**INTRODUZIONE** *02-03-23*

**SINTASSI BASE DI MYSQL**

Un database è un insieme di dati organizzati secondo un modello logico. Il software che gestisce il database è detto DBMS. Noi considereremo MySQL, che è un DBMS relazionale (cioè, adotta come modello logico il modello relazionale), dunque i dati sono organizzati in tabelle.

Per estrarre informazioni d’interesse da un database, bisogna formulare **un’interrogazione (query).** Per farlo useremo il linguaggio SQL, che è un linguaggio dichiarativo: si specificano le proprietà del risultato che si vuole ottenere, e non come lo si vuole ottenere (come invece fa, per esempio, C++). Il risultato che si otterrà sarà una tabella, detta **result set**.

Partiamo dal considerare un semplice database, costituito da una sola tabella:

Immagine che contiene tavolo

Descrizione generata automaticamente

Vediamo un primo esempio di query:

* INDICARE COGNOME E NOME DELLE PERSONE DI ETA’ INFERIORE A 40 ANNI

**attributi d’interesse**

**condizioni**



**fonte dei dati**



La sintassi di una query SQL è fatta da 3 parti:

qui si indicano, separati da virgola, gli **attributi d’interesse**, e cioè quali attributi dovrà avere il risultato, nell’ordine in cui sono indicati (si dice che questi sono gli attributi *proiettati*);

SELECT *Lista\_Attributi*



FROM *Insieme\_Tabelle*



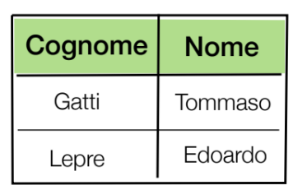
qui si indica la **fonte dei dati**, e cioè la tabella (o l’insieme di tabelle opportunamente combinate, vedi dopo) da cui si vogliono ricavare le informazioni d’interesse;

WHERE *Lista\_Condizioni*;



qui si indicano le **condizioni**: finiranno nel risultato solo le righe di *Insieme\_Tabelle* che le soddisfano (si dice che queste sono le righe *selezionate*). Alla fine si mette ‘;’.



Il codice sarà dunque il seguente:

3° SELECT Cognome, Nome

1° FROM Persona

2° WHERE Età < 40;

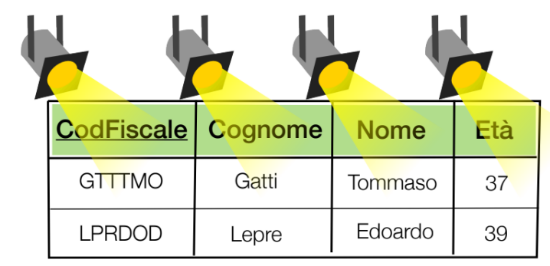
I numeri a lato indicano l’ordine con cui il DBMS processa il codice:

1. Prende la tabella indicata in FROM (*Persona*)
2. La controlla riga per riga, lasciando solo le righe che soddisfano le condizioni indicate in WHERE (e quindi quelle che assumono per l’attributo *Età* un valore ≥ 40);
3. Delle righe rimaste, lascia solo gli attributi indicati in SELECT, nell’ordine in cui sono scritti (e quindi rimarranno solo gli attributi *Cognome* e *Nome,* in quest’ordine).

Questo è anche l’ordine consigliato nello scrivere una query:

1. Prima si indica da dove si prendono le righe;
2. Poi quali righe si vogliono selezionare;
3. Infine quali attributi si vogliono proiettare.

Per proiettare tutti gli attributi, si scrive **SELECT** \*

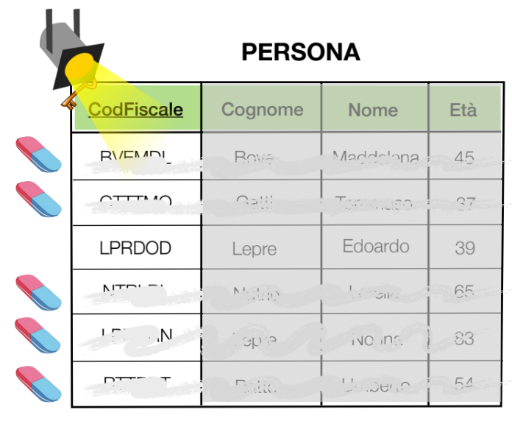
* INDICARE TUTTE LE INFORMAZIONI DELLE PERSONE DI ETA’ INFERIORE A 40 ANNI

SELECT \*

FROM Persona

WHERE Età < 40;

Condizioni più espressive si ottengono usando gli **operatori logici** (AND, OR, NOT):

* INDICARE IL CODICE FISCALE DELLE PERSONE DI ETA’ INFERIORE A 40 ANNI IL CUI COGNOME E’ LEPRE

SELECT CodFiscale

FROM Persona

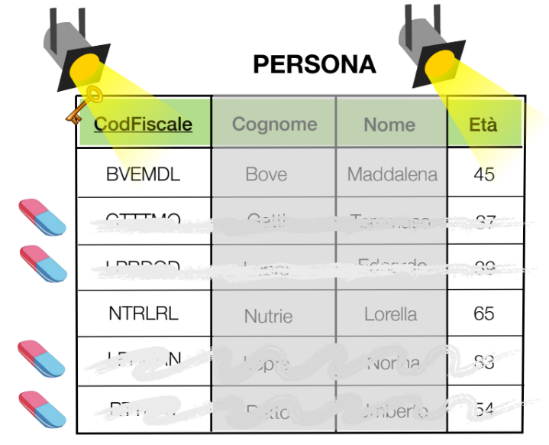
WHERE Età < 40

AND Cognome = ‘Lepre’;



**NB:** i valori di tipo stringa si scrivono tra singoli apici.

Finiranno nel risultato solo le righe che soddisfano entrambe le condizioni.

* INDICARE IL CODICE FISCALE E L’ETA’ DELLE PERSONE IL CUI COGNOME E’ NUTRIE O IL CUI NOME E’ MADDALENA

SELECT CodFiscale, Età

FROM Persona

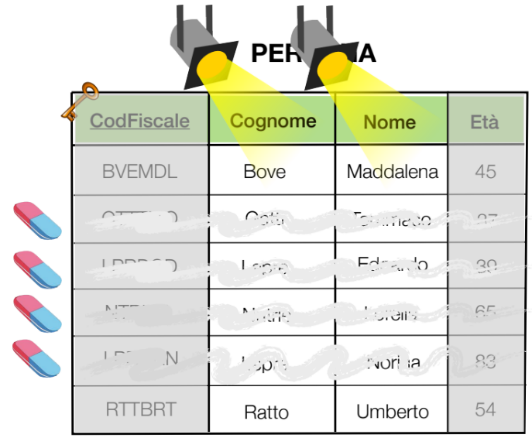
WHERE Cognome = ‘Nutrie’

OR Nome = ‘Maddalena’;

Finiranno nel risultato solo le righe che soddisfano almeno una delle due condizioni.



Se vogliamo che il valore di un attributo (numerico o data) sia compreso tra certi valori, si usa la direttiva **BETWEEN** (che include anche gli estremi) o una congiunzione logica di <= e >= (o < e >, se non si vogliono includere gli estremi):

* INDICARE COGNOME E NOME DELLE PERSONE DI ETA’ COMPRESA TRA 45 E 60 ANNI

SELECT Cognome, Nome

FROM Persona

WHERE Età BETWEEN 45 AND 60;

*oppure*

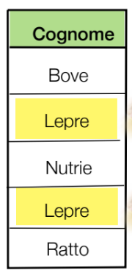
SELECT Cognome, Nome

FROM Persona

WHERE Età >= 45 AND Età <= 60;

**DISTINCT**

In una tabella non ci possono essere righe duplicate. I valori degli attributi non chiave, tuttavia, possono ripetersi in righe diverse, e questo può dare fastidio in certe query:

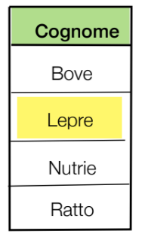
* INDICARE I COGNOMI DELLE PERSONE DI ETA’ ALMENO PARI A 38 ANNI

SELECT Cognome

FROM Persona

WHERE Età >= 38;

Nel risultato compare due volte la stessa riga, quando invece basterebbe che comparisse una volta sola.   
Per non avere duplicati nel risultato, si usa la direttiva **DISTINCT** dopo SELECT:

* INDICARE I COGNOMI DELLE PERSONE DI ETA’ ALMENO PARI A 38 ANNI

SELECT DISTINCT Cognome

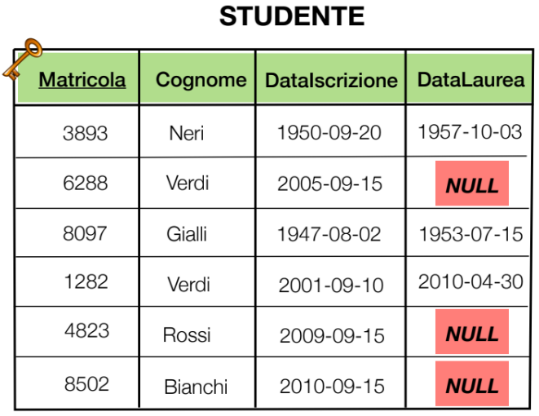
FROM Persona

WHERE Età >= 38;

La direttiva DISTINCT viene sempre processata alla fine: una volta effettuata la proiezione, se ci sono delle righe duplicate, queste vengono rimosse.

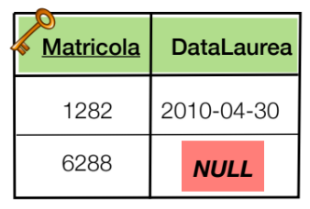
**VALORI NULL**

I valori NULL sono dei valori mancanti, per varie ragioni. Solo gli attributi non chiave possono assumere NULL come valore.



In questa tabella, il valore NULL in “DataLaurea” sta ad indicare che lo studente non si è ancora laureato.

Per trattare i valori NULL non si usano gli operatori di confronto, ma le direttive **IS NULL** e **IS NOT NULL**. Vediamo in che occasione potrebbero servire:

* INDICARE MATRICOLA E DATA DI LAUREA DEGLI STUDENTI IMMATRICOLATI TRA L’ANNO 2001 E L’ANNO 2005

SELECT Matricola, DataLaurea

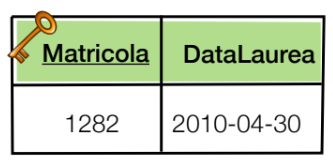
FROM Studente

WHERE DataIscrizione BETWEEN ‘2001-01-01’ AND ‘2005-12-31’;

**NB:** Le date si scrivono tra singoli apici ed in formato YYYY-MM-DD.



Nel risultato compaiono anche righe con DataLaurea a NULL, dunque studenti che non si sono ancora laureati. Avrebbe più senso escludere questi studenti, dunque una query più utile è la seguente:

* INDICARE MATRICOLA E DATA DI LAUREA DEGLI STUDENTI LAUREATI IMMATRICOLATI TRA L’ANNO 2001 E L’ANNO 2005

SELECT Matricola, DataLaurea

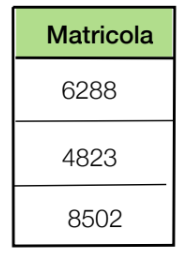
Dunque le righe che assumono valore NULL per l’attributo DataLaurea non compariranno nel risultato.

FROM Studente



WHERE DataLaurea IS NOT NULL

AND DataIscrizione BETWEEN ‘2001-01-01’ AND ‘2005-12-31’;

* INDICARE MATRICOLA E DATA DI LAUREA DEGLI STUDENTI NON ANCORA LAUREATI

SELECT Matricola, DataLaurea

Dunque le righe che assumono valore diverso da NULL per l’attributo non compariranno nel risultato.

FROM Studente

WHERE DataLaurea IS NULL;



**GESTIONE DELLE DATE**

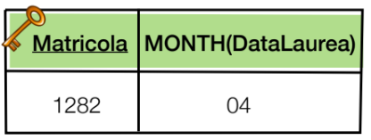
Le date possono essere confrontate usando gli operatori maggiore, minore, uguale, combinazioni di essi, o BETWEEN, e una data sarà minore di un’altra se viene prima.

Per ricordare, può essere utile tracciare una retta temporale, ed una data è minore di un’altra se viene prima:



Esistono poi le funzioni **DAY(*data*)**, **MONTH(*data*)** e **YEAR(*data*)** che restituiscono rispettivamente giorno, mese e anno di *data*, espressi come numeri (e quindi si possono confrontare con altri numeri).

* INDICARE MATRICOLA E MESE DI LAUREA DEGLI STUDENTI IMMATRICOLATI DOPO IL 2000

SELECT Matricola, MONTH(DataLaurea)

FROM Studente

Chiede il mese di laurea, dunque devono essere laureati.

WHERE DataLaurea IS NOT NULL



AND YEAR(DataIscrizione) > 2000;

Analizziamo ora questa query: “indicare il cognome degli studenti che si sono laureati 5 anni fa”. Questo significa che, partendo dalla data corrente, la data di laurea deve essere di 5 anni prima.

A questo scopo si usa la variabile di sistema **CURRENT\_DATE**, che contiene il valore della data corrente. Vediamo allora il codice:

* INDICARE IL COGNOME DEGLI STUDENTI CHE SI SONO LAUREATI 5 ANNI FA

**NB:** una query deve esprimere la condizione indipendentemente da quando viene eseguita.

Non sarebbe stato corretto dunque mettere come condizione “YEAR(DataLaurea) = 2023 – 5”, perché in un anno diverso dal 2023 non esprimerebbe più la condizione “che si sono laureati 5 anni fa”.

Notare poi che non ho messo come condizione “DataLaurea IS NOT NULL”, perché tanto le righe che hanno DataLaurea a NULL vengono escluse, essendo NULL != YEAR(…) - 5.

SELECT DISTINCT Cognome

FROM Studente



WHERE YEAR(DataLaurea) = YEAR(CURRENT\_DATE) – 5;

In MySQL, non è possibile sommare/sottrarre le date tra loro usando gli operatori classici +/-, ma vanno usate delle funzioni apposite.

La prima è la funzione **DATEDIFF(*data\_più\_recente*, *data\_più\_vecchia*)**, che restituisce il numero di giorni che separano le due date.

* INDICARE MATRICOLA E DA QUANTI GIORNI RISULTAVANO ISCRITTI GLI STUDENTI AD OGGI LAUREATI CHE NON SI ERANO LAUREATI IL 15 LUGLIO 2005

**NB**: va messo nel SELECT, perché questa è la parte dove si indicano quali attributi dovrà avere il risultato (che quindi possono essere sia attributi della tabella indicata in FROM, che altri attributi che sono stati calcolati tramite funzioni, come in questo caso).

Si ha quindi che, una volta processato il WHERE, per ogni riga rimasta viene proiettato il valore dell’attributo *Matricola* e il numero di giorni tra il 2005-07-15 e il valore di *DataIscrizione* presente in tale riga.

SELECT Matricola, DATEDIFF(‘2005-07-15’, DataIscrizione)



FROM Studente

WHERE DataIscrizione < ‘2005-07-15’

AND DataLaurea > ‘2005-07-15’;



*Ad oggi laureati che non si erano laureati il 15 luglio 2005*, dunque si devono essere iscritti prima del ‘2005-07-15’ e si devono essere laureati dopo il ‘2005-07-15’.

Si hanno poi le funzioni **DATE\_ADD** e **DATE\_SUB** che permettono rispettivamente di sommare e sottrarre intervalli di tempo ad una data, restituendo la data risultante.

Per esprimere un intervallo di tempo si usa la parola chiave **INTERVAL,** in questo modo:

INTERVAL *numero* [YEAR|MONTH|DAY]

Detto questo, la sintassi di DATE\_ADD e DATE\_SUB è la seguente:

DATE\_ADD(*data*, INTERVAL *numero* [YEAR|MONTH|DAY])

DATE\_SUB(*data*, INTERVAL *numero* [YEAR|MONTH|DAY])

* INDICARE LA MATRICOLA E IL MESE DI ISCRIZIONE DEGLI STUDENTI CHE SI SONO LAUREATI DOPO CINQUE ANNI ESATTI DAL GIORNO DELL’ISCRIZIONE

SELECT Matricola, MONTH(DataIscrizione)

FROM Studente

WHERE DataLaurea = DATE\_ADD(DataIscrizione, INTERVAL 5 YEAR);

In questo modo verranno mantenuti solo gli studenti che hanno una *DataLaurea* pari a *DataIscrizione* + intervallo di 5 anni esatti.

Notare che anche qui non ho messo come condizione “DataLaurea IS NOT NULL”, perché tanto le righe che hanno DataLaurea a NULL vengono escluse, essendo NULL != DATE\_ADD(…).



Si può anche sommare/sottrarre un intervallo di tempo ad una data usando direttamente gli operatori +/-:

* INDICARE LA MATRICOLA E IL MESE DI ISCRIZIONE DEGLI STUDENTI CHE SI SONO LAUREATI DOPO CINQUE ANNI ESATTI DAL GIORNO DELL’ISCRIZIONE

SELECT Matricola, MONTH(DataIscrizione)

FROM Studente

WHERE DataLaurea = DataIscrizione + INTERVAL 5 YEAR;

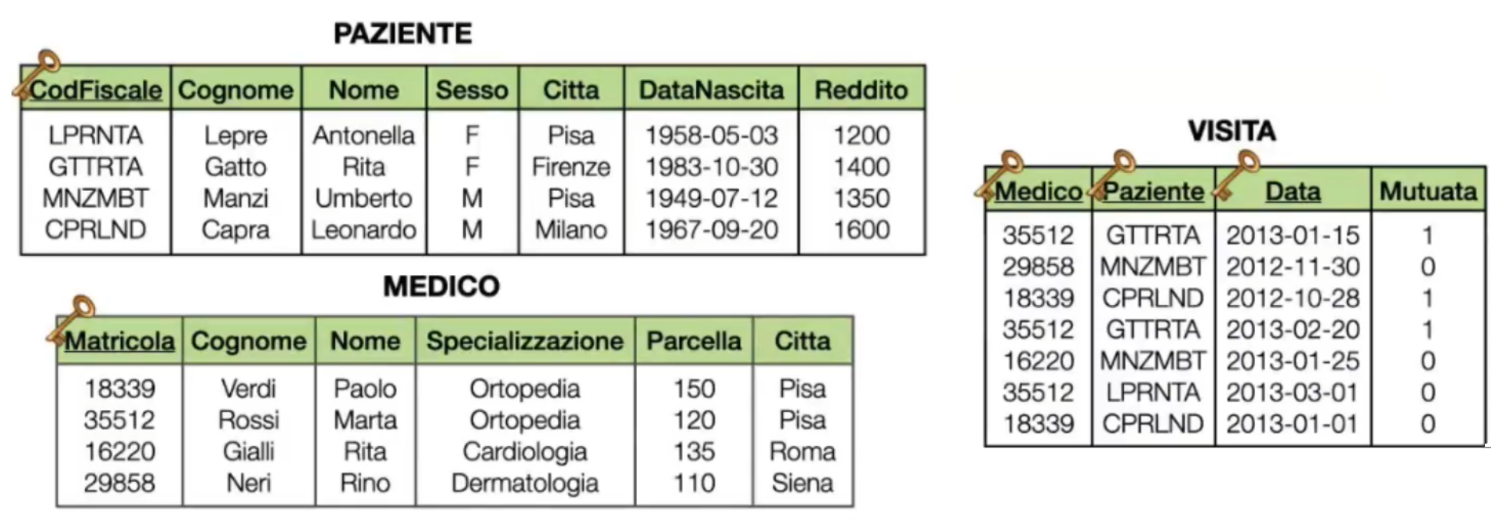
Equivalente alla query sopra, e più comodo.



