[Intro 2](#_Toc137034343)

[Creazione, cancellazione, rinomina, cambio database 3](#_Toc137034344)

[Schemi 4](#_Toc137034345)

[Creazione, cancellazione, inserimento dati tabelle 5](#_Toc137034346)

[Viste 8](#_Toc137034347)

[Stored Procedures 9](#_Toc137034348)

[Tipi di variabili 11](#_Toc137034349)

[Queries 13](#_Toc137034350)

[Join 19](#_Toc137034351)

[Subqueries 21](#_Toc137034352)

[Cicli 23](#_Toc137034353)

[Funzioni 24](#_Toc137034354)

## Intro

**SQL Server** is a relational database management system, or RDBMS, developed and marketed by Microsoft. Similar to other RDBMS software, SQL Server is built on top of SQL, a standard programming language for interacting with relational databases.

SQL Server consists of two main components:  
- **Database Engine**: consists of a relational engine (also known as the query processor, it contains the components that determine the best way to execute a query) that processes queries and a storage engine (in charge of storage and retrieval of data from the storage systems such as disks and SAN) that manages database files, pages, indexes, etc. The database objects such as [stored procedures](https://www.sqlservertutorial.net/sql-server-stored-procedures/basic-sql-server-stored-procedures/), [views](https://www.sqlservertutorial.net/sql-server-views/), and [triggers](https://www.sqlservertutorial.net/sql-server-triggers/) are also created and executed by the Database Engine.  
- **SQLOS**: provides many operating system services such as memory and I/O management. Other services include exception handling and synchronization services.

Le keywords non sono case sensitive, i nomi di database, colonne, tables si. Per distinguere una query dall’altra, si usa il punto e virgola alla fine (;). Per inserire un commento in una sola riga, basta inserire due volte di seguito il carattere "--" (si possono mettere anche a metà riga, e rendono commento tutto quello che c’è alla loro destra), per i commenti su più linee invece si mettono “\*/” per iniziare e “\*/” per finire.

Le operazioni algebriche possibili sono le classiche 4 (+, -, \*, /), e il modulo (%); ci sono le operazioni bitwise (& è l’AND, | è l’OR, ^ è l’OR esclusivo); ci sono le operazioni di confronto (>, <, >=, <=, <> è non uguale, =); ci sono le operazioni composte (+= aggiungi all’uguale, -= sottrai all’uguale, \*= moltiplica all’uguale, /= dividi all’uguale, %= modulo all’uguale, &= AND all’uguale, |\*= OR all’uguale, ^-= OR esclusivo all’uguale).

## Creazione, cancellazione, rinomina, cambio database

Per creare un database

# CREATE DATABASE Nome\_database;

Per osservare tutti i database presenti

# SELECT name FROM master.sys.databases ORDER BY name;

Per modificare il nome di un database, o tasto destro sull’esplora oggetti a sinistra, oppure

Alter database Nome\_Database Modify Name = Nuovo\_nome\_database

Per eliminare un database (non si può fare se il database è in uso, e i database di sistema non si possono droppare), o tasto destro sull’esplora oggetti a sinistra, oppure

# DROP DATABASE Nome\_database;

Se altri utenti sono collegati, per cancellarlo va messo in modalità utente singolo  
Alter Database Nome\_Database Set Single\_User With Rollback Immediate

Per cambiare il database che si sta utilizzando, o dal menu a finestra in alto a sinistra, oppure con la seguente query  
Use [Nome\_Database]  
Go

Per creare un backup del database in uso (backup creato al path dato):

# BACKUP DATABASE databasename TO DISK = 'filepath';

## Schemi

Uno schema è una collezione di oggetti, tra cui tabelle, viste, indici… Ogni schema è associato ad uno username (ovvero il proprietario dello schema), e appartiene ad un solo database. Tuttavia, un database può avere diversi schemi, in base alla funzione dello stesso. Due tabelle di due schemi diversi possono avere lo stesso nome: schema1.tab e schema2.tab.

Per creare uno schema, il comando è il seguente:

# CREATE SCHEMA customer\_services; GO

Li possiamo trovare nella sezioni Security 🡪 Schemas.

Per avere una lista di tutti gli schemi:

# SELECT s.name AS schema\_name, u.name AS schema\_owner FROM sys.schemas s INNER JOIN sys.sysusers u ON u.uid = s.principal\_id ORDER BY s.name;

Si possono alterare gli schemi, ad esempio aggiungendo oggetti ad uno schema:

# ALTER SCHEMA sales TRANSFER OBJECT::dbo.offices;

Per cancellare uno schema, il comando è il seguente:

# DROP SCHEMA logistics;

## Creazione, cancellazione, inserimento dati tabelle

Per creare una tabella si può premere col tasto destro sull’esplora oggetti a sinistra. Si specificano i nomi delle colonne, il tipo di dato nella colonna (eventualmente con anche la lunghezza massima tra parentesi per alcuni tipo le stringhe), la colonna considerata chiave primaria (freccia a sinistra col tasto destro) e se sono accettati valori nulli.   
Finita di impostare, si clicca col tasto destro sulla “finestra” e si salva col nome.

