

Tecniche di SQL-Data Understanding & Preparation per usi attuariali mediante l'uso di Software Open-Source - Corso SIA

Manuel Caccone
manuel.caccone@gmail.com

24/9/2024

Programma del corso

Parte 1 - Trattazione teorica

In questa sezione è presente una scaletta del corso, in particolare si presentano i punti fondamentali che verranno affrontati. In particolare:


Argomenti teorici:

- a) Introduzione al linguaggio SQL;
- b) Il linguaggio SQL per la selezione dati;
- c) Il linguaggio SQL per aggregare i dati: Window functions, subquery, CTE;
- d) SQL DDL: Primary Key, Indici e normalizzazione;
- e) SQL e NO-SQL: le differenze
- f) Le connessioni mediante R, Python e SAS

Parte 2 - Trattazione pratica

Esercitazione pratiche:

2.0) Partiamo dalla fine

2.1) Operiamo da 

2.2) Operiamo da 

2.3) Operazioni NO-SQL da 

└ Parte 1 - Quanti tipi di dati abbiamo?

Parte 1 - Quanti tipi di dati abbiamo?

└ Parte 1 - Quanti tipi di dati abbiamo?

Parte 1 - Quanti tipi di dati abbiamo?

Quando ci interfacciamo con la realtà del dato, possiamo essenzialmente trovare tre tipi di dati:

- Dati strutturati
- Dati semi-strutturati
- Dati non strutturati

Parte 1 - Quanti tipi di dati abbiamo?

In particolare i dati **strutturati** sono dati in posizione fissa all'interno di record definiti (numeri, testi, date), con una struttura rigida e con dati e metadati ben separati. In buona sostanza vi sono le informazioni relative ai campi della tabella (dati) contenute in un altro tipo di tabella (di solito menzionata come *schema* o *dictionary table*). Non vi sono strutture gerarchiche, nel senso che ogni unità minimale della tabella può contenere un solo tipo di informazione non annidata (un numero ma non un elenco di numeri)

Parte 1 - Quanti tipi di dati abbiamo?

I dati **non-strutturati** invece sono caratterizzati da assenza di schema (ad esempio immagini, video, audio) o da schemi molto leggeri; se lo schema non è presente come nel caso di oggetti multimediali, la modalità di gestione di questi dati cambia radicalmente rispetto a quelli con uno schema (dati strutturati). Oggi si parla di Internet-of-Things (IoT) e per questo si utilizzano soprattutto dati non-strutturati. L'esempio più semplice è il motore di ricerca Google per i quali si parla di *Web Information Retrieval*.

Parte 1 - Quanti tipi di dati abbiamo?

Per quanto attiene ai dati **semi-strutturati**, questi sono dati tipicamente vengono espressi da linguaggi agglomerativi rispetto all'esigenza di contenere, in unico corpo, dati con schema e senza schema: XML è il linguaggio alla base dei dati semi-strutturati. Basti pensare ad una pagina web con differenti contenuti: da questa si possono estrarre (*scraping*) informazioni in maniera standardizzata. Hanno una struttura flessibile, metadati e dati convivono nello stesso "corpo" separati da *tag*, vi sono strutture gerarchiche.

Parte 1 - Introduzione al linguaggio SQL

Parte 1 - Introduzione al linguaggio SQL

Il linguaggio SQL (*Structured Query Language*) è un linguaggio (ISO) utilizzato per interagire con i dati archiviati in qualcosa chiamato **database relazionale**. Si può pensare ad un database relazionale come a una raccolta di tabelle, non solo come insieme di righe e colonne, come un foglio di calcolo, ma come unità strutturale minima. L'unità minimale generica è rappresentata dalla cella.

Parte 1 - Introduzione al linguaggio SQL

Non è un semplice linguaggio per interrogare un database relazione, si divide in tre sotto-linguaggi principali:

- **DDL** (*Data Definition Language*): linguaggio che permette di creare,eliminare,modificare gli oggetti in un database (tabelle e viste). I comandi DDL definiscono la struttura del database;
- **DML** (*Data Manipulation Language*): linguaggio che permette di leggere,inserire,modificare,eliminare i dati di un database. Le interrogazioni in SQL appartengono a DML;
- **DCL** (*Data Control Language*): permette di fornire o revocare agli utenti i permessi necessari per poter utilizzare i comandi di DDL e DML su specifici oggetti-dati di un database;
- **Trigger**: si tratta di altre componenti (azioni eseguite dal DBMS che soddisfano determinate condizioni), SQL dinamico ed **embedded**, esecuzione *client-server*, gestione di transazioni, sicurezza.

Parte 1 - Introduzione al linguaggio SQL

Il linguaggio SQL si basa sia sull'algebra che sul calcolo relazionale, perciò esprime le interrogazioni in modo misto: in parte dichiarativo e in parte procedurale. Ad ogni modo l'interrogazione SQL viene passata all'ottimizzatore di interrogazioni (query optimizer), un componente del DBMS che analizza l'interrogazione e ne costruisce una versione equivalente in un linguaggio procedurale basato sull'algebra relazionale interno al DBMS.

Parte 1 - Introduzione al linguaggio SQL

Con il linguaggio SQL possiamo essenzialmente (semplificando all'estremo):

- selezionare unità strutturali (passi di SELECT);
- modificare unità strutturali (passi di INSERT);
- cancellare unità strutturali (passi di DELETE);
- creare unità strutturali (passi di CREATE).

Parte 1 - Introduzione al linguaggio SQL - Base statements

Statement Box

```
SELECT [DISTINCT] Attribute  
FROM Table  
WHERE condition
```

L'interrogazione SQL seleziona, tra le righe che appartengono al prodotto cartesiano delle tabelle elencate nella clausola FROM, queste soddisfano le condizioni espresse nell'argomento della clausola WHERE. Il risultato di un'interrogazione SQL è una tabella (non necessariamente una relazione matematica!) le cui colonne si ottengono dalla valutazione delle espressioni che appaiono nella clausola SELECT.

Parte 1 - Introduzione al linguaggio SQL - Estrazione plenaria e Ridenominazione

Statement Box

```
SELECT [DISTINCT] Attribute AS Attribute2  
FROM Table  
WHERE condition
```

SELECT può anche comparire il carattere speciale * (asterisco), che rappresenta la selezione di tutti gli attributi delle tabelle elencate nella clausola FROM. Inoltre ad ogni attributo può essere individuato un *alias* con la clausola AS.

Parte 1 - Introduzione al linguaggio SQL - Variabili di range

Statement Box

```
SELECT A1,A2,A3  
FROM Table1 AS I, Table2 AS D  
WHERE A1 = D.A2
```

Per evitare ambiguità tra attributi aventi lo stesso nome in tabelle diverse, si possono anche definire variabili di range. Queste possono essere anche utilizzate per disporre di un “duplicato” di una tabella, utile ai fini di una interrogazione successiva.

Parte 1 - Introduzione al linguaggio SQL - NOT,AND,OR

Statement Box

```
SELECT A1,A2,A3  
FROM Table1  
WHERE A1 = 'a'  
AND A2 > 99
```

La clausola **WHERE** consente di attribuire una espressione booleana combinando predicati semplici con gli operatori AND (operatore \cap dell'insiemistica), OR (operatore \cup dell'insiemistica) e NOT (operatore \neq dell'insiemistica). Da notare che l'OR e l'AND hanno la stessa priorità nella selezione del filtro operato nella clausola **WHERE**, tuttavia NOT ha una priorità maggiore, motivo per cui l'utilizzo delle parentesi è più consono.

Parte 1 - Introduzione al linguaggio SQL - Operatore LIKE

Statement Box

```
SELECT *  
FROM Table1  
WHERE A1 LIKE "lt"
```

L'operatore LIKE permette di operare confronti fra stringhe, molto utile quando si lavora con operazioni di filtro parziale su stringhe. Come da riquadro, osserviamo che il simbolo “_” rappresenta un confronto con un carattere arbitrario, mentre “%” rappresenta un confronto con una stringa di lunghezza arbitraria. Nell'esempio del riquadro osserviamo che si stanno cercando tutti i record di A1 che contengano la coppia di caratteri “lt” rispettivamente come terzultimo e penultimo carattere.