**OPERATORI DI AGGREGAZIONE** *16-03-23*

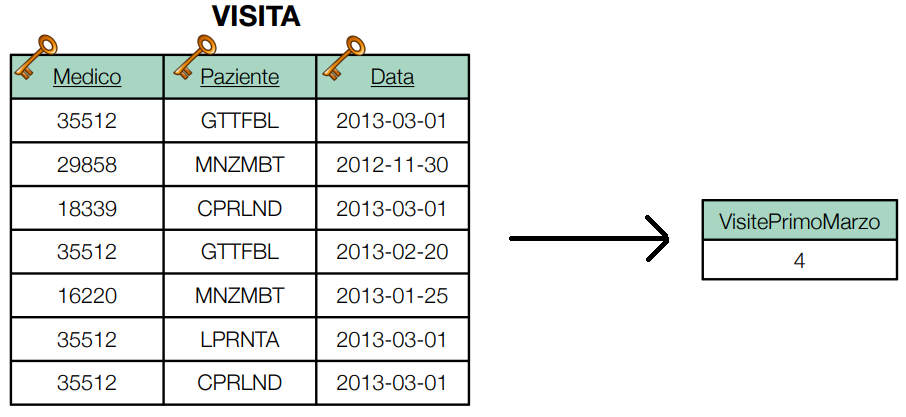
Un **operatore di aggregazione** permette di eseguire calcoli i cui operandi sono i valori assunti da un attributo in un insieme di record. In particolare, un operatore di aggregazione *collassa* tutti i record in uno solo formato da un solo attributo numerico, il cui valore è ciò che l’operatore ha calcolato. Per questo motivo, se si mette un operatore di aggregazione nel SELECT (e qui vanno messi, essendo questa la parte dove si indica quali attributi dovrà avere il risultato), non si possono proiettare attributi dell’*Insieme\_Tabelle* indicato nel FROM (perché l’operatore di aggregazione collasserà tutti i record in uno solo formato da un solo attributo numerico il cui valore è ciò che ha calcolato, dunque i valori degli altri attributi andranno persi), se non come argomento di altri operatori di aggregazione (così tutti i record verranno collassati in uno solo formato da tanti attributi numerici quanti sono gli operatori di aggregazione, e il valore di ciascuno sarà ciò che il corrispondente operatore di aggregazione avrà calcolato).

Vediamo alcuni operatori di aggregazione utili, considerando il seguente database (negli esempi seguenti, per motivi di spazio, alcuni attributi non verranno mostrati):



**COUNT(\*)**

L’operatore **COUNT(\*)** conta i record.

* INDICARE IL NUMERO DI VISITE EFFETTUATE IN DATA 1° MARZO 2013

SELECT COUNT(\*) AS VisitePrimoMarzo

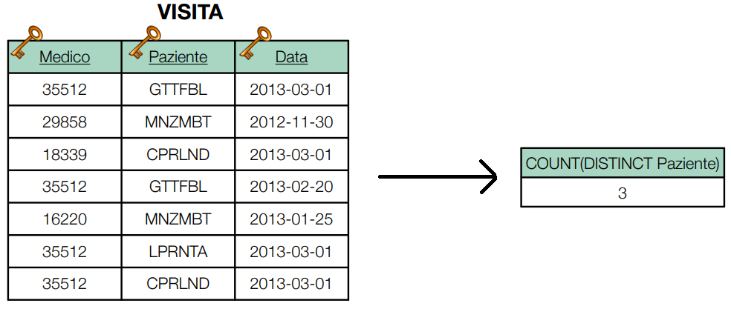
FROM Visita

WHERE Data = ‘2013-03-01’;

Prima seleziono i record della tabella *Visita* con *Data* = ‘2013-03-01’, e cioè le visite effettuate il 1° Marzo 2013. Poi uso COUNT(\*) per contare questi record, e quindi per contare queste visite.   
Il comando ***AS* *nome***serve a ridenominare un attributo (o assegnare un alias ad una tabella, vedi dopo).

**COUNT(DISTINCT *LISTA\_ATTRIBUTI*)**

L’operatore **COUNT(DISTINCT *Lista\_Attributi*)** conta i valori diversi assunti da un attributo (o un insieme di attributi) in un insieme di record.

* INDICARE IL NUMERO DI PAZIENTI VISITATI NEL MESE DI MARZO 2013

SELECT COUNT(DISTINCT Paziente)

FROM Visita

WHERE MONTH(Data) = 3 AND YEAR(Data) = 2013;

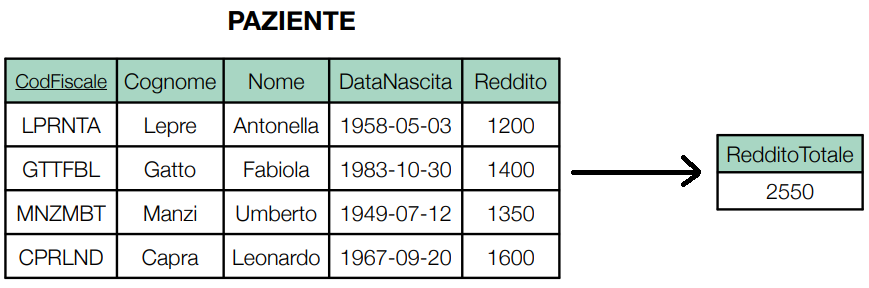
Prima seleziono i record della tabella *Visita* con MONTH(Data) = 3 e YEAR(Data) = 2013, e cioè le visite effettuate a Marzo 2013. Poi uso COUNT(DISTINCT *Paziente*) per contare il numero di valori diversi assunti dall’attributo *Paziente* in questi record, e quindi il numero di pazienti visitati in queste visite.

Notare che non sarebbe stato corretto usare COUNT(\*), e quindi contare queste visite, perché un paziente può essere stato visitato più volte, e quindi erroneamente lo avrei contato più volte.

**SUM(ATTRIBUTO) E AVG(ATTRIBUTO)**

Gli operatori **SUM(*Attributo*)** e **AVG(*Attributo*)** rispettivamente sommano e calcolano la media dei valori assunti da un attributo in un insieme di record.

* SUPPONENDO CHE UMBERTO MANZI SIA IL MARITO DI ANTONELLA LEPRE, CALCOLARE IL REDDITO TOTALE DELLA FAMIGLIA MANZI.

SELECT SUM(Reddito) AS RedditoTotale

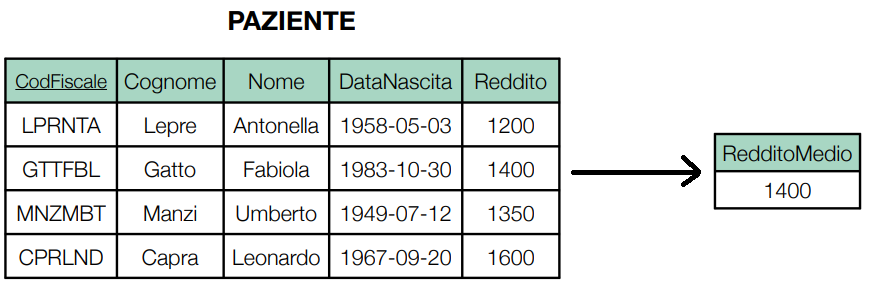
FROM Paziente

WHERE (Nome = ‘Umberto’ AND Cognome = ‘Manzi’)

OR (Nome = ‘Antonella’ AND Cognome = ‘Lepre’);

Prima seleziono i record della tabella *Paziente* che mi interessano. Poi uso SUM(*Reddito*) per sommare i valori assunti dall’attributo *Reddito* in questi record.

* CALCOLARE IL REDDITO MEDIO DEI PAZIENTI NATI DOPO IL 1950

SELECT AVG(Reddito) AS RedditoTotale

FROM Paziente

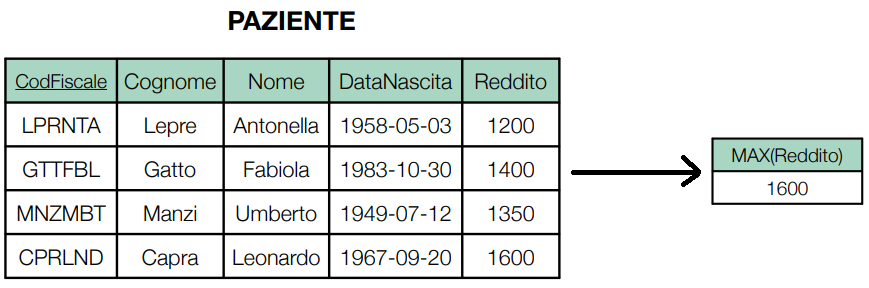
WHERE YEAR(DataNascita) > 1950;

Prima seleziono i record della tabella *Paziente* che mi interessano. Poi uso AVG(*Reddito*) per calcolare la media dei valori assunti dall’attributo *Reddito* in questi record.

**MAX(ATTRIBUTO) E MIN(ATTRIBUTO)**

Gli operatori **MAX(*Attributo*)** e **MIN(*Attributo*)** ricavano rispettivamente il valore massimo e il valore minimo tra i valori assunti da un attributo in un insieme di record.

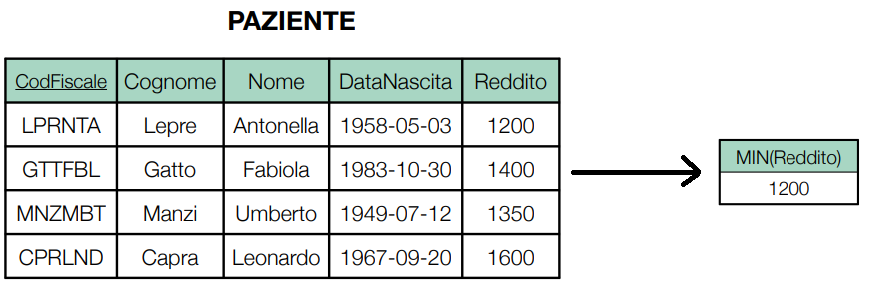
* RICAVARE IL REDDITO MASSIMO TRA QUELLI DI TUTTI I PAZIENTI

SELECT MAX(Reddito)

FROM Paziente;

Uso MAX(*Reddito*) per ricavare il valore massimo tra i valori assunti dall’attributo *Reddito* in tutti i record di *Paziente*.

* RICAVARE IL REDDITO MINIMO TRA QUELLI DI TUTTI I PAZIENTI

SELECT MIN(Reddito)

FROM Paziente;

Uso MIN(*Reddito*) per ricavare il valore minimo tra i valori assunti dall’attributo *Reddito* in tutti i record di *Paziente*.

Attenzione a non commettere un errore del genere:

* INDICARE QUAL È IL REDDITO MASSIMO TRA QUELLI DI TUTTI I PAZIENTI, E IL NOME E COGNOME DI CHI LO DETIENE

SELECT MAX(Reddito), Nome, Cognome



FROM Paziente;



Non va assolutamente fatta così. Ricordiamo infatti che un operatore di aggregazione collassa tutti i record in uno solo formato da un solo attributo numerico, il cui valore è ciò che l’operatore ha calcolato. Questo significa che i valori degli altri attributi andranno persi, e quindi verrà perso anche il *Nome* e *Cognome* del paziente con il reddito massimo.

**QUERY SU PIU’ TABELLE**

In alcuni casi, l’informazione d’interesse può essere frammentata su più tabelle. E’ necessario dunque riunire i vari pezzi, e a tale scopo si usano i join. Vanno messi nel FROM (visto che è qui che si indica l’insieme di tabelle opportunamente combinate da cui si vogliono ricavare le informazioni d’interesse)

**INNER JOIN**

Date due tabelle, **l’inner join** combina ogni record della prima con tutti i record della seconda che verificano una determinata condizione (ed eventuali attributi omonimi tra le due tabelle compaiono due volte a seguito del join). I record che non sono mai stati combinati vengono scartati.

La sintassi è la seguente:

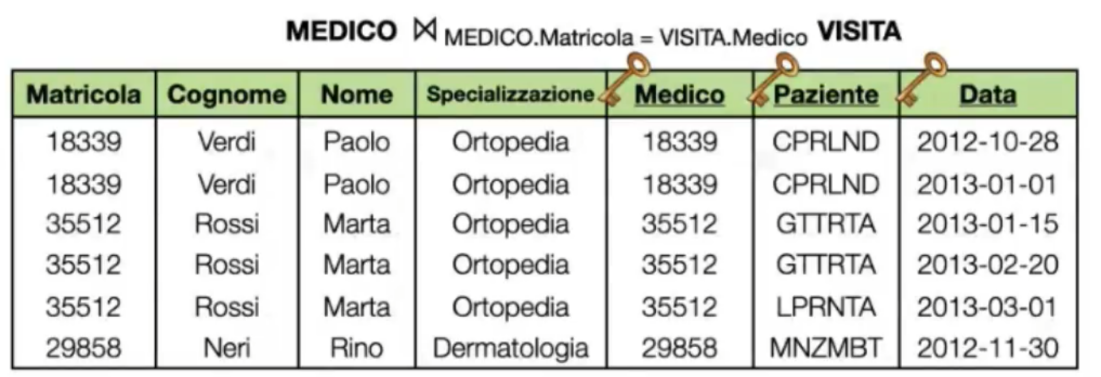
*Tabella\_1* INNER JOIN *Tabella\_2* ON *Condizione*

* INDICARE NOME E COGNOME DEI MEDICI CHE HANNO EFFETTUATO ALMENO UNA VISITA

SELECT DISTINCT M.Nome, M.Cognome

FROM Visita V INNER JOIN Medico M ON V.Medico = M.Matricola;

Mi serve la tabella *Visita* per sapere la matricola dei medici che hanno effettuato almeno una visita, e la tabella *Medico* per sapere nome e cognome di questi medici. Per unire allora i vari pezzi d’informazione, facciamo un inner join tra le due tabelle sulla condizione *Medico* = *Matricola*, così da associare, ad ogni visita, il medico che l’ha fatta. Questo è ciò che si ottiene a seguito del join:



Fatto ciò, di queste visite più informative, proietto il nome e cognome dei medici che le hanno fatte, che quindi saranno nome e cognome di medici che hanno effettuato almeno una visita (e metto *distinct*, perché uno stesso medico può avere fatto più visite, ma a me interessa che il suo nome e cognome compaia nel result set una volta sola).

Quei *V* e *M* sono degli **alias di tabelle**, e servono per indicare senza ambiguità quale attributo stiamo considerando (indicando la tabella d’origine di tale attributo).

**NATURAL JOIN**

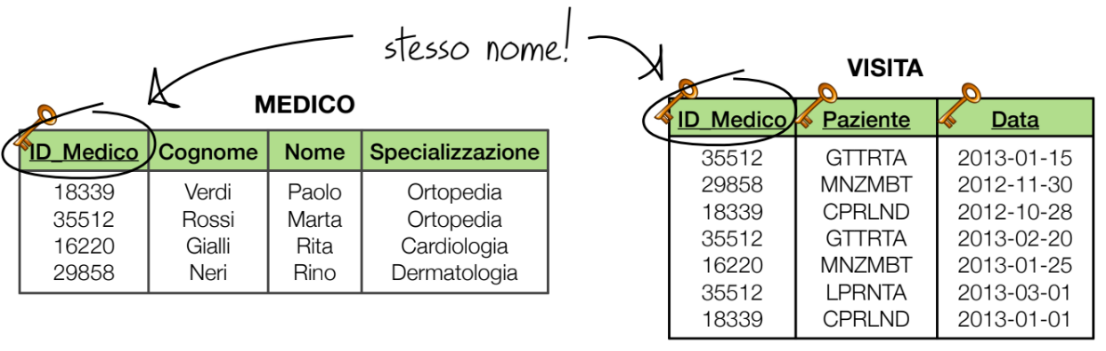
Date due tabelle, il **natural join** combina ogni record della prima con tutti i record della seconda aventi valori uguali su tutti gli attributi omonimi (e questi compaiono una volta sola a seguito del join). I record che non sono mai stati combinati vengono scartati.

La sintassi è la seguente:

*Tabella\_1* NATURAL JOIN *Tabella\_2*

* INDICARE NOME E COGNOME DEI MEDICI CHE HANNO EFFETTUATO ALMENO UNA VISITA

Solo per questo esempio, supponiamo che il database sia il seguente:

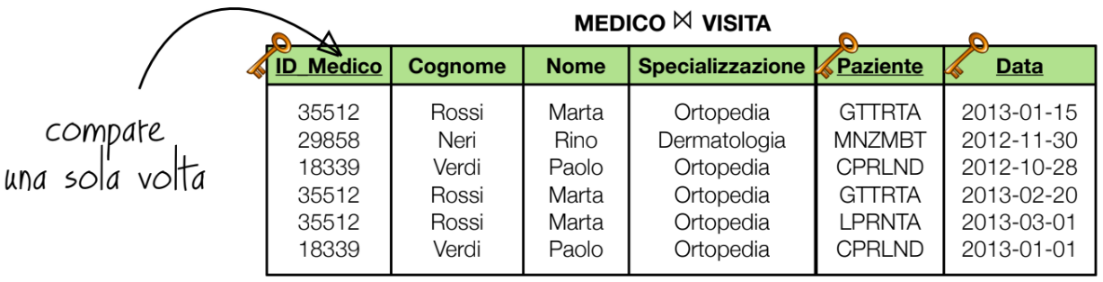


La query, allora, è la seguente:

SELECT DISTINCT M.Nome, M.Cognome

FROM Visita V NATURAL JOIN Medico M;

Mi serve la tabella *Visita* per sapere la matricola dei medici che hanno effettuato almeno una visita, e la tabella *Medico* per sapere nome e cognome di questi medici. Per unire allora i vari pezzi d’informazione, visto che le due tabelle hanno entrambe un attributo di nome *ID\_Medico*, ci faccio un natural join, così da associare, ad ogni visita, il medico che l’ha fatta. Questo è ciò che si ottiene a seguito del join:



Fatto ciò, di queste visite più informative, proietto il nome e cognome dei medici che le hanno fatte, che quindi saranno nome e cognome di medici che hanno effettuato almeno una visita (e metto *distinct*, perché uno stesso medico può avere effettuato più visite, ma a me interessa che il suo nome e cognome compaia nel result set una volta sola).