Altro metodo è

# CREATE SCHEMA production; go

# CREATE TABLE production.categories ( category\_type INT IDENTITY (1, 1), category\_name VARCHAR (255) NOT NULL category\_id INT (255) NOT NULL CONSTRAINT category\_id PRIMARY KEY (category\_type, category\_name), FOREIGN KEY category\_id REFERENCES Production.price(category\_name), CHECK (category\_type > 100), );

I cosiddetti vincoli (constraints) definiscono alcune regole/caratteristiche delle singole variabili in ogni tabella. Ce ne sono di diversi tipi:  
- IDENTITY crea un valore autogenerato che aumenta sempre di un valore costante (il primo numero è il numero da cui parte l’ordine, il secondo è l’incremento) (non è necessario inserire dati per colonne con IDENTITY).   
- NOT NULL definisce che una colonna non può avere valori mancanti;  
- UNIQUE definisce che tutti i valori di una colonna sono diversi tra di loro, non ci sono ripetizioni (simile al primary key, ma ce ne possono essere di più in una tabella);  
- PRIMARY KEY definisce la chiave primaria della tabella (automaticamente acquisisce le proprietà di UNIQUE); può essere formata da più colonne se specificato con CONSTRAINT nome\_constraint PRIMARY KEY (colonna1, colonna2);  
- FOREIGN KEY è un campo che si collega a chiavi primarie di altre tabelle, dunque non può essere cancellata;   
- CHECK stabilisce delle condizioni che vanno rispettate in una colonna;  
- DEFAULT stabilisce un valore dato a tutti i nuovi dati inseriti in una tabella, se non specificato qualcosa di diverso;  
- CREATE INDEX per ritrovare dati dai database più velocemente

Tutte queste condizioni si possono anche aggiungere posteriormente con il comando (con il constraint di interesse):

# ALTER TABLE sales.quotations ADD CHECK (category.id >30);

Per eliminare una tabella tasto destro sull’esplora oggetti a sinistra, oppure:

# DROP TABLE sales.delivery;

Per inserire dati in una tabella, si può premere col tasto destro e selezionare “edit first 200 rows” (“modifica le prime 200 righe”) per inserirli in modo manuale, oppure per ogni riga (ovviamente si possono mettere più righe di seguito):

# INSERT INTO hr.person(first\_name, last\_name, gender) OUTPUT inserted.person\_id VALUES('John','Doe', 'M');

Oppure

# INSERT INTO hr.person(first\_name, last\_name, gender) VALUES('John','Doe', 'M'), ('John','Doe', 'M'), ('John','Doe', 'M');

Per modificare la tabella, premere col tasto destro sull’esplora oggetti a sinistra e andare su “design” (“progetta”). Qua si possono aggiungere colonne, impostare la chiave primaria, cancellare colonne, creare relazioni fra tabelle diverse…

Per modificare il nome di una tabella, tasto destro sulla tabella e “Rename”, oppure:

# EXEC sp\_rename 'sales.contr', 'contracts';

Per aggiungere una nuova colonna ad una tabella già creata:

# ALTER TABLE sales.quotations ADD description VARCHAR (255) NOT NULL;

Per modificare una colonna già esistente:

# ALTER TABLE sales.quotations ALTER COLUMN c VARCHAR (255) NOT NULL;

Per eliminare una (o più) colonne già esistente in una tabella:

# ALTER TABLE sales.price\_lists DROP COLUMN discount, surcharge;

Se la colonna ha un vincolo (constraint), prima va eliminato il vincolo (constraint) e poi la colonna:

# ALTER TABLE sales.price\_lists DROP CONSTRAINT ck\_positive\_price; ALTER TABLE sales.price\_lists DROP COLUMN price;

Per aggiungere una colonna composta (computed column) ad una tabella già creata, si fa così (si possono fare operazioni algebriche, aggiungere funzioni, spazi, caratteri, stringhe…):

# ALTER TABLE persons ADD full\_name AS (first\_name + ' ' + last\_name);

Per modificare una colonna intera (si possono anche modificare più colonne) in una tabella:

# UPDATE sales.taxes SET n\_sales += 2;

Per cancellare una o più righe (TOP/BOTTOM n per quel numero di righe, con PERCENT facoltativo per il %), c’è il comando DELETE:

# DELETE TOP (5) PERCENT FROM production.product\_history WHERE model\_year = 2017;

Per porre una variabile come chiave primaria di una tabella (ovvero la colonna che identifica in modo univoco ogni riga), va specificato questo al momento della creazione della tabella:

# CREATE TABLE production.categories ( category\_id INT IDENTITY (1, 1) PRIMARY KEY, category\_name VARCHAR (255) NOT NULL );

Se le chiavi primarie sono due, si pone in modo differente:

# CREATE TABLE sales.participants( activity\_id int, customer\_id int, PRIMARY KEY(activity\_id, customer\_id) );

Per aggiungere dopo la creazione di una tabella una chiave primaria:

# ALTER TABLE sales.events ADD PRIMARY KEY (event\_id);

Invece una chiave esterna (foreign key) è una colonna, o un gruppo di colonne, che identifica in modo univoco righe di un’altra tabella. Si crea in questo modo:

# CREATE TABLE procurement.vendors ( vendor\_id INT IDENTITY PRIMARY KEY, vendor\_name VARCHAR(100) NOT NULL, group\_id INT NOT NULL, CONSTRAINT fk\_group FOREIGN KEY (group\_id) REFERENCES procurement.vendor\_groups(group\_id));

Per creare relazioni (la reference dev’essere sempre una chiave primaria di una tabella) tra tabelle diverse, oltre al designer, si può usare  
Alter Table Foreign\_Key\_Table add constraint Foreign\_Key\_Table\_Column\_FK  
Foreign Key (Colonna\_Chiave) references Tabella\_Chiave\_Primaria (Colonna\_Chiave\_Primaria)

Quando si crea una relazione di questo tipo (con una tabella esterna che ha il valore come chiave primaria), se si prova ad inserire un valore nella tabella diverso da tutti quelli della tabella dov’è chiave primaria da errore.

Per creare un indice in una tabella (ovvero in che ordine vengono ordinati i risultati delle query), si usano i comandi relativi ad ‘index’:

# CREATE INDEX ID\_EMPLOYEE\_ID ON hr.employees (EMPLOYEE\_ID);

Per cancellarlo:

# DROP INDEX ID\_JOB\_ID ON hr.employees;

## Viste

Si possono salvare i risultati di queries dandogli un nome, ovvero creando le views (viste); si tratta di tabelle:

# CREATE VIEW sales.daily\_sales AS SELECT \* FROM sales.customers;

In questo modo si potrà richiamare la vista formata dalla query nominandola.

Ovviamente si possono cancellare delle viste, col seguente comando:

# DROP VIEW sales.daily\_sales

Si possono anche rinominare delle viste, cliccando col tasto destro su di essa dal menù a sinistra.

Per vedere tutte le viste presenti, si può usare il seguente comando:

# SELECT OBJECT\_SCHEMA\_NAME(v.object\_id) schema\_name, v.name FROM sys.views as v;

Per richiamare una vista, si fa come se fosse una tabella (il risultato infatti è una tabella):

# SELECT \* FROM nome\_vista

Attenzione quando si crea una vista, non si può inserire l’order by all’interno di una vista. Se serve un order by, va inserito nella query in cui si richiama la vista.

Le viste sono comode perché permettono di ripetere operazioni su database che si aggiornano, senza dover modificare il codice ogni volta. In generale, le viste:  
- permettono l’automazione di operazioni;  
- permettono di far ricevere solo alcune informazioni a personale con determinate autorizzazioni;  
- permettono di rendere più semplice e comprensibile il codice in un’operazione;

Per avere le informazioni di una vista:

# EXEC sp\_helptext ‘nome\_vista’ ;

## Stored Procedures

Le stored procedures sono gruppi di statements salvati nel database che possono essere richiamati nominandoli.

# CREATE PROCEDURE nome\_procedura AS BEGIN SELECT \* FROM sales.customers; END;

Tutto ciò che sta tra BEGIN e END è la procedura stessa, chiamata col nome\_procedura. Se è tutto corretto, una volta eseguita viene il risultato “Commands completed successfully.”

Le procedure salvate si possono trovare il “programmabilità” 🡪 “Stored Procedures”, nel menù a sinistra.

Per eseguire una stored procedures, basta fare:

# EXECUTE nome\_procedura

Per modificare una procedura già esistente, basta fare:

# ALTER PROCEDURE nome\_procedura AS BEGIN SELECT \* FROM sales.customers; END;

E il risultato dovrebbe restituire ancora “Commands completed successfully.”

Per cancellare una procedura esistente, basta fare:

# DROP PROCEDURE nome\_procedura

Si possono creare procedure con alcuni parametri da inserire di volta in volta in base a ciò che interessa fare. Per fare ciò, basta fare così:

# CREATE PROCEDURE uspFindProducts(@min\_list\_price AS DECIMAL, @max\_list\_price AS DECIMAL) AS BEGIN SELECT product\_name, list\_price FROM production.products WHERE list\_price >= @min\_list\_price AND list\_price <= @max\_list\_price ORDER BY list\_price; END;

Per eseguire una stored procedures con parametri, basta fare:

# EXECUTE uspFindProducts 900, 1000;

Oppure

# EXECUTE uspFindProducts @min\_list\_price = 900, @max\_list\_price = 1000;

E, come nei casi precedenti, si può fare ALTER PROCEDURE per modificarla.

