preso la patente B, ma c'è differenza fra il concetto di patente B e l'oggetto in plastica riposto nel portafogli.

Studio di una soluzione

Poniamo di avere, nella nostra solita tabella Person, i signori James e Jill, con id 1 e 2 rispettivamente. Jill “ha” le patenti A e B, James solo la B. Ipotizziamo anche di avere la seguente tabella **License**:

id	name	examcost
1	A	200
2	B	250
3	C	1000

Dobbiamo trovare un modo di associare i dati delle varie patenti alle singole persone (James e Jill nel nostro caso).
Questo si ottiene tramite una terza tabella che viene detta associativa.

Definizione di tabella associativa

Regola generale: un rapporto n-a-n fra t1 e t2 **si scompone in due rapporti 1-n** verso una tabella intermedia che viene detta **associativa (a)**. t1 **n-a-n** t2 diventa t1 **1-n** a e t2 **1-n** a. **a** deve contenere le chiavi primarie delle due tabelle lato 1 (t1 e t2), che fungeranno da chiavi esterne verso le rispettive tabelle.
Graficamente:

Implementazione pratica

Detta così è formalmente corretta ma oscura. Vediamo di chiarirlo col nostro esempio. t1 è Person. Viene lasciata così com'è. t2 è License, e vale lo stesso discorso. Dobbiamo creare "a", la tabella associativa, che chiameremo LICENSECERTIFICATION:

```
CREATE TABLE LICENSECERTIFICATION
(
  ID INT PRIMARY KEY AUTO_INCREMENT,
  PERSONID INT,
  LICENSEID INT,
  FOREIGN KEY(PERSONID) REFERENCES PERSON(ID),
  FOREING KEY(LICENSEID) REFERECES LICENSE(ID)
);
```

Esprimere la relazione fra Person e License sotto forma di righe di LicenseCertification

Ricordiamoci che Jill (id=2) ha la patente A e B, James (id=1) solo la B, che la patente A ha id=1 e la B id = 2. Possiamo esprimere queste informazioni in questo modo, nella tabella LicenseCertification:

id	personid	licenseid
1	1	2
2	2	1
3	2	2

Rimettere assieme i pezzi in una relazione n-n

L'id è chiave primaria per la certificazione, come al solito, ma i due id servono ad associare fra loro due righe. La riga 1 di Person (James) è associata solo alla riga 2 di License, mentre la riga 2 di Person (Jill) è associata sia alla riga 1 che alla riga 2 di License. La riga 2 è associata sia a James che a Jill (entrambi "hanno" la patente B). E adesso come rimettere tutto assieme? Abbiamo tre tabelle, quindi due predicati di join (n-1):

```
SELECT PERSON.NAME, PERSON.SURNAME, LICENSE.TYPE FROM PERSON INNER JOIN LICENSECERTIFICATION ON PERSON.ID = LICENSECERTIFICATION.PERSONID INNER JOIN LICENSE ON LICENSECERTIFICATION.LICENSEID = LICENSE.ID;
```

Attributi associativi

La tabella LicenseCertification viene detta associativa perchè il suo scopo è esprimere l'associazione fra altre due tabelle ("entità"). Nasce per dire quali righe devono essere legate, e quali no, ma risponde anche a un'altra necessità.

Appurato il fatto che James e Jill hanno entrambi la patente B, James potrebbe averla presa prima o dopo di Jill. La patente di James potrebbe essere scaduta, e quella di Jill no, o viceversa. Il costo sostenuto per l'esame potrebbe essere diverso fra i due, e da patente in patente. Queste informazioni non riguardano la persona, e non riguardano la patente. Riguardano il loro rapporto, la loro associazione, e infatti vengono detti attributi associativi.

Servono a fornire maggiore dettagli sull'associazione di una riga di t1 a una riga di t2. Miglioriamo la tabella sopra fornendo due attributi associativi:

Introduzione di attributi associativi

```
CREATE TABLE LICENSECERTIFICATION
(
  ID INT PRIMARY KEY AUTO_INCREMENT,
  PERSONID INT,
  LICENSEID INT,
  OBTAINEDON DATE,
  COST INT,
  FOREIGN KEY(PERSONID) REFERENCES
PERSON(ID),
  FOREING KEY(LICENSEID) REFERECES
LICENSE(ID)
);
```

id (pk)	personid (fk)	licenseid (fk)	obtainedon	cost
1	1	2	2020-01-01	2000
2	2	1	2021-01-01	1000
3	2	2	2019-01-01	1500

Interpretazione della tabella associativa

Leggendo i dati noitamo che Jill ha preso prima la patente B (2019) e poi la A (2021). Notiamo anche che abbiamo dovuto specificare due date di ottenimento (una per certificazione), per cui sarebbe stato impossibile mettere questa informazione in un solo campo "licensedate" in Person.