Parte 1 - Introduzione al linguaggio SQL - Gestione NULL

Statement Box

```
SELECT *  
FROM Table1  
WHERE A1 > 99 OR A2 IS NULL
```

Per poter includere (IS NULL) o escludere (IS [NOT] NULL) dalla selezione dei record delle tabelle, il predicato va gestito nella clausola WHERE

Parte 1 - Introduzione al linguaggio SQL - JOIN esplicito

Statement Box

```
SELECT [DISTINCT] Attribute  
FROM Table1 { JOIN Table2 ON JoinCond}  
WHERE OtherCond
```

L'inserimento del predicato JOIN permette di combinare le operazioni di selezione dei record con quella di unione tra righe di tabelle diverse. Come osserviamo il predicato non appare in corrispondenza della clausola WHERE, resta separata per “pulizia” e separazione delle operazioni. Si evidenzia inoltre che è possibile accavallare più condizioni JOIN in corrispondenza di FROM.

Parte 1 - Introduzione al linguaggio SQL - JOIN naturale

Statement Box

```
SELECT [DISTINCT] Attribute  
FROM Table1 NATURAL JOIN Table2
```

Seppur non molto conosciuto, laddove due tabelle condividono lo stesso campo, per queste il predicato “NATURAL JOIN” permette, con una sintassi più asciutta, lo stesso risultato del JOIN esplicito.

Parte 1 - Introduzione al linguaggio SQL - JOIN esterni

La caratteristica dei JOIN esterni è quella di trascurare, nell'unione dei record, sottinsiemi della tabella “ospitante” od “ospitata” che non hanno controparti. Il JOIN esterno esiste di tre tipi:

- **LEFT JOIN**: sinistro, mantiene insieme di righe mantenendo i record della tabella “ospitante” e tralasciando quelli della tabella “ospitata” che non hanno controparte;
- **RIGHT JOIN**: destro, mantiene insieme di righe mantenendo i record della tabella “ospitata” e tralasciando quelli della tabella “ospitante” che non hanno controparte;
- **FULL JOIN**: completo, mantiene insieme di righe mantenendo i record della tabella “ospitante” e “ospitata” anche se non hanno controparte.

Parte 1 - Introduzione al linguaggio SQL - ORDER BY

Statement Box

```
SELECT *  
FROM Table1  
ORDER BY AttrOrd [ASC|DESC] {, AttrOrd2 [ASC|DESC] }
```

Se si necessita di un ordine per le righe di una tabella, l'SQL permette di specificare un ordinamento sulle righe del risultato di un'interrogazione tramite la clausola ORDER BY. Come da riquadro si osserva che le righe vengono ordinate in base al primo attributo *Attrord* in elenco, per quelle che hanno stesso valore dell'attributo *Attrord*, si considerano i valori degli attributi successivi in sequenza. Con il predicato ASC si ha un qualificatore in ascendenza dei valori, con DESC in discendenza.

Parte 1 - Introduzione al linguaggio SQL - Operatori aggregati

Una delle estensioni di SQL più rilevanti è quella che supera l'algebra razionale fin qui trattata. Nella clausola `SELECT` si possono ottenere anche campi calcolati mediante operazioni di conteggio, minimo, massimo, media e totale.

└ Parte 1 - Introduzione al linguaggio SQL

Parte 1 - Introduzione al linguaggio SQL - Operatori aggregati - COUNT

Statement Box

```
SELECT count(*) | ([DISTINCT|ALL] Attr)
FROM Table1
WHERE AttrOrd
```

Nel riquadro si sta proponendo di contare tutti i record della tabella che soddisfano una certa condizione di WHERE. Si tratta di COUNT come operatore aggregato di conteggio: agiscono sul risultato delle interrogazioni di FROM e WHERE, inoltre mediante altri attributi che agiscono in sequenza con le parentesi tonde (e poi quadre) si possono conteggiare record unici (DISTINCT) od anche plenari (ALL) che soddisfano anche un criterio.

└ Parte 1 - Introduzione al linguaggio SQL

Parte 1 - Introduzione al linguaggio SQL - Operatori aggregati - SUM,AVG,MAX,MIN

Statement Box

```
SELECT A1,A2,MAX(A3)  
FROM Table1  
WHERE AttrOrd
```

Gli operatori **SUM,AVG,MAX,MIN** non costituiscono un meccanismo di selezione, sono solo operazioni di calcolo che restituiscono un valore quando sono applicate ad un insieme di righe.

└ Parte 1 - Introduzione al linguaggio SQL

Parte 1 - Introduzione al linguaggio SQL - Interrogazione con raggruppamento - GROUP BY

Statement Box

```
SELECT A1,AVG(A2)  
FROM Table1  
GROUP BY A1
```

Spesso si ha l'esigenza di applicare l'operatore aggregato separatamente a sottinsiemi di righe, ciò si potrà ottenere grazie alla clausola GROUP BY che permette la specifica di tale suddivisione. Ricordiamo che in una interrogazione di questo tipo, possono comparire nella *target list* solo attributi che compaiono nella **GROUP BY** (se A2 assume più valori per uno stesso valore unico di A1, occorre calcolare il campo che lo rende univoco).

└ Parte 1 - Introduzione al linguaggio SQL

Parte 1 - Introduzione al linguaggio SQL - Interrogazione con raggruppamento - GROUP BY

Statement Box

```
SELECT A1,AVG(A2)  
FROM Table1 JOIN Table2 ON A3=A4  
GROUP BY A1 HAVING AVG(A2) > 99
```

Si possono anche imporre le condizioni di selezione sui gruppi, ovviamente diversa dalla condizione che seleziona insiemi di righe che debbano formare i gruppi (clausola WHERE). Per fare ciò si utilizza HAVING che apparirà dopo la clausola GROUP BY.

└ Parte 1 - Introduzione al linguaggio SQL

Parte 1 - Introduzione al linguaggio SQL - Query completa

Statement Box

SELECT [DISTINCT] *lista-select*
FROM *lista-from*

WHERE *condition*

GROUP BY *listaGruppo*

HAVING *qualificazionegruppo*

ORDER BY *Attrord*

Possiamo osservare in questo riquadro la query completa nell'interrogazione più completa e più semplice.

Parte 1 - Funzioni avanzate di SQL

Parte 1 - Funzioni avanzate di SQL - Subquery in WHERE

A volte risulta necessario fare delle trasformazioni intermedie ai nostri dati, prima di selezionare, filtrare o calcolare le informazioni di cui si ha bisogno. Le **subquery** sono molto importanti e rappresentano il modo più comune per effettuare queste trasformazioni intermedie. Si possono piazzare ovunque, in SELECT, FROM, WHERE o GROUP BY; essenzialmente si tratta di una query annidata in un'altra che possa essere eseguita da sola.

Parte 1 - Funzioni avanzate di SQL - Subquery in WHERE

Statement Box

```
SELECT A1  
FROM Table1  
WHERE A1 (  
    SELECT AVG(A1)  
    FROM Table1  
)
```

Come possiamo osservare la subquery è utile per poter filtrare dei dati laddove un campo non è stato ancora calcolato, dunque per poter permettere al database di effettuare prima la query annidata, in un secondo momento la query “esterna”.

Parte 1 - Funzioni avanzate di SQL - Subquery in FROM

Possiamo inoltre a volte necessitare di una *reshaping* della tabella per poter effettuare operazioni di consultazione, filtraggio e calcolo di una tabella che non si presenta in maniera, ad esempio, aggregata. E' il caso della ricerca dei primi 3 record (ad esempio) di sinistri che hanno registrato un eccesso rispetto alla media in un determinato periodo. Nel nostro Db magari non abbiamo la media dei sinistri, quindi abbiamo prima la necessità di costruire una tabella intermedia. Con la subquery in FROM possiamo annidare questo passaggio.