Si possono anche porre parametri opzionali, con valori che se non dati hanno un default (come il primo dei due in questo caso):

# CREATE PROCEDURE uspFindProducts(@min\_list\_price AS DECIMAL = 0, @max\_list\_price AS DECIMAL) AS BEGIN SELECT product\_name, list\_price FROM production.products WHERE list\_price >= @min\_list\_price AND list\_price <= @max\_list\_price ORDER BY list\_price; END;

## Tipi di variabili

Tra le variabili utilizzabili con Microsoft SQL Server ci sono:  
- gli interi (integer), definiti da INT (ci sono anche altri formati in base a quanto è grande o piccolo il numero)  
- i decimali, definiti con DECIMAL (p,s) (con p come numero totale di cifre significative memorizzate, e s come numero massimo di numeri decimali alla destra del punto).   
- il bit, che può assumere valore 1 (equivalente al TRUE), 0 (equivalente al FALSE) o NULL.  
- il CHAR (n), ovvero le stringhe, con n che è la lunghezza massima consentita della stringa.   
- l’NCHAR (n), ovvero la stringa in Unicode, con n che è la lunghezza massima consentita della stringa. Rispetto a CHAR si usa quando le stringhe hanno lunghezza definita, e non variano molto.   
- il VARCHAR (n), ovvero una stringa non Unicode, con n che è la lunghezza massima consentita della stringa.   
- il NVARCHAR (n), ovvero una stringa Unicode con lunghezza definita, con n che è la lunghezza massima consentita della stringa.   
- il DATETIME2, ovvero la data e l’ora, nel formato YYYY-MM-DD hh:mm:ss. È composta da due parti, la data e l’ora.   
- il DATE, ovvero la data, nel formato YYYY-MM-DD.   
- il TIME, ovvero l’orario, nel formato hh:mm:ss.

Si possono anche creare, assegnare, cancellare variabili. Queste variabili si possono poi usare nelle queries. Si creano col seguente comando (specificando il tipo) (il nome deve iniziare con la chiocciola) (se ne possono dichiarare di più con una virgola tra una e l’altra):

# DECLARE @model\_year INT;

Per assegnare un valore ad una variabile:

# SET @model\_year = 2018;

Si possono anche assegnare valori trovati con una query:

# SET @product\_count = ( SELECT COUNT(\*) FROM production.products);

Per le date, ci sono funzioni particolari utili:

# SELECT DAY(employees.HIRE\_DATE) FROM hr.employees

# SELECT MONTH(employees.HIRE\_DATE) FROM hr.employees

# SELECT YEAR(employees.HIRE\_DATE) FROM hr.employees

Ci sono funzioni anche per le differenze tra date (tre esempi, ma anche **week, hour, minute, second**…):

# SELECT DATEDIFF(DAY, employees.HIRE\_DATE, employees.CONTRACT\_END) FROM hr.employees

# SELECT DATEDIFF (MONTH, employees.HIRE\_DATE, employees.CONTRACT\_END) FROM hr.employees

# SELECT DATEDIFF (YEAR, employees.HIRE\_DATE, employees.CONTRACT\_END) FROM hr.employees

Per controllare se un valore è una data, si usa *ISDATE(data)*.

Per modificare un tipo di un dato, si può usare il comando CAST:

# SELECT CAST(10.95 AS INT)

Si può fare la seguente operazione:   
- da decimale a intero (AS INT)  
- da decimale a decimale con lunghezza diversa (AS DEC(3,0))  
- da stringa a data (AS DATETIME)  
- da data a Varchar (AS VARCHAR)  
- da NULL a intero (AS INT)  
…

Per creare una sequenza di numeri:

# CREATE SEQUENCE stud\_seq    AS INT START WITH 101   INCREMENT BY 1;

## Queries

Si usa il comando SELECT per scegliere quali colonne visualizzare in una tabella, che viene definita con FROM (SELECT e FROM sono gli unici due comandi obbligatori in una query).

# SELECT \* FROM sales.customers;

In questo modo selezioniamo tutte le colonne nella tabella *nome\_tabella*. Però attenzione che con tabelle molto grandi questo può creare problemi. Se volessimo selezionare una o più colonne faremmo così.

# SELECT first\_name FROM sales.customers;

# SELECT first\_name, last\_name FROM sales.customers;

Invece, se volessimo filtrare le righe in base ad una condizione (definita con simboli come =, >, <, >=, <=, !=), aggiungeremmo il WHERE. L’ordine dell’operazione, processata dal programma, è FROM 🡪 WHERE 🡪 SELECT.

# SELECT \* FROM sales.customers WHERE state = 'NY';

Si possono anche selezionare diverse condizioni (con AND/OR; in caso ci siano entrambi, la priorità va sull’AND) nel WHERE:

# SELECT \* FROM sales.customers WHERE state = 'NY' AND city = 'New York';

Tra le condizioni possibili c’è anche l’appartenere ad un range (si può anche fare NOT BETWEEN se interessano valori al di fuori di un certo range; si usa questo anche per le date):

# SELECT \* FROM sales.customers WHERE zip\_code BETWEEN 10000 AND 11000;

Tra le condizioni possibili c’è anche l’appartenere ad un insieme di valori definiti (si può fare anche il contrario con NOT IN; in generale equivale a fare degli OR con le virgole tra ogni singolo elemento tra parentesi):

# SELECT \* FROM sales.customers WHERE zip\_code IN (10950,10562);

Tra le condizioni possibili ci sono le wildcards, ovvero caratteri che implicano una sostituzione di uno o più caratteri in una stringa. Tra le varie possibilità, c’è l’avere una stringa che contiene una parte di stringa definita in una posizione random (se interessino le stringhe diverse, basta mettere NOT LIKE) (con i % possono esserci da 0 a molti caratteri mancanti):

