Appunti sui DataBase Relazionali e sul linguaggio SQL

Autore Pietro Suffritti

Versione 1.0

basato in origine sul documento "Introduction to Structured Query Language ver. 4.71" di James Hoffmann, reperibile all' URL HTTP://w3.one.net/~jhoffman/sqltut.htm

Queste pagine nascono a causa della richiesta rivoltami dai membri dell' ERLUG di Modena di tenere un corso introduttivo sull' SQL ANSI (un grazie particolare a quel rompiscatole di Andrea "Andy" Papotti che con le sue battute e richieste mi ha spinto a fare questa robaccia).

Non avendo io la minima esperienza né di SQL ANSI (normalmente ammetto di lavorare con il transact-sql di SQLServer 7.0) né tantomeno di corsi ho ricercato su internet una guida introduttiva all' SQL ANSI su cui basarmi ed ho trovato l'ottima "Introduction to Structured Query Language" di James Hoffmann a cui ho attinto a piene mani per sviluppare questi appunti, ampliandoli con una disgressione su cos'è un database e traducendoli in italiano. Spero di non avere introdotto troppi errori, ma tengo a precisare che il medesimo James Hoffmann mi ha autorizzato via mail a fare tutto ciò. Presto vedrò di tradurre anche pedissequamente il suo documento e metterlo sempre qui, distaccando quindi in seguito questo documento sempre di più dal suo originale, ed aggiungendovi nuove sezioni. In ogni caso riconosco al suddetto quanto di buono può esserci in questo documento mentre mi assumo la colpa di qualsiasi boiata io possa avervi introdotto,e come nel documento originale vi segnalo l' esistenza di un SQL Club nell' area forum di Yahoo rintracciabile a questo link: SQL Club on Yahoo! .

Voglio comunque chiarire che era mia intenzione creare un documento in cui la semplicità di comprensione fosse INCREDIBILMENTE più importante della precisione (di manuali di SQL e' pieno il mondo, se dovete usarlo davvero fate riferimento a quelli), per cui invoco la clemenza della corte per le incredibili semplificazioni che ho qui utilizzato nel tentativo (orrore!) di uscire dal linguaggio tecnico per avvicinarmi a qualcosa che anche mia madre, insegnante di scuola materna dotata di un ottimo cervello ma di preparazione informatica quasi nulla, potesse comprendere. Spero che i tecnici mi perdoneranno e che i non tecnici possano trarne un qualche vantaggio.

Ringrazio Decio Biavati per avermi aiutato traducendo l' ultimo pezzettino residuo del documento di Hoffmann ancora presente in inglese. Colgo l' occasione per dirgli però che se invece che in formato doc l' avesse fatto in html avrei apprezzato ancora di più :-)

Indice

INTRODUZIONE SUI DATABASE

cos'è un database

In tutti i corsi che io ho frequentato la definizione di cosa sia un database è stata o allegramente saltata (dando per scontato che tutti sapessero di cosa si stava parlando) o definita con un criptico linguaggio pseudomatematico per me di difficile comprensione, pertanto tento di esporre qui quello che io ho capito essere un database (con buona pace di professori e puristi dell' informatica)

Al di là dei termini complessi e da iniziati un database (del tipo "un insieme strutturato di dati che in un momento n fornisce una rappresentazione semplificata di una realtà in evoluzione") un database non è altro che una specie di "contenitore" che ci permette di gestire grossi quantitativi di informazioni simili in maniera ordinata e (si spera) più semplice e veloce che con grossi libroni cartacei o documenti di tipo foglio di calcolo o testo.

Piccola storia dei database assolutamente incompleta

Probabilmente il più glorioso (ed a tutt' oggi utilizzato) antenato dei database relazionali odierni può essere identificato con la sana e vecchia agenda telefonica. in effetti penso che se guardiamo come è fatta la nostra agenda scopriremo una notevole affinità con i sui parenti più tecnologici attuali. Infatti è organizzata tramite un indice (la serie di linguette sul fianco che ci permette di accedere più rapidamente a tutti i nominativi che iniziano con una certa lettera) che gestisce una tabella composta da colonne che identificano il tipo di dato sotto riportato (nome, numero di telefono, a volte indirizzo) all' interno delle registrazioni (vogliamo chiamarle con il termine inglese "record"?) che , pur differendo l' una dall' altra per i dati riportati al loro interno "hanno tutti la stessa struttura", cioè riportano le stesse informazioni nella medesima maniera.

Il primo tentativo di cui io sono a conoscenza compiuto dagli informatici per riversare questo tipo di oggetti in un "coso" trattabile dalle loro grosse calcolatrici corrisponde al nome di CSV (Comma Separated Value), cioè un banalissimo file di testo dove ogni informazione (numero di telefono di pino, nome di gino, indirizzo di tino) è separata dalle altre tramite un carattere particolare (normalmente una virgola) ed ogni record (vedi definizione spora riportata, cioè la riga della nostra agenda) è separato dagli altri tramite un' altro carattere (normalmente il carattere di "a capo"). Ovviamente questo sistema era decisamente embrionale, in quanto comunque per trovare all' interno di un file del genere una informazione specifica era spesso necessario scorrerselo quasi tutto ed in modo poco pratico per trovare quanto si ricercava.

la logica evoluzione del CSV fu l'ISAM (Indexed Sequential Access Method), che differiva dal CSV solo per il fatto che i record non erano "buttati dentro a casaccio" (cioè in ordine di inserimento), bensì veniva definito un ordinamento (tipicamente nel caso della nostra agenda l' ordine alfabetico sui cognomi) che veniva sfruttato sia in scrittura ("dove devo mettere il record del telefono di Anna?" "sotto la lettera A tra Anita ed Antonio") sia in lettura ("dove hai infilato il numero di telefono di Pietro?" " lettera P, tra Paolo e Pinocchio") permettendo in questo modo di abbreviare incredibilmente i tempi di ricerca di una data informazione. Per riuscire poi a gestire ancora meglio il tutto si crearono anche delle specie di "archivi sussidiari", detti indici, in cui veniva registrato solo l' ordine dei vari record senza tutte le altre informazioni, il che permetteva di andare a svolgere le proprie ricerche in questo "riassunto" in modo molto più veloce (meno roba da leggere=ci metto di meno a leggerla) e poi "puntare diritti" sul database completo per leggere tutto il record una volta che si sapeva dov'era

A questo punto parecchi matematici di notevole ingegno si misero a cercare metodi per rendere ancora più veloce l' accesso alle informazioni che ci servivano sfornando sistemi di ricerca dai nomi fantasiosi come "ricerca dicotomica" o " a farfalla" (che non è nostro interesse qui approfondire) , sviluppando così tutti quei sistemi oggi definiti "Database non relazionali" (in contrapposizione ai database relazionali che vedremo dopo) di cui forse i più famosi sono stati i database B-Trieve e DBIII-like (clipper, DBIII, DBIV ecc)

con l'evoluzione di questi ultimi i database ebbero una notevole diffusione e quindi iniziarono a nascere richieste di affidabilità e di prestazioni sempre maggiori, con uno sviluppo teorico notevole dietro ad essi, che permetteva a questo punto di affrontare diversi problemi, di cui di seguito elenco quelli più conosciuti (forse da me: sicuramente un teorico dei database potrà aggiungerne altri milioni). sia chiaro, alcuni di questi problemi venivano egregiamente

affrontati dai database non relazionali, ma raramente tutti insieme ed in maniera affidabile

1. La ridondanza dei dati:
   ergo "ma non esiste proprio un modo per evitare di rimettere nel mio database della contabilità duemila volte l' indirizzo del mio cliente principale?"
2. L' uniformità dei dati
   "oddio, come ho chiamato la Johns Coopers and Lybrand Incorporated l' ultima volta? JCL? J.C.L. ? J.C.L Inc.? J.C.& L. Inc?"
3. L'indipendenza dalla piattaforma
   "scusa Aristide, ti ricordi sul tuo sistema come devo fare per vedere il contenuto di una tabella? perché su quello di Asdrubale devo fare così, su quello di Antonio cosà e sul mio in un modo completamente diverso"
4. La sicurezza delle transazioni
   "ARGH! stavo mettendo dentro al database tutte le fatture dell' ultimo semestre quando è andata via la corrente ed adesso non so se l' ultima me l' ha presa o no! cosa faccio, devo ricostruire l' intera tabella delle fatture per essere certo che non ci siano valori doppi o inseriti a metà??"
5. La possibilità di gestire correttamente un ambiente multiutente
   "Mi spieghi perché io sono certo di avere inserito l' ordine per le ultime dieci scatole di pirulazio a nome del mio miglior cliente e quelle sono finite invece al peggior cliente di un' altro commerciale? eppure SO di averle fermate al magazzino io per primo.."