Parte 1 - Funzioni avanzate di SQL - Subquery in FROM

Statement Box

```
SELECT A1,AVGA2
FROM (
SELECT A2.A1,
      AVG (A2.A2) AS AVGA2
FROM A2
WHERE A1 > 1999
)
WHERE AVGA2 > AVG (AVGA2)
ORDER BY A1
LIMIT 3
```

Osserviamo come difatti riusciamo con un unico *statement* quello che avremmo fatto in più passaggi, sia di “codice” che di calcolo.

Parte 1 - Funzioni avanzate di SQL - Subquery in SELECT

La subquery inserita nella SELECT restituisce solo un valore aggregato, quindi può essere abbastanza utile dal momento che non è possibile includerlo in una query “semplice” che non sia raggruppata. Per esempio se si volesse comparare il numero di sinistri totale degli assicurati in portafoglio con il numero totale di rischi anno. Possiamo farlo in un unico passaggio usando una subquery in SELECT.

Parte 1 - Funzioni avanzate di SQL - Subquery in SELECT

Statement Box

```
SELECT A1,  
(SX1+SX2+SX3) AS SX,  (  
SELECT SUM(A2),  
  FROM Table2  
  WHERE (Table2.A1 - Table2.A1+1) =1  
)  
FROM Table1 JOIN Table2 ON Table1.A1=Table2.A1  
WHERE A1 = 1999
```

Possiamo osservare come delle operazioni che normalmente richiedono dei campi calcolati separatamente, possono tranquillamente essere effettuate “one shot”.

Parte 1 - Funzioni avanzate di SQL - Common Table Expressions

Le **CTE** sono una sorta di rimodulazione delle subquery in modo tale che la gestione delle “operazioni annidate” avvenga nella maniera più chiara e definita possibile. Sono delle subquery in una forma “dichiarata” prima della query principale. In estrema sintesi permettono la creazione di “tabelle temporanee”/subquery da sfruttare in seguito nella query principale. Ipotizziamo di avere due tabelle, come nel caso precedente, con dati di portafoglio da un lato e dati dei sinistri analitici dall'altro. Entrambe condividono l'ID del PH.

Parte 1 - Funzioni avanzate di SQL - Common Table Expressions

Statement Box

```
WITH s AS(  
    SELECT ID, SUM(SX1+SX2+SX3) AS SX  
    FROM Table1  
    WHERE A1=1999  
)  
SELECT A1,ID,SX, SUM(PR1+PR2)  
FROM Table2 AS c JOIN s ON c.ID=s.ID
```

Parte 1 - Funzioni avanzate di SQL - Common Table Expressions

Le **CTE** presentano numerosi vantaggi rispetto ad una subquery scritta all'interno di una query principale, tra questi senz'altro:

- La CTE è depositata nella memoria;
- la CTE migliora la *performance* della query;
- migliora l'organizzazione del codice laddove ci sono tante query da innestare, le CTE possono essere dichiarate in maniera del tutto sequenziale.
- in base all'ultima punto, se, ad esempio, si hanno tre CTE sequenziali, la terza può ricevere informazioni dalle precedenti due.
- una CTE può essere autoreferenziale (SELF JOIN)

Parte 1 - Funzioni avanzate di SQL - Recap

In breve:

- **Joins:** combinano due o più tabelle per operazioni semplici o di aggregazione;
- **Subqueries:** permettono di aggirare i limiti delle *join* ma richiedono molto tempo macchina;
- **nested subqueries:** permettono accuratezza e riproducibilità di operazioni più complesse;
- **CTE:** permettono l'organizzazione delle subquery e permette il legame con altre CTE e sè stesse.

Parte 1 - Funzioni avanzate di SQL - Recap

- **Joins:** utili quando si lavora con DB con più di una tabella (totale dei sinistri per assicurato?);
- **Subqueries:** utili per ottenere campi calcolati da altre tabelle in join con altre (quanti sinistri per ogni assicurato sono stati registrati in un certo anno?);
- **nested subqueries:** utili per selezionare informazioni calcolate a cascata da più tabelle (quanti sinistri per ogni assicurato in un certo anno superano la media dei sinistri dell'anno x?);
- **CTE:** sono grandiose per organizzare un gran numero di informazioni disparate (Possiamo creare una tabella costo per frequenza sinistri?).

Parte 1 - Funzioni avanzate di SQL - Window functions

Adesso affrontiamo una limitazione che spesso troviamo nel linguaggio SQL. Spesso si necessita di raggruppare i risultati con funzioni di aggregazione, dunque recuperare informazioni aggiuntive senza raggruppare i dati per ogni singolo valore non aggregato, risulta un errore.

Parte 1 - Funzioni avanzate di SQL - Window functions

Statement Box

```
SELECT  
A1,A2,A3,  
AVG(A4) AS AVGA4  
FROM Table1
```

Come possiamo osservare non possiamo confrontare i valori aggregati con dati non aggregati. Usando una **WINDOW FUNCTION** è possibile aggirare questa limitazione: questa è una classe di funzioni che eseguono calcoli su un set di risultati che è stato già generato, indicato come “finestra”. Grazie a questa possiamo avere campi calcolati direttamente in **SELECT** senza usare *subquery* o *CTE*.

Parte 1 - Funzioni avanzate di SQL - Window functions

Sono disponibili tre tipi principali di funzioni **WINDOW FUNCTIONS**: funzioni di aggregazione, classificazione e di restituzione valore. Nell'immagine sottostante, possiamo osservare un riepilogo delle funzioni che rientrano in ciascun gruppo:

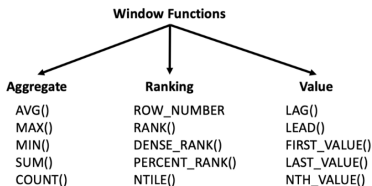


Figure 1: Riepilogo funzioni di WF

Parte 1 - Funzioni avanzate di SQL - Window functions

Dunque in un caso pratico, riprendendo una domanda delle sezioni precedenti, potremmo chiederci: quanti sinistri sono stati registrati nel periodo $X|X-1$ e come possiamo confrontarli con la media dei sinistri dell'anno di riferimento?

Parte 1 - Funzioni avanzate di SQL - Window functions

Statement Box

```
SELECT  
A1,(NX1+NX2) AS NSX,  
AVG(NX1+NX2) OVER() AS AVGNSX  
FROM Table1 WHERE A1=1999
```

Come possiamo notare con il predicato **OVER** abbiamo direttamente ottenuto un campo calcolato nella clausola **SELECT**, senza passare per la clausola **GROUP BY** o una subquery.

Parte 1 - Funzioni avanzate di SQL - Window functions

Statement Box

```
SELECT  
A1,(NX1+NX2) AS NSX,  
RANK() OVER( ORDER BY NX1+NX2) AS RANKSX  
FROM Table1 WHERE A1=1999
```

Un altro modo di usare la window function è il calcolo del RANK di una variabile quantitativa. Possiamo dunque ottenere in un'unica query il risultato dell'ordinamento (ASC di default) ascendente o discendente (DESC) di una variabile direttamente nella clausola SELECT.

Parte 1 - Funzioni avanzate di SQL - Window functions

Un altro modo di usare il comando `OVER()` è quello di unirlo al comando **PARTITION BY** per il quale si possono calcolare diversi valori aggregati di una variabile in base a categorie designate da un'altra, nella stessa colonna. Ad esempio, riprendendo la domanda di prima: quanti sinistri sono stati registrati nel periodo $X|X-1$ e come possiamo confrontarli con la media dei sinistri dell'anno di riferimento **per classi di rischio**?

Parte 1 - Funzioni avanzate di SQL - Window functions

Statement Box

```
SELECT  
A1,(NX1+NX2) AS NSX,  
AVG(NX1+NX2) OVER (PARTITION BY A2) AS AVGNSX  
FROM Table1 WHERE A1=1999
```

Possiamo ottenere inoltre lo stesso calcolo aggregato immettendo più variabili di categorizzazione (oltre a A2), alimentando nelle parentesi, con virgola, i campi (A2,A3,..). I tempi di esecuzione sono molto veloci soprattutto laddove vi sono enormi quantità di dati.