# SELECT \* FROM sales.customers WHERE email LIKE '%yahoo.com%';

Tra le condizioni possibili c’è anche l’avere una stringa che contiene una stringa definita all’inizio:

# SELECT \* FROM sales.customers WHERE email LIKE 'de%';

Tra le condizioni possibili c’è anche l’avere una stringa che contiene una stringa definita alla fine:

# SELECT \* FROM sales.customers WHERE last\_name LIKE '%l';

Tra le condizioni possibili c’è anche l’avere una stringa che contiene una stringa alla inizio e una alla fine:

# SELECT \* FROM sales.customers WHERE last\_name LIKE 'd%l';

Tra le condizioni possibili c’è anche l’avere una stringa che contiene una stringa alla fine con un carattere indifferente (definito con \_) (con l’underscore \_ si definisce esattamente un carattere mancante):

# SELECT \* FROM sales.customers WHERE last\_name LIKE '\_a%';

Tra le condizioni possibili c’è anche l’avere una stringa che contiene un carattere a scelta tra una serie di caratteri possibili (compresi tra le parentesi quadre, e con posizione in base alla posizione: stringa che inizia con B o C, e con seconda lettera a o e):

# SELECT \* FROM sales.customers WHERE last\_name LIKE '[BC][ae]%';

Tra le condizioni possibili c’è anche l’avere una stringa che contiene un carattere a scelta tra un range di caratteri compresi tra due lettere (comprese tra le parentesi quadre, e con posizione in base alla posizione; per fare un carattere non compreso in un range, basta fare [^B-F], e in generale il contrario di […] è [^…]):

# SELECT \* FROM sales.customers WHERE last\_name LIKE '[B-F]%';

Tra le condizioni possibili c’è anche il considerare solo valori nulli (se solo valori nulli, basta mettere IS NOT NULL):

# SELECT \* FROM sales.customers WHERE phone IS NULL;

Per comprendere in una stringa caratteri speciali tipo %/’ , va usato un carattere ESCAPE (include il carattere appena dopo l’escape):

# SELECT \* FROM sales.customers WHERE email LIKE '%30!%%' ESCAPE '!';

Per ordinare i dati in base ad una condizione particolare, si usa la condizione ORDER BY (si può usare anche con colonne non comprese nel SELECT). L’ordine dell’operazione, processata dal programma, è FROM 🡪 WHERE 🡪 SELECT 🡪 ORDER BY. Senza nient’altro, ordina le righe in ordine crescente.

# SELECT \* FROM sales.customers WHERE state = 'NY' ORDER BY first\_name;

Per ordinarle in modo crescente o decrescente in base all’interesse, si aggiunge ASC (predefinito se non specificato) e DESC nell’order by dopo la colonna.

# SELECT \* FROM sales.customers WHERE state = 'NY' ORDER BY first\_name ASC;

# SELECT \* FROM sales.customers WHERE state = 'NY' ORDER BY first\_name DESC;

Possono anche essere definite due condizioni o più per l’order by: in questo caso, viene tutto ordinato in base alla prima, e per valori identici della prima condizione poi viene ordinato il tutto in base alla seconda, e così via (se l’ordine è il nome e il cognome, con prima il nome, le righe sono in ordine alfabetico in base al nome, e per nomi uguali in base al cognome).

# SELECT \* FROM sales.customers WHERE state = 'NY' ORDER BY first\_name ASC, last\_name ASC;

Si possono anche inserire funzioni all’interno della condizione dell’order by, come la lunghezza della stringa con len(nome\_colonna).

# SELECT \* FROM sales.customers WHERE state = 'NY' ORDER BY len(first\_name) ASC;

Invece per osservare il numero di righe/occorrenze raggruppati in base ad una particolare colonna, si utilizza il GROUP BY, e dentro alla SELECT si inseriscono il nome della colonna di interesse e la condizione COUNT(\*), che conta il numero di righe per ogni diversa occorrenza della colonna selezionata (si ottiene lo stesso risultato anche usando il SELECT DISTINCT senza il GROUP BY). L’ordine dell’operazione, processata dal programma, è FROM 🡪 WHERE 🡪 GROUP BY 🡪 SELECT 🡪 ORDER BY.

# SELECT city, COUNT(\*) FROM sales.customers WHERE state = 'NY' GROUP BY city ORDER BY city;

Si può anche aggiungere WITH ROLLUP nella clausola del GROUP BY per avere il totale di tutti i gruppi (funziona anche se ci sono più variabili nella clausola GROUP BY):

# SELECT city, COUNT(\*) FROM sales.customers WHERE state = 'NY' GROUP BY WITH ROLLUP city ORDER BY city;

Altre operazioni che si possono fare, oltre al COUNT(), sono la media (AVG), la somma (SUM), il minimo (MIN), il massimo (MAX), la deviazione standard (STDEV), la varianza (VAR), la concatenazione di stringhe (CONCAT(parola, parola)), la lunghezza della stringa (LEN), la stringa resa minuscola (LOWER) o maiuscola (UPPER), la stringa rimpiazzata (REPLACE(colonna, vecchio, nuovo)), la stringa ripetuta tot volte (REPLICATE(parola, volte)), la stringa girata al contrario (REVERSE), una stringa da un numerico (STR), togliere gli spazi vuoti (TRIM). Si possono anche fare operazioni combinate sfruttando i valori delle colonne e operazioni algebriche tra esse.

