I database e il linguaggio SQL

Introduzione al concetto di Database

Cos'è un Database?

Un database è un insieme di dati:

- in relazione tra loro

- archiviati (generalmente su un supporto informatico)

- organizzati secondo determinate strutture

- creato per :
    - supportare software, applicazioni web, eccetera
    - analizzare e controllare i dati
    - archiviarli

Cos’è un DBMS

Un DBMS è un software informatico che permette di creare e gestire un Database.

Esempi: SQL Server, Azure SQL Database, Oracle, MySQL, MongoDB, Neo4J, PostgreSQL, Access*

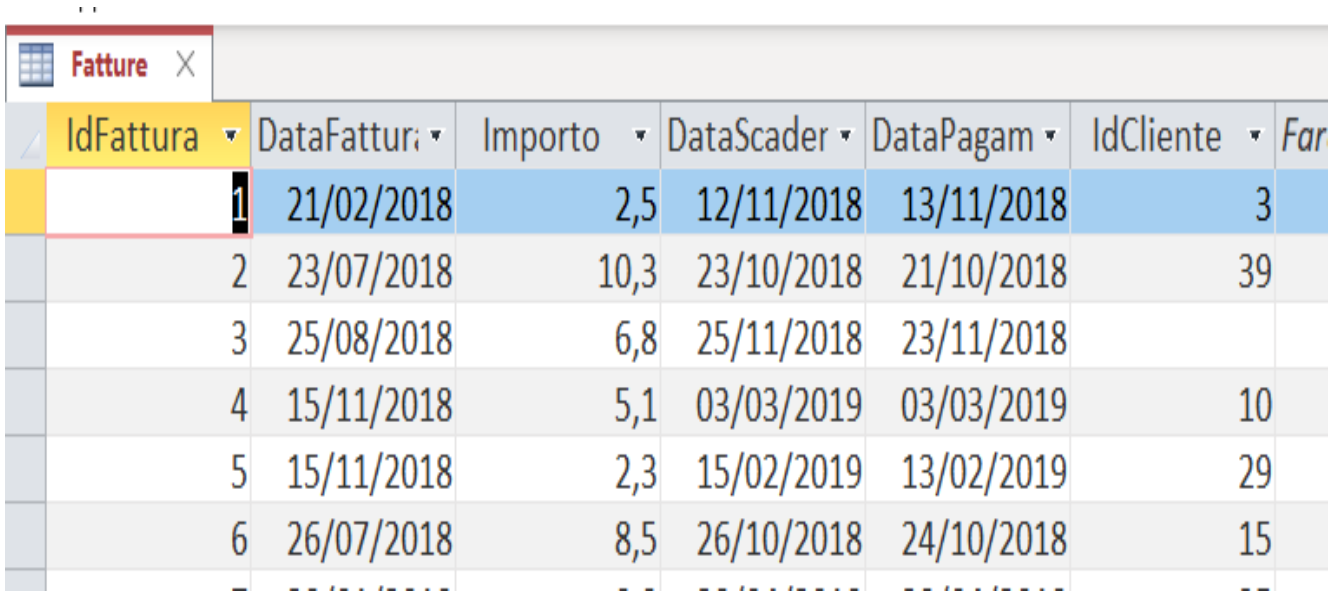
*con delle limitazioni

**Excel non è propriamente un DBMS, ma se usato con molta attenzione è possibile organizzare i dati al suo interno in modo simile a quanto accade in un database

Un R-DBMS è un DBMS specifico per database relazionali

Database relazionali

I **database relazionali** sono database in cui i dati sono organizzati in tabelle



Fatture

IdFattura	DataFattura	Importo	DataScader	DataPagam	IdCliente	Far
1	21/02/2018	2,5	12/11/2018	13/11/2018	3	
2	23/07/2018	10,3	23/10/2018	21/10/2018	39	
3	25/08/2018	6,8	25/11/2018	23/11/2018		
4	15/11/2018	5,1	03/03/2019	03/03/2019	10	
5	15/11/2018	2,3	15/02/2019	13/02/2019	29	
6	26/07/2018	8,5	26/10/2018	24/10/2018	15	

Database non relazionali

I **database non relazionali** sono database in cui i dati non sono organizzati in tabelle (grafi, documenti JSON, eccetera)

DB Relazionali: cos’è una tabella?

Fatture

IdFattura	DataFattur	Importo	DataScader	DataPagam	IdCliente	Far
1	21/02/2018	2,5	12/11/2018	13/11/2018	3	
2	23/07/2018	10,3	23/10/2018	21/10/2018	39	
3	25/08/2018	6,8	25/11/2018	23/11/2018		
4	15/11/2018	5,1	03/03/2019	03/03/2019	10	
5	15/11/2018	2,3	15/02/2019	13/02/2019	29	
6	26/07/2018	8,5	26/10/2018	24/10/2018	15	

- Insieme di righe e colonne
- Le righe non sono numerate (non esiste la prima riga, la seconda riga…)
- Colonne con un nome univoco
- Posso inserire dei **vincoli** sui dati che andranno a popolare tabelle:
    - Vincoli di Tipo
    - Vincoli NULL/NOT NULL
    - Vincoli di Chiave Primaria
    - Vincoli di Chiave Esterna
    - Vincoli Check

Linguaggio SQL

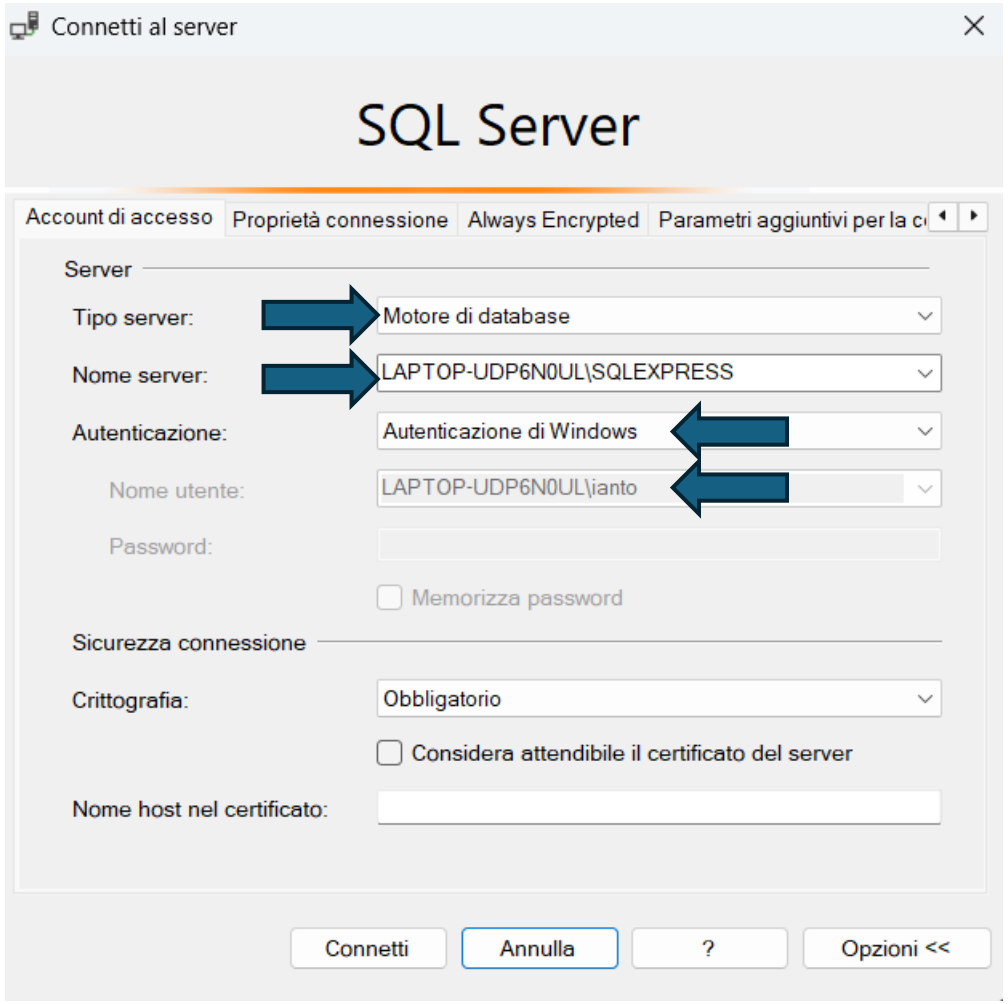
- Il linguaggio di programmazione per lavorare con i Database Relazionali e le tabelle si chiama SQL.
- Esiste qualche differenza tra l'SQL utilizzato su MySQL, quello utilizzato su Oracle, su SQL Server, Azure SQL Database, Access, eccetera.
- Nelle prossime slide faremo riferimento al linguaggio SQL per **Microsoft SQL Server** (molto simile ad Azure SQL Database)
- La maggior parte dei concetti è comune a tutti i Database relazionali.
- L'SQL è un linguaggio Dichiarativo e English Like
- L'SQL si divide in DDL, DML, DQL e DCL

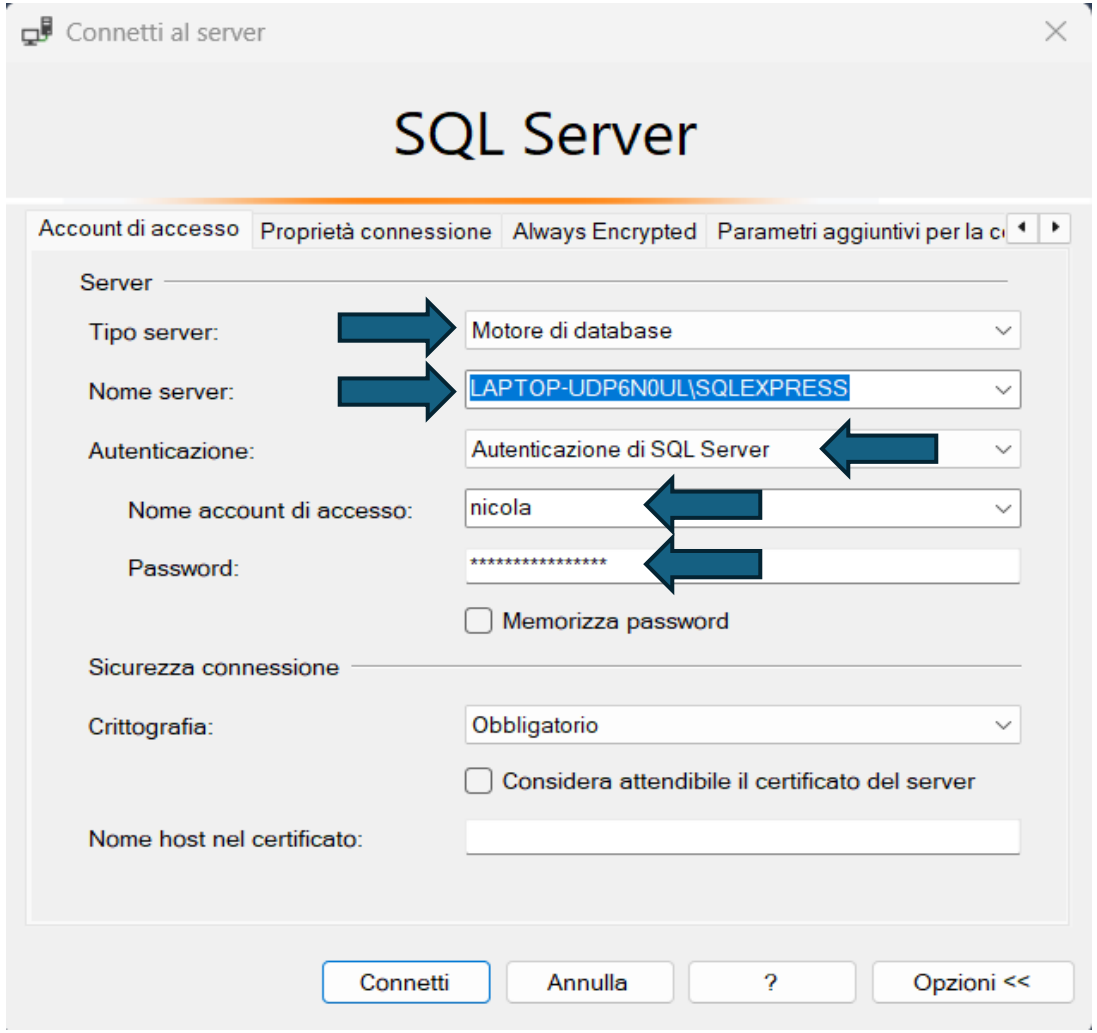
Connettiamoci a un Database SQL Server

Una volta installato Microsoft SQL Server, possiamo connetterci ad esso in svariati modi.

- Tramite SQL Server Management Studio
- Tramite un plugin di Visual studio Code
- Tramite terminale
- Tramite Excel, Access, Power BI, …
- Tramite Python, C#, …
- Ulteriori modalità

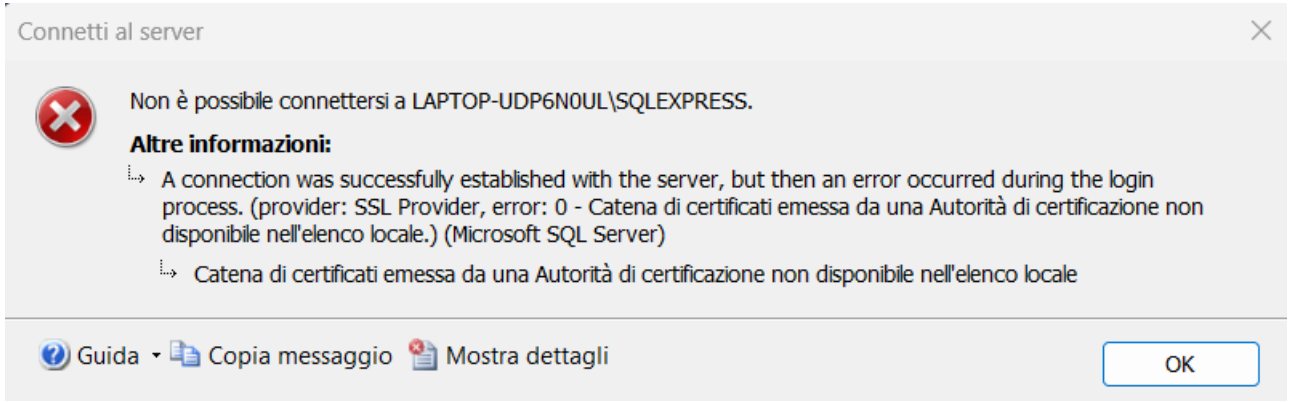
Esempio connessione tramite SQL Server Management Studio





Errore sul certificato

Se avete installato una versione abbastanza recente di SQL Server sul vostro PC, in fase di connessione otterrete probabilmente il **messaggio di errore** mostrato in basso. È legato ad un possibile problema di sicurezza nella comunicazione dei dati tra SQL Server e SQL Server Management Studio.



Sulla vostra personale installazione **utilizzata solo a fini di studio**, potete risolvere il problema spuntando la voce **"Considera attendibile il certificato del Server"** nella schermata di login mostrata nella slide precedente.

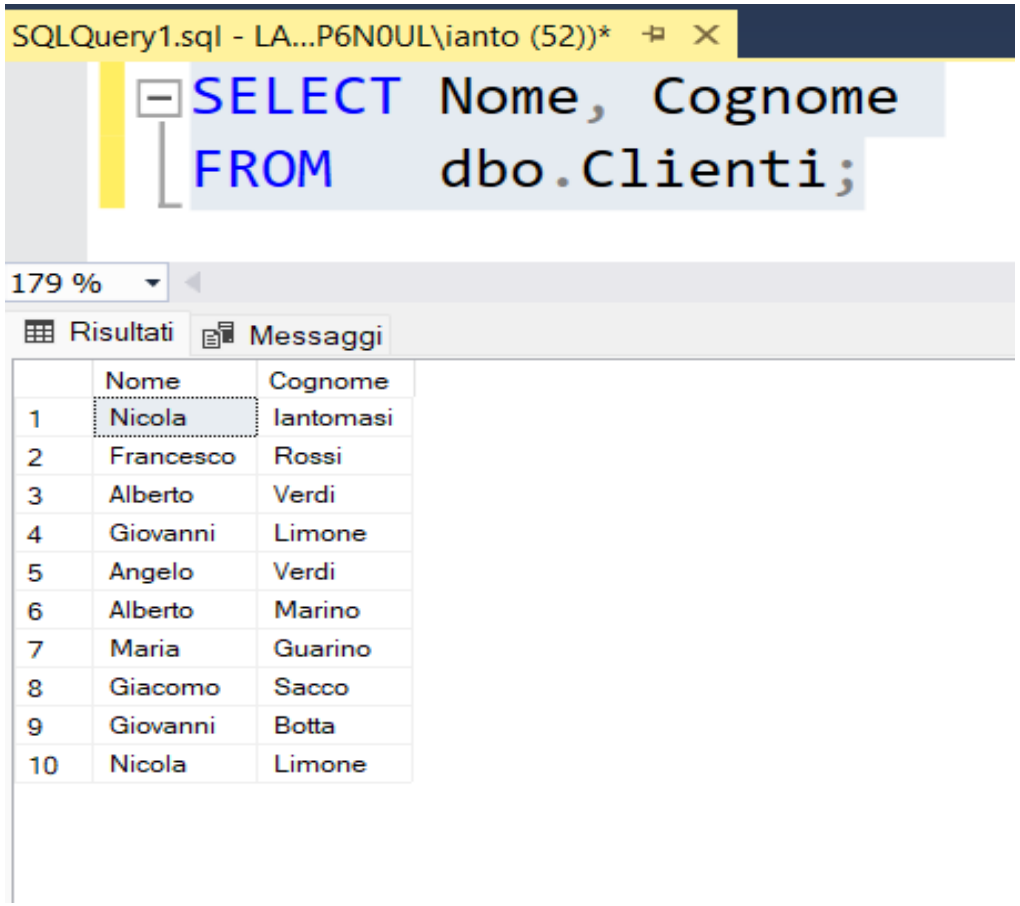
Se invece avete questo errore in ambito aziendale, avvisate il vostro DBA (Database Administrator) o sistemista che saprà come risolvere nel modo più idoneo al contesto.

Prime query SQL

La mia prima query

Visualizzare il contenuto delle colonne *Nome* e *Cognome* della tabella *Clienti*

```
SELECT Nome, Cognome
FROM   dbo.Clienti;
```



	Nome	Cognome
1	Nicola	Iantomasi
2	Francesco	Rossi
3	Alberto	Verdi
4	Giovanni	Limone
5	Angelo	Verdi
6	Alberto	Marino
7	Maria	Guarino
8	Giacomo	Sacco
9	Giovanni	Botta
10	Nicola	Limone

Visualizziamo una colonna in più

Visualizzare il contenuto delle colonne *Nome*, *Cognome*, *DataNascita* della tabella *Clienti*

```
SELECT Nome, Cognome, DataNascita
FROM   dbo.Clienti;
```

Select "star"

Visualizzare tutte le colonne della tabella *clienti*

```
SELECT IdCliente, Nome, Cognome,
   DataNascita, RegioneResidenza
FROM   dbo.Clienti;

SELECT *
FROM   dbo.Clienti;
```

Commento

In un database reale alcune tabelle potrebbero contenere tantissime righe. Eseguire una semplice query del tipo

"SELECT * FROM Tabella"

potrebbe comportare un importante carico di lavoro per il sistema.

Nella prossima query è utilizzata la sintassi *"SELECT TOP 50"* che permette di visualizzare soltanto 50 righe della tabella, un numero che è spesso sufficiente per farci un'idea sul suo contenuto.

Clausola TOP

Visualizzare 50 righe della tabella *Clienti*

```
SELECT TOP 50 *
FROM   dbo.Clienti;
```

Commento

La query precedente estrae 50 righe casuali della tabella.

Nelle tabelle di un database relazione NON c'è il concetto di "prima riga", "seconda riga", eccetera.

Per specificare un ordinamento è necessario inserire una clausola ORDER BY. La query seguente ad esempio estrae i primi 3 clienti secondo l'ordinamento ascendente della colonna nome.

Clausola Order BY

Visualizzare i primi 3 *clienti* secondo l'ordinamento ascendente della colonna *nome*.

```
SELECT TOP 50 *
FROM   dbo.Clienti
ORDER  BY Nome ASC;
```

Commento

Per l'ordinamento decrescente possiamo usare ORDER BY Nome DESC. Se non specifichiamo ASC o DESC il valore di default è ASC.

Facciamo attenzione alle performance perché la clausola Order By è molto costosa in termini di tempi di esecuzione!

ATTENZIONE: con le impostazioni di DEFAULT, il NULL è il primo valore in ordine crescente

Rinominare le colonne in output

Riportare *Nome*, *Cognome* e *IdCliente*. La colonna contenente il valore *IdCliente* deve avere come titolo *CodiceCliente*.

```
SELECT Nome, Cognome, IdCliente AS CodiceCliente
FROM   dbo.Clienti;
```

Commento

Per raggiungere lo scopo è bastato utilizzare la parola *AS*.

Ovviamente non si tratta di una rinomina sulla tabella reale salvata all'interno del database, ma soltanto di modificare quanto si vede nell'output della query

Filtrare i dati

Filtrare i dati

Visualizzare *nome*, *cognome* e *regione di residenza* dei clienti residenti in *Lombardia*

```
SELECT Nome, Cognome, RegioneResidenza
FROM   dbo.Clienti
WHERE  RegioneResidenza = 'Lombardia';
```

Commento

Per inserire il filtro abbiamo usato la nuova parola chiave **WHERE**.

Osserviamo che la parola 'Lombardia' è racchiusa tra apici.

Se non usassimo gli apici, il codice restituirebbe un errore perché andrebbe a cercare i valori contenuti all'interno di un'ipotetica colonna di nome Lombardia

Filtri più complessi

Visualizzare *nome*, *cognome* e *regione di residenza* dei clienti residenti in *Lombardia* e che si chiamano *Alberto*

```
SELECT Nome, Cognome, RegioneResidenza
FROM   dbo.Clienti
WHERE  RegioneResidenza = 'Lombardia'
  AND  Nome = 'Alberto';
```

Commento

Quando inserisco più di un filtro, essi devono essere connessi con un AND

oppure con un OR. Analizziamo come cambia il risultato lanciando la query usando l'OR

```
SELECT Nome, Cognome, RegioneResidenza
FROM   dbo.Clienti
WHERE  RegioneResidenza = 'Lombardia'
    OR  Nome = 'Alberto';
```

Attenzione alla lingua italiana!

Estrarre *nome* e *cognome* dei clienti numero *1* e *5*

```
SELECT Nome, Cognome
FROM   dbo.Clienti
WHERE  IdCliente = 1
   OR  IdCliente = 5;
```

Commento

Quando eseguiamo filtri con valori numerici NON dobbiamo utilizzare gli apici. Scriviamo dunque semplicemente

IdCliente = 1

invece di

IdCliente = '1'

Una scorciatoia per la OR

Estrarre *nome* e *cognome* dei clienti numero *1* e *5* (stesso esercizio di prima)

```
SELECT Nome, Cognome
FROM   dbo.Clienti
WHERE  IdCliente IN (1,5);
```

Filtri su date

Visualizzare tutte le colonne delle *fatture* emesse dal giorno *1 gennaio 2022* in poi

```
SELECT *
FROM   dbo.Fatture
WHERE  DataFattura >= '20220101';
```

Commento

Quando scriviamo filtri su colonne di tipo "date", ricordiamoci di scrivere le date nel formato internazionale e standard *AAAAMMDD*.