Parte 1 - Funzioni avanzate di SQL - Normalizzazioni e Primary Key

La normalizzazione è il processo per eliminare la ridondanza dei dati e migliorare l'integrità dei dati nella tabella. La normalizzazione aiuta anche a organizzare i dati nel database. È un processo in più fasi che imposta i dati in forma tabellare e rimuove i dati duplicati dalle tabelle relazionali. Prima di continuare con il concetto di normalizzazioni a più livelli dobbiamo definire:

- **candidate Key:** è un insieme di una o più colonne in grado di identificare un record in modo univoco in una tabella ed ognuna di queste è, appunto, candidata come chiave primaria (*primary key*);
- **super key:** è un insieme di più di una chiave che può identificare un record in modo univoco in una tabella, laddove la chiave primaria è un sottoinsieme della *super key*.

Parte 1 - Funzioni avanzate di SQL - Normalizzazioni 1NF e 2NF e Primary Key

Possiamo trovare vari tipi di normalizzazioni:

- **1NF**: una tabella viene definita nella sua prima forma normale se l'atomicità della tabella è 1; si intende come l'atomicità la condizione che una singola cella non può contenere più valori. Deve contenere solo un attributo a valore singolo.
- **2NF**: la prima condizione affinché la tabella sia in 2NF è che la tabella debba essere in 1NF. La tabella non dovrebbe possedere una dipendenza parziale, cioè che il sottoinsieme corretto della chiave candidata dovrebbe fornire un attributo “non primo”. Dunque se esiste una chiave primaria con due campi ed almeno un campo che dipende da un sottoinsieme della chiave primaria, allora la tabella non rispetta la 2NF;

Parte 1 - Funzioni avanzate di SQL - Normalizzazione 3NF e Foreign Key

Abbiamo inoltre la normalizzazione **3NF**: la prima condizione affinché vi sia una normalizzazione **3NF** è che sia valida la condizione 2NF prima, in seguito che non abbia dipendenze funzionali transitive. Per capirlo osserviamo la seguente tabella:

A1	A2	A3	A4
1	A	a	K1
2	B	b	K2
3	B	c	K2

Parte 1 - Funzioni avanzate di SQL - Normalizzazione 3NF e Foreign Key

Questa possiede più di una *primary key* (A1,A4), inoltre osserviamo che vi sono dipendenze funzionali transitive, ossia al variare di almeno una delle celle di A2 implica il variare delle celle di A4. Nel momento che una tabella possiede due *primary key*, la si può normalizzare separandole e rendendo una delle due come *foreign key*. La maggior parte dei sistemi di database sono database normalizzati fino alla terza forma normale in DBMS. Dunque, nel disegnare un DB, dobbiamo prenderlo in considerazione.

Parte 1 - Funzioni avanzate di SQL - Normalizzazioni e Primary Key

La domanda è: perché è necessaria?

Senza la normalizzazione in SQL, potremmo dover affrontare molti problemi come:

- Inserimenti anomali: si verifica quando non possiamo inserire dati nella tabella senza la presenza di un altro attributo;
- Aggiornamenti anomali: è l'incoerenza dei dati che deriva dalla ridondanza dei dati e da un aggiornamento parziale dei dati;
- Cancellazioni anomale: si verifica quando alcuni attributi vengono persi a causa dell'eliminazione di altri attributi.

Come si fa? La normalizzazione è una tecnica di *design* di costruzione di un RDBMS (*Relational Data Base Management System*).

└ Parte 1 - Funzioni avanzate di SQL

Parte 1 - Funzioni avanzate di SQL - Normalizzazioni e Primary Key

Riportandoci alla tabella :

A1	A2	A3	A4
1	A	a	K1
2	B	b	K2
3	B	c	K2

Questa viene normalizzata in 3NF se viene divisa in

■ Table1:

A1	A2
1	A
2	B
3	B

└ Parte 1 - Funzioni avanzate di SQL

Parte 1 - Funzioni avanzate di SQL - Normalizzazioni e Primary Key

■ Table2:

A1	A3
1	a
2	b
3	c

■ Table3:

A4	A2
1	K1
2	K2

Parte 1 - Funzioni avanzate di SQL - DDL

Per poter operare una normalizzazione dovremmo conoscere i comandi di **DDL (Data Definition Language)** che ci permettono di costruire le tabelle con il design della 3NF che abbiamo visto precedentemente. I comandi di DDL sono:

- **CREATE**: questo comando viene utilizzato per creare il database o i suoi oggetti (come tabella, indici, funzioni, viste, stored procedure e trigger);
- **DROP**: questo comando viene utilizzato per eliminare oggetti dal database;
- **ALTER**: viene utilizzato per modificare la struttura del database;
- **TRUNCATE**: viene utilizzato per rimuovere tutti i record da una tabella, inclusi tutti gli spazi allocati per i record vengono rimossi.

Parte 1 - Funzioni avanzate di SQL - DDL - CREATE TABLE

Statement Box

CREATE TABLE

```
tableName (  
  A1 int(3) NOT NULL,  
  A2 varchar(20),  
  PRIMARY KEY (A1)
```

Esistono essenzialmente vari tipi di campi da utilizzare, dipende anche dal DMBS utilizzato (MySQL?). Quelli più utilizzati sono:

- **INT**: variabile di intero;
- **VARCHAR**: variabile stringa con grandezza specificata;
- **DOUBLE(size,d)**: precisione doppia con *size* come grandezza e *d* nella parte decimale;
- **BOOL**: variabile booleana

Parte 1 - Funzioni avanzate di SQL - DDL - DROP TABLE/TRUNCATE

Statement Box

```
DROP TABLE / TRUNCATE TABLE  
tableName
```

La differenza tra i due comandi sta nel:

- TRUNCATE è normalmente ultraveloce ed è l'ideale per eliminare i dati da una tabella temporanea;
- TRUNCATE conserva la struttura della tabella per un uso futuro, a differenza di DROP che ne elimina la sua struttura completa.
- Non è possibile eseguire il rollback (UNDO) dell'eliminazione di tabelle o database utilizzando l'istruzione DROP, quindi deve essere utilizzata con saggezza.

Parte 1 - Funzioni avanzate di SQL - DDL - ALTER TABLE

Statement Box

```
ALTER TABLE tableName  
ADD A3 varchar(20);
```

Statement Box

```
ALTER TABLE tableName  
DROP COLUMN A3;
```

Statement Box

```
ALTER TABLE tableName  
ALTER COLUMN A3 INT(3);
```

Parte 1 - Alternative NO-SQL

Parte 1 - Alternative NO-SQL - DB relazionali e DB non relazionali

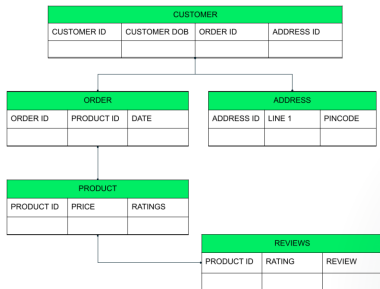
Per riassumere la differenza tra i database relazionali e non relazionali: i database relazionali memorizzano i dati in righe e colonne come un foglio di calcolo mentre i database non relazionali no, utilizzano un modello di archiviazione più adatto per il tipo di dati che sta memorizzando. Dunque possiamo dividere in

- **RDBMS**: possiede proprietà *ACID* (Atomicity, Consistency, Isolation & Durability), necessaria normalizzazione e accuratezza. Non è scalabile nè flessibile, inoltre la performance è legata al design del DB.
- **NoSQL** (*Not Only SQL*): utilizzano un modello di archiviazione in “*cloud*”, tra i più conosciuti il *Document DB* (JSON-Like structure).