**PRODOTTO CARTESIANO**

Date due tabelle, il **prodotto cartesiano** combina ogni record della prima con tutti i record della seconda.

La sintassi è la seguente:

*Tabella\_1* CROSS JOIN *Tabella\_2*

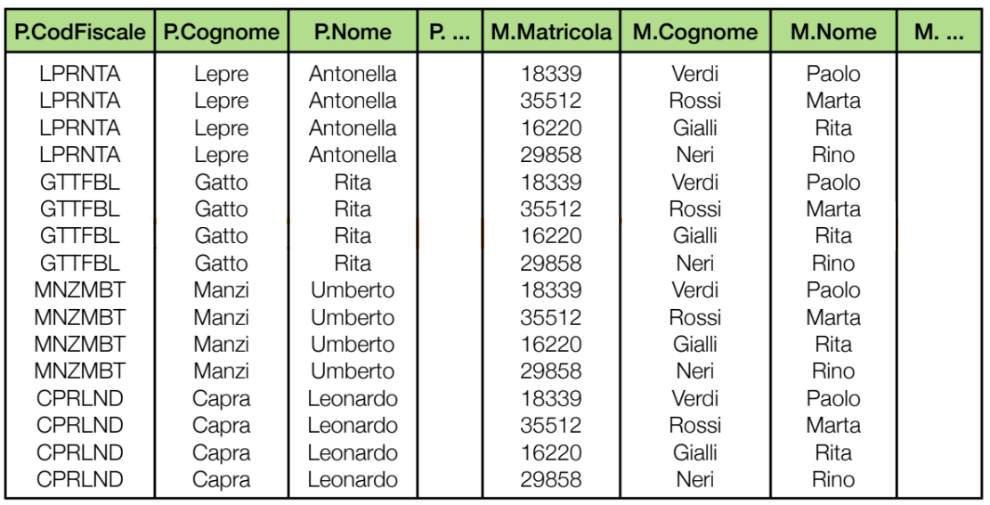
* INDICARE IL NUMERO DI PAZIENTI AVENTI NOME UGUALE AD ALMENO UN MEDICO DELLA CLINICA

SELECT COUNT(DISTINCT P.CodFiscale)

FROM Paziente P CROSS JOIN Medico M

WHERE P.Nome = M.Nome;

Faccio un cross join tra la tabella *Paziente* e la tabella *Medico* così da combinare ogni paziente con ogni medico. Questo è ciò che si ottiene a seguito del join:



Di tutte le combinazioni, seleziono quelle dove il nome del paziente è uguale al nome del medico. Infine, conto il numero di distinti codici fiscali che ci sono nelle combinazioni selezionate, e quindi il numero di pazienti aventi nome uguale ad almeno un medico della clinica (non conto le combinazioni selezionate, e quindi non uso count(\*), altrimenti conterei il numero totale di volte che un paziente ha lo stesso nome di un medico, e visto che un paziente può avere nome uguale a più medici, otterrei potenzialmente un risultato diverso).

**OUTER JOIN**

Date due tabelle, **l’inner join** combina ogni record della prima con tutti i record della seconda che verificano una determinata condizione (ed eventuali attributi omonimi tra le due tabelle compaiono due volte a seguito del join). Mantiene anche i record della prima tabella (left outer join) o della seconda tabella (right outer join) che non sono mai stati combinati, completandoli con valori NULL.

La sintassi è la seguente:

*Tabella\_1* LEFT OUTER JOIN *Tabella\_2* ON *Condizione*

*Tabella\_1* RIGHT OUTER JOIN *Tabella\_2* ON *Condizione*

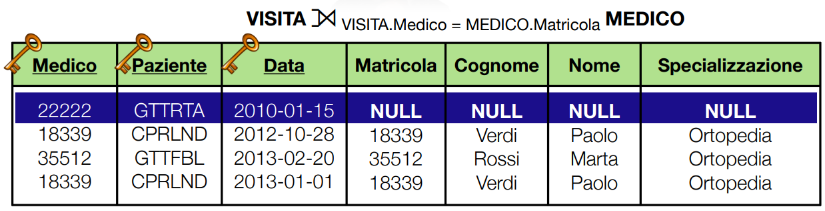
* INDICARE LE VISITE EFFETTUATE DA MEDICI CHE NON LAVORANO PIÙ PRESSO LA CLINICA

SELECT V\*

FROM Visita V LEFT OUTER JOIN Medico M ON V.Medico = M.Matricola

WHERE M.Matricola IS NULL;

Faccio un left outer join tra *Visita* e *Medico* sulla condizione *V.Medico* = *M.Matricola*, così da associare, ad ogni visita, il medico che l’ha fatta. Le visite che non verranno mai combinate saranno quelle fatte da medici che non compaiono nella tabella *Medico*, e quindi che non lavorano più nella clinica; queste, a seguito del join, verranno completate con valori NULL:



Seleziono ora le visite fatte da medici che non lavorano più nella clinica (che sono, come già detto, quelle che, a seguito del join, sono state completate con valori NULL, e quindi hanno *M.Matricola* a NULL).

Infine, visto che mi chiede di indicare le visite, proietto solo gli attributi originari dalla tabella *Visita* (e per farlo, si può scrivere SELECT V.\*).

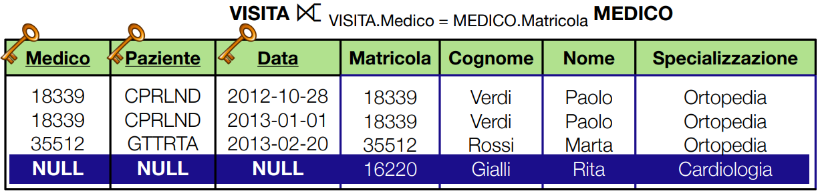
* INDICARE MATRICOLA, COGNOME E SPECIALIZZAZIONE DEI MEDICI CHE NON HANNO ANCORA EFFETTUATO VISITE.

SELECT M.Matricola, M.Cognome, M.Specializzazione

FROM Visita V RIGHT OUTER JOIN Medico M ON V.Medico = M.Matricola

WHERE V.Medico IS NULL;

Faccio un right outer join tra *Visita* e *Medico* sulla condizione *V.Medico* = *M.Matricola*, così da associare, ad ogni visita, il medico che l’ha fatta. I medici che non verranno mai combinati saranno dunque quelli che non hanno ancora effettuato visite, e questi, a seguito del join, verranno completati con valori NULL:



Seleziono ora i medici che non hanno mai effettuato visite (che sono, come già detto, quelli che, a seguito del join, sono stati completati con valori NULL, e quindi hanno *V.Medico* a NULL).

Infine, proietto *Matricola*, *Cognome* e *Specializzazione* di questi medici.

**JOIN MULTIPLO**

E’ possibile fare il join tra più di due tabelle.

* INDICARE NOME E COGNOME DEI PAZIENTI VISITATI DAL DOTT. RINO NERI.

SELECT DISTINCT P.Nome, P.Cognome

FROM Visita V

INNER JOIN

Medico M On V.Medico = M.Matricola

INNER JOIN

Paziente P On V.Paziente = P.CodFiscale

WHERE M.Nome = ‘Rino’ AND M.Cognome = ‘Neri’;

Faccio un inner join tra *Visita* e *Medico* sulla condizione *V.Medico* = *M.Matricola*, così da associare, ad ogni visita, il medico che l’ha fatta.

Faccio poi un inner join tra ciò che ho ottenuto prima e *Paziente* sulla condizione *V.Paziente*   
= *P.CodFiscale*, così da associare, ad ogni visita e al medico che l’ha fatta, il paziente che è stato visitato.

Di queste visite più informative, seleziono quelle dove il medico che ha effettuato la visita è Rino Neri.

Infine, delle visite selezionate, proietto nome e cognome dei pazienti che sono stati visitati (e metto *distinct*, perché uno stesso paziente può essere stato visitato più volte da Rino Neri, ma a me interessa che il suo nome e cognome compaia una volta sola nel result set).

**SELF JOIN**

In alcuni casi, può essere utile fare un join tra una tabella e se stessa. Di solito, questo permette di risolvere query che nel testo contengono espressioni come “*più di una volta*”.

* INDICARE IL CODICE FISCALE DEI PAZIENTI CHE SONO STATI VISITATI PIÙ DI UNA VOLTA DA UNO STESSO MEDICO DELLA CLINICA NEL MESE CORRENTE

SELECT DISTINCT V1.Paziente

FROM Visita V1

INNER JOIN

Visita V2 ON (

V1.Medico = V2.Medico

AND V1.Paziente = V2.Paziente

AND V1.Data <> V2.Data

)

WHERE MONTH(V1.Data) = MONTH(CURRENT\_DATE)

AND YEAR(V1.Data) = YEAR(CURRENT\_DATE)

AND MONTH(V2.Data) = MONTH(CURRENT\_DATE)

AND YEAR(V2.Data) = YEAR(CURRENT\_DATE);

Faccio un inner join tra *Visita* e se stessa sulla condizione V1.Medico = V2.Medico AND   
V1.Paziente = V2.Paziente AND V1.Data <> V2.Data. Ogni record risultante, quindi, sarà costituito da due visite fatte ad uno stesso paziente, da uno stesso medico, in data diversa (e quindi i pazienti indicati nei record sono quelli che sono stati visitati più di una volta da uno stesso medico).

Di questi record, seleziono quelli dove entrambe le visite sono state fatte nel mese corrente.

Infine, di ogni record selezionato, proietto il codice fiscale del paziente visitato in entrambe le visite (e quindi il codice fiscale dei pazienti che sono stati visitati più di una volta da uno stesso medico nel mese corrente. Metto poi *distinct*, visto che i record selezionati sono combinazioni di due visite fatte allo stesso paziente, dallo stesso medico, in data diversa, nel mese corrente. Il codice fiscale del paziente comparirà dunque tante volte quante sono le possibili combinazioni di due sue visite del genere, ma a me interessa che compaia una volta sola nel result set).

**DERIVED TABLE**

Una **derived table** è una tabella definita attraverso una query incapsulata nel FROM di un’altra query.   
E’ utile per mantenere risultati intermedi, ed è volatile: viene calcolata solo quando bisogna processare la query in cui è incapsulata, dopodiché viene eliminata dalla memoria.

Bisogna sempre assegnare un alias ad una derived table, usando il comando *AS alias*.

* INDICARE LA MATRICOLA DEI MEDICI CHE NON HANNO MAI VISITATO DI GIOVEDÌ

SELECT DISTINCT V1.Medico

FROM Visita V1

LEFT OUTER JOIN

(

SELECT V2.Medico

FROM Visita V2

WHERE DAYOFWEEK(V2.Data) = 4

) AS D

ON V1.Medico = D.Medico

WHERE D.Medico IS NULL;

Prima definisco una derived table, assegnandole l’alias *D*, in cui metto le matricole dei medici che hanno visitato di giovedì.

A questo punto faccio un left outer join tra *Visita* e la derived table sulla condizione   
V1.Medico = D.Medico, e così ogni visita verrà combinata al medico che l’ha fatta, ma solo se tale medico ha anche visitato di giovedì.

Le visite che non verranno mai combinate saranno dunque visite fatte da medici che non hanno mai visitato di giovedì, e queste, a seguito del join, verranno completate con valori NULL (in particolare, avranno D.Medico a NULL). Seleziono allora queste visite, e proietto la matricola dei medici che le hanno fatte, che sarà quindi la matricola di medici che non hanno mai visitato di giovedì (*distinct*, perché un medico può aver fatto più visite non di giovedì, ma a me interessa che la sua matricola compaia una volta sola nel result set).

**RAGGRUPPAMENTO**  *30-03-23*

Il **raggruppamento** suddivide un insieme di record in gruppi di record che assumono lo stesso valore per uno o più attributi, detti *attributi di raggruppamento.*

La sintassi, con indicato anche l’ordine di esecuzione, è la seguente:

4° SELECT …

1° FROM …

2° WHERE …

3° **GROUP BY *Lista\_Attributi***

A seguito del raggruppamento, ogni gruppo collassa in un unico record. Per questo motivo, il raggruppamento viene spesso usato insieme ad operatori di aggregazione, che verranno applicati non su tutto l’insieme di record, ma su ogni singolo gruppo. In questo modo, ogni singolo gruppo collasserà in un unico record formato da tanti attributi numerici quanti sono gli operatori di aggregazione, e il valore di ciascuno sarà ciò che il corrispondente operatore di aggregazione avrà calcolato.

Oltre agli operatori di aggregazione, è possibile proiettare solo gli attributi di raggruppamento. Si avrà così che, per ogni gruppo collassato, oltre ad eventuali valori calcolati dagli operatori di aggregazione, verrà indicato anche il valore degli attributi di raggruppamento che i record di tale gruppo assumevano (e questo è possibile, perché in ogni gruppo è costante il valore degli attributi di raggruppamento, e quindi il DBMS “sa” che valore indicare per ogni gruppo, anche a seguito del collasso).

Il raggruppamento è utile per risolvere query che presentano nel testo espressioni come “*per ogni*”.

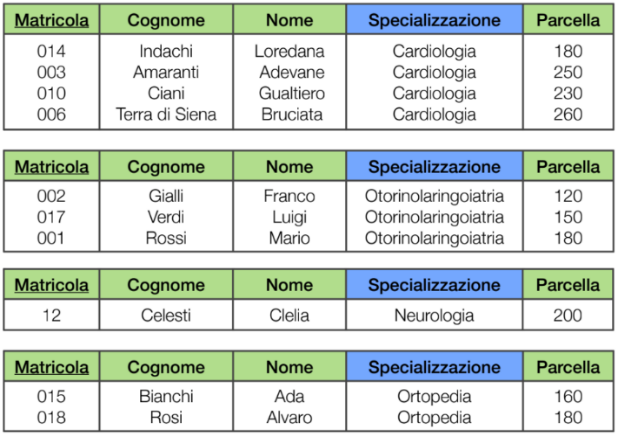
* DATA OGNI SPECIALIZZAZIONE, INDICARNE IL NOME E LA PARCELLA MEDIA DEI MEDICI

SELECT M.Specializzazione, AVG(M.Parcella) AS ParcellaMedia

FROM Medico M

GROUP BY M.Specializzazione;

Il DBMS fa questo: data la tabella *Medico*, la partiziona in gruppi di record che assumono lo stesso valore per l’attributo *Specializzazione* (e quindi, raggruppa i medici per specializzazione):



Fatto ciò, applica l’operatore di aggregazione su ogni singolo gruppo e affianca, ad ogni gruppo collassato, il valore che i record di tale gruppo assumevano per l’attributo di raggruppamento (e così, per ogni specializzazione, indica il nome e la parcella media dei medici):



* PER OGNI SPECIALIZZAZIONE, INDICARNE IL NOME, LA PARCELLA MINIMA E IL COGNOME DEL MEDICO A CUI APPARTIENE

SELECT M.Specializzazione, MIN(M.Parcella), M.Cognome



FROM Medico M

GROUP BY M.Specializzazione;

Non va assolutamente fatta così. Ricordiamo infatti che un operatore di aggregazione collassa ogni gruppo di record in uno solo formato da un solo attributo numerico, il cui valore è ciò che l’operatore ha calcolato. Questo significa che i valori degli altri attributi andranno persi, e quindi verrà perso anche il *Cognome* del *Medico* con il reddito massimo.

Gli unici attributi che si possono proiettare, come già detto, sono quelli di raggruppamento (perché in ogni gruppo è costante il valore di tali attributi, e quindi il DBMS “sa” che valore indicare per ogni gruppo, anche a seguito del collasso).

Questa query, dunque, va risolta “a pezzi”:

1. Prima bisogna trovare, per ogni specializzazione, il nome e la parcella minima;
2. Poi bisogna trovare i medici che hanno, come parcella, la parcella minima della loro specializzazione (informazione che si trova nel risultato calcolato nel punto precedente).

Un modo di risolverla, allora, è il seguente:

SELECT M.Specializzazione, D.ParcellaMinima, M.Cognome

FROM Medico M

NATURAL JOIN

(



SELECT M.Specializzazione, MIN(M.Parcella) AS M.ParcellaMinima



FROM Medico M



GROUP BY M.Specializzazione

) AS D

WHERE M.Parcella = D.ParcellaMinima;

1. Prima definisco una derived table, con alias D, in cui indico, per ogni specializzazione, il nome e la parcella minima.
2. Poi faccio un natural join tra *Medico* e la derived table (natural join, in quanto le due tabelle hanno l’attributo *Specializzazione* omonimo), così da associare, ad ogni medico, la parcella minima della sua specializzazione.

Di questi medici con l’informazione aggiuntiva, seleziono quelli che hanno la parcella uguale alla parcella minima della loro specializzazione.

I medici selezionati, dunque, saranno quelli che hanno la parcella minima della loro specializzazione. Di questi, allora, proietto la loro specializzazione, la loro parcella e il loro cognome.

**CONDIZIONI SUI GRUPPI**

Le **condizioni sui gruppi** vengono valutate gruppo per gruppo, e verranno lasciati solo i gruppi che le soddisfano. Sono espresse tramite operatori di aggregazione.

La sintassi, con indicato anche l’ordine di esecuzione, è la seguente:

5° SELECT …

1° FROM …

2° WHERE …

3° GROUP BY *…*

*4°* **HAVING *Lista\_Condizioni***

* INDICARE LE SPECIALIZZAZIONI DELLA CLINICA CON PIU’ DI DUE MEDICI

SELECT Specializzazione

FROM Medico

GROUP BY Specializzazione

HAVING COUNT(\*) > 2

Raggruppo i medici per specializzazione, e lascio i gruppi che sono composti da più di due medici   
(e cioè, i gruppi composti da più di 2 record, e quindi i gruppi per cui, applicando su di essi l’operatore COUNT(\*), questo restituisce un valore > 2).

Di ogni gruppo, poi, proietto la specializzazione (ricordiamo che ogni gruppo collassa in un unico record a prescindere se si usano o meno operatori di aggregazione nel SELECT. Si avrà così che, per ogni gruppo collassato, verrà indicato il valore dell’attributo *Specializzazione* che i record di tale gruppo assumevano. Non serve poi distinct, perché il raggruppamento era stato fatto in base al valore dell’attributo *Specializzazione*, dunque per ogni gruppo collassato verrà indicato un valore diverso di tale attributo).

* INDICARE LE SPECIALIZZAZIONI CON LA PIÙ ALTA PARCELLA MEDIA

SELECT Specializzazione

FROM Medico

GROUP BY Specializzazione

HAVING MAX(AVG(Parcella));



Non va fatta così. MySQL, infatti, non permette di annidare gli operatori di aggregazione.   
In più, questa “condizione” non esprime nulla: non c’è un operatore di confronto e non c’è un secondo membro.

Anche questa query, dunque, va risolta “a pezzi”:

1. Prima bisogna trovare la media delle parcelle dei medici di ogni specializzazione;
2. Poi bisogna calcolare la più alta parcella media (e quindi il massimo delle parcelle del result set trovato al punto precedente);
3. Infine, bisogna prendere le specializzazioni che hanno, come parcella media, la più alta parcella media (e cioè, il valore trovato al punto precedente).