Per risolvere tutti questi problemi in maniera soddisfacente si è dovuto cambiare il modo di "pensare" i database, portando così alla nascita dei database relazionali

I Database Relazionali

allora cosa sono questi famigerati Database relazionali? il concetto che sta alla base è , come spesso succede nell' informatica, molto meno "strampalato" di quello che si pensa: invece di fare un' enorme tabella in cui è contenuto TUTTO il database si divide lo stesso in tante tabelle che contengono dati logicamente correlati e per metterle insieme si usano delle relazioni tra l' una e l' altra tabella. vediamo di fare un qualche esempio per capirci meglio.

Per il nostro esempio prenderemo in esame il database di una piccola gestione di una videoteca di famiglia.Vogliamo tenere registrati tutti i film che abbiamo in modo da sapere su di essi le seguenti informazioni:

1. il titolo
2. il regista
3. l' attore protagonista
4. l' attrice protagonista
5. l' anno di uscita nelle sale del film
6. il tipo di supporto
7. se l' ho in casa o se l' ho prestato a qualcuno
8. a chi l' ho eventualmente prestato: nome, cognome e numero di telefono
9. il genere del film
10. se nostra figlia di 5 anni può vederlo o non è proprio il caso
11. un breve riassunto della trama

nel modo "classico" avremmo realizzato un' unica "tabellona" più o meno di questo tipo:

Videoteca												
cod	titolo	regista	attore	attrice	anno	supporto	PrestNome	PrestCogn	PrestTel	Genere	VM18	trama
1	là dove scorre il fiume	Redford	DeNiro	Stone	85	DVD				Dramma	no	xxx
2	Proposta indecente	Spielberg	Redford	Pfeiffer	91	VHS	Gigi	Salvioli	05808086	sexy	si	yyy

in questo modo abbiamo registrato poche informazioni e di difficile uso. proviamo allora a ragionarci sopra e vediamo che vogliamo in realtà registrare 3 diversi tipi di dati, nella fattispecie:

- i dati della cassetta vera e propria: codice, titolo, supporto, genere, vietata si/no , trama
- i dati di persone che lavorano nei film: regista, attore, attrice
- i dati dei nostri amici a cui potremmo voler prestare la cassetta

a questo punto prendiamo in mano un database relazionale e creiamo non una ma tre tabelle:

Cassette										
CodCass	Titolo	CodReg	CodAtt1	CodAtt2	Anno	Supporto	CodPrest	Genere	VM18	trama
1	là dove scorre il fiume	1	2	3	85	DVD	0	dramma	no	xxx
2	Proposta indecente	4	1	5	85	VHS	1	Sexy	SI	yyy

Personaggi del cinema			
CodPers	Cognome	Nome	VintoOscar
1	Redford	Robert	si
2	DeNiro	Robert	si
3	Stone	Sharon	no
4	Spielberg	Stephen	si
5	Pfeiffer	Michelle	no

Amici				
CodAmico	Cognome	Nome	Telefono	Indirizzo

0	non prestata			
1	Salvioli	Gigi	0547778899	via Garibaldi 5
2	Persiceto	Gennaro	0547558877	via Mazzini 6

in questo modo, oltre ad inserire molti più dati abbiamo già risolto due dei problemi citati sopra: inseriremo una sola volta Robert de Niro per tutte le volte che vorremo usarlo in punti differenti del nostro database ed avremo sempre la certezza di chiamarlo Redford, Robert e non r.redford,robertredford, Robert Redford ecc...

in compenso abbiamo un problema : come faccio a capire che la cassetta di proposta indecente l' ho prestata a Gigi Salvioli e non ad un fantomatico signor 1 ? risposta: tramite delle relazioni

una delle cose che dovrebbe saltare immediatamente all' occhio guardando le tabelle dell' esempio è il fatto che ogni record ha un suo codice univoco (spesso chiamato "chiave primaria") che identifica lui e solo lui (non ci sono né possono essere due personaggi del cinema con CodPers=1) e che tali codici vengono richiamati all' interno della tabella delle cassette: si può quindi dire che ho stabilito una relazione tra i campi della tabella cassette CodReg, CodAtt1 e CodAtt2 e il campo CodPers della tabella personaggi del cinema, così come ho stabilito una relazione tra CodAmico della tabella Amici e CodPrest (risolvendo peraltro il fastidio di avere caselle vuote per indicare che la cassetta è in mio possesso). in questo modo tramite un apposito sistema di definizione delle relazioni si possono creare database estremamente complessi ma di consultazione abbastanza semplice. con una appropriata gestione degli indici sulle tabelle inoltre si può ottenere un tempo di accesso ai dati decisamente soddisfacente

SQL questo sconosciuto

Abbiamo appena finito di dire che uno dei requisiti fondamentali per poter utilizzare un database relazionale e' avere un "linguaggio" che ci permetta di interrogarlo in maniera opportuna, cosa che gli informatici (che complottano tra di loro usando sigle strane per non farsi capire dai non iniziati) chiamano RDBMS, cioè Relational DataBase Management System. come spesso succede nel mondo dell' informatica anche in questo caso e' nato uno "standard" che avrebbe permesso a tutti di parlare la stessa lingua, ed appena e' nato sono nati così tanti dialetti che non parlano l' uno con l'altro da potercisi affogare. Questo standard si chiama ANSI SQL (Structured Query Language della American National Standard Institute), e ovviamente sebbene TUTTI dicano che il loro linguaggio e' ANSI Compliant (compatibile con l' ansi SQL) in realtà tutti hanno differenze sostanziali l' uno con l'altra. i più diffusi linguaggi SQL oggi sono (ovviamente) quelle dei più diffusi database, cioè

- Oracle SQL
- Transact - SQL
- PostgreSQL
- MySQL
- SQLInformix
- DBII SQL
- ecc...

non potendo fare un corso su ognuno di essi (in primis perché molti non li conosco) noi faremo invece riferimento all' ANSI SQL. ma prima di partire con il linguaggio soffermiamoci un' attimo sugli altri punti mancanti delle richieste che abbiamo fatto ai nostri database relazionali: la possibilità di riuscire a recuperare la situazione in caso di guasto e

di lavorare in un sistema multiutente

Commit e Rollback, ovvero come limitare i danni

In tutti i database relazionali minimamente decenti esistono due comandi fondamentali che rispondono per l' appunto al nome di commit e rollback. questi due comandi provvedono a fare in modo che una determinata operazione venga effettuata come un tutto unico o che non venga effettuata per niente. vediamo di capire cosa significa.

Mettiamo che voi stiate gestendo il database di una banca e che dobbiate effettuare un bonifico bancario; questa operazione in realtà si divide in almeno due distinte operazioni:

1. aumentare il saldo del conto corrente del destinatario della cifra in esame
2. diminuire il saldo del conto corrente del mittente della stessa cifra più quella dovuta per le commissioni

ora, cosa succederebbe se a metà dell' operazione il sistema avesse dei problemi (stile corrente che salta, un errore logico come la mancanza del record del destinatario, un processore fuso per il troppo calore,un impiegato impazzito con grosso martellone che picchia sul server)? a seconda dell' ordine in cui le effettuate potete avere i seguenti risultati:

1. i soldi vengono tolti dal cc del mittente e non arrivano in quello del destinatario: risultato il destinatario del bonifico vuole il vostro sangue
2. i soldi NON vengono tolti dal cc del mittente, arrivano in quelli del destinatario e la banca ce li rimette: risultato dovete cercarvi in fretta un'altro lavoro DOPO averli sborsati voi

entrambi i risultati non brillano per piacevolezza, ma per fortuna vi vengono incontro i due comandi suddetti. come funzionano?

Quando si fa qualcosa che deve venire gestito come un tutt' uno si inizia la procedura con una istruzione BEGIN TRANSACTION, che indica il punto di inizio del nostro codice "a rischio" . a questo punto il nostro database non esegue più le variazioni direttamente sul disco ma in una particolare area temporanea (normalmente in memoria) e se la tiene parcheggiata lì finché non sente un comando COMMIT. a quel punto in un' unica soluzione fa TUTTE le variazioni che ha precalcolato in contemporanea. nel caso qualcosa andasse storto l' intera transazione andrebbe persa e quindi il bonifico non verrebbe fatto, sicuramente una situazione meno spiacevole di quella citata prima (anche perché può venire rifatto). E' inoltre possibile forzare questo avvenimento tramite l' istruzione ROLLBACK nel caso che il problema sia di tipo software (esempio ho gli estremi del mittente ma in tabella non esiste il destinatario, quindi sebbene sia già partito a togliere i soldi dal primo non posso metterli sul secondo) . un tipico esempio di gestione commit-rollback può essere la seguente:

```
BEGIN TRANSACTION;                // inizio della transazione

USE banca1;                       // utilizza il database di nome BANCA1

UPDATE conticorrenti              // aggiorna la tabella dei conti correnti

SET saldo = saldo + 1000000       // aggiungendo un milione al saldo

WHERE ccn =  181818               // del conto corrente numero 181818
```

```
ON ERROR ROLLBACK;                  // in caso di un qualunque errore effettua un
rollback ed annulla tutto

USE banca2;                         // utilizza il database di nome BANCA2

UPDATE conticorrenti                // aggiorna la tabella dei conti correnti

SET saldo = saldo - 1000000         // sottraendo un milione al saldo

WHERE ccn =  161616;                // del conto corrente numero 161616

ON ERROR ROLLBACK;                  // in caso di un qualunque errore effettua un
rollback ed annulla tutto

COMMIT;                             // effettua tutta la transizione in una volta
```

attenzione, l' esempio fatto sopra e' un classico esempio di più ISTRUZIONI inserite in un' unica transazione , ma in genere se non specifichiamo nulla ogni singola istruzione viene gestita come una transazione a se stante, pertanto se durante l' esecuzione della nostra update su due milioni di campi crolla il sistema, le procedure di ripristino provvederanno a fare si che NESSUNA modifica della due milioni che stavamo facendo compaia, il che e' incredibilmente preferibile al non sapere su quali record la modifica e' già stata apportata e su quali no