Ecco alcuni dei simboli matematici che possiamo utilizzare quando facciamo dei confronti nei filtri

=, >, <, >=, <=, <>, !=

Filtri su funzioni

Visualizzare tutte le colonne delle *fatture* emesse nel *2022*

```
SELECT *
FROM   dbo.Fatture
WHERE  YEAR(DataFattura) = 2022;
```

Commento

La colonna *DataFattura* contiene l'intera data, non solo l'anno. Sarebbe sbagliato dunque scrivere la query in questo modo

```
SELECT *
FROM   dbo.Fatture
WHERE  DataFattura = 2022;
```

Nella query corretta invece abbiamo applicato a *DataFattura* la **funzione** *YEAR*. Tale funzione ha lo scopo di convertire una data nel numero intero corrispondente al relativo anno.

Filtri su date

Visualizzare tutte le colonne delle *fatture* emesse nel *2022* (stesso esercizio di prima)

```sql
SELECT *
FROM   dbo.Fatture
WHERE  DataFattura >= '20220101'
   AND DataFattura < '20230101';
```

Commento

In alcuni casi scrivere codice che NON utilizzi funzioni potrebbe portare a dei miglioramenti delle performance.

Evitiamo invece di scrivere la query precedente in questo modo

```
SELECT *
FROM   dbo.Fatture
WHERE  DataFattura >= '20220101'
   AND DataFattura <= '20221231';
```

perché potenzialmente potrebbe portare a dei risultati errati se la colonna *DataFattura* contenesse informazioni anche su ore minuti e secondi di registrazione.

Una mail dal capo...

"L'ufficio fiscale mi ha confermato che le *fatture* devono avere tutte l'*iva* al 20 per cento. Per piacere mi controlli se c'è qualcuna errata?"

```
SELECT *
FROM   dbo.Fatture
WHERE  Iva <> 20 OR Iva IS NULL;
```

Commento

Limitando il filtro a *Iva <> 20*, non avremmo estratto la riga contenente NULL. In generale occorre fare molta attenzione nella gestione dei NULL. Ad esempio le due query seguenti sono "errate" perché restituiranno in tutti i casi zero righe

```
SELECT *
FROM   dbo.Fatture
WHERE  Iva = NULL;
```

```
SELECT *
FROM   dbo.Fatture
WHERE  Iva <> NULL;
```

Filtri sui NULL

Visualizzare tutte le righe della tabella *Fatture* dove la colonna *Iva* è *NULL*.

```
SELECT *
FROM   dbo.Fatture
WHERE  Iva IS NULL;
```

Filtri sui NOT NULL

Visualizzare tutte le righe della tabella *Fatture* dove la colonna *Iva* NON è *NULL*.

```
SELECT *
FROM   dbo.Fatture
WHERE  Iva IS NOT NULL;
```

Ricerca all'interno di stringhe

Estrarre *nome* e *cognome* dei *clienti* il cui nome contiene la lettera "*n*"

```
SELECT Nome, Cognome
FROM   dbo.Clienti
WHERE  Nome LIKE '%n%';
```

Commento

Abbiamo utilizzato un nuovo operatore di confronto: LIKE. Per cercare i nomi che INIZIANO con "n" avrei dovuto scrivere

```
WHERE Nome LIKE 'n%'
```

Per cercare i nomi che TERMINANO con "n" avrei dovuto scrivere

```
WHERE Nome LIKE '%n'
```

Invece con il filtro usato nella query

```
WHERE Nome LIKE '%n%'
```

estraiamo i record con la n in qualsiasi posizione (iniziale, finale e al centro)

Raggruppare i dati

Raggruppare i dati

Riportare per ogni *anno* il numero di *fatture* emesse

```
SELECT  YEAR(DataFattura) as Anno,
        COUNT(*) AS Conteggio
FROM    dbo.Fatture
GROUP BY YEAR(DataFattura);
```

Commento 1

Poiché la domanda chiede di riportare delle informazioni *"per ogni anno"*, l'output della nostra query dovrà avere tante righe quanti sono i valori distinti dell'espressione *YEAR(DataFattura)*.

Nella tabella *dbo.Fatture* abbiamo di default una riga per ogni fattura, per restituire un output che abbia una riga per ogni anno devo inserire la clausola *GROUP BY*.

Commento 2

Il processo di scrittura di questa query prevede quindi di

1) compilare la clausola FROM

2) compilare la clausola GROUP BY

3) fare copia e incolla nella SELECT di ciò che c'è nella GROUP BY

4) aggiungere nella SELECT altre informazioni aggregate utilizzando SUM, AVG, MIN, MAX, COUNT.

COUNT(*) conta in generale il numero di righe in cui ogni valore della colonna raggruppata è presente. Poiché nella tabella delle *fatture* c'è una riga per ogni fattura, in questo contesto *COUNT(*)* conta il numero di fatture.

Commento 3

Facciamo un'ulteriore considerazione sulla frase *"fare copia e incolla nella SELECT di ciò che c'è nella GROUP BY"*

Verifichiamo infatti che provando a lanciare la query senza GROUP BY otteniamo un errore.

```
SELECT YEAR(DataFattura) as Anno,
    COUNT(*) AS Conteggio
FROM   dbo.Fatture;
```

Commento 4

Invece se non inseriamo le colonne raggruppate nella SELECT, l'output della query diventa impossibile da interpretare.

```
SELECT    COUNT(*) AS Conteggio
FROM      dbo.Fatture
GROUP BY YEAR(DataFattura);
```

Raggruppamenti con la somma

Riportare per ogni *anno* il numero di *fatture* emesse e la somma degli *importi*

```
SELECT YEAR(DataFattura) AS Anno,
    COUNT(*) AS Conteggio,
    SUM(Importo) AS TotaleImporto
FROM      dbo.Fatture
GROUP BY YEAR(DataFattura);
```

Un altro esempio

Riportare il numero di *clienti* per ogni *regione*

```
SELECT    RegioneResidenza,
    COUNT(*) AS NumeroClienti
FROM      dbo.Clienti
GROUP BY RegioneResidenza;
```

Aggregazioni senza raggruppamenti

Visualizzare il numero di *righe* presenti nella tabella *Fatture*

```
SELECT COUNT(*) AS NumeroRighe
FROM   dbo.Fatture;
```

Commento

Le funzioni di aggregazione COUNT, MIN, MAX, SUM, AVG possono essere utilizzate anche senza GROUP BY. Il risultato sarà una sola riga contenente il valore totale per tutto il contenuto della tabella (o per quelle righe che rispettano un'eventuale condizione scritta nella WHERE).

La query seguente invece restituisce un errore perché mescola una funzione di aggregazione e una colonna non raggruppata:

```
SELECT COUNT(*) AS NumeroRighe, Importo
FROM   dbo.Fatture;
```

Count Distinct

Riportare il numero di *clienti* associati ad almeno una *fattura* con importo superiore a *50* euro.

```sql
SELECT COUNT(DISTINCT IdCliente) AS NumeroClienti
FROM   dbo.Fatture
WHERE  Importo > 50;
```

Commento 1

Occorre fare molta attenzione perché in questo caso COUNT(*) non restituisce il numero corretto. Il ragionamento è il seguente:

- COUNT(*) conta il numero di righe
- nella tabella fatture c'è una riga per ogni fattura
- quindi in questo caso COUNT(*) è il numero di fatture, non il numero di clienti.

Un cliente avrebbe potuto fare più fatture con importo superiore a 50€.

Per conteggiarlo una volta sola è necessario scrivere

SELECT COUNT(DISTINCT IdCliente).

Commento 2

Riscrivere la query senza Distinct non produce il risultato corretto. Infatti scrivere

```
SELECT COUNT(IdCliente)
FROM   dbo.Fatture
WHERE  Importo > 50;
```

è solo una "scorciatoia" equivalente alla query

```
SELECT COUNT(*)
FROM   dbo.Fatture
WHERE  Importo > 50
    AND IdCliente IS NOT NULL;
```

Aggregazione + Where

Calcolare la somma degli *importi* delle *fatture* del *2022*.

```
SELECT SUM(Importo) AS Importo
FROM   dbo.Fatture
WHERE  YEAR(DataFattura) = 2022;
```

Aggregazione + Where + Group by

Riportare per ogni *fornitore* il numero di *fatture* emesse a *marzo 2022*

```
SELECT IdFornitore,
   COUNT(*) AS Conteggio
FROM  dbo.Fatture
WHERE YEAR(DataFattura) = 2022
   AND MONTH(DataFattura) =  3
GROUP BY IdFornitore;
```

Raggruppamenti su più colonne

Riportare la somma degli *importi* raggruppati per *anni* e *mesi*, considerando soltanto le *fatture* di *tipologia A*.

```
SELECT YEAR(DataFattura) AS Anno,
    MONTH(DataFattura) AS Mese,
    SUM(Importo) AS Fatturato
FROM  dbo.Fatture
WHERE Tipologia = 'A'
GROUP BY YEAR(DataFattura),
    MONTH(DataFattura);
```

Commento

Il risultato della query precedente (come quello di una QUALSIASI ALTRA QUERY) non ha un ordinamento pronosticabile. Per specificare un ordinamento occorre inserire una clausola ORDER BY. Occorre sempre valutare se è preferibile eseguire l'ordinamento in un linguaggio di front end, e non direttamente nel database.

```
SELECT YEAR(DataFattura) AS Anno,
   MONTH(DataFattura) AS Mese,
   SUM(Importo) AS Fatturato
FROM  dbo.Fatture
WHERE Tipologia = 'A'
GROUP BY YEAR(DataFattura),
   MONTH(DataFattura)
ORDER BY Anno, Mese;
```

Group by + order by + top

Riportare l'*anno* con il *fatturato* maggiore

```
SELECT TOP 1 YEAR(DataFattura) AS Anno,
      SUM(Importo) AS Fatturato
FROM      dbo.Fatture
GROUP BY YEAR(DataFattura)
ORDER BY Fatturato DESC;
```

Group by per analizzare colonne

Approfondiamo il contenuto della colonna *Regione* della tabella *Clienti* (variabile qualitativa)

```
SELECT    RegioneResidenza,
      COUNT(*) AS FrequenzaAssoluta
FROM      dbo.Clienti
GROUP BY RegioneResidenza
ORDER BY FrequenzaAssoluta DESC;
```

Group by per analizzare colonne

Ripetiamo l'analisi sulla colonna *Iva* della tabella *Fatture*. Osserviamo che questa tipologia di query mostra anche i NULL

```
SELECT    Iva,
      COUNT(*) AS FrequenzaAssoluta
FROM      dbo.Clienti
GROUP BY Iva
ORDER BY FrequenzaAssoluta DESC;
```

Group by per analizzare colonne

Approfondiamo il contenuto della colonna *Importo* (variabile quantitativa) della tabella *Fatture*

```
SELECT    SUM(Importo) AS Totale,
    COUNT(Importo) AS NumeroNonNull,
    COUNT(*) - COUNT(Importo) AS NumeroNull,
    AVG(Importo) AS Media,
    STDEVP(Importo) AS DeviazioneStandard
FROM      dbo.Fatture;
```

Commento

Se la colonna importo fosse un intero, allora su SQL Server prima di applicare la media devo fare una conversione tramite la funzione **CONVERT**

```
SELECT    SUM(Importo) AS Totale,
  COUNT(Importo) AS NumeroNonNull,
  COUNT(*) - COUNT(Importo) AS NumeroNull,
  AVG(CAST(Importo AS DECIMAL(18,2))) AS Media,
  STDEVP(Importo) AS DeviazioneStandard
FROM      dbo.Fatture;
```

Espressione Case When

Visualizzare le colonne *IdFattura*, *Importo*, *Tipologia* e *Tipologia_descrizione* della tabella delle Fatture. Valorizzare *la Tipologia_descrizione* in questo modo:

- se *tipologia* = 'A' allora 'Acquisto'

- se *tipologia* = 'V' allora 'Vendita'

```
SELECT IdFattura,
  Importo,
  Tipologia,
  CASE WHEN Tipologia = 'A' THEN 'Acquisto'
       WHEN Tipologia = 'V'  THEN 'Vendita'
       ELSE NULL
  END AS Tipologia_descrizione
FROM  dbo.Fatture;
```

Case When e Group By

Classificare le *fatture* in tre categorie di prezzo

- Basso se l'*importo* è compreso tra 0 e 30 (0 incluso, 30 escluso)
- Medio se l'*importo* è compreso tra 30 e 70 (30 incluso, 70 escluso)
- Alto se l'*importo* è maggiore o uguale di 70.

Contare il numero di *fatture* per ogni categoria.

Nella prossima slide...

Case When e Group By

```sql
SELECT
    CASE WHEN Importo >= 0 AND Importo < 30 THEN 'Basso'
         WHEN Importo >= 30 AND Importo < 70 THEN 'Medio'
         WHEN Importo >= 70 THEN 'Alto'
         ELSE 'Non classificata'
    END AS Tipologia_cliente,
    COUNT(*) AS Numero
FROM dbo.Fatture
GROUP BY
    CASE WHEN Importo >= 0 AND Importo < 30 THEN 'Basso'
         WHEN Importo >= 30 AND Importo < 70 THEN 'Medio'
         WHEN Importo >= 70 THEN 'Alto'
         ELSE 'Non classificata'
    END;
```

Combinare i dati di più tabelle

Combinare dati da più fonti

Per riportare in una sola query le informazioni contenute in più colonne, le abbiamo scritte separate con una virgola all'interno della *SELECT*.

Analogamente, per riportare informazioni contenute in più tabelle, potremmo pensare di scriverle nella *FROM* e separarle anche in questo caso da una virgola.

Lanciamo le tre query in basso e analizziamo il risultato

```
SELECT * FROM dbo.Fatture;
```

```
SELECT * FROM dbo.Clienti;
```

```
SELECT *FROM  dbo.Fatture, dbo.Clienti;
```

Due tabella nella FROM

L'ultima query combina i dati delle due tabelle. Tuttavia molti record non hanno ragione di esistere. Ad esempio a lato dei dati della *fattura 1*, vorremmo vedere solo i dati del *cliente 1* perché è quello l'unico cliente a cui è stata emessa la fattura (questa informazione è presente nella colonna *IdCliente* della tabella *Fatture*).

161 %

Risultati | Messaggi

	IdFattura	Tipologia	Importo	Iva	IdCliente	DataFattura	IdFornitore	IdCliente	Nome	Cognome	DataNascita	RegioneResidenza
1	1	A	120.00	20	1	2018-01-01	1	1	Nicola	Iantomasi	1980-10-17	NULL
2	2	V	32.00	20	2	2017-03-01	1	1	Nicola	Iantomasi	1980-10-17	NULL
3	3	A	45.00	20	3	2017-06-01	1	1	Nicola	Iantomasi	1980-10-17	NULL
4	4	V	67.00	20	3	2019-01-30	1	1	Nicola	Iantomasi	1980-10-17	NULL
5	5	A	12.00	20	5	2018-01-01	1	1	Nicola	Iantomasi	1980-10-17	NULL
6	6	V	31.00	20	6	2017-03-01	2	1	Nicola	Iantomasi	1980-10-17	NULL
7	7	A	12.00	20	3	2017-06-01	2	1	Nicola	Iantomasi	1980-10-17	NULL
8	8	V	54.00	20	8	2019-01-30	2	1	Nicola	Iantomasi	1980-10-17	NULL

Filtriamo il risultato

A tal fine, potrei pensare di filtrare i dati così ottenuti con una *WHERE*. Lanciamo questa query, il risultato è quello desiderato

```
SELECT *
FROM   dbo.Fatture, dbo.Clienti
WHERE  dbo.Fatture.IdCliente = dbo.Clienti.IdCliente;
```

Risultati | Messaggi

	IdFattura	Tipologia	Importo	Iva	IdCliente	DataFattura	IdFornitore	IdCliente	Nome	Cognome	DataNascita	RegioneResidenza
1	1	A	120.00	20	1	2018-01-01	1	1	Nicola	Iantomasi	1980-10-17	NULL
2	2	V	32.00	20	2	2017-03-01	1	2	Francesco	Rossi	1982-12-17	Molise
3	3	A	45.00	20	3	2017-06-01	1	3	Alberto	Verdi	1983-01-03	Campania
4	4	V	67.00	20	3	2019-01-30	1	3	Alberto	Verdi	1983-01-03	Campania
5	5	A	12.00	20	5	2018-01-01	1	5	Angelo	Verdi	1980-04-07	Puglia
6	6	V	31.00	20	6	2017-03-01	2	6	Alberto	Marino	1980-02-03	Lombardia
7	7	A	12.00	20	3	2017-06-01	2	3	Alberto	Verdi	1983-01-03	Campania
8	8	V	54.00	20	8	2019-01-30	2	8	Giacomo	Sacco	1980-10-11	Lombardia
9	9	A	67.00	20	3	2018-01-01	2	3	Alberto	Verdi	1983-01-03	Campania

Sintassi con la Join

Scriviamo ora un'altra sintassi completamente equivalente per raggiungere lo stesso obiettivo

```
SELECT *
FROM   dbo.Fatture
INNER JOIN dbo.Clienti
    ON dbo.Fatture.IdCliente = dbo.Clienti.IdCliente;
```

Un primo esercizio con la Join

Visualizzare tutte le colonne della tabella fatture aggiungendo il nome e il cognome dei clienti.

```
SELECT dbo.Fatture.*,
       dbo.Clienti.Nome,
       dbo.Clienti.Cognome
FROM dbo.Fatture
 INNER JOIN dbo.Clienti
   ON dbo.Fatture.IdCliente = dbo.Clienti.IdCliente;
```

Un secondo esercizio

Visualizzare tutte le colonne della tabella fatture aggiungendo il *nome* del *fornitore*. Sembra simile al precedente, ma facciamo attenzione al risultato!

```
SELECT dbo.Fatture.*,
       dbo.Fornitori.Denominazione
FROM   dbo.Fatture
INNER JOIN dbo.Fornitori
   ON dbo.Fatture.IdFornitore = dbo.Fornitori.IdFornitore;
```

Commento

Le fatture *12* e *13* sono sparite. Analizzando i record sulla sola tabella fattura, vediamo che non hanno un fornitore associato.

```
SELECT *
FROM   dbo.Fatture
WHERE  IdFattura IN (12, 13);
```

Risultati | Messaggi

	IdFattura	Tipologia	Importo	Iva	IdCliente	DataFattura	IdFornitore
1	12	A	57.00	20	7	2019-01-30	NULL
2	13	V	87.00	20	2	2016-01-03	NULL

Commento

Utilizzando una inner join queste due fatture NON SARANNO VISIBILI.
Prima abbiamo detto che la query con la INNER JOIN è equivalente a

```
SELECT dbo.Fatture.*, dbo.Fornitori.Denominazione
FROM   dbo.Fatture, dbo. Fornitori
WHERE  dbo.Fatture.IdFornitore = dbo.Fornitori.IdFornitore;
```

e abbiamo già visto che un filtro del tipo "*colonna* = *null*" non restituisce mai risultati.

Per preservare le due fatture senza fornitori associati devo cambiare la tipologia di JOIN: da INNER JOIN devo passare a **LEFT JOIN**.

Ecco un primo grande vantaggio della sintassi che utilizza le tipologie di JOIN invece della virgola e del WHERE

Left Join

Ecco la nuova versione della query con la Left Join

```
SELECT dbo.Fatture.*,
       dbo.Fornitori.Denominazione
FROM   dbo.Fatture
LEFT JOIN dbo.Fornitori
   ON dbo.Fatture.IdFornitore = dbo.Fornitori.IdFornitore;
```

Commento

Utilizzando *Left join* **saranno visibili anche** le fatture per le quali la colonna IdFornitore:

1) è *null*

2) contiene valori non presenti nella colonna *IdFornitore* della tabella *Fornitori*

Ricorda:

Se non specifico la tipologia di Join, sarà eseguita una *Inner Join*.

Per ragioni di chiarezza, leggibilità e manutenibilità del codice, consiglio di specificare sempre la tipologia di *Join*.

Esercizio 3: join + where

Estrarre tutte le fatture emesse a *clienti* nati a *febbraio*. Visualizzare tutte le colonne della tabella *Fatture* aggiungendo il *nome* e il *cognome* dei *clienti*.

```
SELECT dbo.Fatture.*,
       dbo.Clienti.Nome,
       dbo.Clienti.Cognome
FROM   dbo.Fatture
INNER JOIN dbo.Clienti
   ON dbo.Fatture.IdCliente = dbo.Clienti.IdCliente
WHERE  MONTH(dbo.Clienti.DataNascita) = 2;
```

Commento

In questo caso non ha senso utilizzare la *Left Join* perché, anche se ci fosse una fattura senza cliente associato (il che sarebbe indice di un database con gravi problemi di qualità dei dati) quest'ultima non potrebbe rispettare il filtro richiesto (cliente associato nato a febbraio).

Utilizzo di AS nella From

Possiamo alleggerire notevolmente il codice delle query contenenti *Join* utilizzando l'AS anche nella From. La query precedente può essere ad esempio riscritta così

```
SELECT Fa.*,
       Cl.Nome,
       Cl.Cognome
FROM   dbo.Fatture AS Fa
INNER JOIN dbo.Clienti AS Cl
   ON Fa.IdCliente = Cl.IdCliente
WHERE  MONTH(Cl.DataNascita) = 2;
```

Esercizio 4: Join + Where + Group by

Riportare per ogni anno il numero di *clienti* nati nel *1980* che hanno associate almeno una *fattura* con *importo* superiore a 10 euro

```
SELECT YEAR(f.DataFattura) AS Anno,
       COUNT(DISTINCT f.IdCliente) AS NumeroClienti
FROM   dbo.Fatture AS f
INNER JOIN dbo.Clienti AS c
    ON f.IdCliente = c.IdCliente
WHERE YEAR(C.DataNascita) = 1980
   AND f.importo > 10
GROUP BY YEAR(f.DataFattura);
```

Commento

Il procedimento logico per scrivere la query può essere:

1) compilare la clausola *From* inserendo le tabelle nelle quali sono presenti i dati oggetto di analisi.

Se abbiamo più di una tabella, combinarle con un opportuna *Join*. Nel nostro caso

```
FROM   dbo.Fatture as f
INNER JOIN dbo.Clienti as c
   ON F.IdCliente = C.IdCliente
```

Commento

2) inserire nella clausola *Where* i filtri pre-raggruppamento che individuano il perimetro di analisi. Nel nostro caso

```
WHERE YEAR(C.DataNascita) = 1980
  AND   F.importo > 10
```

Commento

3) la combinazione delle due tabelle inserite nella *From* porta ad avere una fonte dati con una riga per ogni fattura. Poiché l'estrazione richiesta vuole i dati divisi per anno, dovrò raggruppare per anno.

Nel nostro caso

```
GROUP BY YEAR(f.DataFattura)
```

Commento

4) Copiare e incollare quanto scritto nella *Group By* nella *Select*. A questo punto aggiungere eventuali altre informazioni aggregate richieste tramite SUM, MIN, AVG, COUNT.