Un modo di risolverla, allora, è il seguente:

SELECT M.Specializzazione

La *tabella 2°* è costituita da un unico record con un unico attributo, e quindi da un unico valore.

Con la condizione ‘HAVING AVG(M.Parcella) = *tabella 2°’*, dunque, confronto il valore restituito da AVG gruppo per gruppo con l’unico valore della *tabella 2°*.

FROM Medico M

GROUP BY M.Specializzazione

HAVING AVG(M.Parcella) =



(



SELECT MAX(D.MediaParcelle)



FROM (

SELECT M2.Specializzazione, AVG(M2.Parcella) AS MediaParcelle



FROM Medico M2



GROUP BY M2.Specializzazione

) AS D

);

1. Raggruppo i medici per specializzazione e, di ogni gruppo, trovo la parcella media;
2. Trovo la parcella media più alta;
3. Raggruppo i medici per specializzazione e lascio i gruppi che hanno la parcella media uguale alla più alta parcella media. Dei gruppi rimasti, infine, proietto la specializzazione.

Facciamo ora un confronto tra la clausola HAVING e la clausola WHERE:

* Le condizioni nel WHERE sono valutate record per record, prima del raggruppamento;
* Le condizioni nell’HAVING sono valutate gruppo per gruppo, dopo il raggruppamento.

Quale delle due clausole usare per esprimere una certa condizione scritta nel testo di una query?

* Se può essere espressa usando un operatore di aggregazione, allora va scritta come condizione sui gruppi, usando la clausola **HAVING**;
* Altrimenti, va scritta come condizione sui record, usando la clausola **WHERE**.
* INDICARE LE SPECIALIZZAZIONI CON PIÙ DI DUE MEDICI DI PISA

SELECT Specializzazione

FROM Medico

WHERE Città = ‘Pisa’

GROUP BY Specializzazione

HAVING COUNT(\*) > 2;

Prima seleziono i medici di Pisa.

Poi raggruppo questi medici per specializzazione e lascio i gruppi con più di due medici.

Infine, dei gruppi rimasti, proietto le specializzazioni (che quindi saranno specializzazioni con più di due medici di Pisa).

**COMMON TABLE EXPRESSION (CTE)**

Le **CTE** sono delle tabelle dotate di identificatore definite tramite query prima della query vera e propria.   
Possono essere utilizzate dalle query immediatamente successive (altre CTE o la query vera e propria) indicandole semplicemente tramite il loro identificatore (e non ricopiando tutto il codice, come si faceva con le derived table). Sono utili per mantenere risultati intermedi, e sono volatili: vengono calcolate solo quando bisogna processare la query che le utilizza, dopodiché vengono eliminate dalla memoria.

La sintassi per dichiarare le CTE è la seguente:

**WITH**

CTE1

name1 AS (query1),

CTE2

name2 AS (query2),

…



...,

CTEN

nameN AS (queryN)



query vera e propria;

* INDICARE IL NUMERO DI PAZIENTI DI SIENA, MAI VISITATI DA ORTOPEDICI

WITH

ortopedici AS

(

SELECT M.Matricola AS Medico

FROM Medico M

WHERE M.Specializzazione = ‘Ortopedia’

),

paz\_visitati\_ortopedici AS

(

SELECT V.Paziente

FROM ortopedici O

NATURAL JOIN Visita V

)

SELECT COUNT(\*)

FROM Paziente P

LEFT OUTER JOIN

paz\_visitati\_ortopedici PVO ON P.CodFiscale = PVO.Paziente

WHERE P.Citta = ‘Siena’

AND PVO.Paziente IS NULL;

1. La prima CTE *ortopedici* contiene le matricole dei medici ortopedici.
2. La seconda CTE *paz\_visitati\_ortopedici* contiene i codici fiscali dei pazienti visitati da ortopedici. Questi li trovo facendo un natural join tra *ortopedici* e *Visita* (natural join, in quanto le due tabelle hanno l’attributo *Medico* omonimo) così da associare, ad ogni ortopedico, le visite che ha fatto.

Di queste visite più informative, proietto i pazienti che sono stati visitati (che quindi saranno pazienti visitati da ortopedici).

1. Nella query vera e propria, faccio un left outer join tra *Paziente* e *paz\_visitati\_ortopedici* sulla condizione *P.CodFiscale* = *PVO.Paziente*, e così un paziente di *Paziente* verrà combinato solo se il suo codice fiscale compare in *paz\_visitati\_ortopedici*, e quindi se è stato visitato da un ortopedico.

I pazienti che non verranno mai combinati saranno dunque pazienti che non sono mai stati visitati da ortopedici, e questi, a seguito del join, saranno completati con valori NULL. Seleziono allora questi pazienti, considerando solo quelli di Siena, e conto quanti sono.

**SUBQUERY** *21-04-23*

Le **subquery** sono delle query che possono essere incapsulate in un’altra query. La struttura di una query con subquery è dunque la seguente:

OUTER QUERY

Query con subquery



SUBQUERY

Le subquery possono essere di due tipi:

* Non correlated subquery;
* Correlated subquery.

**NON CORRELATED SUBQUERY**

Una **non correlated subquery** è una subquery il cui result set non dipende dal record della outer query che è attualmente processato. Si incapsula nel WHERE della outer query, viene processata per prima, e si può confrontare un record della outer query con quelli del result set della subquery in due modi:

* Usando **IN**,un record della outer query finirà nel risultato finale solo se i valori che assume su un sottoinsieme di attributi si trovano in almeno un record del result set della subquery;
* Usando **NOT IN**,un record della outer query finirà nel risultato finale solo se i valori che assume su un sottoinsieme di attributi non si trovano in alcun record del result set della subquery.
* INDICARE NOME, COGNOME E PARCELLA DEGLI ORTOPEDICI CHE HANNO EFFETTUATO ALMENO UNA VISITA NELL’ANNO 2013

SELECT M.Nome, M.Cognome, M.Parcella



FROM Medico M



WHERE M.Specializzazione = ‘Ortopedia’

AND M.Matricola IN (



SELECT V.Medico

FROM Visita V



WHERE YEAR(V.Data) = 2013

);

Per scrivere una query con non correlated subquery, si parte dalla query più interna (che è poi l’ordine di processazione di una query del genere):

1. Prima trovo le matricole dei medici che hanno visitato nel 2013;
2. Poi trovo nome, cognome e la parcella dei medici ortopedici che hanno effettuato almeno una visita nel 2013 (e cioè, la cui matricola compare tra quelle dei medici che hanno visitato nel 2013).

* INDICARE I COGNOMI DEI PAZIENTI CHE NON APPARTENGONO ANCHE A UN MEDICO

SELECT DISTINCT P.Cognome

FROM Paziente P

WHERE P.Cognome NOT IN (



SELECT M.Cognome

FROM Medico M



);

1. Prima trovo i cognomi dei medici;
2. Poi trovo i cognomi dei pazienti che non appartengono anche a un medico (e cioè, il cui cognome non è tra i cognomi dei medici)

* INDICARE NOME, COGNOME E SPECIALIZZAZIONE DEI MEDICI CHE HANNO EFFETTUATO VISITE ECCETTO CHE IL GIORNO 1° MARZO 2013

SELECT DISTINCT M.Nome, M.Cognome, M.Specializzazione

FROM Medico M

WHERE M.Matricola IN (

SELECT V.Medico

FROM Visita V



)



AND M.Matricola NOT IN (

SELECT V.Medico

FROM Visita V



WHERE V.Data = ‘2013-03-01’

);

1. Prima trovo le matricole dei medici che hanno effettuato almeno una visita;
2. Poi trovo le matricole dei medici che hanno effettuato almeno una visita il 1° Marzo 2013;
3. Infine, trovo nome, cognome e specializzazione dei medici che hanno effettuato almeno una visita, eccetto il 1° Marzo 2013 (e cioè, la cui matricola compare tra quelle dei medici che hanno effettuato almeno una visita e la cui matricola non compare tra quelle dei medici che hanno effettuato almeno una visita il 1° Marzo 2013).

E’ possibile anche incapsulare una subquery dentro un’altra subquery:

* INDICARE IL NUMERO DI PAZIENTI DI SIENA, MAI VISITATI DA ORTOPEDICI

SELECT COUNT(\*)

FROM Paziente P

WHERE P.Citta = Siena

AND P.CodFiscale NOT IN (

SELECT V.Paziente

FROM Visita V



WHERE V.Medico IN (



SELECT M.Matricola



FROM Medico M



WHERE M.Specializzazione = ‘Ortopedia’

)



);

1. Prima trovo le matricole dei medici ortopedici;
2. Poi trovo i codici fiscali dei pazienti visitati da ortopedici (e cioè, i codici fiscali dei pazienti visitati da medici la cui matricola compare tra le matricole dei medici ortopedici)
3. Infine, conto i pazienti di Siena che non sono mai stati visitati da ortopedici (e cioè, il cui codice fiscale non compare tra quelli dei pazienti visitati da ortopedici)

Esistono poi le cosiddette **subquery scalari**, e cioè subquery (non correlated o correlated) che restituiscono come result set un unico record composto da un unico attributo.

Vediamo la distinzione tra le subquery scalari e le subquery “normali”:

**SUBQUERY SCALARE**

Produce come result set un unico record composto da un unico attributo.

Per confrontare un record della outer query col valore dell’unico attributo dell’unico record restituito dalla subquery scalare si usano =, <>.

Se questo valore fosse numerico, si possono usare anche <=, <, >, >=.

**SUBQUERY “NORMALE”**

Produce come result set un insieme di record.

Per confrontare un record della outer query con i record del result set della subquery si usano IN e NOT IN.

Nel caso in cui la subquery dovesse produrre un unico record, si possono usare al loro posto rispettivamente = e <>.

* INDICARE IL NUMERO DEGLI OTORINI AVENTI PARCELLA PIÙ ALTA DELLA MEDIA DELLE PARCELLE DEI MEDICI DELLA LORO SPECIALIZZAZIONE.

SELECT COUNT(\*)

FROM Medico M1

WHERE M1.Specializzazione = ‘Otorinolaringoiatria’

AND M1.Parcella > (



SELECT AVG(M.Parcella)

FROM Medico M2



WHERE M2.Specializzazione = ‘Otorinolaringoiatria’

);

1. Prima trovo la media delle parcelle dei medici otorini.
2. Poi conto il numero di medici otorini la cui parcella è maggiore della media delle parcelle dei medici otorini (e cioè, la cui parcella è maggiore del valore della parcella media degli otorini. Posso usare ‘>’, perché la subquery restituisce un unico record composto da un unico attributo numerico).

* INDICARE QUAL E’ IL REDDITO MASSIMO, E IL COGNOME DI CHI LO DETIENE.

SELECT DISTINCT P.Reddito, P.Cognome

FROM Paziente P

WHERE P.Reddito = (



SELECT MAX(P2.Reddito)



FROM Paziente P2

);

1. Prima trovo il massimo tra i redditi dei pazienti;
2. Poi trovo il reddito e il cognome dei pazienti di reddito massimo (e cioè, il cui reddito è uguale al valore del massimo tra i redditi dei pazienti. Posso usare ‘=’, perché la subquery restituisce un unico record composto da un unico attributo).

**CORRELATED SUBQUERY**

Una **correlated subquery** è una subquery il cui result set dipende dal record della outer query che è attualmente processato. Si incapsula nel WHERE della outer query, viene ricalcolata ogni volta che viene processato un nuovo record della outer query, e si può confrontare un record della outer query con quelli del result set della correlated subquery negli stessi modi visti per le non correlated subquery.

* INDICARE LA MATRICOLA DEI MEDICI CHE HANNO VISITATO PER LA PRIMA VOLTA ALMENO UN PAZIENTE NEL MESE DI OTTOBRE 2013

SELECT DISTINCT V1.Medico

FROM Visita V1



WHERE YEAR(V1.Data) = 2013



AND MONTH(V1.Data) = 10

AND V1.Paziente NOT IN (

SELECT V2.Paziente

FROM Visita V2



WHERE V2.Medico = V1.Medico

AND V2.Data < V1.Data

);

Per scrivere una query con correlated subquery, si parte dalla query più esterna (che è poi l’ordine di processazione di una query del genere):

1. Trovo le visite effettuate in Ottobre 2013…
2. …e impongo che una visita del genere deve finire nel risultato finale solo se è la prima volta che il medico ha visitato il paziente (e cioè, se il codice fiscale del paziente visitato non compare tra quelli dei pazienti visitati da *tale medico*, ma in *data antecedente*).

Infine, delle visite selezionate, proietto la matricola dei medici che le hanno effettuate (che quindi saranno medici che hanno visitato per la prima volta almeno un paziente nel mese di Ottobre 2013).

* INDICARE NOME E COGNOME DEI PAZIENTI CHE SONO STATI VISITATI DUE VOLTE DAL DOTTOR PAOLO VERDI

SELECT P.Nome, P.Cognome

FROM Paziente P



WHERE 2 = (

SELECT COUNT(\*)

FROM Visita V

INNER JOIN

Medico M ON V.Medico = M.Matricola



WHERE V.Paziente = P.CodFiscale

AND M.Nome = ‘Paolo’

AND M.Cognome = ‘Verdi’

);

1. Trovo i pazienti…
2. …e impongo che un paziente deve finire nel risultato finale solo se è stato visitato due volte dal dottor Paolo Verdi (e cioè, se 2 è uguale al numero di volte che *tale paziente* è stato visitato dal dottor Paolo Verdi).

Infine, dei pazienti selezionati, proietto nome e cognome.

Una correlated subquery può essere inserita anche nel SELECT, a patto che sia una query scalare.

* CONSIDERATO CIASCUN PAZIENTE DI SESSO MASCHILE, INDICARNE IL NOME E IL NUMERO DI VISITE EFFETTUATE

SELECT P.Nome, (SELECT COUNT(\*)

FROM Visita V

WHERE V.Paziente = P.CodFiscale) AS TotaleVisite

FROM Paziente P

WHERE P.Sesso = ‘M’;

Trovo i pazienti di sesso maschile, e di ogni paziente proietto il nome e il numero di visite che ha effettuato (e cioè, quante visite *tale paziente* ha effettuato).

Solo per le correlated subquery è possibile utilizzare il costrutto **EXISTS/NOT EXISTS:**

* Usando **EXISTS**,un record della outer query finirà nel risultato finale solo se il result set della correlated subquery contiene almeno un record;
* Usando **NOT EXISTS**,un record della outer query finirà nel risultato finale solo se il result set della correlated subquery non contiene alcun record.
* UNA VISITA DI CONTROLLO È UNA VISITA IN CUI UN MEDICO VISITA UN PAZIENTE GIÀ VISITATO PRECEDENTEMENTE ALMENO UNA VOLTA. INDICARE MEDICO, PAZIENTE E DATA DELLE VISITE DI CONTROLLO DEL MESE DI GENNAIO 2016

SELECT V1.Medico, V1.Paziente, V1.Data

FROM Visita V1

WHERE YEAR(V1.Data) = 2016



AND MONTH(V1.Data) = 1

AND EXISTS (

SELECT \*

FROM Visita V2

WHERE V2.Medico = V1.Medico



AND V2.Paziente = V1.Paziente

AND V2.Data < V1.Data

);

1. Trovo le visite fatte a Gennaio 2016…
2. …e impongo che una visita deve finire nel risultato finale solo se il medico ha già visitato il paziente precedentemente almeno una volta (e cioè, se esiste almeno una visita fatta da *tale medico*, a *tale paziente,* ma in *data antecedente*).

Infine, delle visite selezionate, proietto il medico, il paziente e la data.

* INDICARE LA MATRICOLA DEI MEDICI CHE HANNO VISITATO PER LA PRIMA VOLTA ALMENO UN PAZIENTE NEL MESE DI OTTOBRE 2013

SELECT DISTINCT V1.Medico

FROM Visita V1

WHERE YEAR(V1.Data) = 2013



AND MONTH(V1.Data) = 10

AND NOT EXISTS (

SELECT \*

FROM Visita V2

WHERE V2.Medico = V1.Medico



AND V2.Paziente = V1.Paziente

AND V2.Data < V1.Data

);

1. Trovo le visite fatte a Ottobre 2013…
2. …e impongo che una visita deve finire nel risultato finale solo se è la prima volta che il medico ha visitato il paziente (e cioè, se non esiste alcuna visita fatta da *tale* *medico*, a *tale paziente*, ma in *data antecedente*).

Infine, delle visite selezionate, proietto la matricola dei medici che le hanno effettuate (che quindi saranno medici che hanno visitato per la prima volta almeno un paziente nel mese di Ottobre 2013).

**DIVISIONE**

La **divisione** è un operatore insiemistico derivato utile a risolvere query che presentano nel testo espressioni come “*tutti*”.

Su MySQL, possiamo esprimerla attraverso un doppio NOT EXISTS, oppure con il raggruppamento.

* INDICARE I PAZIENTI VISITATI DA TUTTI I MEDICI

CON DOPPIO NOT EXISTS:

SELECT P.CodFiscale

FROM Paziente P



WHERE NOT EXISTS (

SELECT \*

FROM Medico M

WHERE NOT EXISTS (

SELECT \*

FROM Visita V



WHERE V.Medico = M.Matricola

AND V.Paziente = P.CodFiscale

)

);

1. Trovo i pazienti…
2. …e impongo che un paziente deve finire nel risultato finale solo se non esiste un medico che non l’abbia mai visitato (e cioè, se non esiste un medico per il quale non esiste una visita fatta da *tale medico* a *tale paziente*). In questo modo, finiranno nel risultato finale solo pazienti visitati da tutti i medici.

Infine, dei pazienti selezionati, proietto il codice fiscale.

CON RAGGRUPPAMENTO:

SELECT V.Paziente

FROM Visita V

GROUP BY V.Paziente

HAVING COUNT(DISTINCT V.Medico) = (

SELECT COUNT(\*)

FROM Medico M

);

Raggruppo le visite per paziente visitato, e lascio i gruppi in cui il paziente è stato visitato da tutti i medici della clinica (e cioè, i gruppi in cui il numero di medici distinti che hanno visitato il paziente è uguale al numero di medici della clinica).

Di ogni gruppo rimasto, infine, proietto il paziente visitato (che quindi sarà un paziente visitato da tutti i medici della clinica).

**DIFFERENZA E MODIFICATORI ANY E ALL**

Vedi da raccoglitore, pag. 2 09/04/21

**STORED PROCEDURE**

Le **stored procedure** sono programmi dichiarativo-procedurali memorizzati nel DBMS. Si possono invocare tramite chiamata e possono prendere in ingresso e/o restituire valori.

Vediamo la sintassi base per creare una stored procedure con un esempio:

* CREARE UNA STORED PROCEDURE CHE STAMPI A VIDEO LE SPECIALIZZAZIONI MEDICHE OFFERTE DALLA CLINICA

Elimino la procedure se già esiste (altrimenti, se già esiste, non la fa creare)

DROP PROCEDURE IF EXISTS mostra\_specializzazioni;

Cambio il delimiter in ‘$$’.