I Semafori , ovvero la gestione della multiutenza

Chiariamo subito una cosa: già linguaggi come Clipper avevano strumenti per gestire la multiutenza sui database, ma in quel caso ancora era completamente demandato al programmatore il fatto di inserire apposite istruzioni per gestire tale eventualità, mentre nei moderni motori database chi gestisce la multiutenza e' il motore stesso (a meno che il programmatore o il DataBase Administrator non vogliano fare giochini strani ed in quel caso se ne prendono la responsabilità). ma come viene gestita questa multiutenza? nella maniera più semplice del mondo: chi primo arriva meglio alloggia. per spiegarsi un po' meglio la cosa funziona in questo modo:

Tutte le volte che noi accediamo ad un determinato record possiamo accedervi in tre diverse modalità: in lettura, in scrittura o in lettura E scrittura. la procedura di accesso ai record SA in che modo noi vogliamo accedere (o perché ha una impostazione di default o perché noi stessi glielo abbiamo detto) e provvede quindi ad imporre un "limite" sul set di record su cui stiamo lavorando che po' essere normalmente uno dei seguenti:

- record libero in lettura e scrittura
- record in sola lettura (divieto di modifica)
- record bloccato sia in lettura che in scrittura

questo tipo di gestione (normalmente definito "a semafori") permette di evitare che due persone tentino contemporaneamente di modificare lo stesso record, permettendo così di evitare i problemi che nascono con la multiutenza

---

Il Linguaggio SQL

Ora che abbiamo visto rapidamente cosa sta alla base di un database relazionale entriamo nel dettaglio del linguaggio SQL, analizzandone i comandi

Basi del comando SELECT

Come abbiamo detto prima nell' introduzione, nei database relazionali le informazioni sono contenute in tabelle. Un esempio di tabella puo' essere rappresentato dalla tabella dei dati degli impiegati riportata qui sotto

TabellaImpiegati					
CFisc	**Nome**	**Cognome**	**Indirizzo**	**Citta**	**Provincia**
ALSNTN60R18F115Z	Alessandro	Antoni	Vicolo Fastidio 23	Sesso	Bologna
FRCBNC58A55F254W	Franca	Bianchi	Via Rua Pioppa 15	Modena	Modena
OSCGRG75F23K242Z	Oscar	Gorgo	Piazza Ugo Bassi 23	Forlì	Forlì Cesena
SMNZNT71D12F251K	Simone	Zanti	Via Radici in piano 115	Sassuolo	Modena

adesso tentiamo di leggere gli indirizzi dei nostri impiegati.Utilizziamo l' istruzione SELECT così:

```
SELECT Nome, Cognome, Indirizzo, Citta, Provincia
FROM TabellaImpiegati;
```

Il risultato della vostra prima QUERY (interrogazione) del database sarà questo:

Nome	**Cognome**	**Indirizzo**	**Citta**	**Provincia**
Alessandro	Antoni	Vicolo Fastidio 23	Sesso	Bologna
Franca	Bianchi	Via Rua Pioppa 15	Modena	Modena
Oscar	Gorgo	Piazza Ugo Bassi 23	Forlì	Forlì Cesena
Simone	Zanti	Via Radici in piano 115	Sassuolo	Modena

Spieghiamo cosa avete appena fatto. Avete appena chiesto al vostro database di farvi vedere tutti i dati contenuti nella tabella TabellaImpiegati, ed in particolare di mostrarvi solo il contenuto delle colonne Nome, Cognome, indirizzo, città e provincia. E' il caso di notare che nei nomi delle colonne e nel nome della tabella non sono presenti spazi, che sono un tipo di carattere non ammesso per specificare i nomi di tali elementi. La sintassi generalizzata di questo comando, per ricevere sulle colonne specificate tutte le righe di una tabella, e' la seguente:

```
SELECT NomeColonna, NomeColonna, ...
FROM NomeTabella;
```

Per ricevere invece tutte le colonne, senza specificarle una ad una, potete usare:

```
SELECT * FROM NomeTabella;
```

Notiamo due cose in particolare da questa prima istruzione:

Ogni comando SQL viene concluso (terminato) da un carattere particolare che e' normalmente il carattere ";". finché SQL non incontra tale carattere continua ad interpretare i caratteri che legge come facenti parti della stesa istruzione. corollario fondamentale di questo fatto e' che possiamo spezzare il comando stesso su più righe allo scopo di ottenere una migliore leggibilità senza particolari problemi (ATTENZIONE: questo carattere VARIA da DBMS a DBMS, CONTROLLATE QUAL'E' il carattere di separazione nel particolare DBMS che state usando).

Selezione Condizionata

Per continuare la discussione sulla selezione, facciamo riferimento a questa nuova tabella di esempio

TabellaStatisticheImpiegati			
CodiceImpiegato	**StipendioAnnuo**	**Benefici**	**Posizione**
010	75000000	15000000	Dirigente
105	65000000	15000000	Dirigente
152	60000000	15000000	Dirigente
215	60000000	12500000	Dirigente
244	50000000	12000000	Impiegato
300	45000000	10000000	Impiegato
335	40000000	10000000	Impiegato
400	32000000	7500000	Apprendista
441	28000000	7500000	Apprendista

Operatori Relazionali

Ci sono sei operatori relazionali in SQL, e dopo averli specificati vedremo come si usano. gli operatori sono:

=	Uguale
<> oppure != (vedere manuali)	Non Uguale
<	Minore di
>	Maggiore di
<=	Minore o uguale di
>=	Maggiore o uguale di

La clausola *WHERE* viene utilizzata per specificare che si desidera vedere solo certe righe della tabella, basandosi per la scelta sul criterio stabilito nella clausola stessa. Penso che un paio di esempi possano rendere più chiaro il funzionamento della stessa.

Se si vuole vedere il Codice Impiegato di quegli impiegati che hanno uno stipendio annuo maggiore di 50 milioni possiamo farlo così:

```
SELECT CodiceImpiegato
FROM TabellaStatisticeImpiegati
WHERE StipendioAnnuo >= 50000000;
```

Notate che abbiamo utilizzato il simbolo >= (Maggiore o uguale di), in quanto vogliamo vedere sia coloro che hanno uno stipendio di 50 milioni che quelli che hanno uno stipendio maggiore di 50 milioni. il risultato sarà il seguente:

```
CodiceImpiegato
------------
010
105
152
215
244
```

la parte del comando *WHERE* che contiene la regola discriminante, in questo caso StipendioAnnuo >= 50000000, e' conosciuta come una *condizione* (un' operazione il cui risultato può essere Vero o Falso). La stessa cosa può essere fatta anche con colonne che contengono testo invece che numeri:

```
SELECT CodiceImpiegato
FROM TabellaStatisticheImpiegati
WHERE POSIZIONE = 'Dirigente';
```

Questo comando mostrerà come risultato il codice impiegato di tutti i dirigenti. Generalmente quando si utilizzano colonne di testo ci si limita agli operatori = e != , e bisogna assicurarsi che tutti i testi utilizzati compaiano nel comando racchiusi tra singoli apici ('). N.B. il singolo apice e' l' identificatore di testo nell' SQL ANSI, ma alcuni DBMS usano altri qualificatori come i doppi apici.