Si possono anche fare operazioni con funzioni matematiche, come: ABS() per il valore assoluto, ASIN()/ACOS()/ATAN() per arcoseno-arcocoseno-arcotangente, SIN()/COS()/TAN()/COT() per seno-coseno-tangente-cotangente, EXP() per l’esponenziale, LOG/LOG10 per il logaritmo naturale e in base 10, PI per il pi greco, POWER() per l’elevamento a potenza, ROUND() per arrotondare, SQRT() per la radice quadrata, SQUARE() per il quadrato di un numero…

A operazioni come quella precedente si può aggiungere un’ulteriore condizione, grazie all’ HAVING. Questa condizione si aggiunge solo alle colonne comprese nel select (mentre il where definisce una condizione su tutte le righe), e permette di porre condizioni per selezionare determinati gruppi creati col GROUP BY. L’ordine dell’operazione, processata dal programma, è FROM 🡪 WHERE 🡪 GROUP BY 🡪 SELECT 🡪 ORDER BY 🡪 HAVING. Anche nell’HAVING si possono inserire funzioni complicate, e operazioni algebriche tra valori delle colonne.

# SELECT city, COUNT(\*) FROM sales.customers WHERE state = 'NY' GROUP BY city HAVING COUNT(\*) > 10 ORDER BY city;

Si possono definire sia un numero di righe da saltare prima di restituire il risultato, sia un numero di righe da considerare. Si utilizza OFFSET n ROWS (con n pari al numero di righe da saltare, 0 se non si vogliono saltare), e si usa FETCH FIRST n ROWS ONLY (con n pari al numero di righe da far vedere) (si può usare sia FIRST che NEXT, hanno lo stesso significato). Se si vuole usare FETCH va inserito anche OFFSET, anche se con 0 righe saltate. Quando si usa OFFSET non si può usare TOP (TOP 10, TOP 10 PERCENT…).

# SELECT \* FROM sales.customers WHERE state = 'NY' ORDER BY first\_name OFFSET 10 ROWS ;

# SELECT \* FROM sales.customers WHERE state = 'NY' ORDER BY first\_name OFFSET 10 ROWS FETCH FIRST 10 ROWS ONLY;

Si può anche definire una colonna che crea un indice sulle righe basato su una o più colonne:

# SELECT \*, ROW\_NUMBER() OVER (ORDER BY customers\_id) FROM sales.customers;

Invece per definire i migliori n valori ordinati, si può usare un opzione inserita nel select: si usa TOP n prima di definire le colonne di interesse (con n pari al numero di righe di interesse).

# SELECT TOP 10 product\_name, list\_price FROM production.products ORDER BY list\_price DESC;

In caso di righe con lo stesso valore, se interessano i migliori 10 risultati con eventuali pareggi, si aggiunge WITH TIES dopo il TOP n:

# SELECT TOP 5 WITH TIES product\_name, list\_price FROM production.products ORDER BY list\_price DESC;

Se invece interessa una certa percentuale del totale di righe, si usa TOP n PERCENT (con n pari alla percentuale di interesse):

# SELECT TOP 5 PERCENT product\_name, list\_price FROM production.products ORDER BY list\_price DESC;

Se interessano solamente i valori unici relativi ad una o più colonne, si utilizza SELECT DISTINCT … (in caso di valori nulli, ne viene salvato uno solo):

# SELECT DISTINCT city FROM sales.customers WHERE state = 'NY';

# SELECT DISTINCT first\_name, city FROM sales.customers WHERE state = 'NY';

Notiamo che è una forma equivalente all’usare GROUP BY su tutte le colonne di interesse: tuttavia, si tende ad usare GROUP BY più spesso quando si fa una funzione aggregata su più colonne.

Se vogliamo creare una nuova colonna a partire da colonne esistenti (anche aggiungendo nuove parti, come lo spazio in mezzo), ovvero un ALIAS, si procede nel seguente modo (con la parte di AS si nomina la nuova colonna creata; si può fare anche senza mettere AS; la nuova colonna creata si può usare in altre parti della query, come l’ORDER BY):

# SELECT first\_name + ' ' + last\_name FROM sales.customers;

# SELECT first\_name + ' ' + last\_name AS 'full\_name' FROM sales.customers;

Si possono anche rinominare le tabelle in modo tale da rendere le queries più comprensibili.

# SELECT first\_name, last\_name FROM sales.customers AS sc;

Si possono fare operazioni di insiemistica, come intersezione, unione ed eccezione.