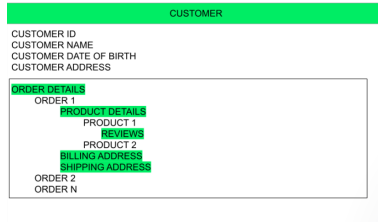
Parte 1 - Alternative NO-SQL

Parte 1 - Alternative NO-SQL - DB relazionali e DB non relazionali ¹

SQL



NoSQL



¹MongoDB (n.d.)

Parte 1 - Alternative NO-SQL - DB relazionali e DB non relazionali

I database NoSQL sono anche altamente scalabili. A differenza dei database relazionali, dove tradizionalmente è possibile ridimensionare solo verticalmente (CPU, spazio su disco rigido, ecc.), i database non relazionali possono essere ridimensionati orizzontalmente. Ciò significa avere i database duplicati su più server, pur rimanendo sincronizzati.


Quando usare un RDBMS? Quando i dati sono prevedibili, in termini di struttura, dimensione e frequenza di accesso, i database relazionali sono ancora la scelta migliore.

Quando usare un NoSQL? Se i dati che stai archiviando devono essere flessibili in termini di forma o dimensione, o se devono essere aperti a modifiche in futuro, allora un database non relazionale è la risposta.

Parte 1 - Alternative NO-SQL - da No-SQL a “NO-SQL”

Come precedentemente è stato visto, esistono database *NotOnly SQL*. Tuttavia qui parleremo di linguaggio “NO SQL”, ossia come, attraverso librerie di software *open-source*, si possa prescindere la logica e dalla rigidità del linguaggio SQL. Ricordando che *SQL* sta per *Structured Query Language*, questo letteralmente si intende come strutturato e rigido rispetto a clausole, predicati con annesse *features* posizionali finora esplorate.

Parte 1 - Alternative NO-SQL - “NO-SQL” - dplyr 4 R

Il pacchetto è uno dei pacchetti dell'insieme *tidyverse* (ggplot2,tidyr,readr,purrr) tramite il quale si possono organizzare operazioni di manipolazione dei dati, aggregazione e pulizia con un linguaggio che richiama l'SQL utilizzabile mediante . Nel pacchetto si lavora con oggetti *dataframe* oppure con un *alias* definito come **tibble**: quest'ultimo si preferisce dal momento che è più “snello” nello *storage* e nello svolgimento dei calcoli (inoltre si possono applicare tutte le funzioni previste).

Parte 1 - Alternative NO-SQL - “NO-SQL” - dplyr 4 R - piping

A caratterizzare il pacchetto è l'utilizzo dell'operatore *pipe* ossia `%>%`. Con questo si possono concatenare operazioni simili a quelle descritte finora (semplici e complesse) mediante l'SQL per poter creare nuove tabelle in maniera molto agevole. In questo senso “NO”-SQL rappresenta la mancanza di una rigida struttura di clausole che si concatenano in maniera univoca, laddove il linguaggio *dplyr* permette la concatenazione libera (o comunque flessibile) delle funzioni mediante l'operatore **pipe**. Tutte le operazioni possibili sono raccolte in un cheat-sheet utilissimo²

²[rstudio/cheatsheets](https://rstudio.com/cheatsheets/) (n.d.a)

Parte 1 - Alternative NO-SQL - “NO-SQL” - dplyr 4 R - piping

SQL

Statement Box


```
SELECT  
A1, A2 AS SX,  
FROM Table1 WHERE  
A2=1999
```


NOSQL

```
library(dplyr)
```

```
Table1 %>%  
  select(A1,A2) %>%  
  rename(SX=A2) %>%  
  filter(SX==1999)
```

Parte 1 - Alternative NO-SQL - “NO-SQL” - dplyr 4 Python

Anche in  python™ risulta possibile operare con lo stesso sistema mediante la libreria **dfply**.

- Già di per sé il software  python™ permette di utilizzare l'oggetto “data.frame” come un oggetto cui “attaccare” istruzioni derivanti da **pandas** mediante l'operatore “.”;
- In questo caso, tuttavia, l'operatore **pipe** è rappresentato da »;
- Inoltre in *dfply* sappiamo che il risultato DataFrame di ogni passaggio di una catena di operazioni è rappresentato da X^3 .

³Allen (n.d.)

Parte 1 - Alternative NO-SQL - "NO-SQL" - dplyr 4 Python - piping

SQL

Statement Box

SELECT

A1, A2 **AS** SX,

FROM Table1 **WHERE**

A2=1999


NOSQL

```
from dfply import *  
import pandas as pd  
data = pd.read_csv('T1.csv')  
(data >>  
    select(X.A1,X.A2) >>  
    mask(X.A2 == 1999) >>  
    mutate(SX = X.A2)  
)
```

Parte 1 - Come connettersi ad un DB da R,Python e SAS

Parte 1 - Come connettersi ad un DB da R,Python e SAS -




In  è possibile connettersi ad un Db MySQL grazie al pacchetto **DBI**: vedremo nella sezione pratica i comandi necessari affinché si ci possa connettere ed inoltrare i comandi fin qui esposti mediante l'utilizzo di SQL. Per farlo utilizzeremo un protocollo **ODBC** (*Open Data Base Connectivity*) che consente di connettere un database di Microsoft Access/Libreria a un'origine dati esterna come, ad esempio, Microsoft SQL Server. Vi sono tuttavia anche origini esterne come Oracle RDBMS, un foglio di calcolo e un file di testo.

Per connettersi a queste origini dati, è necessario eseguire le operazioni seguenti: - Installare il driver ODBC appropriato nel computer in cui si trova l'origine dati; - Definire un nome origine dati/Server (DSN).

└ Parte 1 - Come connettersi ad un DB da R,Python e SAS

Parte 1 - Come connettersi ad un DB da R,Python e SAS -



Nel pacchetto **DBI** avremo tutto quanto serve per poter connettere  → MySQLServer mediante ODBC.



I comandi che utilizzeremo sono:

```
library(DBI)
con <- DBI::dbConnect(odbc::odbc(),
                      Driver   = Driver,
                      Server   = Server,
                      Database = Database,
                      UID       = user,
                      PWD       = pwd,
                      Port      = 1433)
```

└ Parte 1 - Come connettersi ad un DB da R,Python e SAS

Parte 1 - Come connettersi ad un DB da R,Python e SAS -




- Parallelamente a quanto detto per , nel  **python**™ viene utilizzato il pacchetto **SQLAlchemy**: mediante il comando **PyODBC** è possibile connettersi allo stesso MYSQL mediante la creazione di un *engine* specifico.
- SQLAlchemy è il toolkit Python SQL e Object Relational Mapper (ORM) che offre agli sviluppatori di applicazioni tutta la potenza e la flessibilità di SQL, progettato per un accesso al database efficiente e ad alte prestazioni, adattato in un linguaggio di dominio semplice e “Pythonic”.

└ Parte 1 - Come connettersi ad un DB da R,Python e SAS

Parte 1 - Come connettersi ad un DB da R,Python e SAS -




Nel pacchetto **SQLAlchemy** avremo tutto quanto serve per poter connettere  python™ → MySQLServer mediante ODBC.

I comandi che utilizzeremo sono:

```
import sqlalchemy
from sqlalchemy import create_engine
from sqlalchemy.engine import URL
connection_string = "DRIVER={SQL Server Native Client 10.0};SERVER=dagger;DATABASE=test;UID=user;PWD=passw
connection_url = URL.create("mssql+pyodbc", query={"odbc_connect": connection_string})
engine = create_engine(connection_url)
```

Parte 1 - Come connettersi ad un DB da R,Python e SAS -



Anche nel  risulta possibile connettersi ad un MySQL mediante il protocollo ODBC. Tuttavia il driver, già presente sulle macchine Windows, va censito mediante appositi procedimenti⁴.

I comandi che utilizzeremo sono i seguenti:

```
proc sql;  
connect to odbc(user=userid password=password dsn=dsn_name);  
  create table work.job204 as  
  select * from connection to odbc  
  (select * from employees where jobcode=204);  
quit;
```

⁴Microsoft (n.d.)

Parte 2 - Partiamo dalla fine

Parte 2.0 - Partiamo dalla fine: come si crea un DB?