Nel nostro caso devo riportare il numero di clienti. Dovrò scrivere quindi

```
SELECT YEAR(DataFattura), COUNT(DISTINCT F.IdCliente)
```

Commento

5) Inserire rinomine nella *Select* con AS per rendere l'output più chiaro. Nel nostro caso

```
SELECT YEAR(f.DataFattura) AS Anno,
    COUNT(Distinct IdCliente) AS NumeroClienti
```

Altre tipologie di Join

Le due query seguenti sono equivalenti:

```
SELECT dbo.Fatture.*, dbo.Fornitori.Denominazione
FROM   dbo.Fatture
LEFT JOIN dbo.Fornitori
   ON dbo.Fatture.IdFornitore = dbo.Fornitori.IdFornitore;
```

```
SELECT dbo.Fatture.*, dbo.Fornitori.Denominazione
FROM   dbo.Fornitori
RIGHT JOIN dbo.Fatture
   ON dbo.Fornitori.IdFornitore = dbo.Fatture.IdFornitore;
```

Commento

La ***Right Join*** infatti restituisce:

- le combinazioni che rispettano la condizione di join
- le righe della tabella di destra che non hanno righe associate nella tabella sinistra secondo la condizione di join.

Se voglio preservare le righe di entrambe le tabelle che non hanno combinazioni disponibili devo usare una ***Full Join***.

Commento

Mettiamo a confronto queste quattro query:

- Solo combinazioni valide

```
SELECT dbo.Fatture.*, dbo.Fornitori.Denominazione
FROM   dbo.Fatture
INNER JOIN dbo.Fornitori
   ON dbo.Fatture.IdFornitore = dbo.Fornitori.IdFornitore;
```

- Combinazioni valide più fatture senza fornitori associati

```
SELECT dbo.Fatture.*, dbo.Fornitori.Denominazione
FROM   dbo.Fatture
LEFT JOIN dbo.Fornitori
   ON dbo.Fatture.IdFornitore = dbo.Fornitori.IdFornitore;
```

Commento

- Combinazioni valide più fornitori senza fatture associate

```
SELECT dbo.Fatture.*, dbo.Fornitori.Denominazione
FROM   dbo.Fatture
RIGHT JOIN dbo.Fornitori
   ON dbo.Fatture.IdFornitore = dbo.Fornitori.IdFornitore;
```

- Combinazioni valide più fatture senza fornitori associati più fornitori senza fatture associate

```
SELECT dbo.Fatture.*, dbo.Fornitori.Denominazione
FROM   dbo.Fatture
FULL JOIN dbo.Fornitori
   ON dbo.Fatture.IdFornitore = dbo.Fornitori.IdFornitore;
```

Commento

L'ultima tipologia di Join è la ***Cross Join***, equivalente a separare le tabelle con una virgola. Queste due query sono infatti equivalenti

```
SELECT dbo.Fatture.*, dbo.Fornitori.Denominazione
FROM   dbo.Fatture, dbo.Fornitori;
```

```
SELECT dbo.Fatture.*, dbo.Fornitori.Denominazione
FROM   dbo.Fatture
CROSS JOIN dbo.Fornitori;
```

Esercizio 5: union all

Riportare nome e cognome dei soggetti presenti nella tabella *clienti* e nella tabella *prospect*

```
SELECT Nome, Cognome
FROM   dbo.Clienti
   UNION ALL
SELECT Nome, Cognome
FROM   dbo.Prospect;
```

Commento

Spesso dobbiamo combinare le tabelle accodando semplicemente i dati "gli uni sotto gli altri". Per far questo basta inserire tra le due query la parola chiave ***Union All***.

L'unico vincolo consiste nel fatto che le due query devono avere entrambe lo stesso numero di colonne indicate nella *Select*.

Esercizio 6:

Riportare *nome* e *cognome* dei soggetti presenti nella tabella *clienti* e nella tabella *prospect*, indicando la provenienza in una terza colonna

```
SELECT Nome, Cognome, 'Clienti' AS TabellaProvenienza
FROM   dbo.Clienti
   UNION ALL
SELECT Nome, Cognome, 'Prospect' AS TabellaProvenienza
FROM   dbo.Prospect;
```

Commento

Questo esempio ci mostra come sia possibile inserire un valore costante all'interno di una query, semplicemente racchiudendolo tra apici e assegnandogli un nome con l'AS.

Le colonne di output seguono il nome della prima dbo. Attenzione quindi ad "errori di distrazione". La query seguente infatti viene eseguita senza errori, produce un output con le colonne Nome e Cognome ma contiene informazioni sicuramente errate.

```
SELECT Nome, Cognome
FROM   dbo.Clienti
   UNION ALL
SELECT Cognome, Nome
FROM   dbo.Prospect;
```

Commento

Occorre fare attenzione a non confondere ***Union All*** con ***Union***.

La query seguente che utilizza ***Union*** ha l'effetto aggiuntivo di eliminare tutti i duplicati dal risultato finale, sia quelli generati dopo l'unione, sia quelli eventualmente presenti già nelle tabelle di partenza.

```
SELECT Nome, Cognome
FROM   dbo.Clienti
   UNION
SELECT Cognome, Nome
FROM   dbo.Prospect;
```

Except e intersect

Analogamente ad *Union*, possiamo utilizzare anche ***Intersect*** ed ***Except***.

Intersect riporta i valori comuni alle due query, mentre *Except* riporta i valori presenti nella prima query e non presenti nella seconda.

Entrambe *Intersect* ed *Except*, come *Union*, hanno l'effetto aggiuntivo di eliminare i duplicati dall'output finale.

```
SELECT Nome, Cognome
FROM   dbo.Clienti
  INTERSECT
SELECT Nome, Cognome
FROM   dbo.Prospect;
```

Except e intersect

Query con Except

```
SELECT Nome, Cognome
FROM   dbo.Clienti
  EXCEPT
SELECT Nome, Cognome
FROM   dbo.Prospect;
```

Attenzione agli errori nelle Join!

Facciamo attenzione perché la query seguente compila, ma il risultato ottenuto è privo di senso! Guardiamo ad esempio la fattura 7.

```
SELECT *
FROM   dbo.Fatture AS F
INNER JOIN dbo.Clienti AS C
    ON F.IdFattura = C.IdCliente;
```

Risultati | Messaggi

	IdFattura	Tipologia	Importo	Iva	IdCliente	DataFattura	IdFornitore
1	12	A	57.00	20	7	2019-01-30	NULL
2	13	V	87.00	20	2	2016-01-03	NULL

Commento

Ricorda: le colonne che utilizzo nella condizioni di JOIN devono contenere sempre la stessa tipologia di informazione. Ma da solo questo non basta! Ecco un altro esempio potenzialmente errato.

Mi risulta molto difficile trovare un esercizio per cui questa operazioni abbia senso

```
SELECT *
FROM dbo.Clienti AS C
INNER JOIN dbo.Fornitori AS F
   ON C.RegioneResidenza = F.RegioneResidenza;
```

Consiglio: controlliamo sempre come cambia il conteggio dei dati dopo la JOIN e cerchiamo di capire perché. Potrebbero essere svariati motivi

Commento

Quando tratteremo le ***progettazione*** di un database e i concetti di ***chiave primaria*** e ***chiave esterna*** ci sarà ancora più chiaro il criterio con cui scrivere una condizione nella Join.

Esercizio 7: Having

Riportare le regioni con almeno tre Clienti

```
SELECT RegioneResidenza,
    COUNT(*) AS NumeroClienti
FROM   dbo.Clienti
GROUP BY RegioneResidenza
HAVING COUNT(*)>=3;
```

Commento

In questo caso ci viene richiesto il filtro "riportare solo le regioni che hanno almeno tre clienti"

Osserviamo che questo filtro è diverso da quelli visti negli esercizi precedenti, in quanto deve essere applicato su un insieme di dati già raggruppato.

Si tratta infatti di un filtro "sui dati divisi per regione", e non sui dati originali della tabella clienti dove è presente una riga per ogni cliente.

Per questa nuova tipologia di filtri su dati raggruppati occorre

utilizzare la clausola HAVING.

Esercizio 8

Riportare il fatturato del 2018 per ogni fornitore che ha registrato in quell'anno almeno 3 fatture.

```
SELECT    IdFornitore,
      SUM(Importo) AS Fatturato
FROM      dbo.Fatture
WHERE     DataFattura >= '20180101'
   AND    DataFattura < '20190101'
GROUP BY IdFornitore
HAVING    COUNT(*)>=3;
```

Esercizio 9

Riportare l'anno con il fatturato maggiore

```
SELECT TOP 1 YEAR(DataFattura) AS Anno,
     SUM(Importo) AS Fatturato
FROM     dbo.Fatture
GROUP BY YEAR(DataFattura)
ORDER BY Fatturato DESC;
```

Combinando le clausole ORDER BY e TOP possiamo svolgere tutta una classe di analisi come quella appena vista

Esercizio 10

Riportare la somma degli importi raggruppati per anni e mesi, considerando soltanto le fatture di tipologia A.

```
SELECT YEAR(DataFattura) AS Anno,
   MONTH(DataFattura) AS Mese,
   SUM(Importo) AS Fatturato
FROM  dbo.Fatture
WHERE Tipologia = 'A'
GROUP BY YEAR(DataFattura),
   MONTH(DataFattura);
```

Commento

Il risultato della query precedente (come quello di una qualsiasi altra query) non ha un ordinamento pronosticabile e deterministico.

Per specificare un ordinamento occorre inserire una clausola ORDER BY. Occorre sempre valutare se è preferibile eseguire l'ordinamento in un linguaggio di front end e non nel database.

```
SELECT YEAR(DataFattura) AS Anno,
    MONTH(DataFattura) AS Mese,
    SUM(Importo) AS Fatturato
FROM  dbo.Fatture
WHERE Tipologia = 'A'
GROUP BY YEAR(DataFattura),
    MONTH(DataFattura)
ORDER BY Anno, Mese;
```

Alcune funzioni utili

Finora abbiamo incontrato funzioni di aggregazioni:

SUM, AVG, COUNT, MAX e MIN

E funzioni per le date

YEAR, MONTH, DAY, DATEDIFF, DATEADD, GETDATE

Esistono ovviamente anche funzioni per le stringhe e i numeri

LEFT, RIGHT, SUBSTRING, CONCAT, CHARINDEX, POWER, ABS, +, -, *, &

E per la gestione dei Null

COALESCE, ISNULL, NULLIF

Esercizi

- Estrarre il nome e il cognome dei clienti nati nel 1980

- Estrarre una colonna di nome "Denominazione" contenente il nome, seguito da un carattere "-", seguito dal cognome, per i soli clienti residenti nella regione Lombardia

- Qual è il numero di fatture con iva al 20%

- Riportare il numero di fatture e la somma dei relativi importi divisi per anno di fatturazione.

- Estrarre i prodotti attivati nel 2017 e che sono in produzione oppure in commercio

Esercizi

- Estrarre il totale degli importi delle fatture divisi per residenza dei clienti

- Estrarre il totale, la media e la deviazione standard degli importi divisi per tipologia di fattura

- Considerando soltanto le fatture con iva al 20 per cento, qual è il numero di fatture per ogni anno?

- In quali anni sono state registrate più di 2 fatture con tipologia 'A'?

- In quali regioni sono stati registrate fatture per più di 100 euro?

Esercizi

- Riportare l’elenco delle fatture (Id, importo, iva e data) con in aggiunta il nome del fornitore

- Riportare per ogni fornitore, il relativo nome, regione di residenza e numero di fatture associate. Non mostrare i fornitori con meno di tre fatture

Subquery e CTE

Ripartiamo da una query SQL

```sql
SELECT TOP 1
   C.RegioneResidenza,
   COUNT(*) AS NumeroFatture
FROM  dbo.Fatture AS F
INNER JOIN dbo.Clienti AS C
   ON F.IdCliente = C.IdCliente
WHERE Tipologia = 'A'
GROUP BY C.RegioneResidenza
ORDER BY NumeroFatture DESC;
```

Risolvere problemi complessi combinando soluzioni semplici

Le istruzioni viste nella query precedente sono i mattoni fondamentali con i quali è possibile risolvere un'elevata percentuale di problemi legati all'analisi dichiarativa dei dati.

SELECT – FROM – JOIN - WHERE - GROUP BY – HAVING – ORDER BY - TOP

Per affrontare problemi **più complessi** spesso la strada da seguire sarà **suddividerli in sotto-problemi più semplici**, affrontando ognuno di loro con la classica sintassi, e infine *"mettendo insieme i singoli step in un'unica soluzione"*.

In questa sezione vedremo proprio come *"mettere insieme"* più query semplici per risolvere problemi complessi

Problema: calcolare il fatturato annuo medio

Affrontare il problema in una sola query è quasi impossibile. Spesso si finisce per scrivere una query del genere che **restituisce un errore** di esecuzione.

```
SELECT YEAR(DataFattura) as Anno,
    AVG(SUM(Importo)) as FatturatoMedio
FROM     dbo.Fatture
GROUP BY YEAR(DataFattura);
```

Strategia di soluzione

Invece di provare a risolvere il tutto con una sola query, dividiamo il problema in due sotto-problemi:

1) calcolare la somma degli importi delle fatture divisi per anni

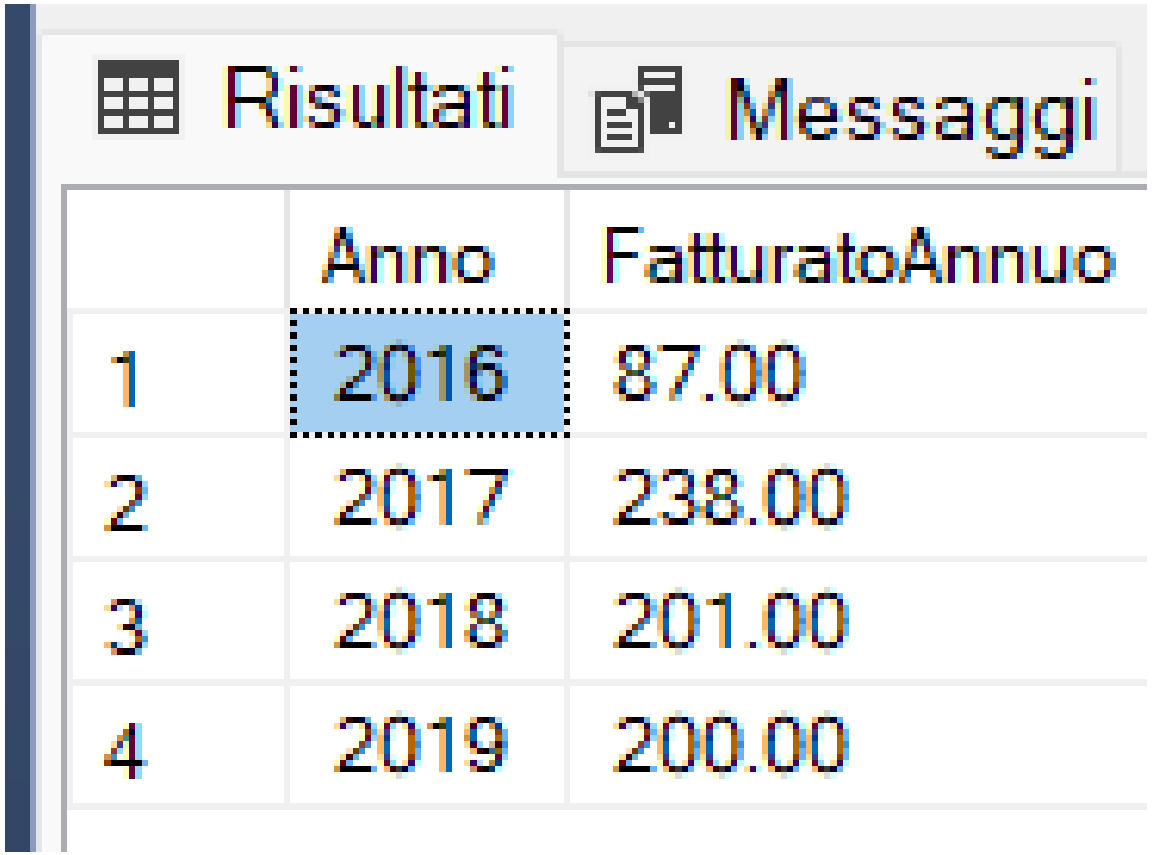
2) calcolare la media di questi valori

Il primo sotto problema è risolvibile con questa semplice query

```
SELECT YEAR(DataFattura) as Anno,
   SUM(Importo) as FatturatoAnnuo
FROM   dbo.Fatture
GROUP BY YEAR(DataFattura);
```

L'output di una query somiglia a una tabella

Osserviamo il risultato di questa dbo.


Risultati Messaggi

	Anno	FatturatoAnnuo
1	2016	87.00
2	2017	238.00
3	2018	201.00
4	2019	200.00

Immaginiamo fosse effettivamente una tabella

Ci appare in tutto e per tutto simile al contenuto di una tabella.

Immaginiamo per un attimo di avere un'ipotetica tabella di nome "Step1" con questi dati. Come risolvereste il secondo problema? Basterebbe scrivere

```
SELECT AVG(FatturatoAnnuo) AS FatturatoAnnuoMedio
FROM   Step1;
```

È sicuramente impensabile creare realmente una tabella per ogni passo intermedio di ogni singolo problema. In pochi giorni avremmo migliaia di tabelle e il database diventerebbe inutilizzabile.

Vediamo **quattro strade alternative** per risolvere questo problema.

Utilizzare le Subquery

Utilizziamo una *SubQuery*

```
SELECT AVG(FatturatoAnnuo) AS FatturatoAnnuoMedio
FROM  (
       SELECT YEAR(DataFattura) as Anno,
           SUM(Importo) as FatturatoAnnuo
       FROM   dbo.Fatture
       GROUP BY YEAR(DataFattura)
      ) AS Step1;
```

Commento alle Subquery

Siccome abbiamo detto che il risultato di una query si mostra in tutto e per tutto come una tabella, l'idea è quello di inserire la query direttamente nella clausola FROM di una query più esterna. Abbiamo in questo caso una ***subdbo*.**

Dobbiamo seguire solo qualche vincolo di sintassi:

- racchiudere la query interna tra parantesi tonde

- assegnarle un nome con AS

- assegnare un nome a tutte le colonne delle subquery che usano una funzione

- non utilizzare la clausola *Order By* nella subquery (a meno che non sia associata ad una TOP)

Utilizzare le CTE

Utilizziamo una *CTE (Common Table Expression)*

```
WITH Step1 as
(
   SELECT YEAR(DataFattura) as Anno,
      SUM(Importo) as FatturatoAnnuo
   FROM   dbo.Fatture
   GROUP BY YEAR(DataFattura)
)
SELECT AVG(FatturatoAnnuo) AS FatturatoAnnuoMedio
FROM   Step1;
```

Commento alle CTE

Questa tecnica è praticamente identica alla Subdbo. L'unica differenza è che il codice degli step intermedi va scritto *"sopra"* la query principale, rendendo il tutto più leggibile.

Utilizzare le tabelle temporanee

Utilizziamo una *tabella temporanea*

```
SELECT   YEAR(DataFattura) as Anno,
   SUM(Importo) as FatturatoAnnuo
INTO     #Step1
FROM     dbo.Fatture
GROUP BY YEAR(DataFattura)

SELECT AVG(FatturatoAnnuo) AS FatturatoAnnuoMedio
FROM   #Step1;
```

Attenzione! Affinché venga creata una tabella **temporanea** è fondamentale che il nome della tabella indicata nella INTO inizi con il carattere #

Commento alle tabelle temporanee

Questa tecnica crea una vera e propria tabella. Infatti, inserire la clausola INTO prima della FROM ha l'effetto di creare e popolare una tabella con il risultato della dbo. Se il nome di tale tabella inizia con un *singolo* simbolo #, si tratterà però di una **tabella temporanea**.

Essa ha due caratteristiche principali:

1) viene eliminata automaticamente appena ci disconnettiamo dal database

2) non è visibile ad altri utenti (a meno che non inseriamo all'inizio un doppio cancelletto, ad esempio ##step1)

È importante osservare che, seppur temporaneamente, **i dati occuperanno spazio sul disco**, in un area di memoria **condivisa da tutti i database dell'istanza**. Attenzione a non utilizzarle per dati di grandi dimensioni!!!

Utilizzare le viste

Utilizziamo una vista

```
CREATE VIEW dbo.FatturatoAnnuo AS
SELECT   YEAR(DataFattura) as Anno,
   SUM(Importo) as FatturatoAnnuo
FROM     dbo.Fatture
GROUP BY YEAR(DataFattura)

SELECT AVG(FatturatoAnnuo) AS FatturatoAnnuoMedio
FROM   dbo.FatturatoAnnuo;
```

Attenzione! Le viste sono oggetti del Database non temporanei

Commento alle viste

Le viste (come le *stored procedure*) sono oggetti *ufficiali* del database. Anche se non occupano memoria in quanto è solo il codice ad essere salvato e non i dati, entreranno nelle procedure di backup, dovranno esserne regolati i diritti di visibilità, eccetera... Quindi occhio a non esagerare nella creazione di viste e ad avere l'ok dal resto del team.

Abbiamo visto che per riutilizzare il codice salvato nella vista, basterà fare riferimento al nome della vista all'interno della clausola FROM di una query, come se si trattasse di una tabella.

È importante sottolineare nuovamente che in una vista *non è salvato nessun dato*. Le viste salvano soltanto l'istruzione di SELECT.

Le **viste materializzate** invece sono una tipologia di implementazione differente.

Altri utilizzi delle Subquery

Riportare *id*, *nome* e *cognome* dei clienti che hanno associata almeno una fattura con *importo* superiore a 50 euro. Risolviamo il problema senza *Subquery*:

```
SELECT DISTINCT C.IdCliente, C.Nome, C.Cognome
FROM   dbo.Fatture as F
INNER JOIN dbo.Clienti as C
   ON F.IdCliente = C.IdCliente
WHERE  F.Importo > 50;
```

Commento – parte 1

La presenza della JOIN nella clausola FROM ci porta ad avere nell'output una riga *per ogni fattura*.

Di conseguenza, se vogliamo restituire un elenco di clienti è fondamentale scrivere una DISTINCT dopo la SELECT, o equivalentemente usare una GROUP BY per utilizzarne l'effetto di rimozione dei duplicati.

Vediamo nella prossima slide come risolvere il problema utilizzando le *Subdbo*. Fino ad ora le abbiamo utilizzate soltanto all'interno della clausola FROM. Vediamo ora come utilizzarle per filtrare i dati all'interno della WHERE.

Commento – parte 2

Per l'esercizio in questione osserviamo che, se da un lato le informazioni necessarie sono presenti in due tabelle distinte (*Fatture* e *Clienti*), dall'altro lato le colonne da visualizzare in output appartengono tutte ad uno sola tabella (*Clienti*).

In questo caso possiamo evitare la JOIN (e dunque anche la DISTINCT) utilizzando le *subquery* nella WHERE in congiunzione con l'IN o con l'EXISTS.