Un delimiter è una sequenza di simboli che, quando viene letta dal DBMS, questo compila tutto il codice compreso tra il precedente delimiter e quello appena letto.

Di default, il delimiter è ‘;’. C’è però un problema: il corpo di una stored procedure può essere costituito da tanti statement, ognuno terminato da ‘;’. Noi, tuttavia, vogliamo che la stored procedure venga compilata tutta insieme, e non uno statement alla volta (altrimenti si avrebbero errori). Serve allora cambiare temporaneamente il delimiter.

Tuttavia

DELIMITER $$

Creo la stored procedure.

* Tra parentesi vanno specificati i parametri d’ingresso/uscita della stored procedure: in questo caso, non ce ne sono.
* Tra BEGIN e END va specificato il corpo della stored procedure. In questo caso, la procedure stampa a video le specializzazioni della clinica (*select* equivale a stampare a video).
* Dopo END va messo il delimiter che si è temporaneamente scelto. In questo modo, quando il DBMS lo leggerà, compilerà la stored procedure.

CREATE PROCEDURE mostra\_specializzazioni()

BEGIN

SELECT DISTINCT M.Specializzazione

FROM Medico M;

END $$

Rimetto il delimiter di default.

DELIMITER ;

Eseguendo questo codice, la stored procedure verrà compilata e salvata nel DBMS.   
Per invocarla, si usa CALL *nome\_procedura()*:

CALL mostra\_specializzazioni()

**VARIABILI**

Una **variabile** viene usata per memorizzare informazioni. Può contenere un solo valore in un certo istante, non più valori.

**VARIABILI LOCALI**

Le **variabili locali** sono usate all’interno di uno stored program per memorizzare informazioni intermedie. Essendo *locali*, non sono visibili all’esterno dello stored program.

Vanno dichiarate tutte insieme all’inizio del body dello stored program, in questo modo:

DECLARE *nome\_variabile* *tipo* DEFAULT *valore\_default*;

Sono due le istruzioni possibili per assegnare un valore ad una variabile: SET o SELECT + INTO.

* SUPPORRE DI ESSERE NEL BODY DI UNA STORED PROCEDURE E CREARE UNA VARIABILE CONTENENTE IL MINIMO NUMERO DI VISITE MENSILI DA EFFETTUARE, IMPOSTATO A 20.

DECLARE min\_visite\_mensili INT DEFAULT 0;

⁝

SET min\_visite\_mensili = 20;

Questo è un esempio di **assegnamento “statico”:** si assegna alla variabile il valore indicato.

*oppure*

DECLARE min\_visite\_mensili INT DEFAULT 0;

⁝

SELECT 20 INTO min\_visite\_mensili;

* SUPPORRE DI ESSERE NEL BODY DI UNA STORED PROCEDURE E CREARE UNA VARIABILE CONTENENTE IL NUMERO DI VISITE EFFETTUATO NEL MESE IN CORSO

DECLARE visite\_mese\_attuale INT DEFAULT 0;

⁝

SET visite\_mese\_attuale = (

SELECT (\*)

FROM Visita V

WHERE MONTH(V.Data) = MONTH(CURRENT\_DATE)

AND YEAR(V.Data) = YEAR(CURRENT\_DATE)

);

Questo è un esempio di **assegnamento “calcolato”:** si assegna alla variabile il valore calcolato dalla query indicata.

*oppure*

DECLARE visite\_mese\_attuale INT DEFAULT 0;

⁝

SELECT COUNT(\*) INTO visite\_mese\_attuale

FROM Visita V

WHERE MONTH(V.Data) = MONTH(CURRENT\_DATE)

AND YEAR(V.Data) = YEAR(CURRENT\_DATE);

**VARIABILI USER-DEFINED**

Le variabili **user-defined** sono variabili che possono essere inizializzate dall’utente senza necessità di dichiarazione e rimangono in vita finché l’utente che le ha inizializzate non si disconnette dal MySQL server.

Il loro nome deve iniziare per @, sono senza tipo (possono contenere qualsiasi tipo di dato, anche di tipi diversi in istanti diversi) e sono visibili ovunque, ma solo dall’utente che le ha inizializzate.

**PARAMETRI DI UNA STORED PROCEDURE** *04-05-23*

Una stored procedure accetta due tipi di parametri:

* **Parametri in ingresso** (si specificano premettendo *IN* al nome del parametro stesso)
* **Parametri in uscita** (si specificano premettendo *OUT* al nome del parametro stesso)

Passando una variabile come parametro d’ingresso, la stored procedure può solo leggere il suo valore, ma non può modificarlo (equivale quindi ad un passaggio per valore). Si può passare direttamente un valore come parametro d’ingresso (e cioè, non una variabile con associato un valore).

Passando una variabile come parametro d’uscita, la stored procedure può modificare il suo valore (equivale quindi ad un passaggio per riferimento).

Di ogni parametro, poi, va specificato il tipo, posponendolo al nome del parametro stesso.

* SCRIVERE UNA STORED PROCEDURE CHE RESTITUISCA IL NUMERO DI PAZIENTI VISITATI DA MEDICI DI UNA DATA SPECIALIZZAZIONE, RICEVUTA COME PARAMETRO

DROP PROCEDURE IF EXISTS tot\_pazienti\_visitati\_spec

DELIMITER $$

CREATE PROCEDURE tot\_pazienti\_visitati\_spec(IN \_specializzazione VARCHAR(100),

OUT totale\_pazienti\_ INT)

BEGIN

SELECT COUNT(DISTINCT V.Paziente) INTO totale\_pazienti\_

FROM Visita V

INNER JOIN

Medico M ON V.Medico = M.Matricola

WHERE M.Specializzazione = \_specializzazione;

END $$

DELIMITER ;

CALL tot\_pazienti\_visitati\_spec(‘Ortopedia’, @quantiPazienti);

Creo la stored procedure *tot\_pazienti\_visitati\_spec* che:

* + prende in ingresso:
    - come parametro d’ingresso *\_specializzazione*, una specializzazione;
  + e restituisce:
    - sulla variabile passata come parametro d’uscita *totale\_pazienti\_*, il numero di pazienti visitati da medici della specializzazione *\_specializzazione*.

Nel corpo della stored procedure, conto i pazienti distinti che sono stati visitati da medici che hanno la specializzazione uguale a quella passata come parametro d’ingresso *\_specializzazione*  
e metto questo valore in *totale\_pazienti\_*.

Invocandola poi con *CALL tot\_pazienti\_visitati\_spec(‘Ortopedia’, @quantiPazienti)*, verrà passato *Ortopedia* come parametro d’ingresso \_*specializzazione*, *@quantiPazienti* come parametro d’uscita *totale\_pazienti\_*,   
e così in *@quantiPazienti* finirà il numero di pazienti visitati da medici ortopedici.

* SCRIVERE UNA STORED PROCEDURE CHE RESTITUISCA LA DATA IN CUI UN PAZIENTE, IL CUI CODICE FISCALE E’ PASSATO COME PARAMETRO, E’ STATO VISITATO PER LA PRIMA VOLTA, E IL NOME E COGNOME DEL MEDICO CHE LO HA VISITATO IN TALE CIRCOSTANZA. IN CASO DI PIU’ MEDICI, PER SEMPLICITA’ SELEZIONARNE UNO.

DROP PROCEDURE IF EXISTS primaVisitaPaziente

DELIMITER $$

CREATE PROCEDURE primaVisitaPaziente(IN \_paziente VARCHAR(16),

OUT dataPrimaVisita\_ DATE,

OUT nomeMedico\_ VARCHAR(100),

OUT cognomeMedico\_ VARCHAR(100))

BEGIN

SELECT MIN(V.Data) INTO dataPrimaVisita\_

FROM Visita V

WHERE V.Paziente = \_paziente;

SELECT M.Nome, M.Cognome INTO nomeMedico\_, cognomeMedico\_

FROM Visita V



Con “SELECT *att1*, *att2*, …, *attN*INTO *att’1*, *att’2*, …, *att’N*”   
si ha che:

* Il valore di *att1* viene assegnato ad *att’1*;
* Il valore di *att2* viene assegnato ad *att’2*;
* …
* Il valore di *attN* viene assegnato ad *att’N*.

INNER JOIN

Medico M ON V.Medico = M.Matricola

WHERE V.Paziente = \_paziente,

AND V.Data = dataPrimaVisita\_

LIMIT 1;

END $$

DELIMITER ;

Creo la stored procedure *tot\_pazienti\_visitati\_spec* che:

* + prende in ingresso:
    - come parametro d’ingresso *\_paziente*, il codice fiscale di un paziente;
  + e restituisce:
    - sulla variabile passata come parametro d’uscita *dataPrimaVisita\_*, la data in cui il paziente con codice fiscale *\_paziente* è stato visitato per la prima volta;
    - sulla variabile passata come parametro d’uscita *nomeMedico\_*, il nome del medico che ha visitato il paziente con codice fiscale *\_paziente* per la prima volta;
    - sulla variabile passata come parametro d’uscita *cognomeMedico\_*, il cognome del medico che ha visitato il paziente con codice fiscale *\_paziente* per la prima volta;

Nel corpo della stored procedure, trovo innanzitutto la data più remota nella quale è stato visitato il paziente con il codice fiscale passato come parametro d’ingresso *\_paziente*: questa sarà la data in cui tale paziente è stato visitato per la prima volta, e quindi assegno questo valore a *dataPrimaVisita\_*.

A questo punto, trovo nome e cognome del medico che ha visitato il paziente con il codice fiscale passato come parametro d’ingresso *\_paziente* in *dataPrimaVisita*\_ (e quindi, nella data in cui tale paziente è stato visitato per la prima volta), e metto questi valori rispettivamente in *nomeMedico\_* e *cognomeMedico\_*.

La keyword **LIMIT *x*** serve a limitare il result set di una query a massimo *x* record. Infatti, nel nostro caso, possono esistere più medici che hanno visitato tale paziente in tale data, ma non possiamo assegnare tutti i loro nomi a *nomeMedico\_* e tutti i loro cognomi a *cognomeMedico\_*, perché una variabile (e anche un parametro) può contenere un solo valore in un certo istante, e non più valori. Quello che facciamo, allora, è usare la keyword LIMIT 1, così che questa query restituisca un solo record: in questo modo, ci sarà un solo valore per *M.Nome* e un solo valore per *M.Cognome*, e quindi correttamente verrà assegnato un solo nome a *nomeMedico\_* e un solo cognome a *cognomeMedico\_*.

**ISTRUZIONI CONDIZIONALI**

Le **istruzioni condizionali** permettono di esprimere condizioni, modificando il flusso di esecuzione.

**ISTRUZIONE IF**

Equivale all’*IF* del C++. La sintassi è la seguente:

*IF if\_condition* THEN

*blocco istruzioni if true*

ELSEIF *elseif\_1\_condition* THEN

*blocco istruzioni elseif\_1*

⁝

Parte facoltativa

ELSEIF *elseif\_N\_condition* THEN



*blocco istruzioni elseif\_N*

ELSE

*blocco istruzioni else*

END IF ;

* SCRIVERE UNA STORED PROCEDURE CHE RICEVA COME PARAMETRO UN INTERO *t* E UNA SPECIALIZZAZIONE *s* E RESTITUISCA IN USCITA *true* SE IL NUMERO DI VISITE DELLA   
  SPECIALIZZAZIONE *s* NEL MESE IN CORSO È SUPERIORE A *t*, *false* SE È INFERIORE, *NULL* SE È UGUALE.

DROP PROCEDURE IF EXISTS visite\_sopra\_soglia

DELIMITER $$

CREATE PROCEDURE visite\_sopra\_soglia(IN \_t INT, IN \_s VARCHAR(100), OUT passed\_ BOOLEAN)

BEGIN

DECLARE visite\_mese\_attuale INT DEFAULT 0;

SET visite\_mese\_attuale = (

SELECT COUNT(\*)

FROM Visita V

INNER JOIN

Medico M ON V.Medico = M.Matricola

WHERE M.Specializzazione = \_s;

AND YEAR(V.DATA) = YEAR(CURRENT\_DATE)

AND MONTH(V.DATA) = MONTH(CURRENT\_DATE)

);

IF visite\_mese\_attuale > \_t THEN

SET passed\_ = true;

ELSEIF visite\_mese\_attuale < \_t THEN

SET passed = false;

ELSE

SET passed = NULL;

END IF;

END $$

DELIMITER ;

CALL visite\_sopra\_soglia(10, ‘Otorinolaringoiatria’, @controllo);

Creo la stored procedure *visite\_sopra\_soglia* che:

* + prende in ingresso:
    - come parametro d’ingresso *\_t*, un intero;
    - come parametro d’ingresso *\_s*, una specializzazione;
  + e restituisce:
    - sulla variabile passata come parametro d’uscita *passed\_*, *true* se il numero di visite della specializzazione *\_s* nel mese in corso è superiore a *\_t*, *false* se è inferiore, *null* se è uguale.

Nel corpo della stored procedure, conto il numero di visite effettuato da medici della specializzazione \_s nel mese in corso e metto questo valore nella variabile *visite\_mese\_attuale\_*.

Faccio poi un IF:

* + Se il numero di visite effettuato da medici della specializzazione *\_s* nel mese in corso è superiore a *\_t*, allora assegno *true* a *passed*;
  + Se invece il numero di visite effettuato da medici della specializzazione \_s nel mese in corso è inferiore a *\_t*, allora assegno *false* a *passed*;
  + Altrimenti, significa che il numero di visite effettuato da medici della specializzazione *\_s* nel mese in corso è uguale a *\_t*. In questo caso, allora, assegno *NULL* a *passed*;

Invocandola poi con *CALL visite\_sopra\_soglia(10, ‘Otorinolaringoiatria’, @controllo)*, verrà passato *10* come parametro d’ingresso *\_t*, *‘Otorinolaringoiatria’* come parametro d’ingresso \_*s*, *@controllo* come parametro d’uscita *passed,* e così in *@controllo* finirà il risultato restituito dalla stored procedure.

**ISTRUZIONE CASE**

Equivale all’istruzione *SWITCH* del C++. La sintassi è la seguente:

CASE

WHEN *condition\_1* THEN

*blocco istruzioni\_1*

⁝

WHEN *condition\_N* THEN

*blocco istruzioni\_N*

END CASE ;

**ISTRUZIONI ITERATIVE**

Le **istruzioni iterative** permettono di ripetere blocchi di codice, dipendentemente dalla veridicità di una condizione.

**ISTRUZIONE WHILE**

La sintassi è la seguente:

WHILE *condition* DO

Il *blocco istruzioni* viene eseguito finché la *condition* è vera. La *condition* viene valutata *prima* di ogni iterazione.

*blocco istruzioni*

END WHILE ;

**ISTRUZIONE REPEAT**

La sintassi è la seguente:

REPEAT

Il *blocco istruzioni* viene eseguito fintantoché   
non si verifica la *condition*. La *condition* viene valutata *dopo* ogni iterazione.

*blocco istruzioni*

UNTIL

*condition*

END REPEAT ;

**ISTRUZIONE LOOP**

La sintassi è la seguente:

Il *blocco istruzioni* viene eseguito fintantoché   
non si verifica una delle *condition* che il programmatore ha specificato all’interno del blocco stesso.

*loop\_label*: LOOP

*blocco istruzioni* *con check di condition*

END LOOP ;

Per uscire da un LOOP o passare all’iterazione successiva di un LOOP si usano rispettivamente le istruzioni di salto **LEAVE *loop\_label*** e **ITERATE *loop\_label***.

Negli esercizi d’esame useremo sempre LOOP. Vediamo un esempio:

* SCRIVERE UNA STORED PROCEDURE CHE RICEVE IN INGRESSO UN INTERO *i* E STAMPA A VIDEO I NUMERI DISPARI DA 1 A *i*, SEPARATI DA VIRGOLA

DROP PROCEDURE IF EXISTS stampa\_dispari

DELIMITER $$

CREATE PROCEDURE stampa\_dispari(IN \_i INT)

BEGIN

DECLARE output VARCHAR(200) DEFAULT ‘1’;

DECLARE counter INT DEFAULT 3;

scan : LOOP

IF counter > \_i THEN

LEAVE scan;

ELSE

SET output = CONCAT(output, ‘,’, counter);

SET counter = counter + 2;

END IF;

END LOOP;

SELECT output;

END $$

DELIMITER ;

Creo la stored procedure *stampa\_dispari* che:

* + prende in ingresso:
    - come parametro d’ingresso *\_i*, un intero, supponendo sia >= 1.

Nel corpo della stored procedure, dichiaro le seguenti variabili:

* + *output,* che conterrà la stringa costituita da tutti i numeri dispari da 1 a \_*i*, separati da virgola. Inizialmente, contiene la stringa ‘1’ (infatti, supposto che *\_i* sia >= 1, 1 è sicuramente il primo numero dispari compreso tra 1 e *i*);
  + *counter,* che userò nel successivo loop per scorrere tutti i numeri dispari da 1 ad *i*.   
    Inizialmente, vale 3 (il primo numero dispari dopo 1);

Entro ora in un loop: partendo da *counter* = 3, fintantoché *counter* non supera *i*, concateno il suo valore a *output* separato da una virgola, faccio avanzare *counter* al prossimo numero dispari e passo alla prossima iterazione del loop.

In questo modo, usciti dal loop, *output* conterrà la stringa costituita da tutti i numeri dispari da 1 a *i* separati da virgola. Stampo allora questa stringa.

**CONCAT(…)** è una funzione che restituisce la concatenazione delle stringhe passate come parametri   
(non c’è limite al numero di parametri che può prendere in ingresso).

**CURSORI**

Un **cursore** permette di scorrere in avanti i record di un result set.

Si dichiara dopo la dichiarazione di tutte le variabili, così:

In questo modo, dichiaro un cursore *nome\_cursore* che permette di scorrere in avanti il result set restituito da *query*.

DECLARE *nome\_cursore* CURSOR FOR

*query*;

Un cursore si usa in questo modo:

* Innanzitutto bisogna aprirlo attraverso l’istruzione **OPEN *nome\_cursore***. Una volta aperto, il cursore punta inizialmente al primo record del result set per cui è stato dichiarato;
* Attraverso l’istruzione **FETCH *nome\_cursore* INTO *lista\_variabili*,** il valore di ogni attributo del record attualmente puntato dal cursore viene messo in *lista\_variabili*. Fatto ciò, il cursore avanza al record successivo del result set;
* Una volta finito di usarlo, va chiuso attraverso l’istruzione **CLOSE *nome\_cursore***.

**HANDLER**

Un **handler** serve (tra le varie cose) a gestire la situazione in cui un cursore è giunto alla fine del result set.