Condizioni più complesse: Condizioni multiple e Operatori Logici

L' operatore *AND* congiunge due o più condizioni, e ritorna tutte e solo le righe che soddisfano TUTTE le condizioni. Per esempio, per mostrare i codici di tutti gli impiegati con uno stipendio superiore ai 40 milioni si utilizza il comando seguente:

```
SELECT CodiceImpiegato
FROM TabellaStatisticheImpiegati
WHERE StipendioAnnuo  > 40000000 AND Posizione = 'Impiegato';
```

L' operatore *OR* collega anch'esso due o più condizioni, ma ritorna una riga se UNA QUALSIASI delle condizioni inserite risulta vera. ad esempio per vedere tutti coloro che hanno uno stipendio annuo inferiore a 40 milioni o ricevono meno di 10 milioni in benefici accessori si utilizza la seguente query:

```
SELECT CodiceImpiegato
```

```
FROM TabellaStatisticheImpiegati
WHERE StipendioAnnuo  < 40000000 OR Benefici < 10000000;
```

AND ed OR possono venire combinati, come nell' esempio seguente:

```
SELECT CodiceImpiegato
FROM TabellaStatisticheImpiegati
WHERE Posizione = 'Dirigente' AND StipendioAnnuo > 60000000 OR Benefici >
12000000;
```

Per prima cosa, SQL trova le righe in cui lo stipendio annuo e' maggiore di 60 milioni e che contengono nella colonna Posizione la parola 'Dirigente', quindi si tiene in memoria questa lista e controlla che soddisfino la condizione OR relativa ad avere benefici per più di 12 milioni. Notate che la condizione AND viene risolta per prima, cosa che modifica nettamente i risultati. spendiamo un paio di parole in più su questo fatto, in quanto se non ben compreso può portare a differenze sostanziali nei risultati delle query ed alla difficile comprensione del perché ciò è avvenuto.

Per generalizzare quello che avviene, SQL calcola i risultati delle operazioni AND per determinare quali sono le righe che soddisfano la condizione (ricordate: TUTTE le condizioni devono essere vere), quindi questi risultati vengono utilizzati per essere confrontati con la condizione OR, e vengono mostrate solo le righe rimanenti dove una qualsiasi delle operazioni collegate dall' operatore OR risulta vera (True). Matematicamente , SQL valuta tutte le condizioni realizzate tramite operatori relazionali, quindi valuta le "coppie" di AND e quindi gli OR (dove entrambi gli operatori sono valutati da sinistra a destra).

Per fare un esempio, guardiamo cosa succede quando il nostro DBMS valuta una determinata condizione, ricordando che i valori booleani "vero" e "falso" in SQL vengono espressi come True e False . il primo passo che il DBMS fa e' di valutare i risultati delle operazioni svolte tramite gli operatori relazionali, quindi si prepara a valutare gli operatori logici sui risultati. mettiamo che dalla valutazione di una condizione complessa al nostro DBMS sia risultato quanto segue:

```
True AND False OR True AND True OR False AND False
```

La prima cosa che fa sarà valutare le coppie di AND, ottenendo quanto segue:

```
True AND False OR True AND True OR False AND False = False OR True OR False
1° coppia          2° Coppia        3°Coppia
```

Quindi parte a valutare gli OR, da sinistra a destra, ottenendo al primo passaggio quanto segue:

```
False OR True OR False = True OR False
1° coppia
```

ed alla fine, valutando l' ultima espressione , arriva al risultato definitivo

```
True OR False = True
```

Il risultato finale e' True (vero) e quindi la riga che ha generato questa sequenza di valori verrà inserita all' interno di quelle passate dalla query. Assicuratevi di leggere attentamente anche la prossima sezione sull' operatore NOT e di

avere ben compreso l' ordine di valutazione degli operatori, in quanto e' un argomento difficile da spiegare in poche righe, pertanto posso solo sperare di essere stato sufficientemente chiaro.

Per variare l' ordine di esecuzione , ad esempio se volete una lista di dipendenti che percepiscono uno stipendio maggiore di 50 milioni o con benefici maggiori di 10 milioni e che siano dei dirigenti, si usano le parentesi, come in questo esempio:

```
SELECT CodiceImpiegato
FROM TabellaStatisticheImpiegati
WHERE Posizione = 'Dirigente' AND (StipendioAnnuo > 50000000 OR Benefici >
10000000);
```

IN , BETWEEN e NOT

Un metodo semplice per usare delle condizioni multiple e' quello di impiegare le due clausole *IN* e *BETWEEN*. Per esempio, se volete vedere tutti i dipendenti che siano dirigenti o impiegati potete fare così:

```
SELECT CodiceImpiegato
FROM TabellaStatisticheImpiegati
WHERE Posizione IN ('Dirigente', 'Impiegato');
```

Oppure per avere la lista di coloro che hanno uno stipendio maggiore o uguale di 30 milioni e minore o uguale a 50 milioni potete impiegare il seguente comando

```
SELECT CodiceImpiegato
FROM TabellaStatisticheImpiegati
WHERE StipendioAnnuo BETWEEN 30000000 AND 50000000;
```

Invece, per mostrare tutti quelli che non sono compresi nell' intervallo, potete usare:

```
SELECT CodiceImpiegato
FROM TabellaStatisticheImpiegati
WHERE StipendioAnnuo NOT BETWEEN 30000000 AND 50000000;
```

Similarmente, la clausola NOT IN restituisce tutte le righe escluse dalla lista generata dalla relativa *IN*, quindi se voleste vedere tutti coloro che non sono ne' dirigenti ne' impiegati potete farlo con il comando

```
SELECT CodiceImpiegato
FROM TabellaStatisticheImpiegati
WHERE Posizione NOT IN ('Dirigente', 'Impiegato');
```

Inoltre, l' operatore NOT può venire utilizzato in congiunzione con AND ed OR per invertirne il risultato, ma va tenuto conto che mentre i due operatori AND ed OR sono binari (usano 2 condizioni) il NOT e' unario, cioè si riferisce ad una sola condizione; inoltre il NOT viene calcolato PRIMA dell' AND e dell' OR .

Ordine in SQL degli operatori logici (tutti funzionano da sinistra a destra)

1. NOT
2. AND
3. OR

Utilizzare la clausola *LIKE ed il carattere jolly %*

Mettiamo di voler estrarre dalla nostra tabella TabellaImpiegati i codici fiscali di tutti gli impiegati il cui nome inizia per "S"; possiamo procedere così:

```
SELECT CFisc
FROM TabellaImpiegati
WHERE NOME LIKE 'A%';
```

Il carattere percento (%) viene utilizzato pre rappresentare ogni possibile carattere (numeri, lettere o segni di interpunzione) o blocchi di caratteri che possono venire trovati dopo il carattere "A". similmente, se volessimo trovare coloro il cui nome termina in "A" potremmo farlo con "%A", o se volessimo trovare quelli il cui nome contiene una A potremmo farlo con "%A%". Visto che il funzionamento dell' operatore Like varia tantissimo da DBMS a DBMS consiglio di verificare la sintassi esatta nel vostro sistema prima di impiegarlo e per vedere quali altre possibilità vi può offrire

---

I Join

In questa sezione discuteremo solo degli *inner joins*, e degli *equijoins*, che , in genere, sono estremamente utili. Per informazioni più dettagliate in merito vi rimando ai link che ho inserito in fondo a questa pagina e che credo possano esservi di un qualche aiuto

Le norme generali sulla buona progettazione dei database suggeriscono che ogni tabella contenga dati relativi solo ad una singola "entità" e che i dati aggiuntivi rispetto ad essa possano venire recuperati tramite relazioni con altre tabelle create tramite i Join. per prima cosa partiamo da un esempio, rappresentato dalle tabelle seguenti:

Antiquari		
IDAntiquario	**CognomeAntiquario**	**NomeAntiquario**
01	Jones	Bill
02	Smith	Bob
15	Lawson	Patricia
21	Akins	Jane
50	Fowler	Sam

Ordini

IDAntiquario	OggettoRicercato
02	Tavolo
02	Scrivania
21	Sedia
15	Specchio

Antichita		
IDVenditore	**IDAcquirente**	**Oggetto**
01	50	Letto
02	15	Tavolo
15	02	Sedia
21	50	Specchio
50	01	Scrivania
01	21	Cassettiera
02	21	Tavolinetto da Caffè
15	50	Sedia
01	15	Portagioielli
02	21	Terracotta
21	02	Libreria
50	01	Piantana

Le Chiavi

Per prima cosa vediamo il concetto di *Chiave*. Una *Chiave Primaria* (o *primary key)* e' una colonna o un gruppo di colonne che identificano in maniera univoca (singola) ogni data riga rispetto alle altre. Tradotto in un linguaggio un po' più semplice, e' quell' insieme di informazioni che mi permettere di distinguere ogni singolo record da ogni altro record. Per esempio, nella tabella Antiquari il campo IDAntiquario distingue in maniera precisa ogni singolo record (non ci sono due record con lo stesso numero nel campo IDAntiquario ne' ci possono logicamente essere). questo significa che non possono esistere due righe della tabella con lo stesso codice in IDAntiquario e che se anche due antiquari avessero lo stesso nome e lo stesso cognome sarebbero comunque differenziato da un codice diverso, permettendoci così di non confonderli tra di loro, e che quindi ci verrà molto più comodo usare il codice rispetto al nome + cognome per collegarci alle altre tabelle del database.