Notiamo anche che James ha avuto bisogno di più soldi (più guide?) per prendere la patente B rispetto a Jill (2000 euro invece di 1500 per la patente con ID=2, vale a dire la B).

Riepilogo dei rapporti n-n

Ricapitolando, e definendo le regole generali, un rapporto n-a-n fra due tabelle t1 e t2 viene **riclassificato** (scomposto) **in due rapporti 1-n fra t1 e una associativa e t2 e la stessa associativa**. t1 e t2 non sono collegati direttamente (infatti t1 non contiene chiavi esterne verso t2, e viceversa), ma indirettamente tramite l'associativa.

Le informazioni che riguardano il rapporto fra le righe di t1 e quelle di t2 sono contenute nell'associativa, e vengono dette **attributi associativi**.

View in MySQL

Alcune query possono diventare complesse e lunghe da scrivere. Molti DBMS, fra cui MySQL, ci offrono la possibilità di salvare le query sotto forma di “view”, vale a dire query (comandi SELECT) salvati che “credono” di essere tabelle (“pseudo tabelle”).

Il problema - riscrivere una query complessa

Riprendiamo l'esempio del rapporto molti a molti. Supponiamo di avere spesso bisogno dei dati delle persone incrociati con quelli delle loro patenti. La query completa potrebbe essere la seguente:

SELECT PERSON.*, LICENSE.TYPE, LICENSECERTIFICATION.OBTAINEDON, LICENSECERTIFICATION.COST FROM FROM PERSON INNER JOIN LICENSECERTIFICATION ON PERSON.ID = LICENSECERTIFICATION.PERSONID INNER JOIN LICENSE ON LICENSECERTIFICATION.LICENSEID = LICENSE.ID

Questa query prenderà tutti i campi di person, il tipo della patente e i dati del suo conseguimento, e li impacchetterà su di una sola riga.

La soluzione - creare una VIEW

Per evitare di riscriverla ogni volta, possiamo fare come segue:

CREATE VIEW VIEWLICENSES AS SELECT PERSON.*, LICENSE.TYPE, LICENSECERTIFICATION.OBTAINEDON, LICENSECERTIFICATION.COST FROM FROM PERSON INNER JOIN LICENSECERTIFICATION ON PERSON.ID = LICENSECERTIFICATION.PERSONID INNER JOIN LICENSE ON LICENSECERTIFICATION.LICENSEID = LICENSE.ID

Dopo di che, per ottenere gli stessi risultati, ci sarà sufficiente scrivere:

SELECT * FROM VIEWLICENSES;

Uso delle View

Viewlicenses è una query salvata che "finge" di essere una tabella. Avrà solo i campi che abbiamo specificato nella sua definizione (il codice "create view" di sopra), ma potremo usarla per le interrogazioni come se fosse una vera tabella:

SELECT * FROM VIEWLICENSES WHERE YEAR(OBTAINEDON) = 2020;

Ci restituirà i dati di tutte le patenti ottenute nel 2020.

Non è insolito avere viste che eseguono i join di tabelle che vengono spesso usate assieme, come in questo caso le tabelle person, licensecertification e license.

Errori comuni nella creazione e nell’uso delle VIEW

Bisogna però fare attenzione alle omonimie. Siccome la vista "crede" di essere una tabella, non possiamo avere due campi con lo stesso nome. Eventuali omonimie in fase di definizione (due campi con lo stesso nome) andranno distinti tramite alias. Notiamo anche che non possiamo agire sui campi che non sono stati specificati esplicitamente in fase di definizione:

SELECT * FROM VIEWLICENSES WHERE LICENSEID = 1;

Non funzionerà, perchè la vista non ha proiettato licenseid dalla tabella licensecertification.

Errori comuni nella creazione e nell’uso delle VIEW (segue)

Dovremo scrivere:

SELECT * FROM VIEWLICENSES WHERE TYPE = 'A';

Ed è type, non license.type. Il riferimento alle tabelle originali si è perso, e viewlicenses è convinta di essere una tabella a sé. Nella pratica, esegue comunque una query sulle tre tabelle, a cui noi applichiamo successivamente dei filtri, dei raggruppamenti o quello che ci servirà