Si dichiara dopo la dichiarazione di tutti i cursori, così:

Innanzitutto:

* *CONTINUE HANDLER*: è un handler la cui esecuzione non comporta l’abort dello stored program;
* *FOR NOT FOUND*: l’handler viene eseguito quando, andando ad eseguire l’istruzione FETCH *nome\_cursore* INTO *lista\_variabili*, non viene prelevato alcun valore. Questo succede quando il cursore *nome\_cursore* è giunto alla fine del result set: è avanzato a tal punto da superare anche l’ultimo record, non puntando più a nulla. Andando allora ad effettuare il prelievo, non verrà prelevato nulla.

In questo modo, dunque, dichiaro un continue handler che, quando un cursore è arrivato alla fine del result set, assegna 1 alla variabile locale *finito* (che chiaramente andrà dichiarata con un valore iniziale diverso da 1)

DECLARE CONTINUE HANDLER FOR NOT FOUND

SET *finito* = 1;

* SCRIVERE UNA STORED PROCEDURE CHE RICEVE IN INGRESSO UNA SPECIALIZZAZIONE *s*   
  E RESTITUISCA I CODICI FISCALI DEI PAZIENTI VISITATI DA UN SOLO MEDICO DI *s*, IN UNA STRINGA DEL TIPO “CodFiscale1, ..., CodFiscaleN”

DROP PROCEDURE IF EXISTS PazientiSingoloMedico

DELIMITER $$

CREATE PROCEDURE PazientiSingoloMedico(IN \_s VARCHAR(20),

OUT codiciFiscali\_ VARCHAR(200))

BEGIN

DECLARE codiceFiscale VARCHAR(200) DEFAULT ‘’;

DECLARE finito INT DEFAULT 0;

DECLARE cursoreCodici CURSOR FOR

SELECT V.Paziente

FROM Visita V

INNER JOIN

Medico M ON V.Medico = M.Matricola

WHERE M.Specializzazione = \_s

GROUP BY V.Paziente

HAVING COUNT(DISTINCT V.Medico) = 1;

DECLARE CONTINUE HANDLER FOR NOT FOUND

SET finito = 1;

SET codiciFiscali\_ = ‘’;

OPEN cursoreCodici;

preleva : LOOP

FETCH cursoreCodici INTO codiceFiscale;

IF finito = 1 THEN

LEAVE preleva;

ELSE

SET codiciFiscali\_ = CONCAT(codiciFiscali\_, ‘,’, codiceFiscale);

END IF;

END LOOP;

CLOSE cursoreCodici;

END $$

DELIMITER ;

Creo la stored procedure *PazientiSingoloMedico* che:

* + prende in ingresso:
    - come parametro d’ingresso *\_s*, una specializzazione;
  + e restituisce:
    - sulla variabile passata come parametro d’uscita *codiciFiscali\_*, una stringa contenente i codici fiscali dei pazienti visitati da un solo medico di *\_s*, nel formato richiesto.

Nel corpo della stored procedure, dichiaro innanzitutto le seguenti variabili:

* + Una variabile d’appoggio *codiceFiscale*;
  + La variabile *finito* usata dall’handler, che inizializzo a 0.

Dichiaro poi un cursore *cursoreCodici* per scorrere il result set contenente i codici fiscali dei pazienti visitati da un solo medico di *\_s*. Subito dopo, dichiaro l’handler.

A questo punto, apro il cursore *cursoreCodici* e entro in un loop: provo a prelevare il codice fiscale attualmente puntato dal cursore e a metterlo nella variabile d’appoggio *codiceFiscale*:

* + se, a seguito di ciò, *finito = 1*, allora l’handler mi ha segnalato che non ho prelevato nulla perché il cursore è arrivato alla fine del result set, quindi esco dal loop;
  + altrimenti, significa che ho prelevato effettivamente un codice fiscale e l’ho messo in *codiceFiscale*. Concateno allora *codiceFiscale* a *codiciFiscali\_* separato da virgola e passo alla prossima iterazione (notare che prima del loop ho assegnato la stringa vuota a *codiciFiscali\_*. Questo l’ho fatto perché, se il chiamante passasse come parametro d’uscita *codiciFiscali\_* una variabile contenente una stringa non vuota, effettuerei le concatenazioni su questa stringa, e quindi *codiciFiscali\_* conterrebbe i codici fiscali + altra roba all’inizio che è inutile).

In questo modo, usciti dal loop*, codiciFiscali\_* conterrà correttamente i codici fiscali dei pazienti visitati da un solo medico di \_*s*, in una stringa del tipo “CodFiscale1, ..., CodFiscaleN”.

Alla fine, chiudo il cursore *cursoreCodici.*

**DATA MANIPULATION**

La **Data Manipulation** è la parte del linguaggio MySQL che permette di inserire, cancellare e aggiornare interi record o valori di attributi delle tabelle di un database.

**INSERIMENTO: INSERT**

Considerata una tabella, **INSERT** permette di inserire nuovi record i cui valori degli attributi possono essere statici (specificati dall’utente) o calcolati (attraverso una query).

La sintassi è la seguente:

Questa è la sintassi per un inserimento con **valori statici**: inserisco all’interno di *Tabella* un nuovo record cui assegno:

* *Valore1* ad *Attributo1*;
* *Valore2* ad *Attributo2*;
* …
* *ValoreN* ad *AttributoN*.

Se ad un certo attributo non assegno alcun valore, questo assumerà il suo *default value* (valore specificato in fase di creazione di *Tabella*).

Se invece assegno un valore ad ogni attributo, si possono non indicare gli attributi ai quali assegno un valore ed indicare semplicemente i valori da assegnare agli attributi, nell’ordine in cui questi compaiono nello schema di *Tabella*.

Attributi ai quali assegno un valore

Tabella in cui inserire un nuovo record



INSERT INTO *Tabella\_Target*(*Attributo1*, *Attributo2*, …, *AttributoN*)

VALUES(*Valore1, Valore2*, …, *ValoreN*);

Valori da assegnare agli attributi

Questa è la sintassi per un inserimento con **valori calcolati**:   
per ogni record del result set della *query*, inserisco all’interno di *Tabella* un nuovo record cui assegno:

* Il valore del primo attributo proiettato nella *query* ad *Attributo1*;
* il valore del secondo attributo proiettato nella *query* ad *Attributo2*;
* …
* il valore dell’ultimo attributo proiettato nella *query* ad *AttributoN*.

Se ad un certo attributo non assegno alcun valore, questo assumerà il suo *default value*.

Se invece assegno un valore ad ogni attributo, si possono non indicare gli attributi ai quali assegno un valore ed indicare semplicemente la *query*, e l’assegnamento seguirà l’ordine in cui gli attributi compaiono nello schema di *Tabella* (e cioè, per ogni record del result set della *query*, verrà inserito all’interno di *Tabella* un nuovo record cui verrà assegnato:

* il valore del primo attributo proiettato nella *query* al primo attributo che compare nello schema di *Tabella*;
* il valore del secondo attributo proiettato nella *query* al secondo attributo che compare nello schema di *Tabella*;
* …;
* il valore dell’ultimo attributo proiettato nella *query* all’ultimo attributo che compare nello schema di *Tabella*.

**NB**: la proiezione della *query* deve essere sempre costituita dallo stesso numero di attributi ai quali assegno un valore.

Attributi ai quali assegno un valore

Tabella in cui inserire nuovi record



INSERT INTO *Tabella\_Target*(*Attributo1*, *Attributo2*, …, *AttributoN*)

*query*

Query che restituisce i valori da assegnare agli attributi



* INSERIRE NEL DATABASE UN NUOVO PAZIENTE, LE CUI INFORMAZIONI SONO:
  + Nome: Elvira
  + Cognome: Passerotti
  + Sesso: F
  + Data di Nascita: 27 Ottobre 1965
  + Città: Pisa
  + Reddito: 1500
  + Codice Fiscale: PSSLVR65R67G702U

INSERT INTO Paziente

VALUES(‘PSSLVR65R67G702U’, ‘Passerotti’, ‘Elvira’, ‘F’, ‘1965-10-27’, ‘Pisa’, 1500);

Inserisco nella tabella *Paziente* un paziente che presenta i valori indicati.

Non ho indicato gli attributi ai quali assegno un valore visto che sto assegnando un valore ad ogni attributo, e quindi ho semplicemente indicato i valori da assegnare agli attributi, nell’ordine in cui questi compaiono nello schema di *Paziente.*

* TRE GIORNI FA, LA DOTTORESSA CLELIA CELESTI HA VISITATO IL PAZIENTE EDOARDO LEPRE,   
  CODICE FISCALE ‘slq6’. LA VISITA NON È STATA INSERITA. LO SI VUOLE FARE IN RITARDO MA NON SI SA PIÙ SE LA VISITA ERA MUTUATA

INSERT INTO Visita(Medico, Paziente, Data)

VALUES(010, ‘slq6’, CURRENT\_DATE - INTERVAL 3 DAY);

Inserisco nella tabella *Visita* una visita che presenta i valori indicati.

Ho indicato gli attributi ai quali assegno un valore visto che c’è l’attributo *Mutuata* al quale non sto assegnando alcun valore (e quindi assumerà il suo *default value*).

* CONSIDERATA LA TABELLA *VisitaVecchia* AVENTE LO SCHEMA RIPORTATO SOTTO, INSERIRE IN *VisitaVecchia* TUTTE LE VISITE EFFETTUATE PRIMA DI DUE ANNI FA.

VisitaVecchia(CognomePaziente, CognomeMedico, DataVisita, Specializzazione)

INSERT INTO VisitaVecchia

SELECT P.Cognome, M.Cognome, V.Data, M.Specializzazione

FROM Visita V

INNER JOIN

Paziente P ON V.Paziente = P.CodFiscale

INNER JOIN

Medico M ON V.Medico = M.Matricola

WHERE V.Data < YEAR(CURRENT\_DATE) – 2;

Innanzitutto, di ogni visita effettuata prima di due anni fa, trovo il cognome del paziente, il cognome del medico, la data della visita e la specializzazione del medico.

A questo punto, per ogni record di questo result set, aggiungo un nuovo record nella tabella *VisitaVecchia* cui assegno il valore di *P.Cognome* a *CognomePaziente*; il valore di *M.Cognome* a *CognomeMedico*; il valore di *V.Data* a *DataVisita* e il valore di *M.Specializzazione* a *Specializzazione*

(non ho indicato gli attributi ai quali assegno un valore visto che sto assegnando un valore ad ogni attributo, e quindi l’assegnamento seguirà l’ordine in cui gli attributi compaiono nello schema di *VisitaVecchia*).

**AGGIORNAMENTO: UPDATE**

Considerata una tabella, **UPDATE** permette di modificare il valore di uno o più attributi di uno o più record con valori statici (specificati dall’utente) o calcolati (attraverso una query).

La sintassi è la seguente:



Tabella in cui effettuare la modifica

Attributi da modificare, con indicato il loro nuovo valore.

UPDATE *Tabella\_Target*

In questo modo, aggiorno *Tabella*, modificando nel modo indicato tutti i record che rispettano la *condizione* indicata.

I *valori* possono essere statici (specificati dall’utente) o calcolati (attraverso una query)



SET *Attributo1* = *Valore1*, *Attributo2* = *Valore2*, …, *AttributoN* = *ValoreN*

WHERE *condizione*;

Condizione per selezionare i record da modificare



* MODIFICARE IN “MUTUATA” TUTTE LE VISITE DEL MESE CORRENTE, EFFETTUATE DA PAZIENTI NATI PRIMA DEL 1925

UPDATE Visita V

SET V.Mutuata = 1

WHERE MONTH(V.Data) = MONTH(CURRENT\_DATE)

AND YEAR(V.Data) = YEAR(CURRENT\_DATE)

AND V.Paziente IN (

SELECT P.CodFiscale

FROM Paziente P

WHERE YEAR(P.DataNascita) < 1925

);

Aggiorno la tabella *Visita*, modificando in “mutuata” tutte le visite del mese corrente effettuate da pazienti nati prima del 1925 (e cioè, il cui codice fiscale compare tra i pazienti nati prima nel 1925).

**CANCELLAZIONE: DELETE**

Considerata una tabella, **DELETE** permette di cancellare uno o più record.

La sintassi è la seguente:



Tabella in cui effettuare la cancellazione

DELETE FROM *Tabella\_Target*

In questo modo, cancello da *Tabella* tutti i record che rispettano la *condizione* indicata.

WHERE *condizione*;



Condizione per selezionare i record da cancellare

* IL DOTTOR ETTORE GRIGI È MORTO. RIMUOVERE TUTTE LE SUE VISITE DAL DATABASE, SUPPONENDO CHE IL MEDICO NON AVESSE OMONIMI NELLA CLINICA.

DELETE FROM Visita V

WHERE V.Medico = (

SELECT M.Matricola

FROM Medico M

WHERE M.Nome = ‘Ettore’

AND M.Cognome = ‘Grigi’

);

Cancello dalla tabella *Visita* tutte le visite fatte dal medico Ettore Grigi (e cioè, le visite in cui la matricola del medico è uguale alla matricola di Ettore Grigi, supponendo che tale medico non avesse omonimi nella clinica).

**RIFERIMENTO ALLA TABELLA TARGET NEL WHERE DI UNA UPDATE/DELETE**

Quando effettuiamo una UPDATE/DELETE, non si può mettere la tabella target nel FROM della query contenuta nella clausola WHERE. Questo è dovuto al fatto che una UPDATE/DELETE, durante l’esecuzione, blocca l’accesso in lettura e scrittura alla tabella target a tutti gli altri programmi (e quindi, anche alla query contenuta nella clausola WHERE).

Non si può quindi fare una cosa del genere:

* RIMUOVERE DAL DATABASE TUTTI I MEDICI DI PISA CHE NON HANNO EFFETTUATO VISITE MUTUATE IL MESE SCORSO

DELETE FROM Medico M



WHERE M.Matricola NOT IN (



SELECT V.Medico

FROM Visita V

INNER JOIN

Medico M2 ON V.Medico = M2.Matricola

WHERE M2.Citta = ‘Pisa’

AND V.Mutuata = 1

AND MONTH(V.Data) = MONTH(CURRENT\_DATE – INTERVAL 1 MONTH)

);

Una soluzione è mettere la query contenuta nella clausola WHERE dentro una derived table (e quindi scrivere “SELECT\* FROM(*query*) AS D”). Questo funziona, perché la derived table viene calcolata per prima, dunque anche prima di eseguire la UPDATE/DELETE, e quindi non si hanno problemi.

DELETE FROM Medico M

WHERE M.Matricola NOT IN (



SELECT \*



FROM

(

SELECT V.Medico

FROM Visita V

INNER JOIN

Medico M2 ON V.Medico = M2.Matricola

WHERE M2.Citta = ‘Pisa’

AND V.Mutuata = 1

AND MONTH(V.Data) = MONTH(CURRENT\_DATE – INTERVAL 1 MONTH)

) AS D

);

**DATABASE ATTIVO** *18-05-23*

Un **database attivo** è un database in grado di reagire autonomamente ai cambiamenti, mediante comportamenti specifici. Per crearne uno, si sfruttano due tipologie di stored programs: i **trigger** e gli **event**.

**TRIGGER**

Un **trigger** è uno stored program che viene eseguito al verificarsi di un inserimento, aggiornamento o cancellazione.

La sintassi per creare un trigger è la seguente:

DROP TRIGGER IF EXISTS *NomeTrigger*;

DELIMITER $$

CREATE TRIGGER *NomeTrigger*

[BEFORE | AFTER] [INSERT | UPDATE | DELETE] ON *TabellaTarget*

FOR EACH ROW

E cioè, creo un trigger *NomeTrigger*, il quale:

* Se è un ***before trigger***, prima di ogni inserimento (aggiornamento/cancellazione) di una row, esegue il *blocco istruzioni*.
* Se è un ***after trigger***, dopo ogni inserimento (aggiornamento/cancellazione) di una row, esegue il *blocco istruzioni*.

BEGIN

*blocco istruzioni*

END $$

DELIMITER ;

I trigger sono utilizzati principalmente per due scopi:

1. **Esprimere le business rules;**
2. **Aggiornare le ridondanze.**

**ESPRIMERE LE BUSINESS RULES**

Le **business rules** sono dei vincoli di integrità generici: “*non si accettano medici con parcella superiore ad una certa parcella massima*”; “*i pazienti devono essere maggiorenni*”; …

In MySQL, si esprimono tramite before trigger. Vediamo un esempio:

* OGNI MESE, LE VISITE NON MUTUATE DI UN MEDICO NON POSSONO ESSERE PIU’ DI 10

DROP TRIGGER IF EXISTS blocca\_non\_mutuate;

DELIMITER $$

CREATE TRIGGER blocca\_non\_mutuate

BEFORE INSERT ON Visita

FOR EACH ROW

BEGIN

DECLARE non\_mutuate\_mese INTEGER DEFAULT 0;

SET non\_mutuate\_mese = (

SELECT COUNT(\*)

FROM Visita V

WHERE V.Medico = NEW.Medico

AND MONTH(V.Data) = MONTH(CURRENT\_DATE)

AND YEAR(V.Data) = YEAR(CURRENT\_DATE)

AND V.Mutuata = 0

);

IF (NEW.Mutuata = 0 AND non\_mutuate\_mese = 10) THEN

SIGNAL SQLSTATE ‘45000’

SET MESSAGE\_TEXT = ‘Visita non inseribile’;

END IF;

END $$

DELIMITER ;

Creo un trigger *blocca\_non\_mutuate* che, prima di ogni inserimento di una row in *Visita* (e cioè, prima di ogni inserimento di una visita), esegue il *blocco istruzioni* indicato:

Dichiaro una variabile e le assegno il numero delle visite del medico di cui si sta provando ad inserire la visita, fatte in questo mese di questo anno, non mutuate. A questo punto:

* + Se la visita che si sta provando ad inserire è non mutuata e il medico che l’ha effettuata ha già fatto 10 visite non mutuate in questo mese di questo anno, allora non può essere inserita. Sollevo allora un errore che ne impedirà l’inserimento, stampando “Visita non inseribile”.
  + Altrimenti, il trigger termina senza fare nulla, e quindi la visita verrà tranquillamente inserita.

In un INSERT trigger, dunque, **NEW** identifica la row che si sta provando ad inserire (nel caso di before trigger) o che è stata appena inserita (nel caso di after trigger).

**AGGIORNARE LE RIDONDANZE**

Le **ridondanze** sono attributi il cui valore è ricavabile da altri attributi di tabelle del database.

In MySQL, si aggiornano tramite:

* After trigger, se la ridondanza è in una tabella diversa da quella in cui si effettua   
  l’inserimento/aggiornamento/cancellazione (si aggiorna opportunamente la ridondanza, dopo che sono stati modificati i dati su cui essa si basa);
* Before trigger, se la ridondanza è dentro la row che si prova ad inserire (si aggiorna opportunamente la ridondanza, prima che venga effettivamente inserita).

Vediamo degli esempi.

* GESTIRE UN ATTRIBUTO RIDONDANTE *UltimaVisita* NELLA TABELLA *Paziente* CONTENENTE LA DATA NELLA QUALE UN PAZIENTE È STATO VISITATO L’ULTIMA VOLTA.