Una *Chiave Esterna* (o *foreign key*) e' invece una colonna presente in una tabella nella quale si registrano dati che sono la chiave primaria di un' altra tabella. per fare un' esempio con il nostro database le colonne IDVenditore e IDAcquirente della tabella antichità sono chiavi esterne in quanto fanno riferimento ai valori contenuti nella chiave primaria della tabella Antiquari, cioè IDAntiquario. In "Informatichese" questa corrispondenza viene chiamata *Integrità Referenziale* (o *referential integrity* per gli inglesofoni), ed impongono il limite che all' interno dei campi della colonna della chiave esterna non possono comparire valori che non siano stati precedentemente inseriti nella colonna della chiave primaria relativa, altrimenti ci si troverebbe di fronte ad un errore chiamato "*Violazione dell' integrità referenziale*" che renderebbe il database *inconsistente*, cioè in pratica inutilizzabile. In pratica, come risulta

chiaro dall' esempio, viene utilizzata la chiave primaria di una certa tabella per fare riferimento ai dati contenuti nel record relativo senza dovere riportarli nella seconda tabella. nel nostro esempio infatti viene utilizzato il campo "IDAntiquario" per sapere chi ha comprato e chi ha venduto un determinato oggetto nella tabella Antichita, senza dovere riportare il nome ed il cognome sia dell' acquirente che del venditore.

Creare un Join

Lo scopo delle Chiavi appena discusse e' quella di correlare i dati attraverso le tabelle, senza dover in questo modo ripetere su tutte le tabelle i dati più frequenti, il che e' il vero scopo e la sostanziale differenza tra i database relazionali ed altri tipi di database. per esempio possiamo trovare i nomi di coloro che hanno comprato una sedia senza dover mettere il nome ed il cognome dell' acquirente nella tabella antichità. possiamo infatti ottenerne il nome mettendo in relazione coloro che hanno comprato una sedia con la lista di tutti i nomi degli antiquari della tabella Antiquari attraverso l' utilizzo dell' IDAcquirente, che mette in relazione i dati nelle due tabelle. per trovare i nomi di chi ha comprato una sedia possiamo , in pratica, utilizzare la seguente query:

```
SELECT CognomeAntiquario, NomeAntiquario
FROM Antiquari, Antichita
WHERE IdAcquirente = IDAntiquario AND Oggetto = 'Sedia';
```

Iniziamo ad analizzare le particolarità di questa query. la prima che balza all' occhio subito e' che ENTRAMBE le tabelle sono coinvolte nella query e citate di fianco alla clausola FROM. la seconda cosa e' che nella clausola WHERE le due chiavi delle due tabelle sono messe in relazione tramite l' uso della condizione `IdAcquirente = IDAntiquario` e quindi ristretto tramite la condizione `Oggetto = 'Sedia'` a coloro che hanno acquistato una sedia. La condizione che mette in collegamento le due tabelle (IDAcquirente=IDAntiquario) e' il nostro *Join* (in italiano connessione). Visto che la condizione che crea il join e' un uguale, questo tipo di join e' chiamato un *equijoin*. Il risultato di questa query saranno due nomi: Smith, Bob e Fowler, Sam.

La *notazione a punti* (*Dot notation*) permette di specificare il nome della tabella a cui si riferisce il campo , evitando così possibili ambiguità. pur essendo più "faticosa" e' nettamente preferibile visto che migliora notevolmente la leggibilità della query e riduce la possibilità di errori. la nostra query precedente in dot notation sarebbe così:

```
SELECT Antiquari.CognomeAntiquario, Antiquari.NomeAntiquario
FROM Antiquari, Antichita
WHERE Antichita.IdAcquirente = Antiquari.IDAntiquario AND Antichita.Oggetto =
'Sedia';
```

Questa notazione non sarebbe (e non e') necessaria se i nomi di tutti i campi di ogni tabella differissero l' uno dall' altro, ma e' comunque un' ottima abitudine da prendere e diventa fondamentale appena la struttura del database cresce a sufficienza da non essere più verificabile a colpo d'occhio

***Clausola DISTINCT* e l'Eliminazione dei Duplicati**

Prendiamo in esame la necessità, nel nostro database di esempio, di avere una lista dei codici e dei nomi esclusivamente degli antiquari che hanno venduto un oggetto. Ovviamente si vorrà che nella lista risultante ogni venditore sia elencato una sola volta, indipendentemente dal numero di oggetti che ha venduto - non vogliamo sapere quanti oggetti ha venduto un antiquario, ma solo che abbia venduto qualcosa (per poterli conteggiare invece fare riferimento alla successiva sezione sulle funzioni di aggregazione). Questo significa che dobbiamo dire ad SQL di eliminare le righe duplicate delle vendite dello stesso antiquario, e quindi elencare ognuno di questi una volta sola. Per

ottenere questo scopo utilizzeremo la clausola *DISTINCT*.

Per prima cosa ci serve un equijoin (come e' stato spiegato subito sopra) alla tabella degli Antiquari per potere ottenere i dati particolareggiati di ogni antiquario, cioè cognome e nome. Ovviamente in merito va mantenuta ben in mente quella che prima e' stata definita l' Integrità referenziale, cioè nel caso in cui non esistesse un record relativo al codice presente nel campo IDVenditore all' interno del campo IDAntiquario nella tabella Antiquari il minimo che può accadere è che il record non venga mostrato, mentre addirittura su alcuni DBMS si verificherebbe un errore di integrità referenziale violata bloccando l' esecuzione della query. come abbiamo detto inoltre vogliamo eliminare le occorrenze multiple dei record nella nostra lista, quindi impieghiamo la clausola Distinct sulle colonne in cui potrebbe verificarsi l' occorrenza multipla stessa

Per aggiungere un' ulteriore livello di complessità della query , vogliamo i risultati in ordine alfabetico prima per cognome e poi per nome. per fare questo possiamo utilizzare la clausola *ORDER BY* . La nostra query finale sarà la seguente:

```
SELECT DISTINCT IDVenditore, CognomeAntiquario, NomeAntiquario
FROM Antichita, Antiquari
WHERE IDVenditore = IDAntiquario
ORDER BY CognomeAntiquario, NomeAntiquario;
```

In questo esempio particolare, visto che tutti hanno venduto almeno un oggetto, avremo una lista completa di tutti gli antiquari, in ordine alfabetico per cognome e nome. Per uso futuro (cioè se un qualche sadico ve lo viene a chiedere) tenete presente che questo tipo di Join è da considerarsi nella categoria degli *inner joins.*

Alias, In e l' utilizzo delle Subquery

In questa sezione parleremo dell' utilizzo degli *Alias*, dell' *In* , dell' utilizzo delle subquery, e come queste possono venire utilizzate nel nostro esempio di 3 tabelle. Per prima cosa diamo un' occhiata alla query riportata qui sotto, che ritorna i cognomi di quegli antiquari che hanno fatto un' ordine e che tipo di oggetto hanno ordinato, mostrando solo gli ordini in essere (se notate infatti esiste un' antiquario che ha ordinato un' oggetto di cui e' già in possesso):

```
SELECT PERS.CognomeAntiquario Cognome, ORD.OggettoRicercato Oggetto Ordinato
FROM Ordini ORD, Antiquari PERS
WHERE ORD.IDAntiquario = PERS.IDAntiquario
AND ORD.OggettoRicercato IN

     (SELECT Oggetto
     FROM Antichita);
```

Questa query ritorna questo risultato:

```
Cognome Oggetto Ordinato
------- ----------------
Smith   Tavolo
Smith   Scrivania
Akins   Sedia
```

```
Lawson  Specchio
```

C'e' veramente un mucchio di cose da vedere in questa query. analizziamola nello specifico :

1. Per prima cosa, nella linea della clausola SELECT compaiono le parole "Cognome" e "Oggetto Ordinato" che forniscono l' intestazione delle colonne del report ritornato dalla query
2. Le parole PERS ed ORD sono *alias*; gli alias sono dei nomi alternativi per le due tabelle che compaiono nella clausola FROM e che vengono utilizzati come prefissi in tutta la dot notation delle colonne utilizzate nella query, in modo da eliminare le ambiguità , in particolar modo nella clausola WHERE dell' equijoin , dove entrambi le colonne si chiamano IDAntiquario e quindi la dot notation si renda indispensabile per evitare confusione tra i due
3. Notate che la tabella Ordini e' quella che compare per prima nella clausola FROM; questo ci permette di essere sicuri che il report verrà tratto dai dati presenti in questa tabella e che la tabella Antiquari viene utilizzata solo per recuperare le informazioni dettagliate (i cognomi)
4. La cosa più importante; l' AND nella clausola WHERE forza a richiamare la subquery contenuta nella clausola IN ("= ANY" o "= SOME" sono due utilizzi equivalenti ad IN). Questo e' quello che fa: la subquery contenuta tra le parentesi viene eseguita, ritornando l' elenco di tutti gli oggetti contenuti nella tabella Antichita non essendoci nessuna clausola WHERE nella subquery; dopodiché, perché una riga nella tabella degli ordini possa venire inserita nel report finale, il valore del campo OggettoRicercato deve essere all' interno della lista degli oggetti restituita dalla subquery. Per chiarire meglio la cosa si può vederla in questo modo: la subquery ritorna un set di oggetti con i quali vengono confrontati gli oggetti presenti nella colonna OggettoRicercato e la condizione di inclusione diventa vera solo se il valore del campo OggettoRicercato è presente nella lista ritornata dalla subquery
5. Inoltre tenete ben presente che quando usate le clausole IN, "= ANY", o "= SOME" NON potete fare riferimento a più colonne diverse. sono i valori che vengono restituiti a potere essere differenti , e il confronto può venire fatto tra valori differenti contenuti in ogni RIGA ma NON in colonne differenti. il viceversa non e' possibile.