Procediamo con il creare un database con funzione di testing utilizzando il servizio gratuito *db4free.net*:

- bisogna raggiungere **db4free.net**;
- **creare un account**:

Nome del database MySQL:	<input type="text" value="provadbsia"/>
Nome utente MySQL:	<input type="text" value="prova_admin"/>
Password utente MySQL:	<input type="password" value="*****"/>
Verifica password utente MySQL:	<input type="password" value="*****"/>
<hr/>	
Indirizzo Email:	<input type="text" value="manuel.caccone@gmail.com"/>
<small>Email addresses of certain domains are not allowed!</small>	
<input checked="" type="checkbox"/> Ho letto ed accetto le condizioni d'uso .	
<input type="button" value="Registrazione"/>	

Figure 2: Step 1

Parte 2.0 - Partiamo dalla fine: come si crea un DB?

- Attenzione: il nome del DB deve essere del tipo **provadbsiaXXX** (dove XXX sono lettere minuscole o numeri a vostra scelta). Il sito non permettere Db omonimi. Alcuni indirizzi mail non sono permessi, inoltre la password è uguale allo user.
- Arrivata una mail all'indirizzo indicato.

Registrazione

Grazie per esserti registrato! Riceverai una email di conferma della registrazione.

Figure 3: Step 2

└ Parte 2 - Partiamo dalla fine

Parte 2.0 - Partiamo dalla fine: come si crea un DB?

- Bisogna cliccare il link ricevuto nella mail per attivare il DB:

Inoltre:

* [db4free.net](https://www.db4free.net) è un servizio per testing, non per hosting. Databases che contengono più di 200 MB di dati saranno azzerati ad intervalli irregolari senza notifica

* Per favore, rimuovi i dati di cui non hai più bisogno, o cancella il tuo account se non ti serve più (<https://www.db4free.net/delete-account.php>). Questo fa in modo che sia più facile eseguire il recover nel caso avvenisse un crash sul server.

<https://www.db4free.net/confirm.php?create=4cd0954ef91a1cd104ec4ad822dac65a> 

Vuoi aiutare a tradurre il sito [db4free.net](https://www.db4free.net)? Per favore vai a <https://www.db4free.net/translate.php>

Figure 4: Step 3

Parte 2.0 - Partiamo dalla fine: come si crea un DB?

- Poi si accede al **pannello phpAdmin**:



Figure 5: Step 4

Parte 2 - Partiamo dalla fine

Parte 2.0 - Partiamo dalla fine: come si crea un DB?

- Una volta effettuato l'accesso si ha un pannello di controllo, dove andremo ad inserire manualmente le nostre tabelle:

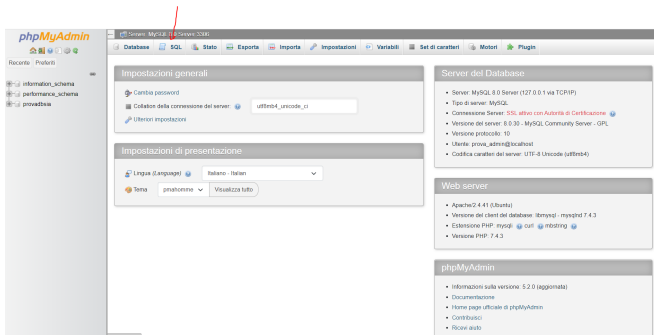


Figure 6: Step 4

Parte 2.0 - Partiamo dalla fine: come si crea un DB?

- il servizio che stiamo utilizzando non è un host professionale, come il sito esplicita non vi è la possibilità di avere dei privilegi da SUPER USER. Dunque l'inserimento del nostro piccolo database attuariale avverrà mediante il caricamento da codice mediante **DDL**, vedi l'allegato **DataEntry.sql**.
- entriamo nella sessione/console SQL come da sessione precedente ed copiamo ed incolliamo il contenuto dello script sql:

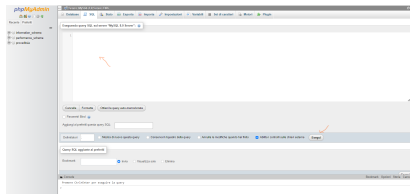



Figure 7: Step 4

Parte 2.1 - Operazioni da

Parte 2.1 - Connettiamoci da

Come abbiamo visto precedentemente, possiamo connetterci da  mediante un protocollo. Abbiamo accennato il protocollo ODBC, tuttavia per il servizio db4free.net è più opportuno che vi sia il protocollo MySQL che possiamo ottenere mediante il pacchetto **RMySQL**:

```
# install.packages('RMySQL', dep=T)
library(RMySQL)
con <- dbConnect(RMySQL::MySQL(),
                 host   = 'db4free.net',
                 dbname = 'provadbsia',
                 user    = 'prova_admin',
                 password = 'prova_admin',
                 port    = 3306)
```

Parte 2.1 - Operazioni semplici da

In questa sezione andremo ad effettuare delle operazioni semplici di selezione dei dati: **vediamo il contenuto del DB:**

```
res=DBI::dbListTables(con)  
res
```

```
## [1] "claims"    "policies"
```

Parte 2.1 - Operazioni semplici da

Primo obiettivo è capire come è fatta la tabella *claims*:

```
res=DBI::dbSendQuery(con,'DESCRIBE claims;')
res=dbFetch(res)
dbClearResult(dbListResults(con)[[1]])
```

```
## [1] TRUE
```

```
## Contenuto Tabella: CLAIMS
```

Field	Type	Null	Key	Default	Extra
ID	int	NO	PRI	NA	
CodAnagrafica	varchar(20)	YES		NA	
ImportoPrestazione	float	YES		NA	
ImportoLiquidato	float	YES		NA	
IDTipoPrestazione	int	YES		NA	
Descrizione	varchar(20)	YES		NA	
IdPrestazione	int	YES		NA	
DescrizionePrestazione	varchar(20)	YES		NA	

Parte 2.1 - Operazioni semplici da

Possiamo selezionare alcuni campi della tabella *CLAIMS* seguendo, come da slide in sezione precedente, la clausola **SELECT**:

```
res=DBI::dbSendQuery(con, 'SELECT ID,
                             CodAnagrafica AS ID_ANA,
                             ImportoLiquidato AS Cost
                             FROM claims
                             LIMIT 10;')
res=dbFetch(res)
dbClearResult(dbListResults(con)[[1]])
```

```
## [1] TRUE
```

```
## Contenuto Tabella: CLAIMS
```

ID	ID_ANA	Cost
1	CA0007	80
2	CA0022	51.6
3	CA0022	36
4	CA0022	12.19
5	CA0023	125
6	CA0027	39.89
7	CA0027	50
8	CA0042	125
9	CAA001	25.53
10	CAA001	34.82

Parte 2.1 - Operazioni semplici da

Possiamo chiederci: **quali sono i costi dei sinistri ordinando per codice anagrafica?**

```
res=DBI::dbSendQuery(con, 'SELECT ID,
                             CodAnagrafica AS ID_ANA,
                             ImportoLiquidato AS Cost
                             FROM claims
                             ORDER BY ID_ANA DESC
                             LIMIT 10;')

res=dbFetch(res)
dbClearResult(dbListResults(con)[[1]])
```

```
## [1] TRUE
```

```
## Contenuto Tabella: CLAIMS
```

ID	ID_ANA	Cost
78	CAA078	25
76	CAA070	19.16
75	CAA070	40
77	CAA070	49.1
73	CAA070	40
74	CAA070	19.16
72	CAA062	50
71	CAA062	36.15
70	CAA062	60
69	CAA062	74.1

Parte 2.1 - Operazioni semplici da

Possiamo chiederci: **quanti sono i sinistri in totale?**

```
res=DBI::dbSendQuery(con, 'SELECT COUNT(ID)
                           FROM claims;')
res=dbFetch(res)
dbClearResult(dbListResults(con)[[1]])
```

```
## [1] TRUE
```

```
## Contenuto Tabella: Numero sinistri
```