Subquery per filtrare i dati con IN

Prima soluzione con IN

```
SELECT IdCliente, Nome, Cognome
FROM   dbo.Clienti
WHERE  IdCliente in (SELECT IdCliente
                     FROM   dbo.Fatture
                     WHERE  Importo > 50);
```

Subquery per filtrare i dati con EXISTS

Seconda soluzione con EXISTS

```
SELECT IdCliente, Nome, Cognome
FROM   dbo.Clienti AS C
WHERE  EXISTS (SELECT IdCliente
               FROM   dbo.Fatture AS F
               WHERE  C.IdCliente = F.IdCliente
                  AND F.Importo > 50);
```

Commento

In generale l'EXISTS è più generale dell'IN. Ad esempio il seguente problema non potrebbe essere risolto con l'IN:

"Estrarre tutti i clienti che hanno una fattura emessa nel mese del loro compleanno".

La risoluzione seguente è errata in quanto nessuno ci assicura che la fattura considerata nell'insieme dopo "IN" sia associata esattamente al cliente estratto.

```
SELECT IdCliente, Nome, Cognome
FROM   dbo.Clienti
WHERE  MONTH(DataNascita) IN (SELECT MONTH(DataFattura)
                              FROM  dbo.Fatture
                              WHERE Importo > 50);
```

Commento

Il risultato corretto si può ottenere invece con l'EXISTS

```
SELECT IdCliente, Nome, Cognome
FROM   dbo.Clienti AS C
WHERE  EXISTS (SELECT *
               FROM   dbo.Fatture AS F
               WHERE  C.IdCliente = F.IdCliente
                  AND MONTH(C.DataNascita) = MONTH(F.DataFattura) );
```

Confronto tra tabelle

Estrarre *id*, *nome* e *cognome* dei clienti senza fatture associate.

```sql
SELECT C.IdCliente, C.Nome, C.Cognome
FROM   dbo.Clienti as C
LEFT JOIN dbo.Fatture as F
   ON C.IdCliente = F.IdCliente
WHERE F.IdCliente IS NULL;

--oppure

SELECT IdCliente, Nome, Cognome
FROM   dbo.Clienti as C
WHERE  NOT EXISTS (SELECT *
                   FROM  dbo.Fatture as F
                   WHERE C.IdCliente = F.IdCliente);
```

Commento su NOT IN

Questa soluzione è ad alto tasso di errore!

```
SELECT IdCliente, Nome, Cognome
FROM   dbo.Clienti
WHERE  IdCliente NOT IN (SELECT IdCliente
                         FROM   dbo.Fatture);
```

L'alta probabilità di errore deriva dal fatto che la presenza di un *NULL* nell'output di una SubQuery inserita dopo una NOT IN porterà la query finale ad avere zero record. Per convincerci, proviamo a lanciare il codice seguente:

```
SELECT *
FROM   dbo.Clienti
WHERE  IdCliente NOT IN (1, NULL);
```

Inoltre un'ipotetica riga con *IdCliente NULL* non verrà comunque estratta a prescindere dal risultato della query dopo la *NOT IN*.

Window Function

Window function: esempi d'uso

Estrarre l'I*dFattura*, la *data*, l'*importo*, l'*IdCliente* e il *totale degli importi delle fatture associate al relativo cliente*. Primo metodo senza Window function:

```
SELECT F.IdFattura,
  F.DataFattura,
  F.Importo,
  F.IdCliente,
  C.TotaleImportoCliente
FROM Fatture as F
INNER JOIN (
     SELECT IdCliente,
        SUM(Importo) AS TotaleImportoCliente
     FROM   dbo.Fatture
     GROUP BY IdCliente) AS C
   ON F.IdCliente = C.IdCliente
      OR (F.IdCliente IS NULL AND C.IdCliente IS NULL);
```

Window function: esempi d'uso

Vediamo come riscrivere la query con le *Window Function*

```
SELECT IdFattura,
    DataFattura,
    Importo,
    IdCliente,
    SUM(Importo) OVER(PARTITION BY IdCliente) AS TotaleImportoCliente
FROM dbo.Fatture;
```

Commento

Ricordiamo che con le impostazioni di default i NULL sono considerati sempre:

- in prima posizione con un ordinamento crescente
- in ultima posizione con un ordinamento decrescente

Window function: esempi d'uso

Calcolare per ogni cliente la fattura con importo massimo.

```
WITH CTE AS
 (
  SELECT *,
     RANK() OVER(PARTITION BY IdCliente
                 ORDER BY Importo DESC) AS Ordinamento
  FROM dbo.Fatture
 )
SELECT *
FROM   CTE
WHERE  Ordinamento = 1;
```

Rank, Dense_Rank e Rownumber

L'output della query precedente cambia se sostituiamo RANK con DENSE_RANK o ROWNUMBER.

Con RANK() due righe con lo stesso ordine avranno lo stesso valore, mentre il valore delle righe successive NON sarà quello successivo, ma effettuerà "un salto" (ad esempio 1,1,3).

Con DENSE_RANK() due righe con lo stesso ordine avranno lo stesso valore, il valore delle righe successive sarà quello successivo (ad esempio 1,1,2).

Con ROW_NUMBER() due righe con lo stesso ordine avranno sempre valori diversi, calcolati non deterministicamente (ad esempio 1,2,3 in un'esecuzione e 2,1,3 in un'altra).

Window function: esempi d'uso

Calcolare per ogni cliente la fattura con data più recente. A parità di *DataFattura*, dare precedenza alla fattura con il valore nella colonna *IdFattura* più grande

```
WITH CTE AS
 (
   SELECT *,
     RANK () OVER(PARTITION BY IdCliente
                  ORDER BY  DataFattura DESC, IdFattura DESC) AS Ordinamento
   FROM dbo.Fatture
)
SELECT *
FROM   CTE
WHERE  Ordinamento = 1;
```

Window function: esempi d'uso

Riportare per ogni fattura il suo peso percentuale su tutte le fatture e sulle fatture in quello specifico anno.

```
SELECT IdFattura, IdCliente, Importo, YEAR(DataFattura) AS anno,
    Importo / SUM(Importo) OVER() AS PercentualeSuTotale,
    Importo / SUM(Importo) OVER(PARTITION BY YEAR(DataFattura)) AS
                                                    PercentualeSuAnno
FROM    dbo.Fatture;
```

Commento: Order By con OVER e SUM

Cosa succede se inserisco un ORDER BY nella clausola OVER dopo una SUM?

```
SELECT IdFattura,
   DataFattura,
   Importo,
   IdCliente,
   SUM(Importo) OVER(ORDER BY DataFattura) AS ImportoCumulato
FROM dbo.Fatture;
```

Ottengo le "somme progressive" (ordinando alla fine il risultato per DataFattura dovrebbe essere più chiaro).

Window function: esempi d'uso

Calcolare il cumulato mensile dell'importo al variare dell'anno

```
WITH ImportiMensili AS (
   SELECT YEAR(DataFattura) AS Anno,
     MONTH(DataFattura) AS Mese,
     SUM(Importo) AS Importo
   FROM dbo.Fatture
   GROUP BY YEAR(DataFattura),
       MONTH(DataFattura) )
SELECT Anno,
   Mese,
   Importo,
   SUM(Importo) OVER(PARTITION BY Anno
                     ORDER BY Mese) AS ImportoCumulato
FROM   ImportiMensili;
```

Window function: esempi d'uso

Con LAG e LEAD prendiamo i valori "precedenti" e "successivi" secondo quanto specificato nell'OVER e nell'ORDER BY.

```
SELECT IdFattura,
       IdCliente,
       DataFattura,
       Importo,
       LAG(Importo) OVER(PARTITION BY IdCliente
                         ORDER BY DataFattura) AS Lag,
       LEAD(Importo) OVER(PARTITION BY IdCliente
                          ORDER BY DataFattura) AS Lead
FROM   Fatture;
```

Casi d'uso della Case when

Casi d'uso della CASE WHEN

Visualizzare le colonne *IdFattura*, *Importo*, *Tipologia* e *Tipologia_descrizione* della tabella delle Fatture. Valorizzare la *Tipologia_descrizione* in questo modo:

- se *tipologia* = *'A'* allora *'Acquisto'*
- se *tipologia* = *'V'* allora *'Vendita'*

```sql
SELECT IdFattura,
    Importo,
    Tipologia,
    CASE WHEN Tipologia = 'A' THEN 'Acquisto'
         WHEN Tipologia = 'V' THEN 'Vendita'
         ELSE NULL
    END AS Tipologia_descrizione
FROM dbo.Fatture;
```

Casi d'uso della CASE WHEN

Classificare le fatture in tre categorie di prezzo:

- basso se l'importo è compreso tra 0 e 30 (0 incluso, 30 escluso)
- medio se l'importo è compreso tra 30 e 70 (30 incluso, 70 escluso)
- alto se l'importo è maggiore o uguale di 70.

Contare il numero di fatture per ogni categoria. ---------->

Casi d'uso della CASE WHEN

```sql
SELECT
    CASE WHEN Importo >= 0 AND Importo < 30 THEN 'Basso'
         WHEN Importo >= 30 AND Importo < 70 THEN 'Medio'
         WHEN Importo >= 70 THEN 'Alto'
         ELSE 'Non classificata'
    END AS Tipologia_cliente,
    COUNT(*) AS Numero
FROM dbo.Fatture
GROUP BY
    CASE WHEN Importo >= 0 AND Importo < 30 THEN 'Basso'
         WHEN Importo >= 30 AND Importo < 70 THEN 'Medio'
         WHEN Importo >= 70 THEN 'Alto'
         ELSE 'Non classificata'
    END;
```

Casi d'uso della CASE WHEN

Oppure

```
WITH CTE AS
(
  SELECT
     CASE WHEN Importo >= 0 AND Importo < 30 THEN 'Basso'
          WHEN Importo >= 30 AND Importo < 70 THEN 'Medio'
          WHEN Importo >= 70 THEN 'Alto'
          ELSE 'Non classificata'
     END AS Tipologia_cliente
  FROM dbo.Fatture
)
SELECT Tipologia_cliente,
   COUNT(*)
FROM CTE
GROUP BY Tipologia_cliente;
```

Casi d'uso della CASE WHEN

Ordinare i clienti secondo questo criterio:

- deve apparire all'inizio il cliente con IdCliente = 7;
- poi gli altri clienti nell'usuale ordine per Nome e Cognome.

```
SELECT   *
FROM     dbo.Clienti
ORDER BY CASE WHEN IdCliente = 7 THEN 1
              ELSE 2
         END ASC,
         Nome ASC,
         Cognome ASC;
```

Casi d'uso della CASE WHEN

Calcolare la somma degli importi delle fatture, considerando un ulteriore incremento del 20% per gli importi delle fatture di tipologia di A.

```
SELECT SUM(CASE WHEN Tipologia = 'A' THEN Importo * 1.20
                ELSE Importo
           END) AS Totale
FROM   dbo.Fatture;
```

Casi d'uso della CASE WHEN

Riportare per ogni Fornitore (Id e Denominazione):
- la somma degli importi delle fatture nell'anno 2017;
- la somma degli importi delle fatture nell'anno 2018.

```
SELECT Fo.IdFornitore,
    Fo.Denominazione,
    SUM(CASE WHEN YEAR(Fa.DataFattura) = 2017 THEN Importo
             ELSE NULL
        END) AS Importo_2017,
    SUM(CASE WHEN YEAR(Fa.DataFattura) = 2018 THEN Importo
             ELSE NULL
        END) AS Importo_2018
FROM    dbo.Fornitori AS Fo
LEFT JOIN dbo.Fatture AS Fa
    ON Fo.IdFornitore = Fa.IdFornitore
GROUP BY Fo.IdFornitore,
    Fo.Denominazione;
```

Piani d'esecuzione e indici

Creiamo una nuova tabella su un nuovo schema

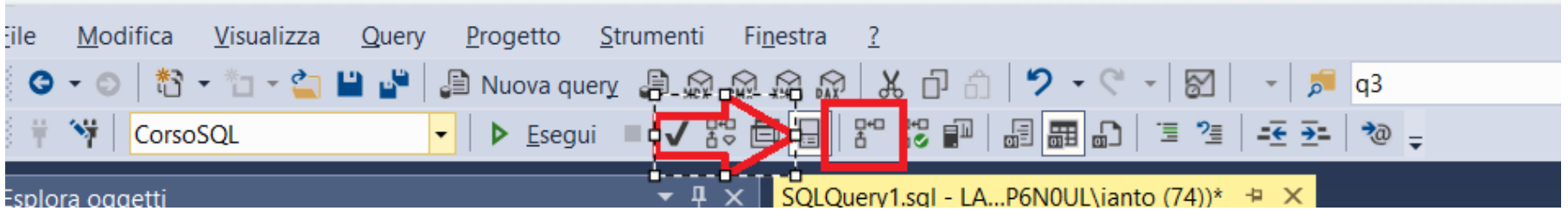
```
CREATE SCHEMA qep;

CREATE TABLE CorsoSQL.qep.Clienti (NumeroCliente INT   NOT NULL,
     Nome varchar(50)    NOT NULL,
     Cognome varchar(50) NOT NULL);

INSERT INTO CorsoSQL.qep.Clienti (NumeroCliente,Nome,Cognome)
VALUES ( 1,'Nicola','Iantomasi'),
   (2,'Giovanni','Rossi'),
   (3,'Alberto','Verdi');
```

Attiviamo la visualizzazione dei QEP

Notiamo che non esistono chiavi primarie sulla tabelle. Analizziamo allora i **piani di esecuzione** (*QEP* o *Query Execution Plan*) di queste tre dbo.

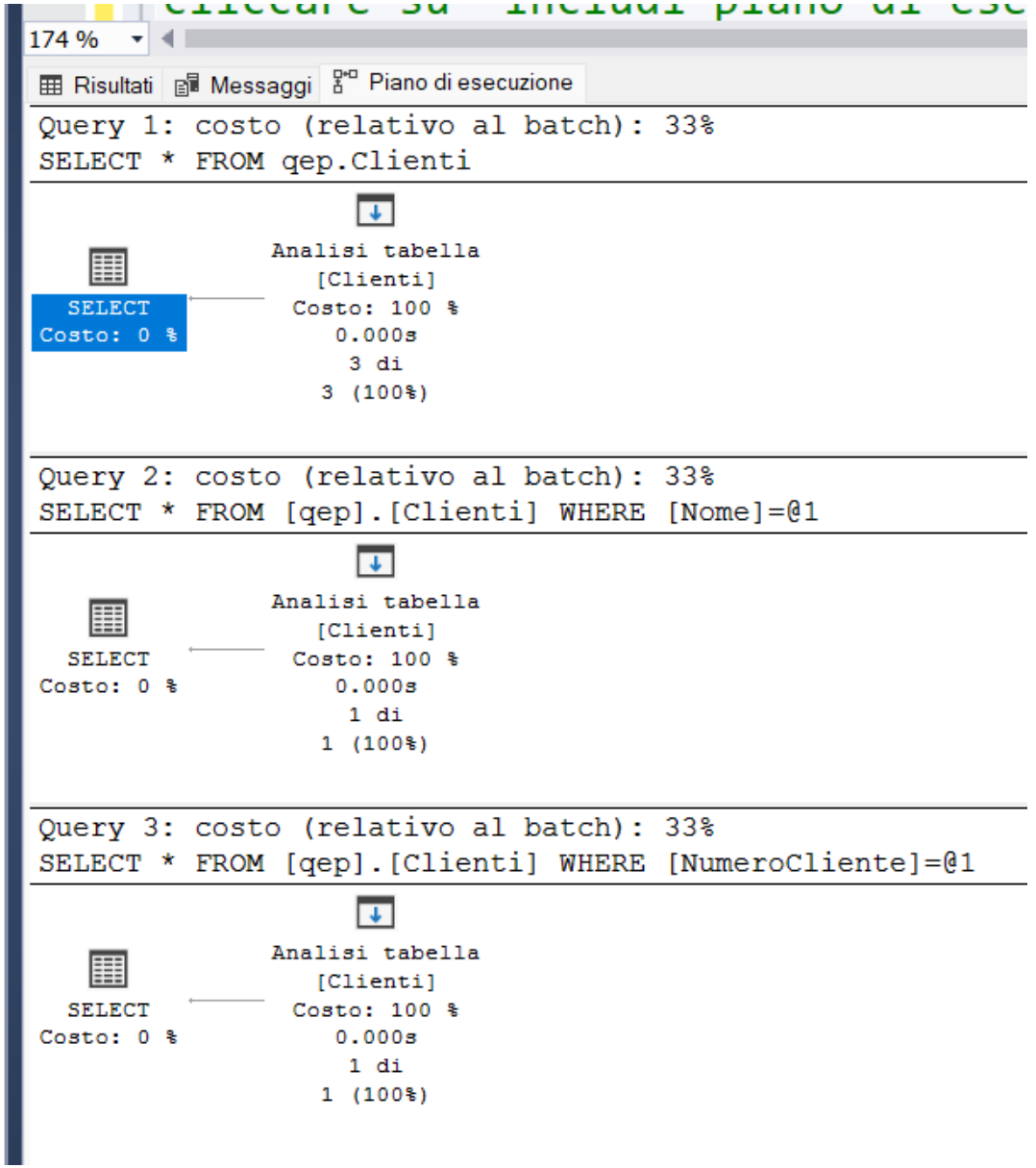
```
SELECT * FROM qep.Clienti;
SELECT * FROM qep.Clienti WHERE Nome = 'Nicola';
SELECT * FROM qep.Clienti WHERE NumeroCliente = 1;
```

Per attivare la visualizzazione del *piano di esecuzione effettivo* ci basta cliccare su "Includi piano di esecuzione effettivo" (oppure CTRL + M da tastiera).



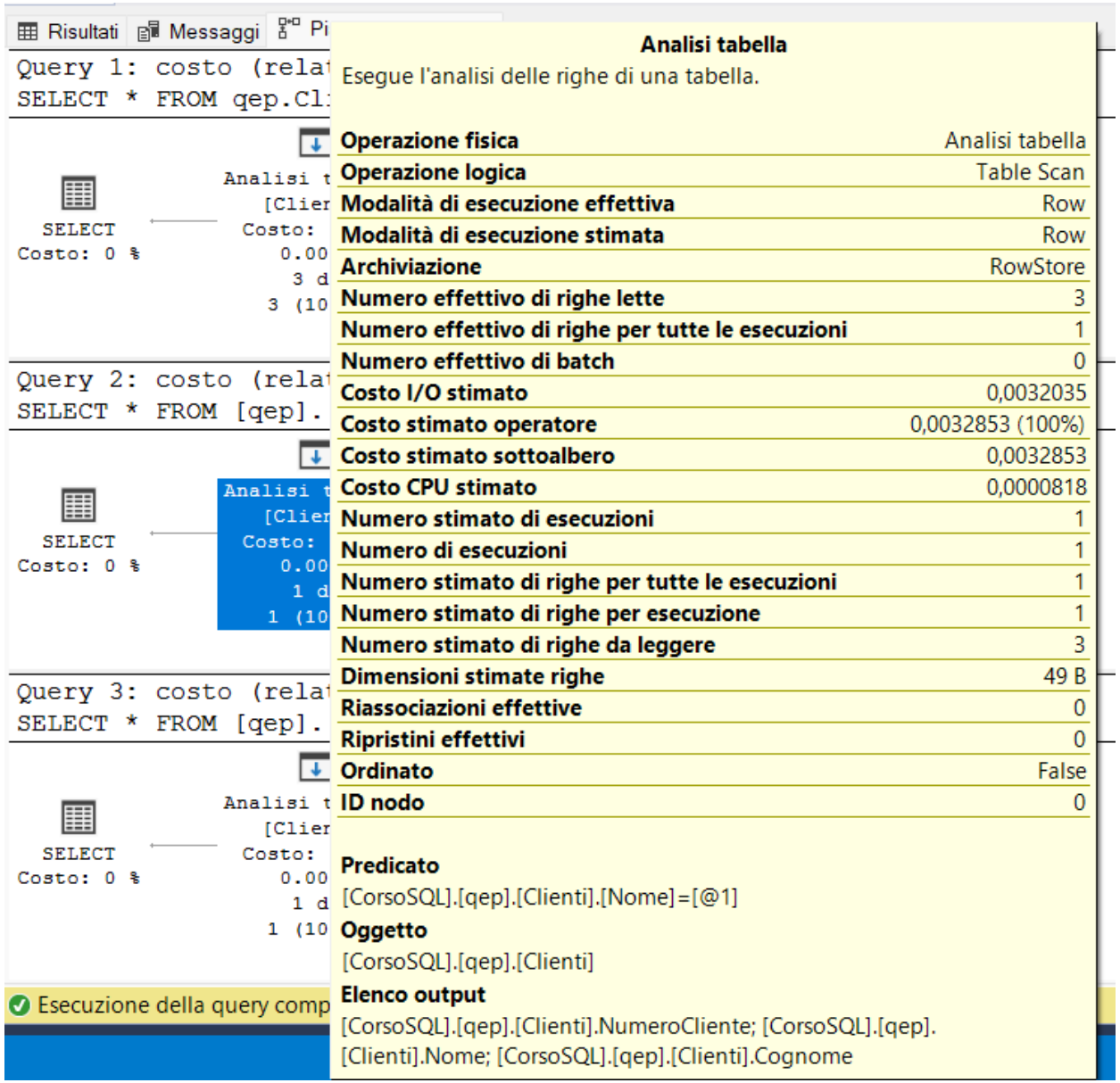
Analizziamo i piani d'esecuzione

I tre piani d'esecuzione (QEP) sono sostanzialmente identici.



Analizziamo i piani d'esecuzione

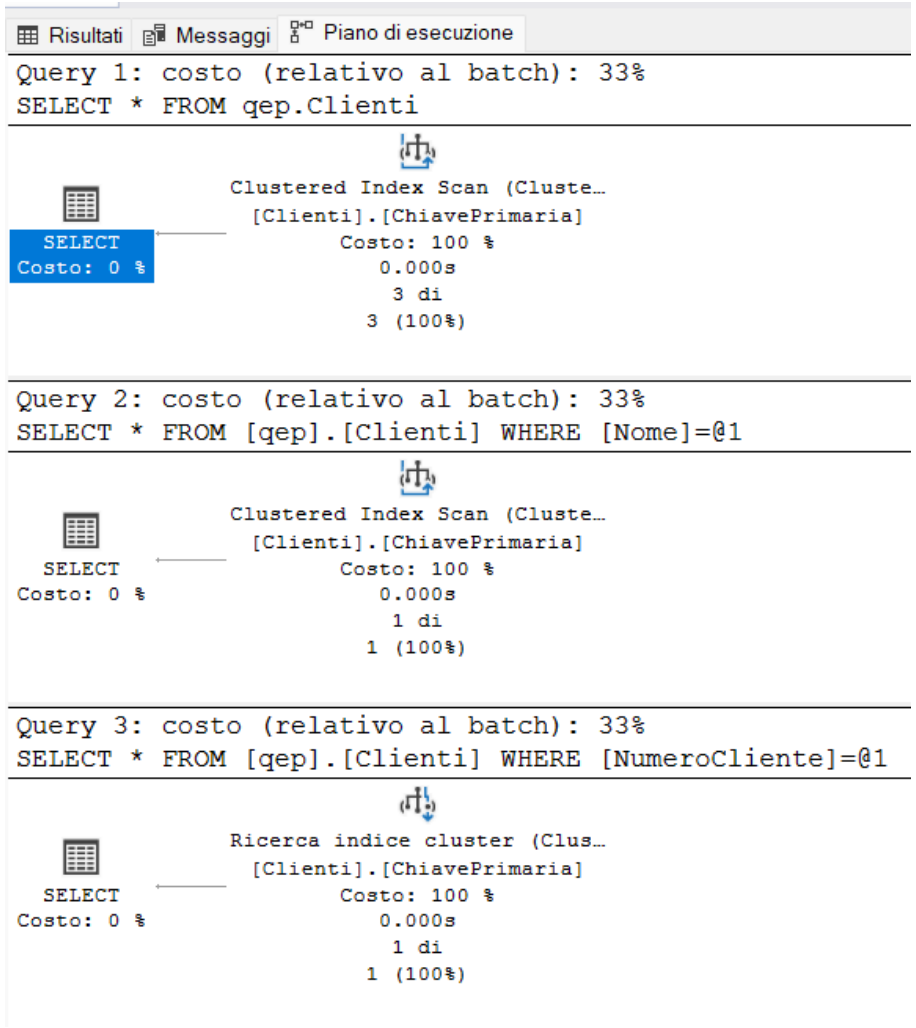
Nella seconda e terza query è presente un predicato nell'operatore di *Analisi Tabella* (*Table Scan*). Ma il motore d'esecuzione dovrà comunque eseguire una **scansione completa** della tabella.