DROP TRIGGER IF EXISTS aggiorna\_ultima\_visita;

DELIMITER $$

CREATE TRIGGER aggiorna\_ultima\_visita

AFTER INSERT ON Visita

FOR EACH ROW

BEGIN

UPDATE Paziente

SET UltimaVisita = CURRENT\_DATE

WHERE CodFiscale = NEW.Paziente;

END $$

DELIMITER ;

Creo un trigger *aggiorna\_ultima\_visita* che, dopo ogni inserimento di una row in *Visita* (e cioè, dopo ogni inserimento di una visita), esegue il *blocco istruzioni* indicato:

Aggiorno la tabella *Paziente*, impostando alla data corrente l’attributo *UltimaVisita* del paziente di cui è stata appena inserita la visita (supponendo una visita venga inserita nel database nello stesso giorno in cui viene effettuata).

In questo caso, dunque, va usato un after trigger: la ridondanza è in una tabella diversa da quella in cui si effettuano gli inserimenti, dunque dopo che viene effettuato un inserimento, la ridondanza va opportunamente aggiornata.

* SCRIVERE UN TRIGGER CHE, OGNI VOLTA CHE VIENE INSERITA UNA NUOVA VISITA, SE ESSA È MUTUATA, IMPOSTI L’ATTRIBUTO *Ticket* IN BASE ALLE FASCE DI REDDITO ANNUE:
  + *Ticket* PARI A EURO 36.15 SE REDDITO FRA EURO 0 ED EURO 15,000
  + *Ticket* PARI A EURO 45.25 SE REDDITO FRA EURO 15,000 ED EURO 25,000
  + *Ticket* PARI A 50.00 EURO SE REDDITO OLTRE 25,000 EURO.
  + SE LA VISITA NON È MUTUATA, INSERIRE NULL.

DROP TRIGGER IF EXISTS imposta\_ticket;

DELIMITER $$

CREATE TRIGGER imposta\_ticket

BEFORE INSERT ON Visita

FOR EACH ROW

BEGIN

DECLARE reddito\_annuo INTEGER DEFAULT 0;

SET reddito\_annuo = (

SELECT P.Reddito \* 12

FROM Paziente P

WHERE P.CodFiscale = NEW.Paziente

);

IF NEW.Mutuata IS TRUE THEN

IF (reddito\_annuo BETWEEN 0 AND 14999) THEN

SET NEW.Ticket = 36.15;

ELSEIF (reddito\_annuo BETWEEN 15000 AND 25000) THEN

SET NEW.Ticket = 45.25;

ELSE

SET NEW.Ticket = 50.00;

END IF;

ELSE

SET NEW.Ticket = NULL;

END IF;

END $$

DELIMITER ;

Creo un trigger *imposta\_ticket* che, prima di ogni inserimento di una row in *Visita* (e cioè, prima di ogni inserimento di una visita), esegue il *blocco istruzioni* indicato:

Dichiaro una variabile e le assegno il reddito annuo del paziente di cui si sta provando ad inserire la visita. A questo punto:

* + Se la visita che si sta provando ad inserire è mutuata, allora, in base al valore del reddito annuo del paziente, imposto opportunamente il valore dell’attributo *Ticket*. Fatto ciò, il trigger termina, e quindi la visita verrà inserita.
  + Altrimenti, imposto il valore dell’attributo *Ticket* a NULL. Fatto ciò, il trigger termina, e quindi la visita verrà inserita.

In questo caso, dunque, va usato un before trigger: la ridondanza è dentro ogni row che si prova ad inserire, dunque va aggiornata opportunamente prima che venga effettivamente inserita).

**EVENT**

Un **event** è uno stored program che viene eseguito al verificarsi di condizioni dettate dal tempo.

Vediamo innanzitutto la sintassi base per creare un ***recurrent event***:

DROP EVENT IF EXISTS *NomeEvent*;

DELIMITER $$

CREATE EVENT *NomeEvent*

*periodicità*

ON SCHEDULE EVERY *numero* DAY | MONTH | YEAR | SECOND | MINUTE | HOUR]



[STARTS *timestamp\_inizio*] [ENDS *timestamp\_fine*]

DO

E cioè, creo un event *NomeEvent* il quale, con la periodicità indicata, esegue il *blocco istruzioni*.

* Se si specifica un *timestamp\_inizio*, la periodicità parte dall’istante indicato. Altrimenti, parte dal momento di creazione dell’event;
* Se si specifica un *timestamp\_fine*, una volta raggiunto l’istante indicato, l’event non tornerà più in esecuzione. Altrimenti, l’event continuerà a ripetersi per sempre.

BEGIN

*blocco istruzioni*

END $$

DELIMITER ;

* CREARE E MANTENERE GIORNALMENTE AGGIORNATA LA RIDONDANZA *TotaleVisite* NELLA TABELLA *Medico* CONTENENTE, PER CIASCUNO, IL TOTALE DI VISITE EFFETTUATE.

DROP EVENT IF EXISTS AggiornaTotaleVisite

DELIMITER $$

CREATE EVENT AggiornaTotaleVisite

ON SCHEDULE EVERY 1 DAY

STARTS ’2023-05-18 23:55:00’

DO

BEGIN

UPDATE Medico M

SET M.TotaleVisite = M.TotaleVisite + (

SELECT COUNT(\*)

FROM Visita V

WHERE V.Medico= M.Matricola

AND V.Data = CURRENT\_DATE

);

END

DELIMITER ;

Creo un event che ogni giorno, partendo dal 2023-05-18 23:55:00, esegue il *blocco istruzioni* indicato:

Aggiorno la tabella *Medico*, aggiungendo al *TotaleVisite* di ogni medico il numero di visite fatte da tale medico in data odierna.

In questo caso, serve specificare il *timestamp\_inizio*, e anche uno molto vicino alla mezzanotte, perché se questo event andasse in esecuzione a metà della giornata, le visite fatte nel resto della giornata non verrebbero contate.

E’ possibile creare anche un ***event a singolo scatto*,** con questa sintassi:

DROP EVENT IF EXISTS *NomeEvent*;

DELIMITER $$

CREATE EVENT *NomeEvent*

E cioè, creo un event *NomeEvent* il quale esegue il *blocco istruzioni* una sola volta, all’istante indicato dal *timestamp.*

ON SCHEDULE AT *timestamp*

DO

BEGIN

*blocco istruzioni*

END $$

DELIMITER ;

**MATERIALIZED VIEW**

Una **materialized view** è una tabella che contiene il risultato già calcolato di una query.

**REFRESH**

Se si modificano le *tabelle raw*, e cioè le tabelle usate per ottenere il risultato della query contenuto nella materialized view, c’è bisogno di aggiornare la materialized view stessa.

Sono due le modalità per farlo:

* **FULL REFRESH:** la materialized view viene svuotata e ripopolata da zero;
* **INCREMENTAL REFRESH:** viene aggiornata solo la parte non più aggiornata della materialized view.

Per quanto riguarda invece *quando* viene aggiornata una materialized view, sono possibili tre politiche:

* **IMMEDIATE:** non appena viene modificata una tabella raw, la materialized view viene aggiornata.   
  Va usato dunque un *after trigger*;
* **DEFERRED:** periodicamente, la materialized view viene aggiornata. Va usato dunque un *event*.
* **ON DEMAND:** la materialized view viene aggiornata su richiesta. Va usata dunque una *stored procedure*, da chiamare quando serve.

Se si adotta una delle ultime due politiche, dunque, la materialized view potrebbe non essere sempre aggiornatissima. Non è un grosso problema: l’idea infatti è che, se si vuole accedere al risultato restituito da una query, ma questa è lenta nel restituirlo, allora, una volta calcolato, conviene salvarselo in una materialized view. A questo punto, conviene accedere al risultato salvato nella materialized view invece di ricalcolarlo ogni volta, tollerando che non sia aggiornatissimo.

**SYNC E FULL REFRESH**

* CREARE UNA MATERIALIZED VIEW CONTENENTE, PER OGNI PAZIENTE, IL NUMERO DI VISITE EFFETTUATE E LA DATA DELL’ULTIMA VISITA.

Innanzitutto, c’è bisogno di creare la materialized view, e cioè creare una tabella. Si fa in questo modo:

Elimino la tabella se già esiste (altrimenti, se già esiste, non la fa creare)

DROP TABLE IF EXISTS MV\_RESOCONTO;

Creo la tabella specificando gli attributi di cui è composta, il loro tipo e altre caratteristiche:

* Con “NOT NULL”, si specifica che l’attributo non può assumere valore NULL;
* Con “DEFAULT x”, si specifica il default value dell’attributo:
  + Se non si specifica alcun default value e l’attributo può assumere valore NULL, allora il default value viene impostato a NULL;
  + Se l’attributo non può assumere valore NULL, è obbligatorio specificare il suo default value.
* Con “PRIMARY KEY (*Lista\_Attributi*)”, si specifica che *Lista\_Attributi* è la chiave primaria della tabella.

CREATE TABLE MV\_RESOCONTO(

Paziente CHAR(100) NOT NULL,

NumVisite INT NOT NULL DEFAULT 0,

UltimaVisita DATE,

PRIMARY KEY (Paziente)

);

A questo punto, popoliamo la materialized view:

**IF** è una funzione che permette di fare delle proiezioni condizionali. Si usa in questo modo:

IF(*condizione*, *true\_value*, *false\_value*)

Se *condizione* è vera, IF restituisce *true\_value*; altrimenti, restituisce *false\_value*

INSERT INTO MV\_RESOCONTO

SELECT P.CodFiscale,

IF(V.Paziente IS NULL, 0, COUNT(\*)) AS NumeroVisite



IF(V.Paziente IS NULL, NULL, MAX(V.Data)) AS DataUltimaVisita

FROM Paziente P

LEFT OUTER JOIN

Visita V ON P.CodFiscale = V.Paziente

GROUP BY P.CodFiscale;

Faccio un left outer join tra *Paziente* e *Visita* sulla condizione *P.CodFiscale* = *V.Paziente*, così da associare, ad ogni paziente, le visite che ha effettuato, mantenendo anche i pazienti che non hanno fatto alcuna visita.

A seguito del join, dunque, avrò:

* + Per ogni paziente che ha fatto delle visite, le visite più informative che ha effettuato;
  + Per ogni paziente che non ha fatto visite, un unico record completato con valori NULL (perché non è stato combinato con alcuna visita. In particolare, questo record avrà V.Paziente a NULL).

Prendo allora i record ottenuti, li raggruppo per paziente, e di ogni gruppo proietto:

* + Il codice fiscale del paziente che ne fa parte;
  + Il numero di visite effettuate da tale paziente. In particolare:
    - Se nel gruppo V.Paziente è diverso da NULL, significa che il paziente ha fatto delle visite. Il gruppo allora sarà costituito da queste visite, e quindi proietto COUNT(\*)   
      (e cioè, il numero di visite).
    - Se nel gruppo V.Paziente è uguale a NULL, significa che il paziente non ha fatto alcuna visita. Il gruppo allora sarà costituito da un unico record completato con valori NULL, e quindi non posso proiettare COUNT(\*), altrimenti conterei erroneamente questo record come visita. In questo caso, allora, proietto 0.
  + La data dell’ultima visita effettuata da tale paziente, facendo lo stesso ragionamento del punto precedente.

Fatto ciò, inserisco il risultato di questa query nella materialized view.

Vediamo ora alcuni modi di gestire l’aggiornamento della materialized view:

IMMEDIATE REFRESH SYNC

DROP TRIGGER IF EXISTS immediate\_refresh\_mv1;

DROP TRIGGER IF EXISTS immediate\_refresh\_mv2;

DELIMITER $$

CREATE TRIGGER immediate\_refresh\_mv1

AFTER INSERT ON Visita

FOR EACH ROW

BEGIN

UPDATE MV\_RESOCONTO

SET MV.NumVisite = MV.NumVisite + 1,

MV.UltimaVisita = CURRENT\_DATE

WHERE MV.Paziente = NEW.Paziente;

END $$

CREATE TRIGGER immediate\_refresh\_mv2

AFTER INSERT ON Paziente

FOR EACH ROW

Gli attributi *NumVisite* e *UltimaVisita* assumeranno il loro default value, che sono:

* 0 per *NumVisite*;
* *NULL* per *UltimaVisita* (non si è specificato alcun default value, dunque è NULL).

BEGIN

INSERT INTO MV\_RESOCONTO(Paziente)



VALUES(NEW.CodFiscale);

END $$

DELIMITER ;

Le tabelle raw della materialized view sono *Visita* e *Paziente*. Supponendo che le uniche modifiche che si possano effettuare ad esse sono solo degli inserimenti, creo:

* + Un after trigger che, dopo ogni inserimento di una row in *Visita* (e cioè, dopo ogni inserimento di una visita), modifica il record della materialized view relativo al paziente del quale è stata appena inserita la visita, incrementando il numero di visite effettuate e impostando alla data corrente la data dell’ultima visita (supponendo che una visita venga inserita nel database nello stesso giorno in cui viene effettuata);
  + Un after trigger che, dopo ogni inserimento di una row in *Paziente* (e cioè, dopo ogni inserimento di un paziente), aggiunge nella materialized view un nuovo record relativo al paziente che è stato appena inserito, con il numero di visite effettuate a 0 e la data dell’ultima visita a NULL (non ha ancora effettuato visite)

ON DEMAND REFRESH FULL

DROP PROCEDURE IF EXISTS on\_demand\_refresh\_mv

DELIMITER $$

CREATE PROCEDURE on\_demand\_refresh\_mv()

BEGIN

TRUNCATE MV\_RESOCONTO;

INSERT INTO MV\_RESOCONTO

SELECT P.CodFiscale,

IF(V.Paziente IS NULL, 0, COUNT(\*)) AS NumeroVisite

IF(V.Paziente IS NULL, NULL, MAX(V.Data)) AS DataUltimaVisita

FROM Visita V

LEFT OUTER JOIN

Paziente P ON P.CodFiscale = V.Paziente

GROUP BY V.Paziente;

END $$

DELIMITER ;

Creo una stored procedure che, quando chiamata, effettua un full refresh della materialized view, e cioè:

* + Svuota la materialized view;
  + La ripopola da zero con i dati aggiornati (e cioè, ricalcolando il risultato della query e inserendolo nella materialized view).

DEFERRED REFRESH FULL

DROP EVENT IF EXISTS deferred\_refresh\_mv

DELIMITER $$

CREATE EVENT deferred\_refresh\_mv

ON SCHEDULE EVERY 1 WEEK

DO

BEGIN

TRUNCATE MV\_RESOCONTO;

INSERT INTO MV\_RESOCONTO

SELECT P.CodFiscale,

IF(V.Paziente IS NULL, 0, COUNT(\*)) AS NumeroVisite

IF(V.Paziente IS NULL, NULL, MAX(V.Data)) AS DataUltimaVisita

FROM Visita V

LEFT OUTER JOIN

Paziente P ON P.CodFiscale = V.Paziente

GROUP BY V.Paziente;

END $$

DELIMITER ;

Creo un event che, ogni settimana, effettua un full refresh della materialized view, e cioè:

* + Svuota la materialized view;
  + La ripopola da zero con i dati aggiornati (e cioè, ricalcolando il risultato della query e inserendolo nella materialized view).

**INCREMENTAL REFRESH**  *22-05-23*

L’incremental refresh permette di aggiornare solo la parte non più aggiornata di una materialized view sfruttando una **log table,** e cioè una tabella che tiene traccia di tutte le modifiche che sono state effettuate alle raw table dopo l’ultimo refresh. La log table, inoltre, deve essere tale che sia possibile effettuare l’incremental refresh della materialized view sfruttando solo la log table e il contenuto vecchio della materialized view stessa, evitando il più possibile l’accesso alle raw table (che potrebbe richiedere troppo tempo).

Ci sono due modalità per effettuare un incremental refresh:

* **PARTIAL REFRESH:** si considerano solo parte degli aggiornamenti fatti alle raw table (e quindi solo parte della log table). La materialized view, dunque, non verrà aggiornata del tutto.
* **COMPLETE REFRESH:** si considerano tutti gli aggiornamenti fatti alle raw table (e quindi tutta la log table). La materialized view, dunque, verrà aggiornata del tutto.

Vediamo un esempio per capire meglio.

* CREARE UNA MATERIALIZED VIEW CONTENENTE, PER OGNI PAZIENTE, IL NUMERO DI VISITE EFFETTUATE E LA DATA DELL’ULTIMA VISITA.

Creiamo e popoliamo la materialized view (è la stessa degli esercizi precedenti):

DROP TABLE IF EXISTS MV\_RESOCONTO;

CREATE TABLE MV\_RESOCONTO(

Paziente CHAR(100) NOT NULL,

NumVisite INT NOT NULL DEFAULT 0,

UltimaVisita DATE,

PRIMARY KEY (Paziente)

);

INSERT INTO MV\_RESOCONTO

SELECT P.CodFiscale,

IF(V.Paziente IS NULL, 0, COUNT(\*)) AS NumeroVisite

IF(V.Paziente IS NULL, NULL, MAX(V.Data)) AS DataUltimaVisita

FROM Paziente P

LEFT OUTER JOIN

Visita V ON P.CodFiscale = V.Paziente

GROUP BY P.CodFiscale;

Creiamo ora la log table:

DROP TABLE IF EXISTS LOG\_TABLE;

CREATE TABLE LOG\_TABLE(

Paziente CHAR(100) NOT NULL,

DataVisita DATE

);

Le tabelle raw della materialized view sono *Visita* e *Paziente*. Supponendo che le uniche modifiche che si possano effettuare ad esse sono solo degli inserimenti, creo una log table dove ogni record è costituito dagli attributi *Paziente* e *DataVisita*. In questo modo:

* + Quando si inserisce una nuova visita, tengo traccia del paziente che l’ha fatta e quando l’ha fatta;
  + Quando si inserisce un nuovo paziente, tengo traccia di chi è, mettendo *DataVisita* a NULL   
    (perché, appunto, non è l’inserimento di una visita. Questo, inoltre, permette di distinguere l’inserimento di una visita dall’inserimento di un paziente).

E, in entrambi casi, è possibile aggiornare la materialized view sfruttando solo il contenuto della log table e il vecchio contenuto della materialized view.

A questo punto, bisogna fare in modo che, ogni volta che viene aggiornata una raw table,   
le informazioni su questo aggiornamento vengano propagate anche nella log table. Servono allora degli after trigger, detti *trigger di push*:

DROP TRIGGER IF EXISTS push\_1;

DROP TRIGGER IF EXISTS push\_2;

DELIMITER $$

CREATE TRIGGER push\_1

AFTER INSERT ON Visita

FOR EACH ROW

BEGIN

INSERT INTO LOG\_TABLE

VALUES(NEW.Paziente, CURRENT\_DATE);

END $$

CREATE TRIGGER push\_2

AFTER INSERT ON Paziente

FOR EACH ROW

BEGIN

INSERT INTO LOG\_TABLE(Paziente)

VALUES(NEW.Paziente);

END $$

DELIMITER ;

Creo:

* + Un after trigger che, dopo ogni inserimento di una row in *Visita* (e cioè, dopo ogni inserimento di una visita), aggiunge nella log table un record che presenta il paziente visitato come *Paziente* e *DataVisita* impostato alla data corrente (supponendo che una visita venga inserita nel database nello stesso giorno in cui viene effettuata);
  + Un after trigger che, dopo ogni inserimento di una row in *Paziente* (e cioè dopo ogni inserimento di un paziente), aggiunge nella log table un record che presenta il paziente inserito come *Paziente* e *DataVisita* impostato a NULL (non è una visita).