COUNT(ID)
78

Parte 2.1 - Operazioni semplici da

Possiamo chiederci: **quanti sono i sinistri suddivisi per tipologia di copertura/peril?**

```
res=DBI::dbSendQuery(con, 'SELECT
                            Descrizione AS Peril,
                            COUNT(ID) AS N_SX
                            FROM claims
                            GROUP BY Peril;')

res=dbFetch(res)
dbClearResult(dbListResults(con)[[1]])
```

```
## [1] TRUE
```

```
## Contenuto Tabella: Numero sinistri per tipologia peril
```

Peril	N_SX
Odontoiatria	12
Visite specialistich	31
Mammografia e ecogra	10
Analisi laboratorio	16
Diagnostica per imma	9

Parte 2.1 - Operazioni semplici da

Potremmo risolvere lo stesso problema andando ad inserire una **window function**, laddove questa ci restituirà la stessa informazione aggregata MA per ogni ID :

```
res=DBI::dbSendQuery(con,'SELECT ID,Descrizione AS Peril,
                           COUNT(*) OVER(PARTITION BY Descrizione) AS NUM_SX
                           FROM claims
                           ;')

res=dbFetch(res)
dbClearResult(dbListResults(con)[[1]])
```

```
## [1] TRUE
```

```
## Contenuto Tabella: Numero sinistri per tipologia peril per ogni ID
```

ID	Peril	NUM_SX
25	Analisi laboratorio	16
62	Analisi laboratorio	16
61	Analisi laboratorio	16
41	Analisi laboratorio	16
39	Analisi laboratorio	16
33	Analisi laboratorio	16
27	Analisi laboratorio	16
26	Analisi laboratorio	16
23	Analisi laboratorio	16
74	Analisi laboratorio	16

Parte 2.1 - Operazioni complesse da

Possiamo chiederci: **qual'è la frequenza sinistri per tipologia di copertura/peril?** Qui possiamo usare una **subquery**:

```
res=DBI::dbSendQuery(con,'SELECT t.Peril as Peril,
                           t.NUM_SX / t.SUMSX AS FREQ_SX
                           FROM
                           (
                             SELECT Descrizione AS Peril, COUNT(ID) AS NUM_SX,
                             (
                               SELECT COUNT(ID)
                               FROM claims
                             ) AS SUMSX
                             FROM claims
                             GROUP BY Peril
                           ) t;
                           ')

res=dbFetch(res)
dbClearResult(dbListResults(con)[[1]])
```

```
## [1] TRUE
```

Parte 2.1 - Operazioni complesse da

```
## Contenuto Tabella: Frequenza sinistri
## per tipologia peril
```

Peril	FREQ_SX
Odontoiatria	0.1538
Visite specialistich	0.3974
Mammografia e ecogra	0.1282
Analisi laboratorio	0.2051
Diagnostica per imma	0.1154

Parte 2.1 - Operazioni complesse da

Passando oltre, possiamo chiederci: **quali sono i rischi-anno nel nostro database?** Per rispondere a questa domanda lo faremo tramite *Nested CTE*:

```
res=DBI::dbSendQuery(con, 'WITH
    s AS (
        SELECT DISTINCT IdAnagrafica AS ID_ANA
        FROM policies
    ),
    r AS (
        SELECT CodAnagrafica AS ID_ANA,
        COUNT(ID) AS NUM_SX
        FROM claims
        GROUP BY ID_ANA
    ),
    t AS (
        SELECT s.ID_ANA as ID_POL,
        IFNULL(r.NUM_SX,0) AS SX
        FROM s LEFT JOIN r ON s.ID_ANA=r.ID_ANA
    )
    SELECT DISTINCT(SX) AS SINISTRI, COUNT(SX) AS RISCHI_ANNO
    FROM t
    GROUP BY SINISTRI
    ORDER BY SINISTRI;')

res=dbFetch(res)
dbClearResult(dbListResults(con)[[1]])
```

```
## [1] TRUE
```


Parte 2.1 - Operazioni complesse da

Contenuto Tabella: Rischi Anno

SINISTRI	RISCHI_ANNO
0	7
1	5
2	2
3	1
4	4
5	3
6	2
7	2
9	1

Parte 2.1 - Operazioni complesse da

Viene spontaneo chiedersi in seguito: **qual è l'indice di sinistrosità e ripetibilità nel nostro database?** Useremo un mix di **CTE, subquery, JOINS**:

```
res=DBI::dbSendQuery(con, 'WITH
    s AS (
        SELECT DISTINCT IdAnagrafica AS ID_ANA
        FROM policies
    ),
    r AS (
        SELECT CodAnagrafica AS ID_ANA,
        COUNT(ID) AS NUM_SX
        FROM claims
        GROUP BY ID_ANA
    ),
    t AS (
        SELECT s.ID_ANA as ID_POL,
        IFNULL(r.NUM_SX,0) AS SX
        FROM s LEFT JOIN r ON s.ID_ANA=r.ID_ANA
    ),
```

Parte 2.1 - Operazioni complesse da

```
'IndSin AS(  
SELECT SUM(w.SINISTRI * w.RISCHI_ANNO) as N,  
SUM(SINISTRI) AS ID  
FROM (  
SELECT DISTINCT(SX) AS SINISTRI, COUNT(SX) AS RISCHI_ANNO  
FROM t  
GROUP BY SINISTRI  
ORDER BY SINISTRI  
) w  
,  
IndSin2 AS(  
SELECT SUM(w.RISCHI_ANNO) as R,  
SUM(SINISTRI) AS ID  
FROM (  
SELECT DISTINCT(SX) AS SINISTRI, COUNT(SX) AS RISCHI_ANNO  
FROM t  
GROUP BY SINISTRI  
ORDER BY SINISTRI  
) w  
)
```

Parte 2.1 - Operazioni complesse da

```

      'IndSin3 AS(
      SELECT  SUM(w.SINISTRI * w.RISCHI_ANNO) / SUM(w.RISCHI_ANNO) as IND_RIP,
      SUM(SINISTRI) AS ID
      FROM (
      SELECT DISTINCT(SX) AS SINISTRI, COUNT(SX) AS RISCHI_ANNO
      FROM t
      GROUP BY SINISTRI
      ORDER BY SINISTRI
      ) w
      WHERE w.SINISTRI>0
      )

      SELECT a.N,b.R, a.N / b.R AS IND_SIN, c.IND_RIP
      FROM IndSin a
      INNER JOIN IndSin2 b
      ON a.ID=b.ID
      INNER JOIN IndSin3 c
      ON b.ID=c.ID
      ;')

res=dbFetch(res)
dbClearResult(dbListResults(con)[[1]])

```

Parte 2.1 - Operazioni complesse da

Contenuto Tabella: Ind. Sinistrosità, Indice Ripetibilità

N	R	IND_SIN	IND_RIP
78	27	2.889	3.9

Parte 2.1 - Operazioni complesse da

Infine: **qual è il premio equo?** Useremo, anche qui, un mix di **CTE**, **subquery**, **JOINS**:

```
res=DBI::dbSendQuery(con, 'WITH
    s AS (
        SELECT DISTINCT IdAnagrafica AS ID_ANA
        FROM policies
    ),
    r AS (
        SELECT CodAnagrafica AS ID_ANA,
        SUM(ImportoLiquidato) AS COST
        FROM claims
        GROUP BY ID_ANA
    ),
    t AS (
        SELECT s.ID_ANA as ID_POL,
        IFNULL(r.COST,0) AS Y
        FROM s LEFT JOIN r ON s.ID_ANA=r.ID_ANA
    )
    SELECT AVG(Y) AS PREMIO_EQUO
    FROM t
;')

res=dbFetch(res)
dbClearResult(dbListResults(con)[[1]])
```