Ok, penso che abbiamo discusso in maniera ampia e dettagliata del comando SELECT e penso che dovreste essere a questo punto in grado di arrivare a fare query di selezione decisamente complesse, per cui passiamo agli altri comandi di SQL

Comandi SQL Vari

Funzioni di Aggregazione

In questa sezione introdurrò le cinque più importanti *Funzioni di Aggregazione*: SUM, AVG, MAX, MIN, e COUNT. Queste funzioni vengono chiamate Di Aggregazione perché servono a calcolare valori dai risultati delle query invece che a ottenerne un listato di tutte le righe. Vediamo il significato di ogni funzione:

- SUM () restituisce il totale di tutte le righe, che soddisfano ogni condizione, della colonna data, quando tale colonna contiene valori numerici
- AVG () restituisce la media della colonna data
- MAX () restituisce il valore più elevato della colonna data
- MIN () restituisce il valore più piccolo della colonna data
- COUNT(*) restituisce il numero di righe che soddisfano le condizioni

proviamo a fare un po' di esempi con le tabelle degli impiegati utilizzate all' inizio di questo documento. guardiamo questi tre esempi:

```
SELECT SUM(StipendioAnnuo), AVG(StipendioAnnuo)
FROM TabellaStatisticheImpiegati;
```

Questa query ritorna la somma totale di tutti gli stipendi annui presenti nella tabella e la media degli stessi

```
SELECT MIN(Benefici)
FROM TabellaStatisticheImpiegati
WHERE Posizione = 'Dirigente';
```

Questa query ritorna il valore del piu' basso importo annuo preso da un dirigente come beneficio accessorio (12 milioni).

```
SELECT COUNT(*)
FROM TabellaStatisticheImpiegati
WHERE POSITION = 'Impiegato';
```

Questa query risponde dandoci il numero di persone con la qualifica di impiegato (3).

Viste

In SQL si può normalmente avere la possibilità di crearsi delle proprie viste, cioè creare delle query "perenni" che possono venire utilizzate come delle vere e proprie tabelle. normalmente questa funzione può venire usata da ogni singolo utente, che sarà colui che vedrà la vista se non specificato diversamente dal DBA (controllate il vostro DBMS e sentite dal vostro DB Administrator se gli utenti hanno questo diritto). per essere più precisi quello che fa una vista e' permettervi di assegnare i risultati di una query ad una nuova tabella "personale" (cioè legata al vostro utente), che potete usare all' interno di altre query , ed il cui nome potete tranquillamente specificare nella clausola FROM come se fosse una qualunque tabella. Generalmente le viste sono dinamiche, cioè i valori nella tabella vengono rigenerati tutte le volte che accedete alla vista stessa. un esempio di creazione di una vista può essere il seguente:

```
CREATE VIEW VistaAnt AS SELECT OggettoRichiesto FROM Ordini;
```

A questo punto possiamo scrivere una query che usa questa vista come una tabella; tabella che in realtà sarà solo un' elenco di tutti gli oggetti richiesti proveniente dalla tabella Ordini:

```
SELECT IDVenditore
FROM Antichita, VistaAnt
WHERE OggettoRichiesto = Oggetto;
```

Questa query mostra il codice del venditore estratto dalla tabella antichità quando il campo OggettoRichiesto in quella tabella compare nella vista VistaAnt, che e' semplicemente la lista di tutti gli oggetti richiesti nella tabella degli ordini, eseguita al momento del lancio della query.Le viste possono essere utilizzate sia allo scopo di restringere la visibilità dei dati a determinati utenti che per semplificare delle query complesse.

Creare Nuove Tabelle

Tutte le tabelle all' interno di un database devono venire create prima o poi. vediamo come possiamo fare. a titolo di esempio prendiamo in esame la creazione della tabella Ordini:

```
CREATE TABLE Ordini
(IDAntiquario    INTEGER  NOT NULL,
OggettoRicercato CHAR(40) NOT NULL);
```

Questo comando dice al DBMS di creare una tabella, gli assegna un nome e definisce ogni colonna nella stessa. ATTENZIONE: notate che questo comando utilizza dei tipi di dato generici, e che i tipi di dato possono variare da DBMS a DBMS, quindi verificate i tipi di dato disponibili sul vostro. Alcuni tipi di dato generalmente disponibili su quasi tutti i DBMS sono:

- Char(x) - una serie di caratteri alfanumerici (generalmente la tabella ASCII o EBCDIC) , in cui x specifica il numero massimo di caratteri permessi nella colonna (spesso conosciuta anche come stringa di caratteri di lunghezza x).
- Integer - una colonna di numeri interi, positivi o negativi, i cui massimi dipendono da DBMS a DBMS.
- Decimal(x, y) - Una colonna di numeri decimali, in cui x esprime il numero massimo di cifre permesso e y il numero di cifre dopo la virgola. per esempio il numero più alto esprimibile in un campo definito Decimal(4,2) e' 99,99.
- Date - Una colonna contenente delle date nel formato tipico specifico del proprio DBMS.
- Logical - Una colonna che può contenere solo uno dei due seguenti valori: TRUE (vero) o FALSE (falso).

Altra cosa da notare e' l' impiego della clausola NOT NULL. Questa clausola impone che il relativo campo non possa mai venire lasciato vuoto (deve sempre essere presente un valore per questo campo). Nel caso si voglia invece permettere l' inserimento di record con quel determinato campo vuoto si usa la clausola NULL o, in molti DBMS, se non viene specificato nulla viene associata per default la clausola NULL.

Modificare la struttura delle tabelle

Sempre con la logica di vedere i comandi all' interno di esempi pratici, vediamo come si puo' aggiungere una nuova colonna "Prezzo" alla tabella antichita per permettere l' inserimento del prezzo di ogni articolo. vista la prossima introduzione dell' euro inseriamo invece che un valore Integer un valore Decimal di dimensioni tali da poterci permettere di esprimere il prezzo sia in lire che in euro:

```
ALTER TABLE Antichita ADD (Prezzo DECIMAL(12,2) NULL);
```

l' ovvio complemento della clausola ADD qui utilizzata per aggiungere una colonna e' la clausola REMOVE che permette di eliminare una colonna specificata. i dati in questa nuova colonna possono essere modificati o inseriti come specificato poco oltre.

Inserire dati in una tabella

Per inserire record (righe) in una tabella si usa il seguente comando (esempio riferito alla tabella Antichita a cui e'

appena stato aggiunto il prezzo) :

```
INSERT INTO Antichita VALUES (21, 01, 'Ottomana', 2000000.00);
```

Questo comando inserisce una nuova riga nella tabella, nell' ordine dei campi predefinito e riempiendo tutti i campi. per poter variare l' ordine di inserimento dei campi e lasciare in bianco il campo Prezzo (che abbiamo precedentemente definito NULL e che quindi accetta valori nulli) si può fare così:

```
INSERT INTO Antichita (IDAcquirente, IDVenditore, Oggetto)
VALUES (01, 21, 'Ottomana');
```

Cancellare dati da una tabella

Adesso proviamo a cancellare la riga di dati che abbiamo appena inserito dalla tabella:

```
DELETE FROM Antichita
WHERE Oggetto = 'Ottomana';
```

Ma, in questo caso, se c'e' un' altra riga che contiene nel campo Oggetto la parla "Ottomana" verrà cancellata anch'essa. per cui siamo un po' più specifici e cancelliamo la riga specificando tutti i dati che abbiamo appena inserito:

```
DELETE FROM Antichita
WHERE Oggetto = 'Ottomana' AND IDAcquirente = 01 AND IDVenditore = 21;
```

Modifica dei dati

Proviamo ad aggiornare il valore di un prezzo in un record dove questo non e' stato inserito. ricordiamoci che in questo caso non si tratta di un inserimento ma di una modifica del valore NULL precedentemente impostato in quel campo, in quanto il record e' già presente nella tabella:

```
UPDATE Antichita SET Prezzo = 500000.00 WHERE Oggetto = 'Sedia';
```

Questo comando imposta il prezzo di tutte le sedie a 500.000£ . anche in questo caso possiamo fare riferimento alla sintassi della clausola WHERE del comando SELECT per essere più o meno specifici nella scelta dei record da modificare. inoltre, esattamente come nel comando SELECT, si possono modificare più colonne con la stessa sintassi usata in questo comando per indicare più colonne (elenco diviso da virgole dopo la clausola SET) .