Effetto delle chiavi primarie

Aggiungiamo una chiave primaria e rilanciamo le stesse tre query di prima.

```
ALTER TABLE    CorsoSQL.qep.Clienti
ADD CONSTRAINT ChiavePrimaria
PRIMARY KEY /*CLUSTERED*/ (NumeroCliente);

SELECT * FROM qep.Clienti ;
SELECT * FROM qep.Clienti WHERE Nome = 'Nicola';
SELECT * FROM qep.Clienti WHERE NumeroCliente = 1;
```



Analisi QEP dopo l'aggiunta della chiave primaria

I primi due piani d'esecuzione sono rimasti sostanzialmente identici a quelli precedenti. Viene effettuata comunque **un'intera scansione** dei dati (*Clustered Index Scan*).

```
SELECT * FROM qep.Clienti ;
```

```
SELECT * FROM qep.Clienti WHERE Nome = 'Nicola';
```

Invece l'algoritmo è cambiato per la terza query dove è eseguito un filtro sulla colonna che è anche chiave primaria: viene effettuata una **ricerca ottimizzata** (*Clustered index seek*).

```
SELECT * FROM qep.Clienti WHERE NumeroCliente = 1;
```

Spiegazione: cosa succede senza chiave primaria

Quando non sono state definite chiavi primarie, i dati sono organizzati fisicamente in una struttura a pila (***heap***) senza nessun tipo di ordinamento.

Di conseguenza, sia se dobbiamo selezionare tutte le righe, sia se dobbiamo fare dei filtri, l'unico algoritmo possibile è *scorrere interamente* la pila (ed eventualmente visualizzare solo quelle che rispettano il filtro).

Il piano di esecuzione sarà dunque lo stesso per tutte e tre le dbo.

Spiegazione: cosa succede con la chiave primaria

Dopo la creazione della chiave primaria, i dati saranno ri-organizzati in un ***clustered index*** (un albero ordinato per la colonna chiave *NumeroCliente*).

Attenzione: a questo punto i termini *"tabella"* e *"clustered index"* fanno riferimento alla stessa identica cosa.

Le ricerche che utilizzano la chiave primaria saranno eseguite in maniera ottimizzata, ***scorrendo efficientemente l'albero*** e visitando soltanto alcune porzioni di esso.

Al contrario, le ricerche che NON utilizzano la chiave primaria saranno eseguite tramite una ***scansione completa dell'albero***, al pari di una query senza nessun filtro.

ORDINAMENTO DELL'OUTPUT DI UNA QUERY

La presenza nel database di strutture fisiche in cui i dati sono salvati secondo un certo ordine non ha nessun impatto sulla seguente affermazione che **rimane sempre valida**:

L'output di una query che non contiene la clausola order by ha un ordinamento non prevedibile e non deterministico.

Tecnicamente esiste un modo per definire l'indice clustered su una colonna diversa dalla chiave primaria. Tuttavia si tratta di un'implementazione che risulta non efficiente nella grande maggior parte dei casi.

Aggiungiamo un indice non clustered

Aggiungiamo un indice *non clustered* sulla colonna *Nome*.

```
CREATE INDEX IX_clienti_nome ON CorsoSQL.qep.Clienti(Nome);
```

La definizione di un indice non clustered ha come risultato la creazione di una struttura d'appoggio **ordinata** (ad albero) che contiene:

- le colonne su cui l'indice è definito (in questo caso la colonna *Nome*);
- la *chiave primaria* (o in generale un riferimento alla struttura principale per recuperare le altre colonne).

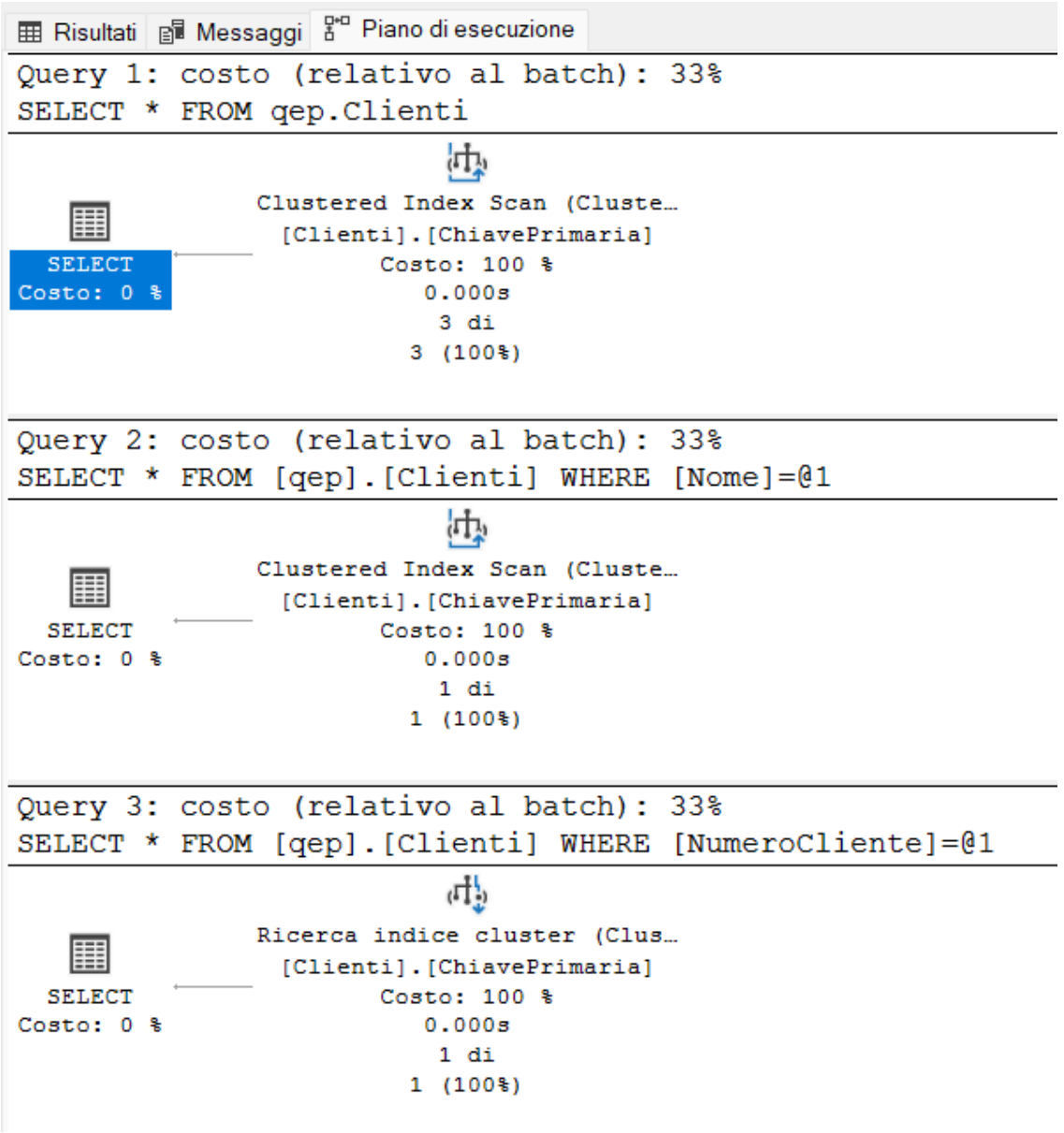
Rilanciamo ora le tre query precedenti

```
SELECT * FROM qep.Clienti;
SELECT * FROM qep.Clienti WHERE Nome = 'Nicola';
SELECT * FROM qep.Clienti WHERE NumeroCliente = 1;
```

I QEP non sono cambiati!

Come vediamo dall'immagine a destra i piani d'esecuzione **non utilizzano** il nuovo indice.

La tabella ha solo tre righe, il motore di esecuzione sceglie di non considerare *l'indice non clustered* in quanto dovrebbe comunque visitare anche l'intera tabella (*indice clustered*) per recuperare i dati della colonna *Cognome*.



Modifichiamo leggermente la query

Modifichiamo la clausola SELECT

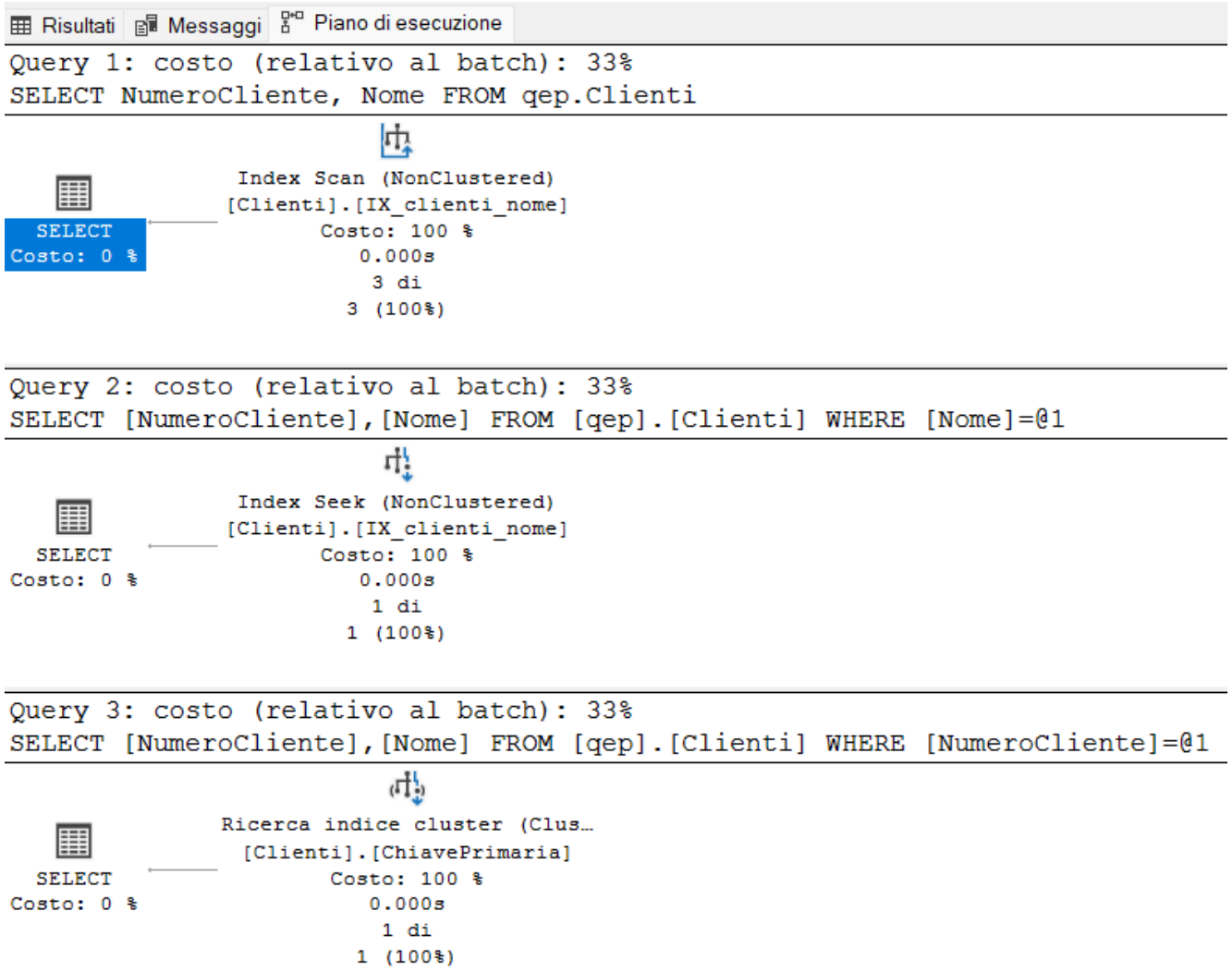
```
SELECT NumeroCliente, Nome
FROM qep.Clienti ;

SELECT NumeroCliente, Nome
FROM qep.Clienti
WHERE Nome = 'Nicola';

SELECT NumeroCliente, Nome
FROM qep.Clienti
WHERE NumeroCliente = 1;
```

Osserviamo che ora **tutte** le colonne richieste nella SELECT appartengono all'indice non clustered.

Questa volta i QEP sono cambiati!

La seconda query è risolta tramite una **ricerca ottimizzata** all'interno dell'indice non clustered!

Anche il QEP della prima query è cambiato: non accede più all'intera tabella (o *indice clustered*) poiché esiste una struttura più piccola che contiene tutte le informazioni.



Aggiungiamo un po' di righe

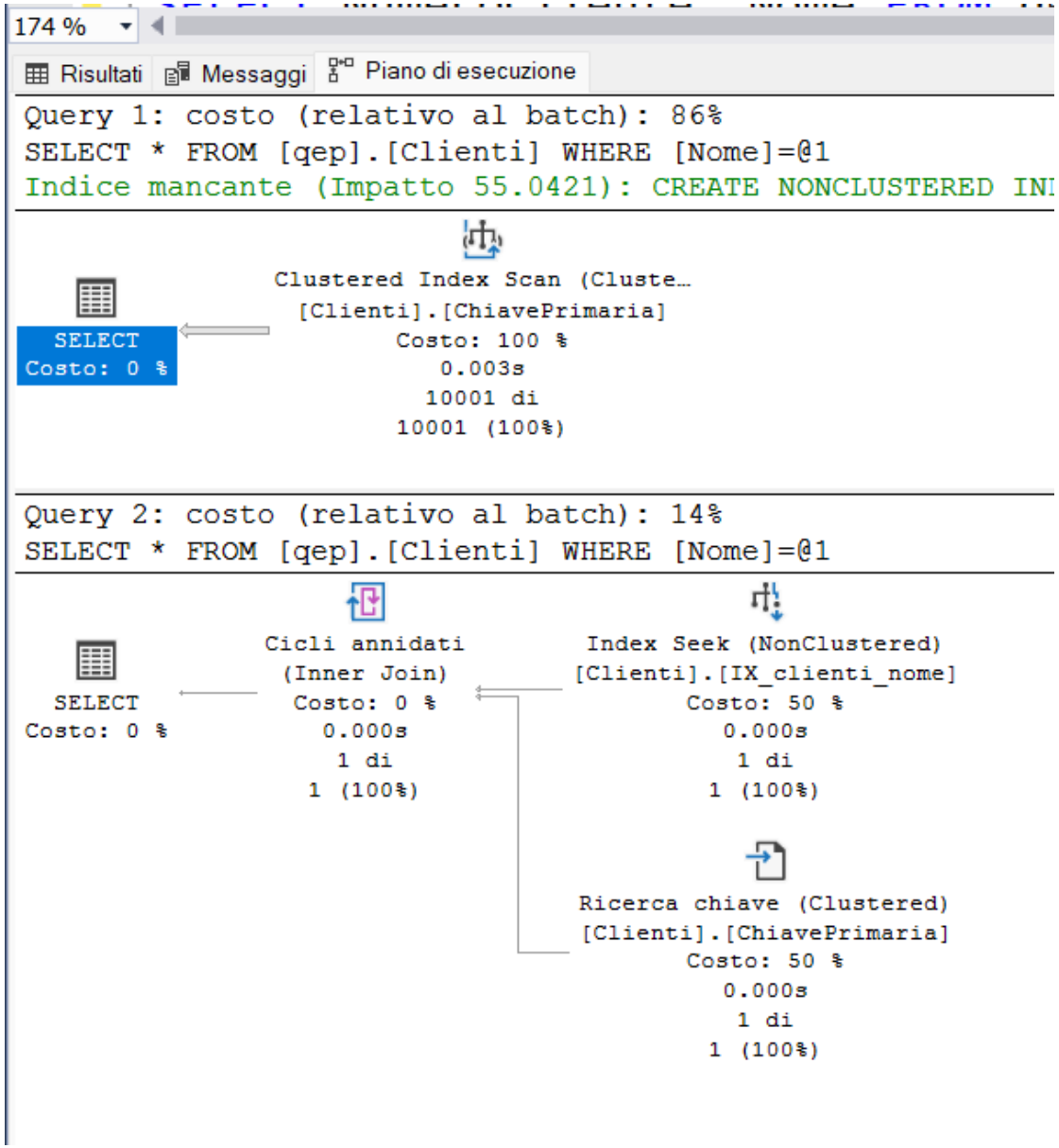
Aggiungiamo nuove righe, tutte con lo stesso valore nella colonna *Nome* **(attenzione: non replicate la strategia usata nella query per inserire le righe).**

```
INSERT INTO CorsoSQL.qep.Clienti (NumeroCliente, Nome, Cognome)
SELECT TOP 10000
       3 + ROW_NUMBER() OVER(ORDER BY(SELECTNULL)),
      'Nicola',
      'Rossi'
FROM   sys.objects a CROSS JOIN sys.objects b;
```

Analizziamo ora il piano d'esecuzione di queste due query:

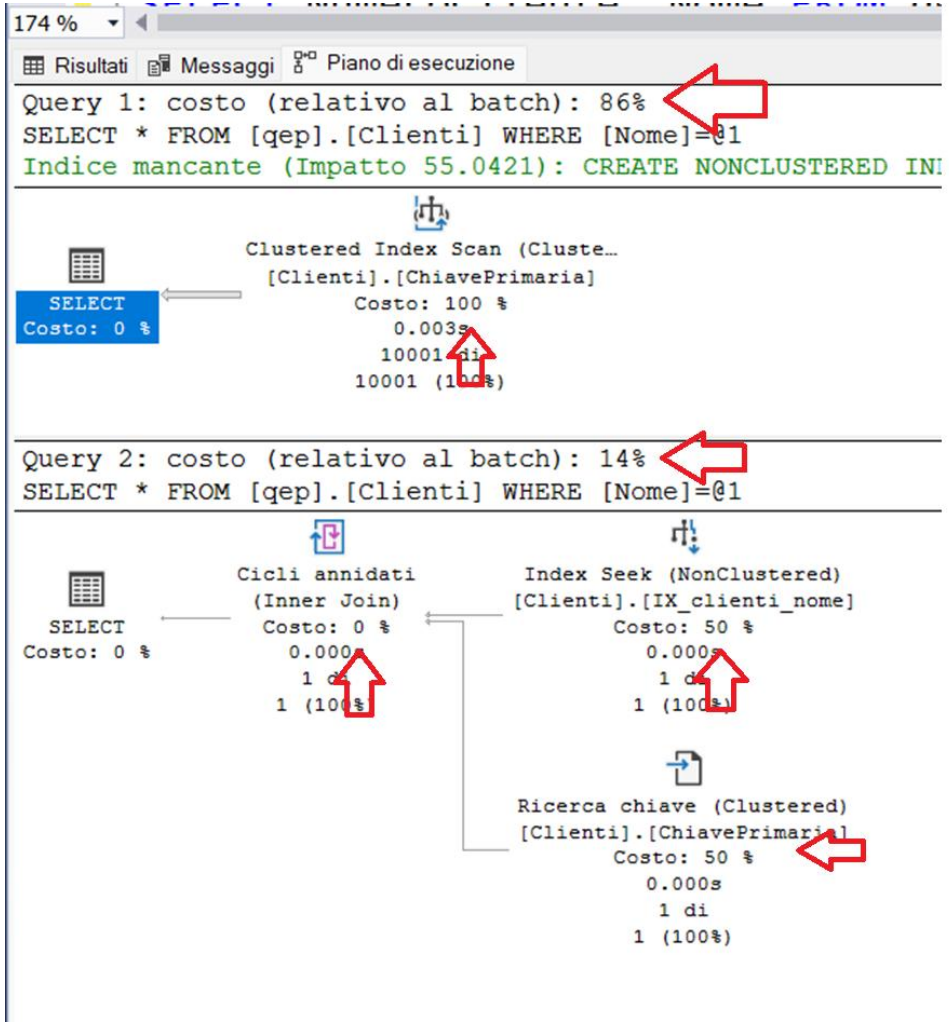
```
SELECT * FROM qep.Clienti WHERE Nome = 'Nicola' ;
SELECT * FROM qep.Clienti WHERE Nome = 'Giovanni';
```

I QEP sono differenti!



ATTENZIONE

Tutte le percentuali presenti nel piano d'esecuzione sono semplicemente delle stime precedenti all'esecuzione effettiva della dbo. Non sono relative all'effettivo tempo di esecuzione.



Spiegazione – parte 1

SQL Server ha due possibili alternative per risolvere le query precedenti:

- eseguire direttamente una semplice scansione dell'*indice clustered* (cioè la tabella).
- fare una ricerca utilizzando l'indice *non clustered* e poi recuperare le altre colonne nell'*indice clustered* (eseguendo a tutti gli effetti una *Join* per combinare le due fonti).

In questo caso, in base al valore presente nel filtro, SQL Server propende una volta per la prima opzione e una volta per la seconda.

Spiegazione – parte 2

Intuitivamente le scelte sembrano comprensibili:

- Il valore *Nicola* è presente tantissime volte nella tabella, non avrebbe senso dover recuperare ogni volta il valore del *Cognome* cercando i rispettivi valori della chiave primaria. Converrà fare direttamente una scansione della tabella.
- Il valore *Giovanni* invece, è presente solo pochissime volte. Dunque varrà la pena fare "il doppio giro".