La log table, dunque, avrà un contenuto del genere:

Immagine che contiene testo, Carattere, numero, schermata

Descrizione generata automaticamente

Vediamo ora come effettuare l’incremental refresh vero e proprio (partial o complete).

ON DEMAND REFRESH INCREMENTAL

DROP PROCEDURE IF EXISTS on\_demand\_refresh\_mv;

DELIMITER $$

CREATE PROCEDURE on\_demand\_refresh\_mv(IN \_up\_to DATE)

BEGIN

WITH aggregated\_log AS (

SELECT LT.Paziente,

SUM(IF(LT.DataVisita IS NULL, 0, 1)) AS NuoveVisite,

MAX(LT.DataVisita) AS NuovaUltima

FROM LOG\_TABLE LT

WHERE LT.DataVisita < \_up\_to

OR LT.DataVisita IS NULL

GROUP BY LT.Paziente

)

REPLACE INTO MV\_RESOCONTO

SELECT AL.Paziente,

IF(MV.Paziente IS NULL, AL.NuoveVisite,

MV.NumVisite + AL.NuoveVisite) AS NumeroVisite,

AL.NuovaUltima AS DataUltimaVisita

FROM aggregated\_log AL

LEFT OUTER JOIN

MV\_RESOCONTO MV ON AL.Paziente = MV.Paziente;

DELETE FROM LOG\_TABLE LT

WHERE LT.DataVisita < \_up\_to

OR LT.DataVisita IS NULL

Creo la stored procedure *on\_demand\_refresh\_mv* che:

* + prende in ingresso:
    - come parametro d’ingresso *\_up\_to*, il valore di *DataVisita* fino al quale considerare i record presenti nella log table (se si passa una data del passato, dunque, verrà effettuato un partial refresh; se si passa CURRENT\_DATE, invece, verrà effettuato un complete refresh).

Nel corpo della stored procedure, come prima cosa, *aggrego il log*, e cioè creo una CTE che contiene solo le informazioni del log che mi interessano, organizzate in modo tale che sia semplice poi aggiornare la materialized view sfruttando tale CTE. In particolare, visto che la nostra materialized view presenta, per ogni paziente, il numero di visite effettuate e la data dell’ultima visita, sarebbe comodo organizzare il log in modo tale da avere, per ogni paziente, il numero di nuove visite effettuate e la data dell’ultima visita. Facciamolo allora:

* + Prendo i record della CTE e considero solo quelli che hanno *DataVisita* < *\_up\_*to o *DataVisita* a NULL. (così da considerare anche i record relativi all’inserimento di un nuovo paziente).   
    Fatto ciò, li raggruppo per *Paziente*, e di ogni gruppo proietto:
    - Il *Paziente* che ne fa parte;
    - Il numero di nuove visite effettuate da tale paziente. L’idea sarebbe contare solo i record relativi all’inserimento di una nuova visita, e cioè quelli che presentano *LT.DataVisita* diverso da NULL. Quello che faccio, allora, è sommare i valori restituiti record per record dalla funzione IF(LT.DataVisita IS NULL, 0, 1): in questo modo, la funzione restituirà 0 per l’eventuale record relativo all’inserimento di tale paziente nella clinica e 1 per ogni record relativo all’inserimento di una sua nuova visita, e quindi la somma di questi valori sarà proprio il numero di record relativi all’inserimento di una nuova visita (e quindi, il numero di nuove visite effettuate da tale paziente);
    - La data dell’ultima visita effettuata da tale paziente.

L’*aggregated\_log*, dunque, avrà un contenuto del genere:

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

A questo punto, usiamo l’*aggregated\_log* e la materialized view per ottenere quelli che dovranno essere i nuovi record da inserire nella materialized view (come nuovi, o aggiornando i vecchi):

* + Faccio un left outer join tra *aggregated\_log* e *MV\_RESOCONTO* sulla condizione   
    *MV.Paziente* = *AL.Paziente*, così da associare ogni record dell’*aggregated\_log* relativo ad un certo paziente al record della materialized view relativo allo stesso paziente, mantenendo anche i record dell’*aggregated\_log* relativi ai pazienti di cui non c’è ancora nulla nella materialized view.  
    A seguito del join, dunque, avrò che:
    - Se un record di *aggregated\_log* è relativo ad un paziente di cui non c’è ancora nulla   
      nella materialized view, allora verrà completato con valori NULL (perché non è stato combinato con alcun record della materialized view. In particolare, questo record avrà *MV.Paziente* a NULL) ;
    - Se un record di *aggregated\_log* è relativo ad un paziente di cui c’è già un record nella materialized view, verrà combinato con tale record.

A questo punto, prendo i record appena ottenuti e proietto:

* + - Il *Paziente*;
    - Il numero aggiornato di visite effettuate da tale paziente. In particolare:
      * Se *MV.Paziente* è NULL, significa che il record è relativo ad un paziente di cui non c’è nulla nella materialized view. In questo caso, allora il numero di nuove visite che tale paziente ha effettuato è *AL.NuoveVisite* (e cioè, è uguale al numero delle sole nuove visite, perché se nella materialized view non c’è nulla di relativo a tale paziente, significa che non aveva fatto altre visite in precedenza);
      * Se invece *MV.Paziente* non è NULL, significa che il record è relativo ad un paziente di cui c’è già un record nella materialized view. In questo caso, allora, allora il numero di visite che tale paziente ha effettuato è *MV.NumVisite* + *AL.NuoveVisite*   
        (e cioè, è uguale al numero di visite che aveva effettuato precedentemente + il numero di nuove visite).
    - La data aggiornata dell’ultima visita di tale paziente. Sia che il record riguardi un paziente di cui non c’è nulla nella materialized view, sia che il record riguardi un paziente di cui c’è già un record nella materialized view, la data dell’ultima visita è *AL.NuovaUltima* (il cui valore è, appunto, la data dell’ultima visita).

Con questa query, dunque, ottengo i nuovi record da inserire nella materialized view.   
L’idea adesso è che, per ogni record restituito da questa query:

* + Se è un record relativo ad un paziente di cui c’è già un record nella materialized view, allora devo semplicemente aggiornare il record presente nella materialized view con i valori del record restituito dalla query;
  + Se è un record relativo ad un paziente di cui non c’è nulla materialized view, allora devo aggiungere questo record nella materialized view.

Per fare ciò, si usa il comando **REPLACE INTO**, che ha la seguente sintassi:

Per ogni record restituito dalla *query*:

* + - Se l’inserimento di questo nella *Tabella\_Target* porterebbe ad avere due record con lo stesso valore sulla chiave primaria, allora aggiorna il record “conflittuale” nella *Tabella\_Target* con i valori del record restituito dalla *query*;
    - Se l’inserimento di questo nella *Tabella\_Target*   
      non porterebbe ad avere due record con lo stesso valore sulla chiave primaria, allora lo inserisce.

**REPLACE INTO *Tabella\_Target***

***query***

Infine, rimuovo dalla log table la parte di log che è stata processata.

**WINDOW FUNCTIONS**

Le **window functions** sono delle funzioni che affiancano ad ogni record *r* un valore ottenuto da un’operazione eseguita su una *partition*, e cioè su un insieme di record logicamente connessi a *r*.

Per definire la partition su cui applicare una window function si usa la clausola **OVER()**.

**WINDOW FUNCTIONS DI TIPO AGGREGATE (SUM, AVG, COUNT…)**

Le **window functions di tipo aggregate** restituiscono un valore effettuando aggregazioni.

* SCRIVERE UNA QUERY CHE INDICHI, PER OGNI CARDIOLOGO, LA MATRICOLA, LA PARCELLA E LA PARCELLA MEDIA DELLA SUA SPECIALIZZAZIONE

SELECT M.Matricola,

Tra le parentesi di OVER() va specificata la partition su cui applicare la window function.

Se non si specifica nulla, la function viene calcolata su tutti i record ottenuti prima di processare il SELECT.

M.Parcella,

AVG(M.Parcella) OVER()

FROM Medico M

WHERE M.Specializzazione = ‘Cardiologia’;

Per ogni cardiologo, proietto:

* + La matricola;
  + La parcella;
  + La media delle parcelle di tutti i cardiologi (e cioè, il risultato della funzione AVG(M.Parcella) calcolata su tutti i record ottenuti prima di processare il SELECT) ottenendo così la parcella media della sua specializzazione.
* SCRIVERE UNA QUERY CHE INDICHI, PER OGNI MEDICO, LA MATRICOLA, LA SPECIALIZZAZIONE,   
  LA PARCELLA E LA PARCELLA MEDIA DELLA SUA SPECIALIZZAZIONE

SELECT M.Matricola,

M.Specializzazione,

M.Parcella,

AVG(M.Parcella) OVER (PARTITION BY M.Specializzazione)

FROM Medico M;

Per ogni medico, proietto:

* + La matricola;
  + La specializzazione,
  + La parcella;
  + La media delle parcelle calcolata sulla partition costituita dai medici della sua specializzazione   
    (e cioè, il risultato della funzione AVG(M.Parcella) calcolata sulla partition costituita dai record ottenuti prima di processare il SELECT che hanno, come valore per l’attributo *Specializzazione*, lo stesso del record attualmente considerato), ottenendo così la parcella media della sua specializzazione.

**WINDOW FUNCTIONS DI TIPO NON AGGREGATE**

Le **window functions di tipo non aggregate** restituiscono un valore senza effettuare aggregazioni.   
Vediamone alcune.

**ROW\_NUMBER()**

La window function **row\_number()** restituisce, per ogni record, la posizione che occupa nella partition su cui la window function è stata applicata. Viene usata per numerare i record in maniera progressiva.

* ASSEGNARE UN NUMERO A OGNI MEDICO NELLA SUA SPECIALIZZAZIONE

SELECT M.Matricola,

ROW\_NUMBER() OVER(PARTITION BY M.Specializzazione)

FROM Medico

Per ogni medico, proietto:

* + La matricola;
  + La posizione che occupa nella partition costituita dai medici della sua specializzazione.

**RANK()**

La window function **rank()** restituisce, per ogni record, la posizione che occupa nella partition cui sui la window function è stata applicata. La partition deve essere ordinata secondo un certo criterio.

Viene usata per stilare classifiche.

* EFFETTUARE UNA CLASSIFICA DELLA CONVENIENZA DEI MEDICI DIPENDENTEMENTE DALLA LORO PARCELLA. RESTITUIRE MATRICOLA, COGNOME, SPECIALIZZAZIONE, PARCELLA E POSIZIONE IN CLASSIFICA.

Senza specificare altro, **ORDER BY *Attributo***ordina i record per *Attributo* crescente (e cioè, come primo record avrò quello che ha il valore di *Attributo* più basso, e poi avrò record con il valore di *Attributo* via via maggiore)

SELECT M.Matricola,

M.Cognome,

M.Specializzazione,

RANK() OVER(ORDER BY M.Parcella)

FROM Medico M

Per ogni medico, proietto:

* + La matricola;
  + Il cognome;
  + La specializzazione;
  + La posizione che occupa nella partition costituita da tutti i medici ordinati per parcella crescente.

Analizziamo un possibile result set:

Immagine che contiene testo, schermata, Carattere, menu

Descrizione generata automaticamente

Possiamo osservare due cose:

* + Se due o più record hanno lo stesso valore sull’attributo usato per ordinare la partition, rank() restituisce la stessa posizione.
  + Vengono saltate un numero di posizioni pari al numero di record che hanno lo stesso valore sull’attributo usato per ordinare la partition (la posizione 3, infatti, non esiste).

Se si vuole che non venga saltata alcuna posizione, va usata la funzione **DENSE\_RANK()**.   
Vediamo un esempio:

* EFFETTUARE UNA CLASSIFICA SENZA GAP DEI MEDICI DI OGNI SPECIALIZZAZIONE DIPENDENTEMENTE DALLA LORO PARCELLA, PARTENDO DALLA PIÙ ALTA. RESTITUIRE MATRICOLA, COGNOME, SPECIALIZZAZIONE, PARCELLA E POSIZIONE NELLA CLASSIFICA.

SELECT M.Matricola,

M.Cognome,

M.Specializzazione,

**ORDER BY *Attributo* DESC**ordina i record per *Attributo* decrescente (e cioè, come primo record avrò quello che ha il valore di *Attributo* più alto, e poi avrò record con il valore di *Attributo* via via minore)

M.Parcella,

DENSE\_RANK() OVER (PARTITION BY M.Specializzazione

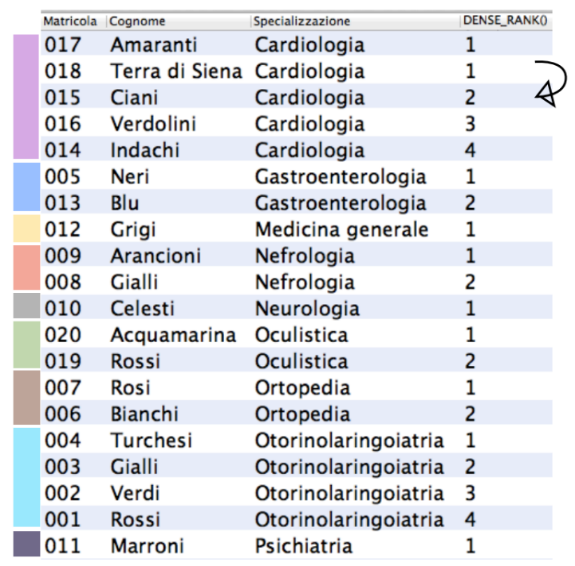
ORDER BY M.Parcella DESC)

FROM Medico M

Per ogni medico, proietto:

* + La matricola;
  + Il cognome;
  + La specializzazione;
  + La parcella
  + La posizione che occupa nella partition costituita dai medici della sua specializzazione, ordinati per parcella decrescente.

Analizziamo un possibile result set:



Possiamo osservare due cose:

* + Anche in questo caso, se due o più record hanno lo stesso valore sull’attributo usato per ordinare la partition, dense\_rank() restituisce la stessa posizione.
  + Non vengono saltate un numero di posizioni pari al numero di record che hanno lo stesso valore sull’attributo usato per ordinare la partition (la posizione 2, infatti, esiste).
* STILARE UNA CLASSIFICA DEI MEDICI IN BASE AL NUMERO DI VISITE EFFETTUATE. RESTITUIRE COGNOME, SPECIALIZZAZIONE, NUMERO DI VISTE EFFETTUATE, POSIZIONE NELLA CLASSIFICA GENERALE, E POSIZIONE NELLA CLASSIFICA PER SPECIALIZZAZIONE.

WITH MediciConVisite AS (

SELECT M.Matricola, M.Cognome, M.Specializzazione, COUNT(\*) AS NVisite

FROM Visita V

INNER JOIN

Medico M ON V.Medico = M.Matricola

GROUP BY V.Medico

)

SELECT MCV.Cognome,

MCV.Specializzazione,

MCV.NVisite

RANK() OVER (ORDER BY MCV.NVisite DESC) AS GlobalRank,

RANK() OVER (PARTITION BY MCV.Specializzazione

ORDER BY MCV.NVisite DESC) AS SpecRank

FROM MediciConVisite MCV

Innanzitutto trovo, per ogni medico, il numero di visite effettuate.

Fatto ciò, per ogni medico con questa informazione aggiuntiva, proietto:

* + Il cognome;
  + La specializzazione;
  + Il numero di visite;
  + La posizione che occupa nella partition costituita da tutti i medici ordinati per numero di visite decrescente;
  + La posizione che occupa nella partition costituita da tutti i medici della sua specializzazione, ordinati per numero di visite decrescente.

**LAG(*attributo*, *k*)**

La window function **lag(*attributo*, *k*)** restituisce, per ogni record, il valore di *attributo* del record che si trova *k* posizioni prima all’interno della partition su cui tale window function è stata applicata.

* CONSIDERATE LE VISITE OTORINOLARINGOIATRICHE DAL 2010 AL 2019, RESTITUIRE, PER CIASCUNA, MATRICOLA DEL MEDICO, CODICE FISCALE DEL PAZIENTE, DATA, E DATA DELLA VISITA PRECEDENTE DEL PAZIENTE CON UN MEDICO DELLA STESSA SPECIALIZZAZIONE

SELECT V.Medico,

V.Paziente

V.Data

LAG(V.Data, 1) OVER (PARTITION BY V.Paziente

ORDER BY V.Data)

FROM Visita V

INNER JOIN

Medico M ON V.Medico = M.Matricola

WHERE M.Specializzazione = ‘Otorinolaringoiatria’

AND YEAR(V.Data) BETWEEN 2010 AND 2019;

Trovo le visite otorinolaringoiatriche dal 2010 e 2019, e di ognuna proietto:

* + Il medico;
  + Il paziente;
  + La data;
  + La data della visita che si trova 1 posizione prima all’interno della partition costituita da tutte le visite otorinolaringoiatriche fatte allo stesso paziente, ordinate per data crescente (e quindi, il valore della visita precedente del paziente con un medico della stessa specializzazione).

**LEAD(*attributo*, *k*)**

La window function **lag(*attributo*, *k*)** restituisce, per ogni record, il valore di *attributo* del record che si trova *k* posizioni dopo all’interno della partition su cui tale window function è stata applicata.

* CONSIDERATE LE VISITE OTORINOLARINGOIATRICHE DAL 2010 AL 2019, RESTITUIRE, PER CIASCUNA, MATRICOLA DEL MEDICO, CODICE FISCALE DEL PAZIENTE, DATA, E DATA DELLA VISITA SUCCESSIVA DEL PAZIENTE CON UN MEDICO DELLA STESSA SPECIALIZZAZIONE

SELECT V.Medico,

V.Paziente

V.Data

LEAD(V.Data, 1) OVER (PARTITION BY V.Paziente

ORDER BY V.Data)

FROM Visita V

INNER JOIN

Medico M ON V.Medico = M.Matricola

WHERE M.Specializzazione = ‘Otorinolaringoiatria’

AND YEAR(V.Data) BETWEEN 2010 AND 2019;

Trovo le visite otorinolaringoiatriche dal 2010 e 2019, e di ognuna proietto:

* + Il medico;
  + Il paziente;
  + La data;
  + La data della visita che si trova 1 posizione dopo all’interno della partition costituita da tutte le visite otorinolaringoiatriche fatte allo stesso paziente, ordinate per data crescente (e quindi, il valore della visita successiva del paziente con un medico della stessa specializzazione).