Gli Indici

Gli indici permettono ad un DBMS di accedere ai dati più rapidamente (nota bene: questa funzionalità è non-standard/ non disponibile su tutti i sistemi).Il sistema crea della strutture dati interne (gli indici) per la selezione più veloce di determinate righe quando la selezione è basata su colonne indicizzate. La struttura indica al DBMS dove si trova una certa riga su una tabella con colonne indicizzate, più o meno come il glossario di un libro indica la pagina in cui

appare una determinata parola. Creiamo un indice per IDAntiquario nella tabella ANTIQUARI

```
CREATE INDEX OID_IDX ON ANTIQUARI (IDAntiquari);
```

Ora un indice sui nomi:

```
CREATE INDEX NAME_IDX ON ANTIQUARI (CognomeAntiquario, NomeAntiquario);
```

Per eliminare un indice si usa l'istruzione DROP:

```
DROP INDEX OID_IDX;
```

In più, si può usare l'istruzione DROP TABLE per eliminare una tabella (attenzione! Questo significa che la tua tabella viene cancellata). Nel secondo esempio, l'indice è relativo a due colonne, aggregate insieme; in questo caso, possono accadere strani comportamenti… consultare il manuale prima di eseguire questa operazione.

Alcuni DBMS non richiedono l'unicità della chiave primaria. In altre parole, se cercassi di inserire un'altra riga nella tabella ANTIQUARI con un IDAntiquario = 2, alcuni sistemi mi lascerebbero fare quest'operazione, anche se non è consigliabile, in quanto si suppone che il valore di questa colonna sia unico per ciascuna riga. Un modo per ovviare a possibili inconvenienti di questo genere è di creare un indice unico sulla colonna che noi vogliamo sia una chiave primaria, per forzare il sistema ad impedire la duplicazione di valori in quella colonna:

```
CREATE UNIQUE INDEX OID_IDX ON ANTIQUARI (IDAntiquario);
```

GROUP BY ed HAVING

Un uso speciale dell'istruzione GROUP BY è di associare una funzione di aggregazione (nella fattispecie COUNT; conta il numero di righe in ciascun gruppo) con dei gruppi di righe. Si assuma che la tabella ANTICHITA abbia la colonna PREZZO e che ciascuna riga abbia un valore inserito in questa colonna. Quello che ci interessa è di vedere il prezzo dell'oggetto più costoso comprato da ciascun acquirente. Si deve impartire via SQL l'ordine di *raggruppare* gli acquisti di ciascun acquirente, e di stampare il prezzo dell'acquisto più costoso:

```
SELECT IDAcquirente, MAX(Prezzo)
FROM ANTICHITA
GROUP BY IDAcquirente;
```

Ora, aggiungiamo l'istruzione di stampare il prezzo massimo di acquisto se supera i 100 dollari; in questo caso si usa l'istruzione HAVING:

```
SELECT IDAcquirente, MAX(Prezzo)
FROM ANTICHITA
GROUP BY IDAcquirente
HAVING Prezzo > 100;
```

Altre Subquery

Un altro uso comune delle Queries coinvolge l'uso di operatori per permettere ad una condizione di WHERE di includere l'output di una SELECT di una Query annidata. Dapprima, si mostri la lista degli acquirenti che hanno comprato un oggetto dispendioso (ovvero con il prezzo dell'articolo che superi di 100 dollari il prezzo medio di tutti gli oggetti comprati):

```
SELECT IDAcquirente
FROM ANTICHITA
WHERE Prezzo >

    (SELECT AVG(Prezzo) + 100
        FROM ANTICHITA);
```

La query annidata fra parentesi calcola il prezzo medio, somma 100, ed usando il risultato, si stampa un IDAcquirente per ciascun oggetto venduto di prezzo superiore. Si può usare DISTINCT IDAcquirente per visualizzare una sola riga per ciascun IDAcquirente selezionato.

Stampa i Cognomi della tabella ANTIQUARI, solo se hanno comprato un oggetto:

```
SELECT COGNOMEANTIQUARIO
FROM ANTIQUARI
WHERE IDAntiquario IN

      (SELECT DISTINCT IDAcquirente
          FROM ANTICHITA);
```

La query annidata (o sottoquery) seleziona una lista di acquirenti, e stampa il cognome di un Antiquario se e solo se il suo IDAntiquario appare nella lista selezionata dalla sottoquery (talvolta chiamata *lista dei candidati*). *Nota:* alcuni DBMS permettono di usare il segno uguale ('=') al posto di IN; per chiarezza, però, si preferisce l'uso di IN, in quanto la sottoquery non ritorna un valore singolo, bensì una lista.

Per un esempio relativo all' UPDATE, sappiamo che il gentiluomo che ha acquistato la libreria ha il Nome sbagliato nel database, e che il nome corretto è Giovanni:

```
UPDATE ANTIQUARI
SET NOMEANTIQUARIO = 'Giovanni'
WHERE IDAntiquario =

    (SELECT IDAcquirente
        FROM ANTICHITA
        WHERE OGGETTO = 'Libreria');
```

In primis, la sottoquery cerca l'IDAcquirente per la persona o le persone che hanno acquistato la libreria, *in secundis* la query esterna aggiorna il Nome.

Regola delle sottoquery: quando si ha una sottoquery come parte di una condizione di WHERE, l'istruzione di SELECT deve avere un numero di colonne pari in numero e tipo a quelle contenute nella condizione di WHERE della

query esterna. In altre parole, se si ha "`WHERE NomeColonna = (SELECT….);`" l'istruzione di Select deve avere solo una colonna, per trovare una corrispondenza con il NomeColonna della condizione di Where esterna; in più, il *tipo* deve essere il medesimo (due interi, due stringhe di caratteri, etc..).

EXISTS ed ALL

EXISTS usa una sottoquery come condizione: la condizione è Vera se la sottoquery ritorna almeno una riga, ed è Falsa se la sottoquery non ritorna nessuna riga. Questa è una funzionalità non intuitiva con quest'uso specifico. Comunque, se si vuole stampare un riassuntivo clienti solo se il negozio ha venduto Sedie, si può scrivere:

```
SELECT NomeAcquirente, CognomeAcquirente
FROM ANTIQUARI
WHERE EXISTS

    (SELECT *
        FROM ANTICHITA
        WHERE Oggetto = 'Sedia');
```

Se nella colonna della tabella ANTICHITA c'è almeno una riga contenente l'articolo sedia, la sottoquery ritorna una o più righe, rendendo Vera la condizione EXISTS; di conseguenza la query stampa gli Antiquari. Se non ci sono righe con l'oggetto Sedia, la query esterna non stamperà alcuna riga.

ALL è un'altra funzionalità inusuale, in quanto la query che contengono l'istruzione ALL possono essere eseguite in altri modi, a volte anche più semplici; diamo un'occhiata alla query di esempio:

```
SELECT IDAcquirente, Oggetto
FROM ANTICHITA
WHERE Prezzo >= ALL

    (SELECT Prezzo
        FROM ANTICHITA);
```

Questa query ritorna l'oggetto con il prezzo più alto (o più di un oggetto se hanno un prezzo uguale e massimo), e l'Acquirente. La sottoquery ritorna una lista di tutti i prezzi nella tabella ANTICHITA, mentre la query esterna passa tutte le righe della tabella ANTICHITA, e se il prezzo è maggiore o uguale di qualunque altro oggetto (ALL = tutti), viene stampato, risultando così l'oggetto con il prezzo più alto. La ragione per cui viene usato il segno di "maggiore o uguale" è che il prezzo dell'oggetto più costoso sarà uguale al prezzo più alto della tabella.

UNION ed Outer Joins (rapida spiegazione non esaustiva)

Si possono presentare casi in cui si voglia vedere il risultato di diverse query insieme, combinando il loro output; in questi casi si usa l'espressione UNION. Per unire l'output delle due query seguenti, stampando tutti gli IDAcquirente, insieme con tutti quelli che hanno mandato un ordine, si esegua:

```
SELECT IDAcquirente
```

```
FROM ANTICHITA
UNION
SELECT IDAntiquario
FROM ORDINI;
```

Si noti che SQL richiede che la lista di colonne di entrambe le Select siano uguali per tipo, colonna per colonna. In questo caso IDAcquirente ed IDAntiquario sono dello stesso tipo (Interi [integer]). Si noti inoltre che SQL elimina automaticamente le righe duplicate usando UNION (come fossero due set di dati); per le query singole si deve usare DISTINCT.