È lecito porsi ora questa domanda:

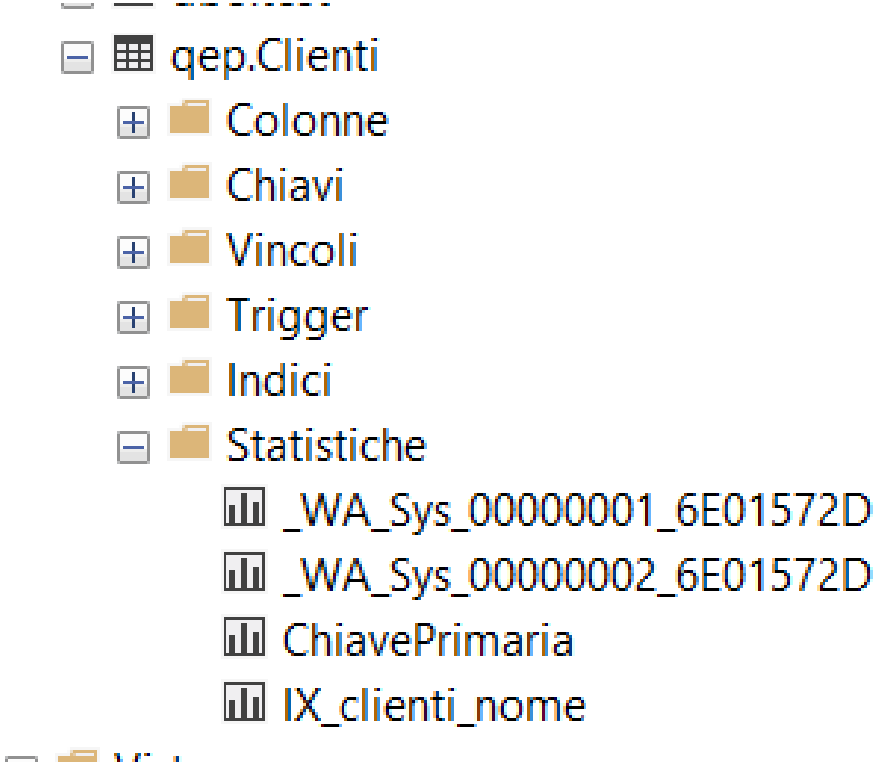
ma come fa SQL Server a prendere questa decisione prima di eseguire la query?

Le statistiche

SQL Server tiene traccia all'interno delle **statistiche** del contenuto delle tabelle (indici e colonne).

Tali statistiche sono aggiornate da SQL Server con una certa regolarità (secondo algoritmi che dipendono dalla versione).

In generale non sarà tenuta traccia di tutti i valori presenti in una colonna, ma solo della distribuzione generale.



Un esempio di statistica di SQL Server

Nel nostro esempio le statistiche sono molto precise poiché sono state apportate poche aggiunte alla tabella e ci sono solo tre valori distinti. Con il tempo potrebbero non essere più così affidabili.

Script ?

Nome tabella: qep.Clienti

Nome statistiche: _WA_Sys_00000002_6E01572D

Statistiche per l'indice '_WA_Sys_00000002_6E01572D'.

Nome	Updated	Rows	Rows Sampled	Steps
_WA_Sys_00000002_6E01572D	mag 10 2024 8:14PM	10003	10003	3

All Density	Average Length	Columns
0.3333333	6.0003	Nome

Histogram Steps

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
Alberto	0	1	0	1
Giovanni	0	1	0	1
Nicola	0	10001	0	1

Ordinamento e rimozione duplicati: chiave primaria

Consideriamo queste quattro query che coinvolgono la colonna *NumeroCliente* che è chiave primaria della tabella *Clienti*

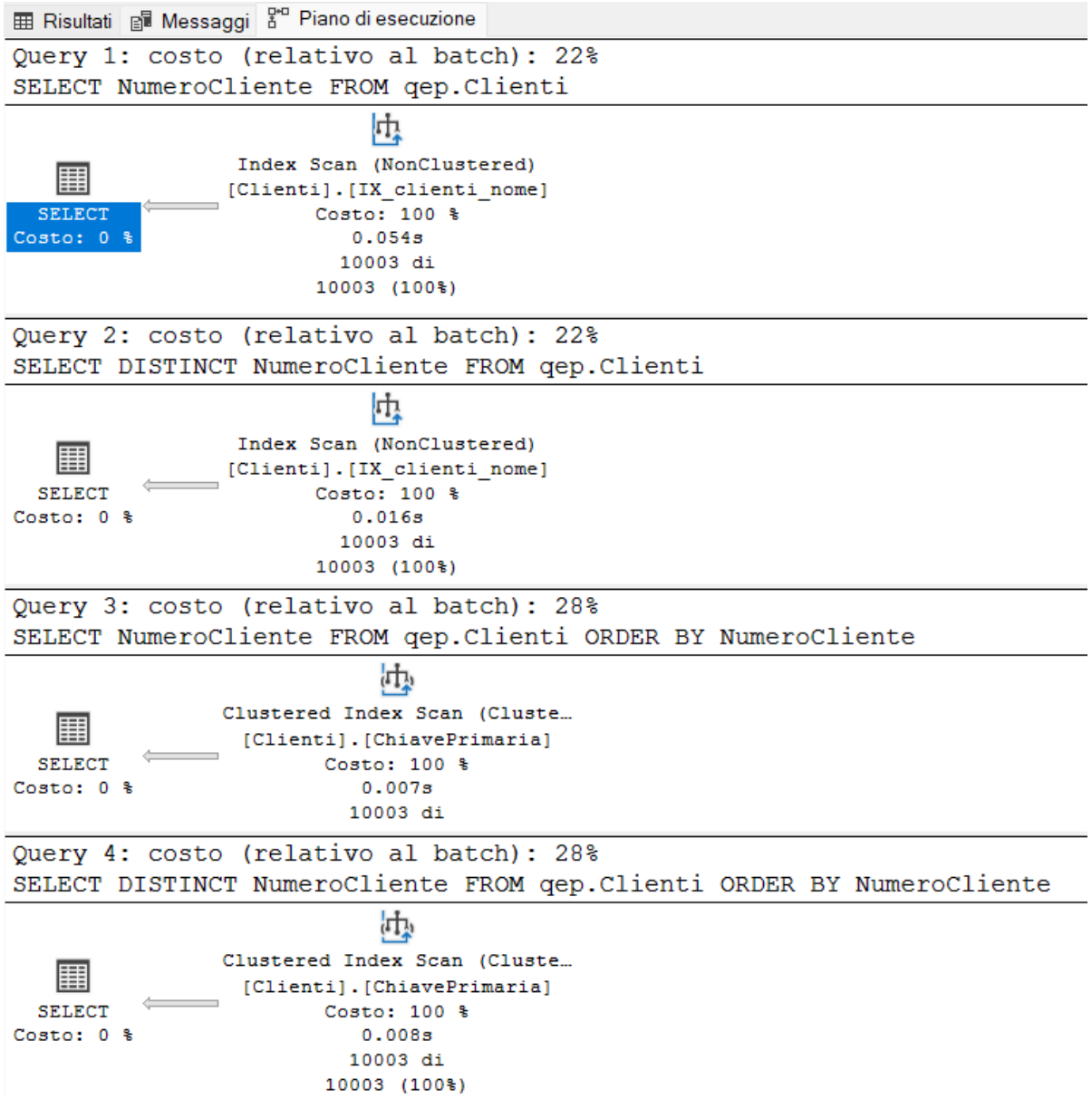
```
SELECT NumeroCliente FROM qep.Clienti;
SELECT DISTINCT NumeroCliente FROM qep.Clienti;
SELECT NumeroCliente FROM qep.Clienti ORDER BY NumeroCliente;
SELECT DISTINCT NumeroCliente FROM qep.Clienti ORDER BY NumeroCliente;
```

Piani d'esecuzione con chiave primaria

Tutte e quattro le query eseguono semplicemente delle scansioni.

La richiesta di rimozione dei duplicatI viene **ignorata**.

Per ottenere i dati ordinati viene semplicemente effettuata la scansione del *Clustered Index* (cioè dell'intera tabella).



Ordinamento e rimozione duplicati: colonna con indice non clustered

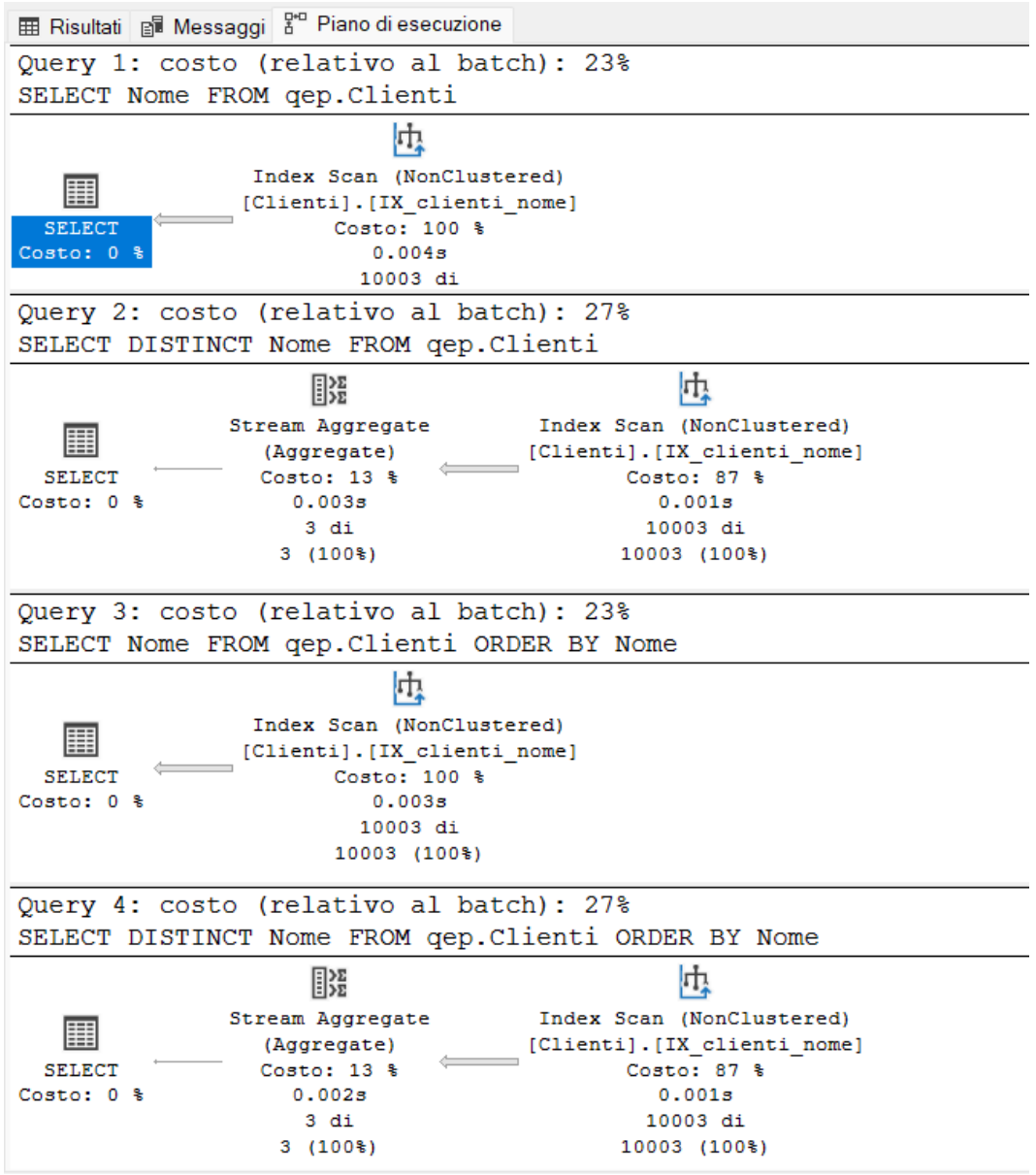
Consideriamo queste quattro query che coinvolgono la colonna *Nome* su cui è costruito un indice non clustered

```
SELECT NumeroCliente FROM qep.Clienti;
SELECT DISTINCT NumeroCliente FROM qep.Clienti;
SELECT NumeroCliente FROM qep.Clienti ORDER BY NumeroCliente;
SELECT DISTINCT NumeroCliente FROM qep.Clienti ORDER BY NumeroCliente;
```

Piani d'esecuzione con colonna con indice non clustered

L'ordinamento è gestito semplicemente tramite una *scansione dell'indice*.

La rimozione dei duplicati è gestita tramite l'operatore ***Stream Aggregate***. Esso è molto efficiente: essendo i valori già ordinati nell'indice, per rimuovere i duplicati basterà confrontare ogni dato con il successivo.



Ordinamento e rimozione duplicati: colonna non indicizzata

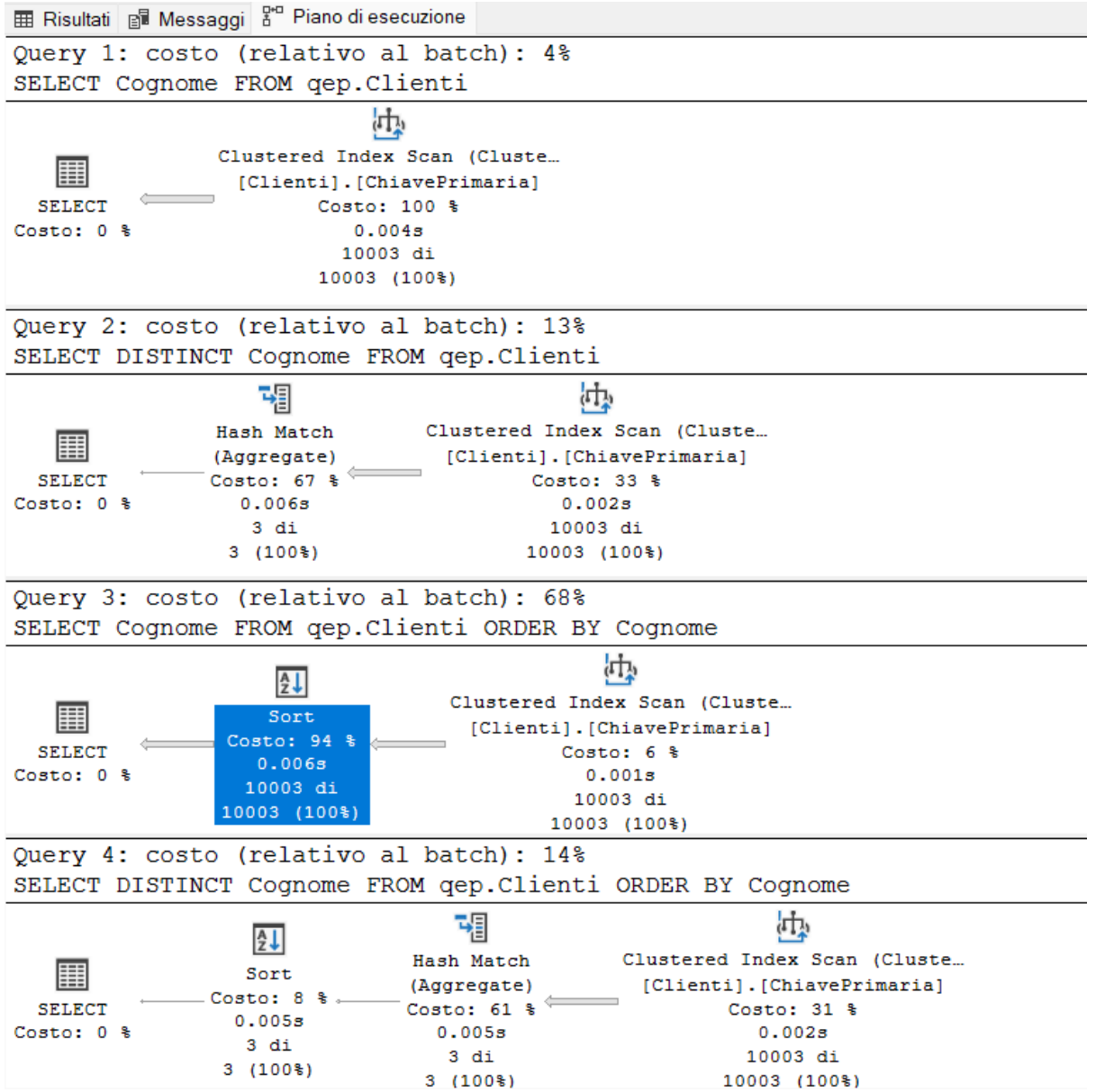
Consideriamo queste quattro query che coinvolgono la colonna *Cognome* che non è indicizzata

```
SELECT Cognome FROM qep.Clienti;
SELECT DISTINCT Cognome FROM qep.Clienti;
SELECT Cognome FROM qep.Clienti ORDER BY Cognome;
SELECT DISTINCT Cognome FROM qep.Clienti ORDER BY Cognome;
```

Piani d'esecuzione con colonna non indicizzata

L'ordinamento è gestito tramite l'operatore ***Sort*** che è uno dei più costosi in termini di tempo.

La rimozione dei duplicati è gestita tramite l'operatore ***Hash Match*****.** Anche questo operatore è abbastanza oneroso: non essendoci una struttura ordinata, per rimuovere i duplicate NON basterà confrontare ogni dato con il solo successivo.



Operatore Group By

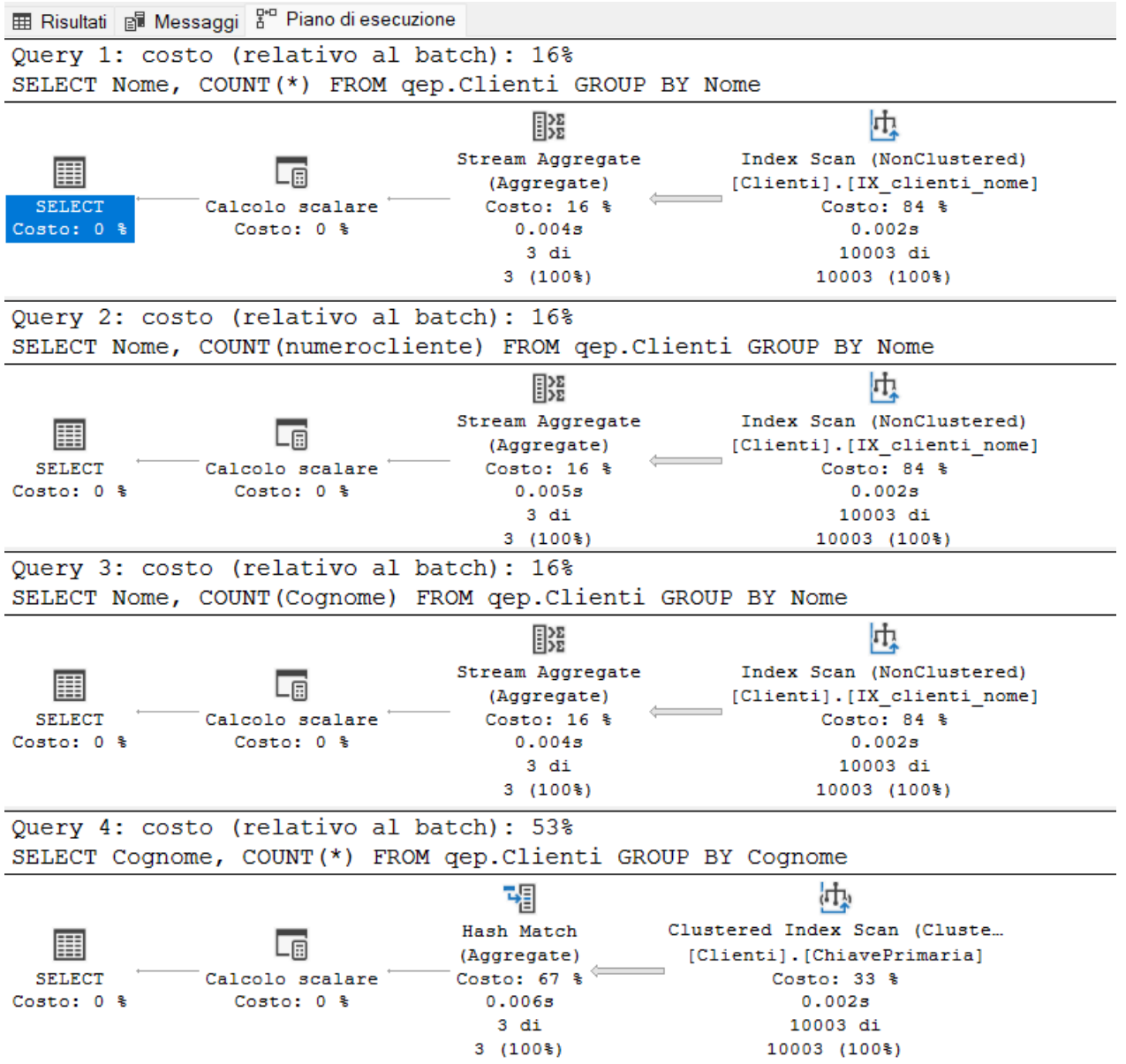
Il comportamento con la *Group By* è molto simile alla *Distinct*. Attenzione però a cosa inseriamo nella *Select* perché potremmo perdere i vantaggi dati dalla presenza di un indice.

```
SELECT Nome, COUNT(*) FROM qep.Clienti GROUP BY Nome;
SELECT Nome, COUNT(numerocliente) FROM qep.Clienti GROUP BY Nome;
SELECT Nome, COUNT(Cognome) FROM qep.Clienti GROUP BY Nome;
SELECT Cognome, COUNT(*) FROM qep.Clienti GROUP BY Cognome;
```

Piani d'esecuzione con group by

Osserviamo che nella terza query non c'è un riferimento alla colonna *Cognome* perché sulla tabella è presente il vincolo NOT NULL associato a tale colonna.

Rimuovendo questo vincolo, il piano d'esecuzione cambierebbe (anche senza modificare i dati contenuti nella tabella).

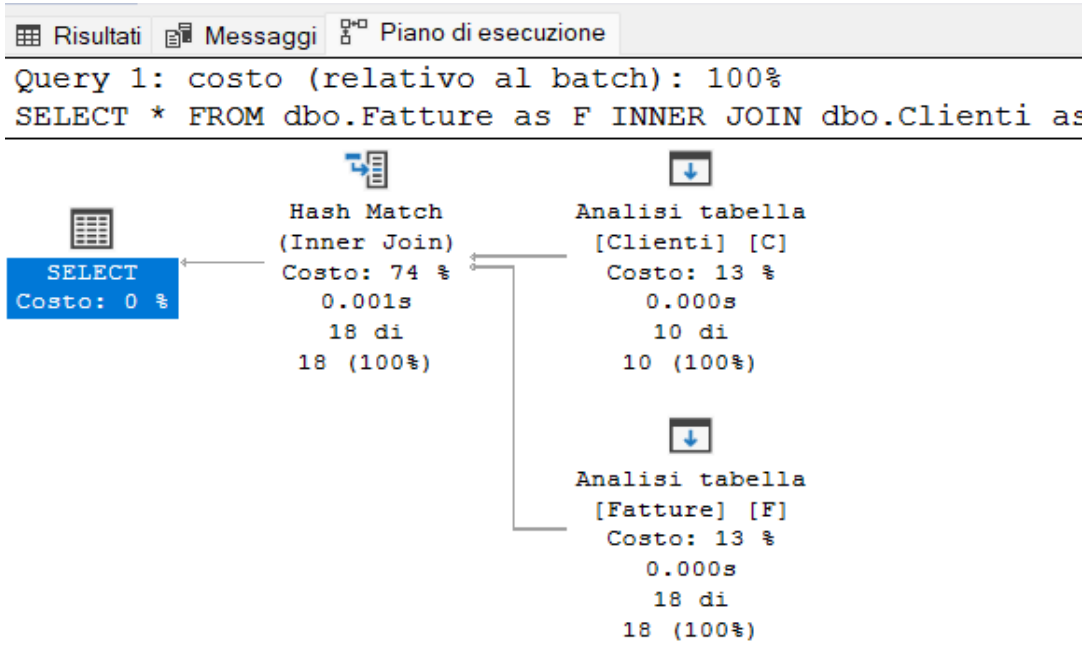


Operatore Join – parte 1

Interroghiamo ora le tabelle dello schema dbo;

Se le colonne su cui è costruita la condizione di Join non sono entrambe ordinate, tendenzialmente nel piano d'esecuzione sarà presente una ***hash join*** (algoritmo abbastanza costoso).