Parte 2.1 - Operazioni complesse da

Contenuto Tabella: Premio Equo

PREMIO_EQUO

128.3

Parte 2.1 - Operazioni complesse da

Infine: qual è il premio equo per peril?

```
res=DBI::dbSendQuery(con,'WITH
    s AS (
        SELECT DISTINCT IdAnagrafica AS ID_ANA
        FROM policies
    ),
    r AS (
        SELECT CodAnagrafica AS ID_ANA,
        SUM(ImportoLiquidato) AS COST,
        Descrizione AS Peril
        FROM claims
        GROUP BY ID_ANA, Peril
    ),
    t AS (
        SELECT s.ID_ANA as ID_POL,
        IFNULL(r.COST,0) AS Y,
        r.Peril
        FROM s LEFT JOIN r ON s.ID_ANA=r.ID_ANA
    )
    SELECT Peril, AVG(Y) AS PREMIO_EQUO
    FROM t
    WHERE Peril IS NOT NULL
    GROUP BY Peril
;')

res=dbFetch(res)
dbClearResult(dbListResults(con)[[1]])
```



Parte 2.1 - Operazioni complesse da

Contenuto Tabella: Premio Equo per Peril

Peril	PREMIO_EQUO
Odontoiatria	85.55
Visite specialistich	84.48
Mammografia e ecogra	50.48
Analisi laboratorio	40.71
Diagnostica per imma	63.53

Parte 2.2 - Operazioni da

Parte 2.2 - Operazioni semplici da

Per connetterci da  python[™] mediante un protocollo MySQL useremo il pacchetto **PyMySQL**:

```
library(reticulate)
py_install("PyMySQL")
py_install("tabulate")
```

Parte 2.2 - Operazioni semplici da

I comandi che utilizzeremo sono:

```

import pymysql
from tabulate import tabulate
import pandas as pd

# Open database connection
db = pymysql.connect(host='db4free.net',
                     user='prova_admin',
                     password = "prova_admin",
                     db='provaadmin')

try:

    with db.cursor() as cur:

        cur.execute('SELECT * FROM claims')

        data = cur.fetchall()

finally:

    db.close()

num_fields = len(cur.description)
field_names = [i[0] for i in cur.description]
df = pd.DataFrame.from_records(data, columns=field_names)
print(df.head().to_markdown())

```

Parte 2.2 - Operazioni semplici da

78

##		ID	CodAnagrafica	ImportoPrestazione	ImportoLiquidato	IDTipoPrestazione	Descr				
##	---	:	----	:	-----	:	----				
##	0		1 CA0007		160		80		8		Odont
##	1		2 CA0022		103.19		51.6		1		Visita
##	2		3 CA0022		36		36		4		Mammografia
##	3		4 CA0022		12.19		12.19		2		Analisi
##	4		5 CA0023		400		125		8		Odont

Parte 2.3 - Operazioni NO-SQL da

Parte 2.3 - Operazioni NO-SQL da

In questa sezione andremo a parlare del pacchetto **dplyr**, in particolare come, quanto visto precedentemente, si possa tradurre mediante **piping**. Domanda “attuariale”: **qual è l'importo medio del liquidato per peril?**

```
res=DBI::dbSendQuery(con,'SELECT * FROM claims;')
res=dbFetch(res)
dbClearResult(dbListResults(con)[[1]])
```

```
## [1] TRUE
claims=res
library(dplyr)
claims %>%
  group_by(peril=Descrizione) %>%
  summarize(Mean_Cost=mean(ImportoLiquidato))
```

```
## # A tibble: 5 x 2
##   peril          Mean_Cost
##   <chr>          <dbl>
## 1 Analisi laboratorio    28.0
## 2 Diagnostica per imma   35.3
## 3 Mammografia e ecogra  40.4
## 4 Odontoiatria          85.6
## 5 Visite specialistich  40.9
```


Parte 2.3 - Operazioni NO-SQL da

In questa sezione andremo a parlare del pacchetto **dplyr**, in particolare come, quanto visto precedentemente, si possa tradurre mediante **piping**. Domanda “attuariale”: **qual è il numero di sinistri per peril?**

```
claims %>%
  group_by(peril=Descrizione) %>%
  summarize(Num_Sx=length(ID))
```

```
## # A tibble: 5 x 2
##   peril          Num_Sx
##   <chr>         <int>
## 1 Analisi laboratorio      16
## 2 Diagnostica per imma       9
## 3 Mammografia e ecogra     10
## 4 Odontoiatria             12
## 5 Visite specialistich     31
```


Parte 2.3 - Operazioni NO-SQL da - Extra

Oltre all'operatore **pipe**, possiamo esplorare le possibilità offerte dal pacchetto **data.table** di . In particolare possiamo operare con una logica veramente **NO-SQL** dal momento che questo pacchetto permette una scrittura più asciutta, meno regolare e più improntata all'utilizzo di grosse dimensioni di dati. Al seguente link⁵ possiamo ottenere una prospettiva su tutte le funzionalità del pacchetto e tutte le *short-cut*.

⁵[rstudio/cheatsheets](https://rstudio.com/cheatsheets/) (n.d.b)

Parte 2.3 - Operazioni NO-SQL da - Extra

In particolare osserviamo come vi siano l'uso tripartito delle parentesi:

```
library(data.table)
```

```
data[ , , ]
```

In particolare:

- alla sinistra dei 3 slot avvengono tutte le operazioni di *filtering* sui dati;
- al centro avvengono le operazioni di *select*, *mutating*, *calculating* che si possono fare sui campi di un dataset;
- a destra invece avvengono tutte le funzioni di *grouping*, *ordering* sui campi.

Parte 2.3 - Operazioni NO-SQL da - Extra

In particolare, per poter rispondere ad una delle domande/problemi posti finora, ad esempio : **qual è l'importo medio del liquidato per peril?**

```
res=DBI::dbSendQuery(con,'SELECT * FROM claims;')
res=dbFetch(res)
dbClearResult(dbListResults(con)[[1]])
```

```
## [1] TRUE
claims=res
library(data.table)
claims=data.table(claims)
claims[,.(Mean_Cost=mean(ImportoLiquidato)),by=.(Peril=Descrizione)]
```

```
##           Peril Mean_Cost
##           <char>      <num>
## 1:      Odontoiatria  85.55000
## 2: Visite specialistich 40.87839
## 3: Mammografia e ecogra 40.38000
## 4:  Analisi laboratorio 27.98875
## 5: Diagnostica per imma 35.29222
```

Parte 2.3 - Operazioni NO-SQL da - Extra

Altro problema: **qual è il premio equo per peril?**

```
claims[,D:=.N]
claims[,.(Equity_Premium=mean(ImportoLiquidato)*length(Descrizione)/unique(D)),by=.(Peril=Descrizione)]
```

```
##              Peril Equity_Premium
##              <char>          <num>
## 1:      Odontoiatria      13.161538
## 2: Visite specialistich  16.246538
## 3: Mammografia e ecogra   5.176923
## 4: Analisi laboratorio    5.741282
## 5: Diagnostica per imma    4.072179
```

References I

Allen, Akinkunle. n.d. “Dplyr-Style Data Manipulation with Pipes in Python.” Accessed September 19, 2022.
<https://towardsdatascience.com/dplyr-style-data-manipulation-with-pipes-in-python-380dcb137000>.

Microsoft. n.d. “Connettersi a Un’origine Dati ODBC (Importazione/Esportazione Guidata SQL Server).” Accessed September 19, 2022.
<https://learn.microsoft.com/it-it/sql/integration-services/import-export-data/connect-to-an-odbc-data-source-sql-server-import-and-export-wizard?view=sql-server-ver16>.

References II

MongoDB. n.d. “Relational Vs. Non-Relational Databases.”

Accessed September 19, 2022.

<https://www.mongodb.com/compare/relational-vs-non-relational-databases>.

rstudio/cheatsheets. n.d.a. “Data Transformation with

Dplyr:.CHEAT SHEET.” Accessed September 19, 2022.

<https://github.com/rstudio/cheatsheets/blob/main/data-transformation.pdf>.

———. n.d.b. “R for Data Science Cheat Sheet Data.table.”

Accessed September 19, 2022. https://s3.amazonaws.com/assets.datacamp.com/blog_assets/datatable_Cheat_Sheet_R.pdf.