L’unione permette di unire in un'unica tabella tutte le righe (che non sono doppie) di due o più tabelle:

# SELECT first\_name, last\_name FROM sales.staffs UNION SELECT first\_name, last\_name FROM sales.customers;

Invece con UNION ALL vengono restituite più volte anche le righe ripetute:

# SELECT first\_name, last\_name FROM sales.staffs UNION ALL SELECT first\_name, last\_name FROM sales.customers;

L'intersezione permette di tenere solo le righe in comune tra due tabelle:

# SELECT first\_name, last\_name FROM sales.staffs INTERSECT SELECT first\_name, last\_name FROM sales.customers;

L'operatore EXCEPT invece permette di confrontare due set di righe, e considera solo quelle del primo gruppo che sono diverse da ogni riga del secondo gruppo:

# SELECT first\_name, last\_name FROM sales.staffs EXCEPT SELECT first\_name, last\_name FROM sales.customers;

Si può usare la funzione CASE per creare una nuova colonna con valori condizionati da altre colonne già presenti (il nome è quello nella riga con END AS, il FROM è la tabella in cui viene inserita la colonna; i WHEN possono essere multipli, e indicano le varie condizioni; di ELSE ce ne può essere solo uno; all’interno dello stesso WHEN ci possono essere più condizioni, con AND tra una e l’altra) (si possono creare anche nested CASE):

# SELECT first\_name, last\_name, state, CASE WHEN state = 'NY' THEN 'New York' ELSE 'Not New York' END AS NY FROM sales.customers;

Si può usare la funzione SUBSTRING per prendere un piccola parte di una stringa di codice (il primo numero è il carattere da cui partire, il secondo quello in cui terminare):

# SELECT SUBSTRING(first\_name,1,3) FROM sales.customers;

Si possono rankare dei risultati con la funzione RANK (in caso di valori uguali li mette allo stesso rank, e poi prosegue il rank in modo da avere un ordine: es. 1° 2° 2° 4° …) (la clausola PARTITION BY è facoltativa, nel caso ranka singolarmente in base alla variabile data):

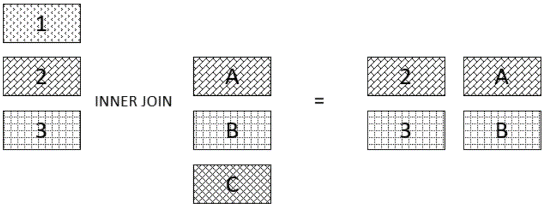
# SELECT first\_name, last\_name RANK() OVER (PARTITION BY city ORDER BY salary) AS ricchezza FROM sales.staffs

La funzione NTILE si comporta in modo simile al RANK, ma semplicemente divide il totale delle righe in quanti gruppi siamo interessati ad avere(in caso di valori uguali li mette allo stesso rank, e poi prosegue il rank in modo da avere un ordine: es. 1° 2° 2° 4° …) (la clausola PARTITION BY è facoltativa, nel caso ranka singolarmente in base alla variabile data) (se 8 righe da dividere in 3, fa 3-3-2):

# SELECT first\_name, last\_name NTILE(5) OVER (PARTITION BY city ORDER BY salary) AS ricchezza FROM sales.staffs

## Immagine che contiene diagramma Descrizione generata automaticamenteJoin

Ci sono 6 tipi diversi di join:   
- inner join (o semplicemente join)  
- left join (o left outer join)  
- right join (o right outer join)  
- full join (o full outer join)  
- cross join  
- self join

Con **l'inner join**, ogni riga della prima tabella viene matchata con una riga della seconda tabella in base alla condizione (la condizione è equivalente ad un where). Se due righe hanno lo stesso valore per la condizione, allora vengono unite. Se la riga della prima tabella non ha valori corrispondenti nella seconda, la riga viene omessa. Si esegue così (la inner di INNER JOIN è facoltativa):

# SELECT product\_name, category\_name, list\_price FROM production.products p INNER JOIN production.categories c ON c.category\_id = p.category\_id ORDER BY product\_name DESC;

Immagine che contiene testo, cruciverba

Descrizione generata automaticamenteCon il **left join**, ogni riga della prima tabella viene restituita, e vengono aggiunti i dati della seconda tabella in base alla condizione di match per le righe in cui il matching è valido (per le righe dove il matching non è valido vengono restituiti valori nulli). Spesso si usa la condizione WHERE (eseguito dopo il join) per limitare i risultati ad un set di interesse.

# SELECT product\_name, order\_id FROM production.products p LEFT JOIN sales.order\_items o ON o.product\_id = p.product\_id WHERE order\_id IS NULL

Con il **right join**, ogni riga della seconda tabella viene restituita, e vengono aggiunti i dati della prima tabella in base alla condizione di match per le righe in cui il matching è valido (per le righe dove il matching non è valido vengono restituiti valori nulli). Spesso si usa la condizione WHERE (eseguito dopo il join) per limitare i risultati ad un set di interesse.

# SELECT product\_name, order\_id FROM sales.order\_items o RIGHT JOIN production.products p ON o.product\_id = p.product\_id ORDER BY order\_id;

Con il **full outer join**, vengono restituite tutte le righe di entrambe le tabelle, con una condizione di matching: quando la condizione è soddisfatta, viene riportata la riga con i dati presi da entrambe le tabelle, altrimenti se la condizione di matching (in entrambe le direzioni) non è soddisfatta viene riportata la riga con i dati di una tabella e valori nulli per il resto.

# SELECT m.name member, p.title project FROM pm.members m FULL OUTER JOIN pm.projects p ON p.id = m.project\_id;

Immagine che contiene testo, cruciverba

Descrizione generata automaticamenteCon il **cross join**, si incrociano due tabelle che non hanno una relazione tra di loro; infatti, il cross join tra una tabella di n righe e una di m righe è una tabella di n\*m righe.