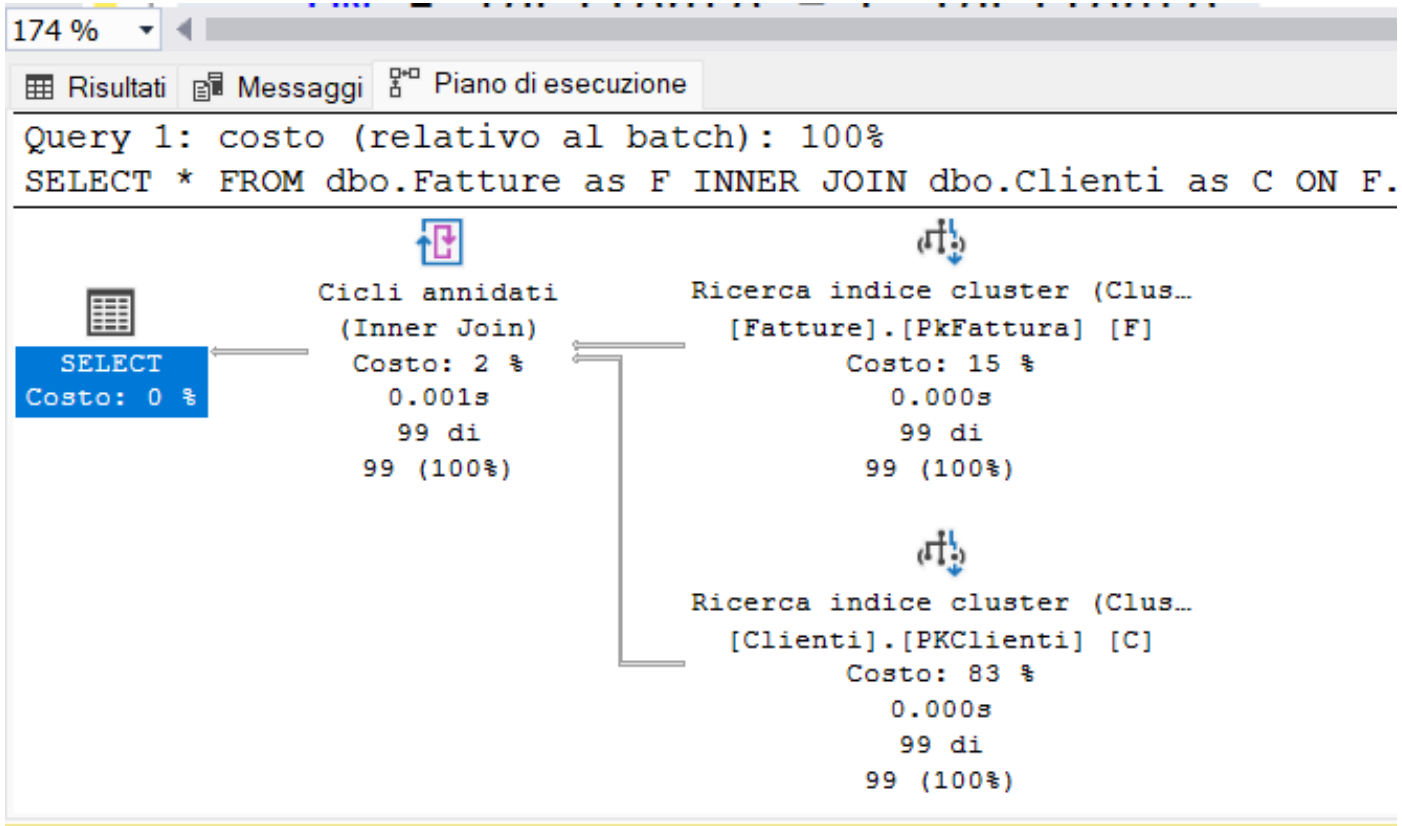
```
SELECT *
FROM   dbo.Fatture as F
INNER JOIN dbo.Clienti as C
  ON F.IdCliente = C.IdCliente;
```



Operatore Join – parte 2

Diminuendo il numero di righe di una o entrambe le tabelle potrebbe essere eseguita una ***nested join***. Si tratta di un vero e proprio *loop di ricerche*. Il costo computazionale può essere molto variabile perché dipende dall'efficienza dell'algoritmo di ricerca e dal numero di ricerche stesse.

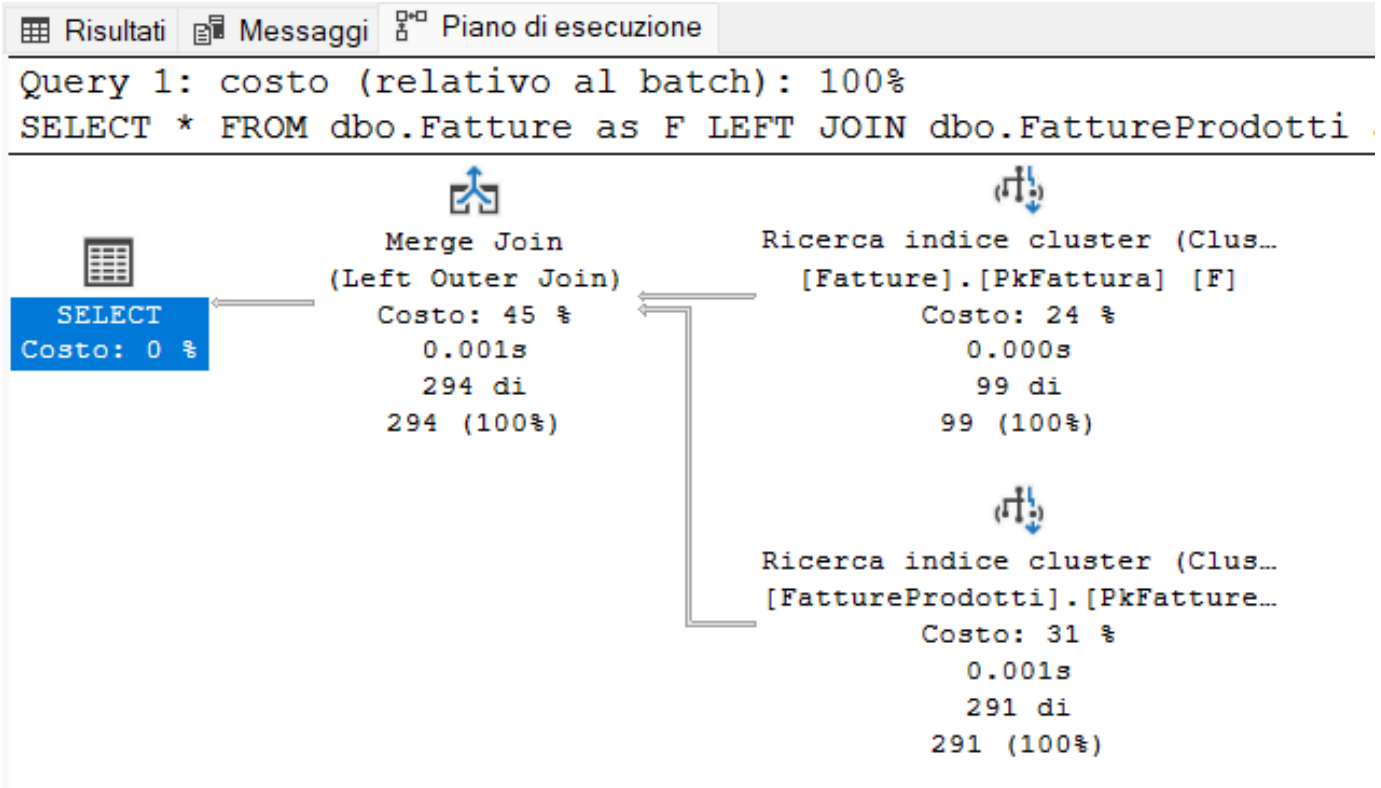
```
SELECT *
FROM   dbo.Fatture as F
INNER JOIN dbo.Clienti as C
   ON F.IdCliente = C.IdCliente
WHERE F.IdFattura < 100;
```



Operatore Join – parte 3

Se le colonne su cui è costruita la condizione di Join sono entrambe ordinate, tendenzialmente vedremo nel piano d'esecuzione una ***merge join.*** Si tratta di un algoritmo molto efficiente che sfrutta il fatto che le colonne da confrontare sono entrambe ordinate.

```
SELECT *
FROM   dbo.Fatture as F
LEFT JOIN dbo.FattureProdotti as C
   ON F.IdFattura = C.IdFattura
WHERE F.IdFattura < 100;
```



Operatore Join – parte 4

Proviamo a creare un indice sulla colonna *IdCliente* della tabella *Fatture*.

```
CREATE INDEX IX_IdCliente ON CorsoSQL.dbo.Fatture(IdCliente);
```

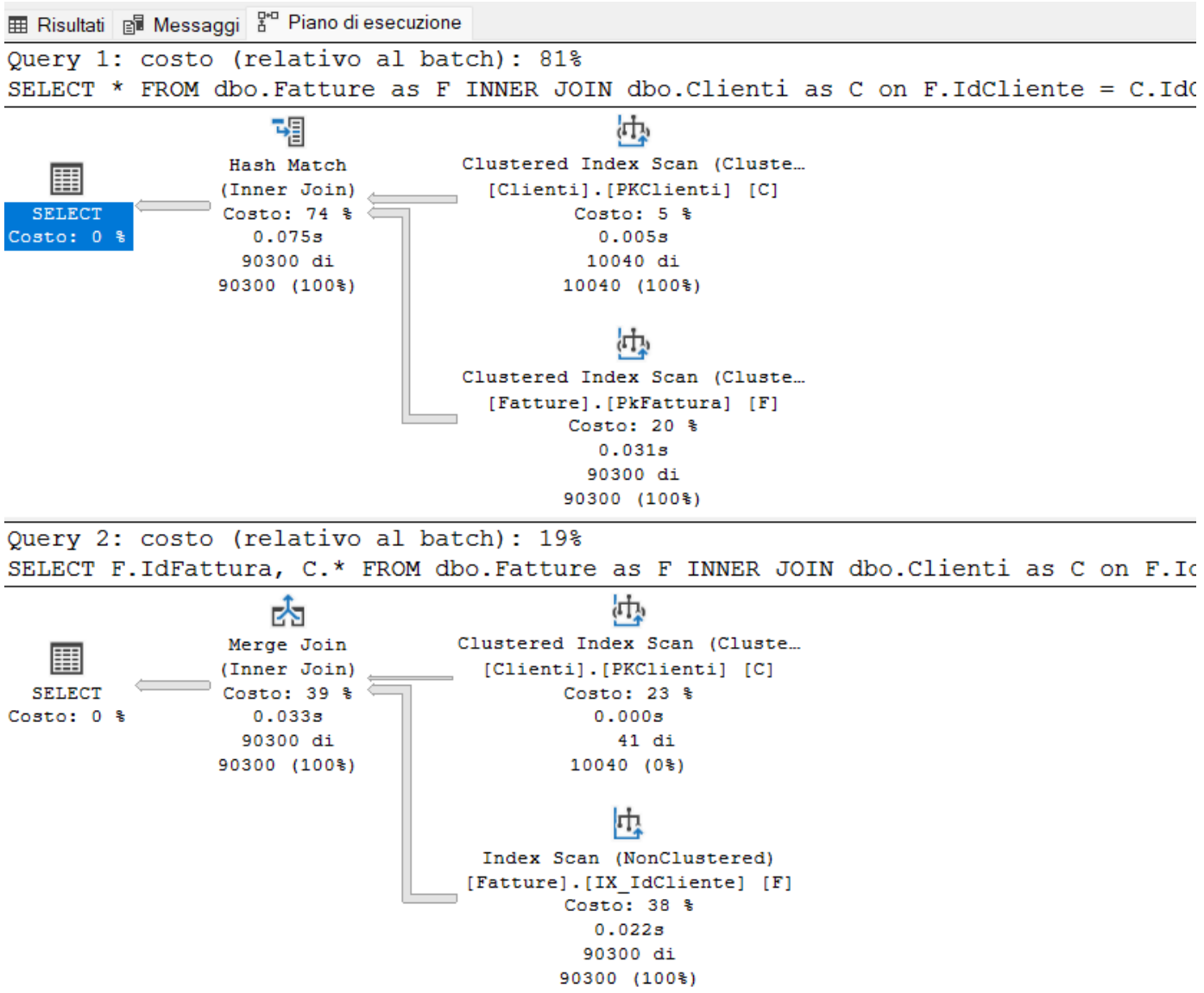
Nella prossima slide vedremo che il QEP di questa query non utilizza tale indice:

```
SELECT *
FROM   dbo.Fatture as F
INNER JOIN dbo.Clienti as C
   ON F.IdCliente = C.IdCliente;
```

L'indice sarà invece utilizzato se modifichiamo la *SELECT* per analizzare dalla tabella *Fatture* le sole colonne presente nell'indice.

```
SELECT F.IdFattura, C.*
FROM   dbo.Fatture as F
INNER JOIN dbo.Clienti as C
   ON F.IdCliente = C.IdCliente;
```

Modificando la Select passiamo da una Hash Join a una Merge Join

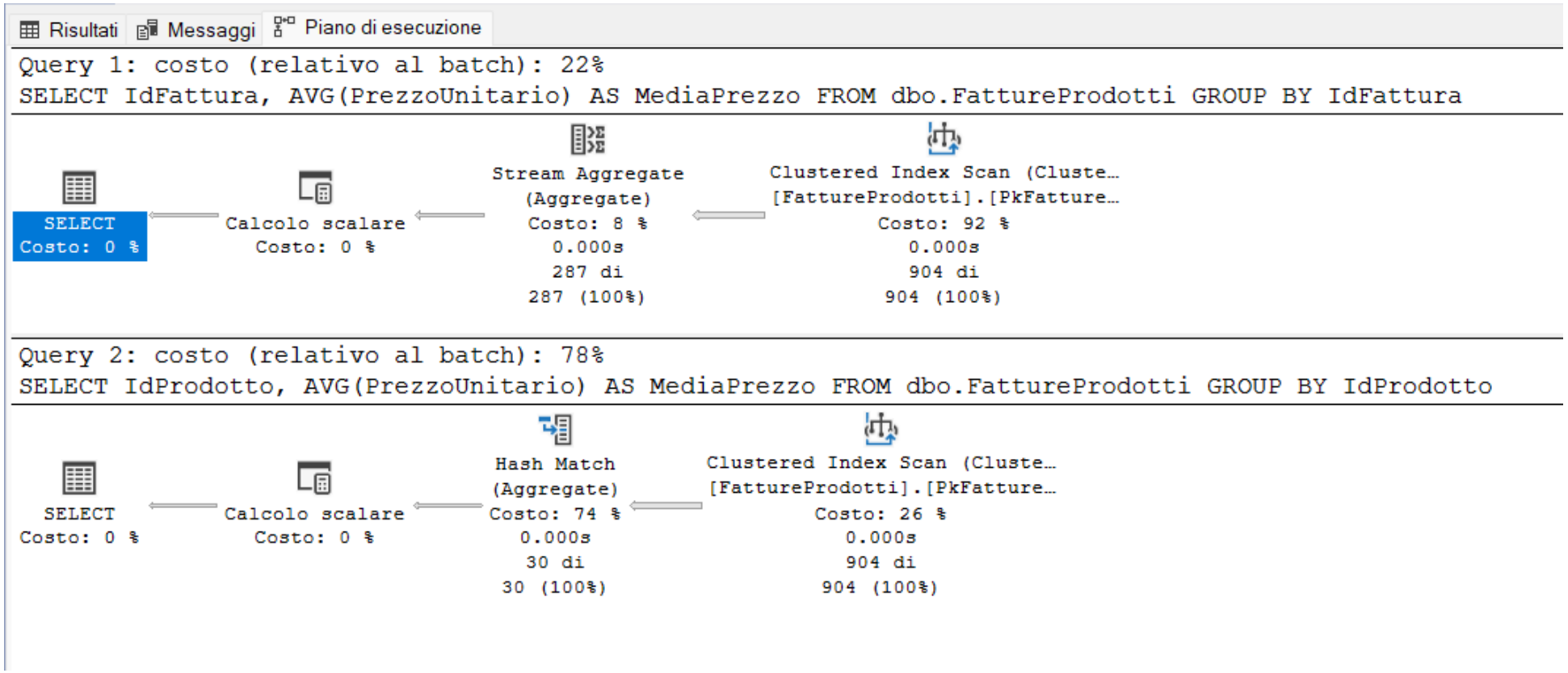


Ordine in un'indice clustered

Confrontiamo queste due query che coinvolgono la tabella *FattureProdotti*. La tabella ha una chiave primaria su questa coppia di colonne *(IdFattura, IdProdotto)*.

```
SELECT    IdFattura,
    AVG(PrezzoUnitario)  AS MediaPrezzo
FROM      dbo.FattureProdotti
GROUP BY IdFattura;

SELECT    IdProdotto,
    AVG(PrezzoUnitario)  AS MediaPrezzo
FROM      dbo.FattureProdotti
GROUP BY IdProdotto;
```

L'ordine delle colonne nella chiave primaria è importante!



Commento

Definendo l'indice clustered come la coppia ordinata *(IdFattura, IdProdotto)* i dati sono ordinati primariamente per *IdFattura*. Solo a parità di *IdFattura*, verrà considerato l'*IdProdotto* nell'ordinamento.

Di conseguenza la rimozione dei duplicati della colonna *IdFattura* è fatta con l'algoritmo efficiente ***Stream Aggregate*** che sfrutta l'ordinamento sulla colonna.

Al contrario, la rimozione dei duplicati della colonna *IdProdotto* è fatta con l'algoritmo più costoso ***Hash Match*** che NON può sfruttare l'ordinamento sulla colonna.

Ordine in un'indice clustered

Considerazioni analoghe si possono fare coinvolgendo la tabella *FattureProdotti* in una JOIN.

```
SELECT   *
FROM     dbo.FattureProdotti AS Fp
INNER JOIN dbo.Fatture AS F
   ON Fp.IdFattura = F.IdFattura ;

SELECT   *
FROM     dbo.FattureProdotti AS Fp
INNER JOIN dbo.Prodotti AS P
   ON Fp.IdProdotto = p.IdProdotto;
```

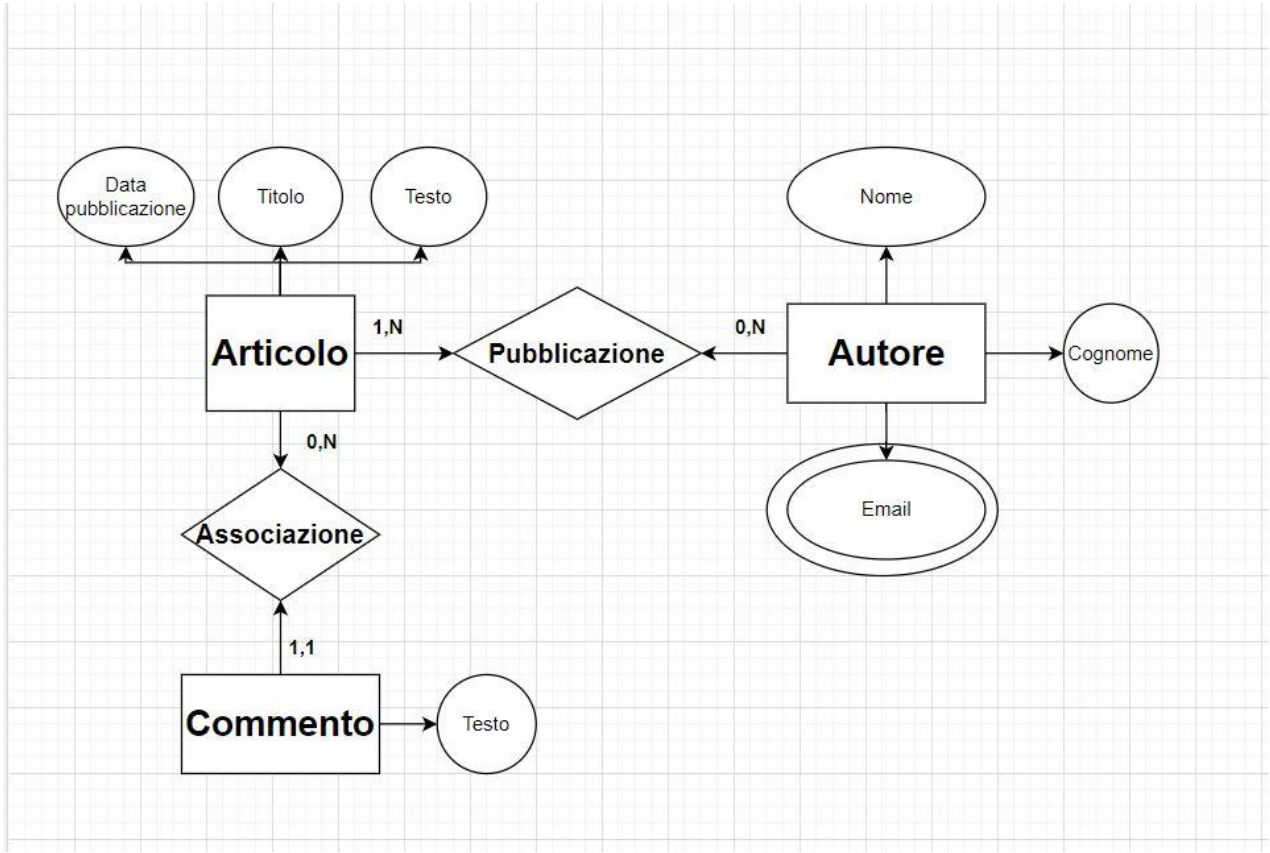
Introduzione alla progettazione di un Database

Progettazione concettuale dei dati

Il primo step per progettare un Database è capire ed elencare **quali dati** devono essere recuperati per rappresentare la determinata area della realtà che il DB deve descrivere.

In molti casi ciò non può essere fatto con un semplice documento di Microsoft Word, ma occorre utilizzare una struttura e una simbologia ben precisa, nota come **diagramma E-R** (diagramma entità-relazione).

Un esempio di diagramma E-R



Caratteristiche del diagramma E-R

Il diagramma E-R (entità relazioni) segue una notazione ben precisa:

- le **entità** del database sono rappresentate all'interno di **rettangoli**;
- ogni entità è definita da una serie di **attributi** rappresentati all'interno di **ellissi** collegate alle entità;
- tra le entità sussistono delle **relazioni**, rappresentate con dei **rombi** collegati alle due (o più) entità coinvolte;
- meno di frequente, anche le relazioni possono avere degli attributi associate.