L'*outer join*, (join esterno) si usa quando una join fra tabelle viene "unita" con le righe non incluse nella condizione di join; questo risulta particolarmente utile se nella query sono contenute delle costanti di tipo testo. Per esempio:

```
SELECT IDAntiquario, 'presente in Ordini ed Antichita'
FROM ORDINI, ANTICHITA
WHERE IDAntiquario = IDAcquirente
UNION
SELECT IDAcquirente, 'presente solo in Antichita'
FROM ANTICHITA
WHERE IDAcquirente NOT IN
   (SELECT IDAntiquario
        FROM ORDINI);
```

La prima query fa una join per stampare tutti gli Antiquari in entrambe le tabelle, e per aggiungere un breve commento dopo ciascun ID. La UNION combina questa lista con la successiva, che viene generata anzi tutto richiamando gli ID non presenti nella tabella ORDINI, infine stampando gli ID esclusi dalla condizione di join. Quindi, ciascuna riga nella tabella ANTICHITA viene esaminata, e se l'IDAcquirente non è nella lista di esclusione, viene stampato con il commento fra apici. Ci possono essere modi più semplici per generare la stessa lista di ID, ma attaccare a ciascuna riga un commento fisso appropriato è ben più complesso se fatto in altri modi.

Questo concetto è utile in situazioni in cui una chiave primaria viene messa in condizione di join con una chiave secondaria, ma la chiave secondaria può essere NULL per alcune chiavi primarie. Ad esempio, in una tabella la chiave primaria è il venditore e in un'altra è il cliente, con il venditore riportato nella stessa riga. Se un venditore non ha clienti, il suo nome non appare nella tabella dei clienti. L'**outer join** si usa allora per ottenere la lista di **tutti** i venditori, insieme con i loro clienti, sia nel caso in cui il venditore abbia clienti sia nel caso in cui non ne abbia, cioè non venga stampato il nome del cliente (un valore di NULL logico) se il venditore non ha clienti, ma è nella tabella dei venditori. In caso contrario, il venditore sarà stampato insieme con ognuno dei suoi clienti.

Un altro punto importante riguardo ai NULL nelle condizioni di join: l'ordine delle tabelle nella lista successiva all'istruzione FROM è molto importante. La regola è che SQL aggiunge la seconda tabella alla prima; se la prima tabella presenta solo righe contenenti NULL nella colonna di join, e la seconda tabella ha una riga con NULL nella colonna di join, tale riga non risulta nella condizione di join, e pertanto deve venire inclusa con i dati di riga della prima tabella. Questa è un altro esempio in cui comunemente viene impiegato un outer join. Il concetto di NULL è importante, e può essere proficuo investigarlo in maniera più approfondita.

BASTA QUERY!!! O non ne avete ancora avuto abbastanza? … adesso si passa a qualcosa del tutto differente…

Sommario della Sintassi - Solo per veri masochisti

Qui potete trovare le forme generalizzate dei comandi discussi in questi appunti, più alcuni aggiuntivi che possono risultare comodi e la cui spiegazione e' data a latere. **ATTENZIONE!!!** non e' detto che tutti questi comandi siano esattamente in questa forma, controllate sul vostro sistema per avere la certezza del loro funzionamento e della loro disponibilità:

ALTER TABLE `<Nome Tabella> ADD|DROP|MODIFY (Specifiche Colonna[e]...vedere Create Table);` --vi permette di aggiungere o cancellare una o più colonne da una tabella, o di cambiare i parametri di una colonna esistente (tipi di dato ecc..); questo comando e' utilizzato spesso anche per cambiare le specifiche fisiche di una tabella (dove e come viene salvata ecc.), ma in questo caso dipende direttamente dal DBMS che state usando, quindi vi rimando ai manuali del vostro database. Oltre che con questo comando le specifiche fisiche della tabella possono venire generalmente date all' interno del comando Create Table quando una tabella viene creata per la prima volta.

Begin Transaction; inizia a considerare i comandi seguenti come facenti parte di un' unico blocco (transazione) che vanno eseguiti in un blocco unico al raggiungimento di una istruzione di commit o scartati tutti insieme al raggiungimento di una istruzione rollback

COMMIT; effettua i cambiamenti fatti sul database dall' ultima istruzione Begin Transaction (in alcuni DBMS dall' ultima commit) e li rende permanenti -- questo blocco di istruzioni viene definito una Transazione

CREATE [UNIQUE] INDEX `<Nome Indice>`
`ON <Nome Tabella> (<Lista Colonne>);` -- UNIQUE e' opzionale e va usato senza parentesi quadre.

CREATE TABLE `<Nome Tabella>`
`(<Nome Colonna> <Tipo di Dato> [(<Dimensione>)] <Limitazioni sulle colonne>, ...altre colonne[PRIMARY KEY (colonna, colonna,...)]);` (sintassi valida anche per ALTER TABLE)
--dove Dimensione viene usato solo su alcuni tipi di dato, e le limitazioni includono quelle qui sotto riportate (controllate automaticamente dal dbms. una tentata violazione causa la generazione di un errore):

1. NULL o NOT NULL (vedi sotto)
2. UNIQUE obbliga a non avere due valori uguali all' interno della colonna
3. PRIMARY KEY dice al database che questa colonna e' la chiave primaria della tabella (utilizzato solo se la chiave primaria e' composta da una sola colonna, altrimenti una clausola PRIMARY KEY (colonna, colonna, ...) comparirà dopo l'ultima definizione di colonna.
4. CHECK permette ad una condizione di essere testata quando i dati vengono inseriti o aggiornati in una determinata colonna; per esempio, CHECK (Prezzo >= 0) impone al sistema di controllare che il prezzo sia maggiore o uguale a zero prima di accettare il valore; a volte viene implementato con l'istruzione CONSTRAINT.
5. DEFAULT inserisce il valore nel database per una determinata colonna se una riga viene inserita senza un valore per la colonna stessa; per esempio, `BENEFITS INTEGER DEFAULT = 10000`
6. FOREIGN KEY funziona esattamente come l'istruzione Primary Key, ma è seguita da: `REFERENCES <Nome Tabella> (<Nome Colonna>)`, che riferisce la chiave riferita ad una chiave primaria.

CREATE VIEW `<Nome Tabella> AS <Query>;`

```
DELETE FROM <Nome Tabella> WHERE <Condizione>;

INSERT INTO <Nome Tabella> [(<Lista Colonne>)]
VALUES (<Lista Valori>);
```

ROLLBACK; -- Annulla le modifiche effettuare al database che sono state effettuate dopo l' ultimo comando COMMIT. ATTENZIONE! vedere il funzionamento delle istruzioni COMMIT, ROLLBACK e BEGIN TRANSACTION sui propri RDBMS particolari in quanto il funzionamento di tali comandi varia moltissimo da un sistema all' altro

```
SELECT [DISTINCT|ALL] <Lista Colonne, Funzioni, Costanti, ecc.>
FROM <Lista di tabelle o di viste>
[WHERE <Condizione/i>]
[GROUP BY <raggruppamento colonna/e>]
[HAVING <condizione>]
[ORDER BY <colonna/e di ordinamento> [ASC|DESC]]; --dove ASC|DESC permettono di fare in modo che l' ordinamento venga effettuato in ordine ascendente (ASC) o discendente (DESC)

UPDATE <Nome Tabella>
SET <Nome Colonna> = <Valore>
[WHERE <Condizione>]; -- se la clausola Where non e' specificata vengono aggiornate tutte le righe come specificate nella clausola SET
```

Sommario dei Link relativi all' SQL

SQL Reference Page
Programmer's Source
DevX
DB Ingredients
SQL Trainer S/W
Web Authoring
DBMS Lab/Links
SQL FAQ
Query List
SQL Practice Site
SQL Course II
Database Jump Site
Programming Tutorials on the Web
PostgreSQL
Adobe Acrobat
A Good DB Course
Tutorial Page
Intelligent Enterprise Magazine

miniSQL
SQL for DB2 Book
SQL Server 7
SQL Reference/Examples
SQL Topics
Lee's SQL Tutorial
Data Warehousing Homepage
MIT SQL for Web Nerds
RDBMS Server Feature Comparison Matrix
Oracle FAQ
Oracle Developer (2000)
Intro to Relational Database Design
SQL Sam, the SQL Server Detective
ZDNet's SQL Introduction
Baycon Group's SQL Tutorial
Dragonlee's SQL Tutorial
A good, but anonymous SQL Tutorial
UC Davis' Oracle PDF's
About.com's Database Advisor
Manas Tungare's SQL Tutorial at FindTutorials.com
A Gentle Introduction to SQL
SQL (News) Wire

vi consiglio vivamente di andare a visitare i link qui sopra riportati , specialmente se volete approfondire gli argomenti qui trattati ed in particolare cose come lo standard SQL-92. ricordatevi comunque che se state utilizzando un database commerciale ben conosciuto (come Oracle, SQLServer ecc) normalmente il posto migliore dove trovare informazioni e' normalmente il sito web del produttore del vostro DBMS