Tipologie di attributi

Oltre ai classici attributi **semplici**, possono essere presenti:

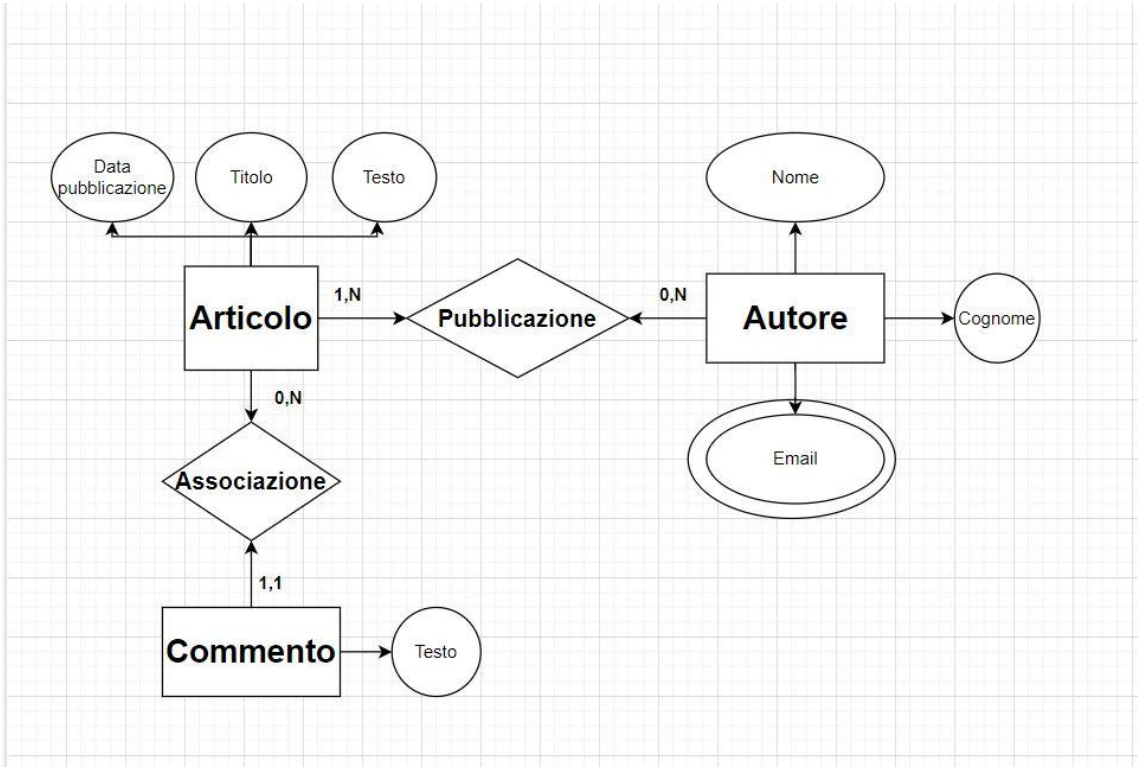
- attributi **composti** da più attributi semplici. Ad esempio *l'indirizzo* è un attributo composto da *via*, *numero civico*, *città* e *CAP*;

- attributi **calcolati** il cui valore dipende da un altro attributo. Ad esempio l'attributo *età* dipende dall'attributo data nascita

- attributi **multi-valore** che possono prevedere più di un valore per una singola istanza di un'entità. Ad esempio uno stesso cliente può avere più di una *mail* o più di un *numero di telefono*.

Tipologie di relazioni

Su ogni freccia tra le entità e le relazioni è presente una coppia di valori (le **cardinalità**): il primo può essere 1 (uno) oppure 0 (zero), mentre il secondo può essere 1 o N. Esploriamone il significato tramite degli esempi:



Tipologie di relazioni

- il primo 1 nella freccia che collegata l'*Articolo* alla *Pubblicazione* indica che ogni articolo deve obbligatoriamente avere un autore;

- il primo 0 (zero) nella freccia che collega l'*Autore* alla *Pubblicazione* indica che può esistere un autore che non abbia un articolo associato;

- il secondo N nella freccia che collega l'*Articolo* alla *Pubblicazione* indica che un articolo può essere associato a più autori;

- il secondo 1 nella freccia che collega il *Commento* all'*Associazione* indica che un commento può essere associato al massimo a un articolo.

Progettazione logica dei dati

La progettazione logica dei dati parte dall'output della progettazione concettuale per costruire un **database**.

Riprendiamo la definizione di database. Un database è un insieme di dati:

- in relazione tra loro

- archiviati (generalmente su un supporto informatico)

- organizzati secondo determinate strutture

- creato per :
    - supportare software, applicazioni web, eccetera
    - analizzare e controllare i dati
    - archiviarli

Torniamo alla progettazione logica

La progettazione logica dei dati parte dall'output della progettazione concettuale per costruire un database.

La prima scelta da fare è stabilire la tipologia di Database. Spesso la scelta cade sui Database **relazionali**.

A questo punto, a partire dalle entità e dalle relazioni, andrò a definire un insieme di **tabelle** (con i rispettivi **vincoli**).

Esempio di progettazione logica

Traduzione di entità e attributi

Vediamo come passare da un diagramma E-R a un insieme di tabelle:

- ogni **entità** diventa una **tabella**;
- gli **attributi semplici** dell'entità diventeranno le rispettive **colonne**;
- identifichiamo le **chiavi primarie** per ogni tabella (attributo o insieme di attributi che identificano un'istanza di un'entità);
- per ogni **attributo multi-valore** creiamo **un'ulteriore tabella** contenente in ogni riga la chiave primaria dell'entità di partenza e un valore dell'attributo.

Tipologia di relazioni

In base al secondo valore di ogni coppia presente sulle frecce del diagramma E-R, possiamo individuare tre tipi di relazione:

- **uno a uno** se per entrambe le entità il valore è 1;
- **uno a molti** se per un'entità il valore è 1 mentre per l'altra è N;
- **molti a molti** se per entrambe le entità il valore è N.

Traduzione delle relazioni

Nel nostro caso *Articoli-Autori* sarà una relazione **molti a molti**, mentre *Commenti-Articoli* sarà una relazione **molti a uno**. Procederemo in questo modo:

- creeremo una **nuova tabella** per la relazione molti a molti, contenente le chiavi primarie delle entità coinvolte;

- **aggiungeremo una colonna nella tabella corretta** per gestire la relazione molti a uno. Nel caso che stiamo seguendo occorrerà aggiungere alla tabella *Commenti* (in cui è presente un 1 come secondo numero) la chiave primaria della tabella *Articoli* (in cui è presente un N come secondo numero).

- sulle chiavi primarie aggiunte nelle nuove tabelle inseriremo il vincolo di **chiave esterna**

Progettazione fisica

Durante la progettazione fisica progettiamo l'effettiva implementazione delle tabelle sul particolare RDBS scelto. Ragioniamo in particolare anche su:

- i tipi delle colonne disponibili nel particolare RDBS scelto
- gli indici clustered e non clustered da aggiungere sulle tabelle
- l'implementazione nello specifico linguaggio dei vincoli sui dati (anche tramite oggetti particolari come i trigger e le stored prodecure)
- la creazione di viste e viste materializzate

Implementazione in SQL

Tramite il linguaggio SQL posso creare concretamente il Database su un particolare R-DBMS (Relational-DBMS).

Eseguirò in particolare il comandi CREATE che fa parte delle istruzioni DDL (Data Definition Language)

Tipi delle colonne

Dopo ogni colonna è indicato il tipo. I tipi utilizzati più frequentemente su SQL Server sono

- **Int:** numeri interi
- **Decimal(18,4):** numeri decimali con 18 cifre totali di cui al più 4 decimali
- **Varchar(250):** per stringhe lunghe al più 250 caratteri
- **Nvarchar(250**): per gestire anche caratteri particolari (Unicode data)
- **Date:** date senza orario (su Oracle sarebbe diverso)
- **Datetime:** date con indicazione dell'orario

Il tipo come vincolo

Il tipo di una colonna è in ultima analisi un **Vincolo**: vincoliamo le colonne ad ammettere solo alcune tipologie di dati.

Ad esempio in una colonna di tipo **int** non possiamo inserire un valore che non sia un numero intero. La scelta del tipo di una colonna è dunque un aspetto cruciale per garantire che i dati del database abbiano una qualità minima necessaria per poter svolgere attività di analisi dei dati.

Null / Not Null

Indicare di fianco ad una colonna l'opzione NOT NULL significa garantire che la colonna non potrà mai contenere NULL.

Viceversa, specificando NULL teniamo aperta questa possibilità.

Non specificare nessuna delle due opzioni equivale a specificare NULL.

Chiave primaria

Inserire un vincolo **di chiave primaria** su una colonna significa garantire che in quella colonna non potrà mai essere presente lo stesso valore in più di una riga della tabella.

Inoltre in una colonna chiave primaria non può essere presente NULL.

La chiave primaria può essere definita anche su un insieme di colonne: in questo caso non potrà ripetersi la combinazione di valori presenti in quelle colonne.

Chiave esterna

Spieghiamo il concetto di **chiave esterna** con un esempio: nella tabella *Commenti* abbiamo inserito la chiave esterna *IdArticolo* che fa riferimento alla chiave primaria *IdArticolo* della tabella *Articoli*. Sono generati così due vincoli

- all'interno della tabella *Commenti* non è possibile inserire un valore di *IdArticolo* che non sia già presente all'interno della chiave primaria *IdArticolo* della tabella *Articoli*;

- non è possibile cancellare una riga nella tabella *Articoli* con un *IdArticolo* già presente all'interno della chiave esterna *IdArticolo* della tabella *Commenti*.

Identity

Se in una colonna indichiamo la proprietà **IDENTITY(1,1)** stiamo chiedendo al database di valorizzare automaticamente quella colonna tramite un numero progressivo che parte da 1 e si incrementa di 1 ad ogni **tentativo** di inserimento.

Osservazione sulle Join

Quando scriviamo una Join nella condizione ON inseriamo nella maggior parte dei casi:

- l'uguaglianza tra la chiave primaria di una tabella e la chiave esterna di un'altra

- l'uguaglianza tra le due chiavi primarie (ma solo se contengono la stessa informazione)

```
SELECT *
FROM   dbo.Articoli AS A
LEFT JOIN dbo.Commenti AS C
   ON A.IdArticolo = C.IdArticolo;
```

Operazioni DDL

Creazione di un nuovo schema o Database

In base al particolare R-DBMS potremmo pensare di creare gli oggetti:

- in uno schema già esistente di un Database già esistente
- In un nuovo Schema all'interno di un database già esistente, con l'istruzione CREATE Schema
- in un nuovo Database con l'istruzione CREATE DATABASE

In base alla scelta avremo ovviamente possibilità diverse di segregare i dati.

ATTENZIONE: su Azure SQL Database l'istruzione CREATE DATABASE **non** è disponibile perché la creazione di un Database richiede logiche differenti

Vincolo di chiave primaria

Creiamo le tabelle relative all'entità *Autore* e all'entità *Commenti* specificando diversamente la chiave primaria. Solo per facilità, continuiamo sullo schema dbo.

```
CREATE TABLE dbo.Autori(
    IdAutore INT NOT NULL,
    Nome VARCHAR(100) NOT NULL,
    Cognome VARCHAR(100) NOT NULL,
    PRIMARY KEY (IdAutore));

CREATE TABLE dbo.Commenti(
    IdCommento INT NOT NULL,
    Testo VARCHAR(1000) NOT NULL,
    PRIMARY KEY (IdCommento));
```

Aggiungere una colonna a una tabella

Gestiamo la relazione tra l'entità *Articolo* e l'entità *Commento* aggiungendo una colonna alla tabella *Commenti*

```
ALTER TABLE dbo.Commenti ADD IdArticolo INT NOT NULL;
```

Aggiungere un vincolo di Chiave esterna

Aggiungiamo alla colonna *IdArticolo* della tabella *dbo.Commenti* un vincolo di chiave esterna che fa riferimento alla chiave primaria *IdArticolo* della tabella dbo.*Articoli*

```
ALTER TABLE dbo.Commenti
ADD FOREIGN KEY (IdArticolo)
    REFERENCES dbo.Articoli(IdArticolo);
```

Vincoli di chiave nella Create

Creiamo la tabella per gestire l'attributo multi-valore *Email* dell'entità *Autore*

```
CREATE TABLE dbo.AutoriEmail(
    IdAutore INT NOT NULL,
    Email VARCHAR(100) NOT NULL,
    PRIMARY KEY (IdAutore,Email),
    FOREIGN KEY (IdAutore) REFERENCES dbo.Autori(IdAutore) );
```

Tabella per relazione molti a molti

Gestiamo la relazione molti a molti tra l'entità *Articolo* e l'entità *Autore*

```
CREATE TABLE dbo.AutoriEmail(
    IdAutore INT NOT NULL,
    Email VARCHAR(100) NOT NULL,
    PRIMARY KEY (IdAutore,Email),
    FOREIGN KEY (IdAutore) REFERENCES dbo.Autori(IdAutore) );
```

Eliminare definitivamente una tabella

Per eliminare **definitivamente** e **permanentemente** una tabella posso eseguire l'istruzione *Drop Table*

```
DROP TABLE CorsoSQL.dbo.Articoli;
```

ATTENZIONE: si tratta di un'istruzione che **non lanceremo praticamente MAI in un ambiente di produzione**, se non in casi molto specifici.

Anteponendo anche il nome del Database, aggiungiamo un ulteriore livello di sicurezza.

Eliminare definitivamente il contenuto di una tabella

Per eliminare **definitivamente** e **permanentemente** il contenuto di una tabella (senza eliminare la struttura) posso eseguire l'istruzione *Truncate Table*

```
TRUNCATE TABLE CorsoSQL.dbo.Articoli;
```

ATTENZIONE: si tratta di un'istruzione che **non lanceremo praticamente MAI in un ambiente di produzione**, se non in casi molto specifici.

Anteponendo anche il nome del Database, aggiungiamo un ulteriore livello di sicurezza.

Istruzioni DML per la modifica dei dati

ATTENZIONE

INSERT, UPDATE, DELETE e MERGE sono istruzioni con effetto PERMANENTE sui dati. Lanciamole con molta attenzione solo quando necessario e con i dovuti test!

Anteponendo anche il nome del Database, aggiungiamo un ulteriore livello di sicurezza.

In alcuni contesti sarà necessario eseguire il COMMIT (conferma) o il ROLLBACK (annullamento) di queste istruzioni.

Ma con le impostazioni di default, il COMMIT avviene automaticamente!

Popolare la tabella Articoli con Insert - values

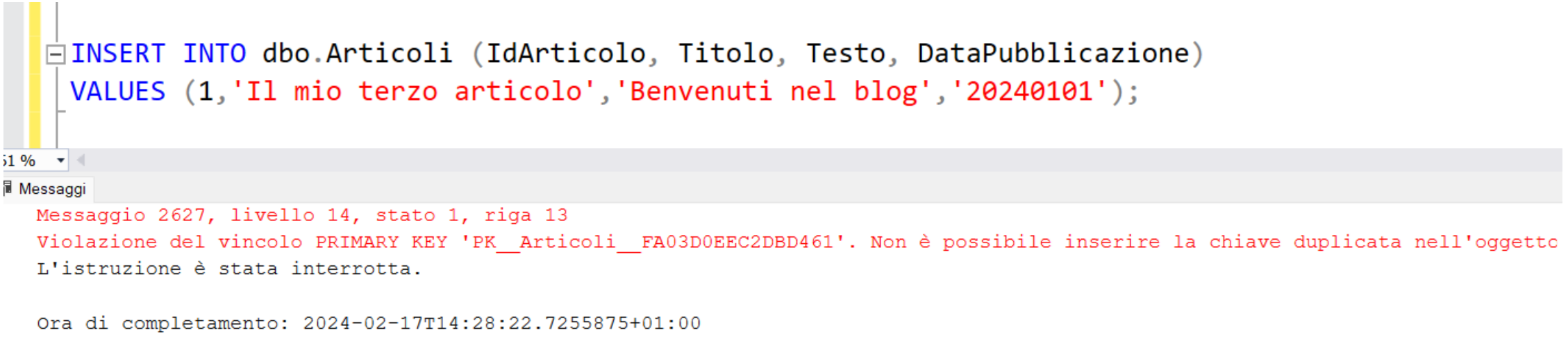
La tabella *Articoli* può essere popolata in svariati modi. Iniziamo inserendo direttamente i valori tramite l'istruzione SQL **Insert**

```
INSERT INTO CorsoSQL.dbo.Articoli (IdArticolo, Titolo,
                                   Testo, DataPubblicazione)
VALUES (1,'Il mio primo articolo','Benvenuti nel blog','20240101');
```

Rispetto dei vincoli di chiave primaria

Analogamente non posso inserire più righe con lo stesso valore di *IdArticolo* per il vincolo di *chiave primaria*.

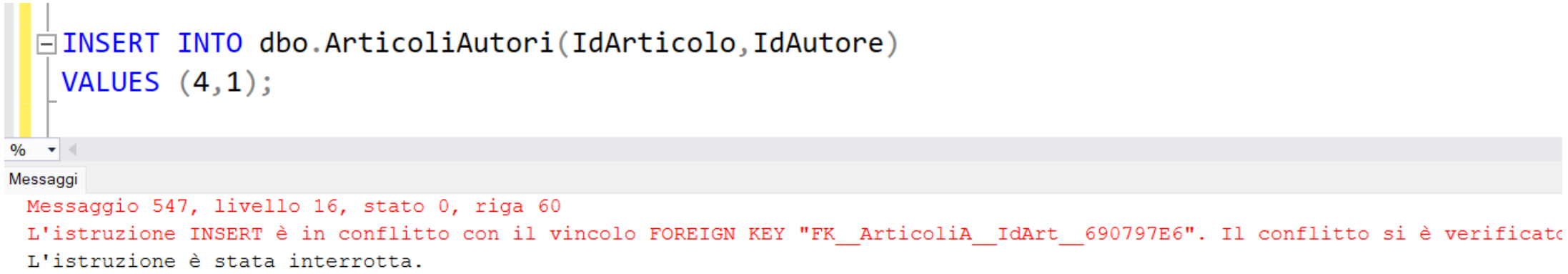
```
INSERT INTO CorsoSQL.dbo.Articoli (IdArticolo, Titolo,
                                   Testo, DataPubblicazione)
VALUES (1,'Il mio terzo articolo','Benvenuti nel blog','20240101');
```



Rispetto dei vincoli di chiave esterna

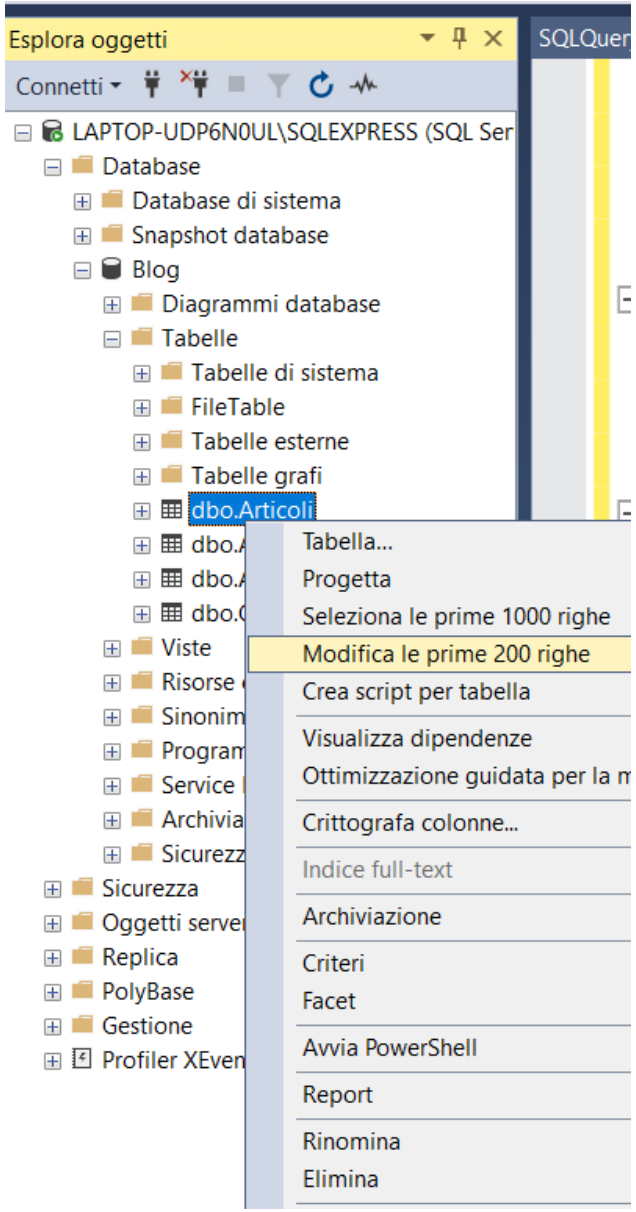
Analogamente non posso inserire all'interno di *ArticoliAutori* un valore di *IdArticolo* che non è stato ancora censito nella tabella *Articoli* per il vincolo di *chiave esterna*.

```
INSERT INTO CorsoSQL.dbo.ArticoliAutori(IdArticolo,IdAutore)
VALUES (4,1);
```



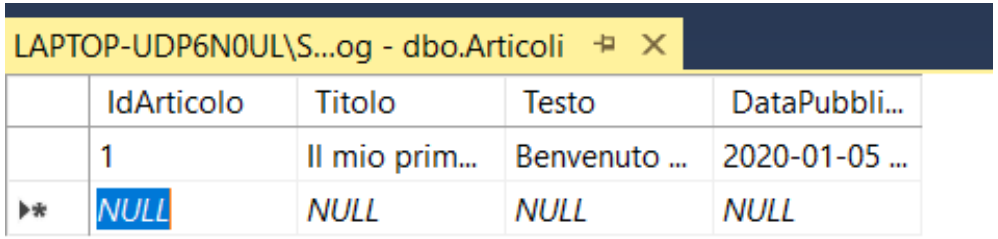
Inserimento tramite Management Studio

Posso inserire i dati nella tabella tramite SQL Server Management Studio



Inserimento tramite Management Studio

Ho una vista simile a Excel, devo però sempre rispettare tutti i vincoli definiti in precedenza!



Inserire i dati tramite Insert Select

Se i dati da inserire sono già presenti in un'altra tabella, possono essere "copiati" tramite l'istruzione *Insert* seguita da una query SQL.

```
INSERT INTO CorsoSQL.dbo.Articoli (IdArticolo, Titolo, Testo,
DataPubblicazione)
SELECT IdArticolo,
   Titolo,
   Testo,
   GETDATE()
FROM dbo.Articoli2
WHERE IdArticolo > 5;
```

Aggiornare i dati

Per modificare permanentemente il titolo della riga della tabella *Articoli* con *IdArticolo* pari a 1 posso usare l'istruzione *Update*

```
UPDATE CorsoSQL.dbo.Articoli
SET   Titolo = 'Nuovo titolo'
WHERE IdArticolo = 1;
```

Cancellare i dati

Per eliminare permanentemente la riga della tabella *Articoli* con *IdArticolo* pari a 1 posso usare l'istruzione *Delete*:

```
DELETE
FROM  CorsoSQL.dbo.Articoli
WHERE IdArticolo = 1;
```

Aggiornare i dati tramite un'applicazione

I dati di un Database possono essere aggiornati anche tramite un applicativo/software costruito tramite linguaggi di **Front End** per creare l'interfaccia e **Back End** per connettersi al Database