# SELECT product\_id, product\_name, store\_id FROM production.products CROSS JOIN sales.stores ORDER BY product\_name, store\_id;

Con il **self join** unisco una tabella con se stessa. Viene usato per fare queries su dati gerarchici (1° query), o per confrontare righe della tabella con altre righe (2° query). Vanno assegnati con nomi diversi le due tabelle della query, anche se sono la stessa.

# SELECT e.first\_name + ' ' + e.last\_name employee, m.first\_name + ' ' + m.last\_name manager FROM sales.staffs e LEFT JOIN sales.staffs m ON m.staff\_id = e.manager\_id ORDER BY manager;

# SELECT c1.city, c1.first\_name + ' ' + c1.last\_name customer\_1, c2.first\_name + ' ' + c2.last\_name customer\_2 FROM sales.customers c1 INNER JOIN sales.customers c2 ON c1.customer\_id > c2.customer\_id AND c1.city = c2.city ORDER BY city, customer\_1, customer\_2;

## Subqueries

Una subquery è una query annidata dentro un’altra query: si possono usare subqueries:  
- al posto di un’espressione  
- con IN e NOT IN  
- con ANY o ALL  
- con EXISTS e NOT EXISTS  
- con UPDATE, INSERT o DELETE  
- nel FROM

Al posto di un’espressione:

# SELECT order\_id, order\_date, (SELECT MAX (list\_price) FROM sales.order\_items i WHERE i.order\_id = o.order\_id) AS max\_list\_price FROM sales.orders o order by order\_date desc;

Con IN e NOT IN:

# SELECT product\_id, product\_name FROM production.products WHERE category\_id IN (SELECT category\_id FROM production.categories WHERE category\_name = 'Mountain Bikes' OR category\_name = 'Road Bikes');

Con l’ANY:

# SELECT product\_name, list\_price FROM production.products WHERE list\_price >= ANY (SELECT AVG (list\_price) FROM production.products GROUP BY brand\_id);

Con l’ALL:

# SELECT product\_name, list\_price FROM production.products WHERE list\_price >= ALL (SELECT AVG (list\_price) FROM production.products GROUP BY brand\_id);

Con EXISTS e NOT EXISTS:

# SELECT customer\_id, first\_name, last\_name, city FROM sales.customers c WHERE EXISTS (SELECT customer\_id FROM sales.orders o WHERE o.customer\_id = c.customer\_id AND YEAR (order\_date) = 2017) ORDER BY first\_name, last\_name;

Nel FROM:

# SELECT AVG(order\_count) average\_order\_count\_by\_staff FROM (SELECT staff\_id, COUNT(order\_id) order\_count FROM sales.orders GROUP BY staff\_id) t;

## Cicli

Si possono creare cicli come nei vari linguaggi di programmazione. Il ciclo IF ELSE si fa nel seguente modo:

# IF @sales > 10000000 BEGIN PRINT 'Great! The sales amount in 2018 is greater than 10,000,000'; END ELSE BEGIN PRINT 'Sales amount in 2017 did not reach 10,000,000'; END

Il ciclo WHILE si fa così:

# DECLARE @counter INT = 1; WHILE @counter <= 5 BEGIN PRINT @counter; SET @counter = @counter + 1; END

Al ciclo WHILE si possono dare altri statement, come BREAK (per interrompere il ciclo ed uscire al raggiungimento di una condizione) o CONTINUE (termina l’iterazione corrente e continua col codice successivo)

## Funzioni

Le funzioni scalari accettano uno o più parametri e restituiscono un solo valore. Permettono di semplificare un codice, permettendo di ridurre a poche parole parti di codice lungo e ripetitivo. Si possono inserire cicli al loro interno, come WHILE, IF… Possono richiamare a loro volta altre funzioni, ma non possono modificare i dati presenti in una tabella.

Dopo averne creata una, si possono trovare le funzioni scalari nel menù a sinistra, sotto Database 🡪 Nome\_Database 🡪 Programmabilità 🡪 Funzioni 🡪 Funzioni a valori scalari.

Si inserisce il nome della funzione, preceduto dallo schema dove va inserita, con i parametri tra parentesi, e poi tra il BEGIN e l’END si inseriscono i passaggi da compiere per ottenere il valore di interesse.

# CREATE FUNCTION hr.guadagnonetto( @SALARY DEC(8,2), @COMMISSION\_PCT DEC(2,2) ) RETURNS DEC (8,2) AS BEGIN RETURN @SALARY \* (1 - @COMMISSION\_PCT); END;

Si richiama la funzione come una funzione built-in (richiamata da sola col SELECT o inserita in una query).

# SELECT hr.guadagnonetto(1000,0.1) AS net\_sale;

# SELECT hr.guadagnonetto(SALARY, COMMISSION\_PCT) FROM hr.employees

Per modificare una funzione scalare, si usa il comando ALTER:

# ALTER FUNCTION hr.guadagnonetto( @SALARY DEC(8,2), @COMMISSION\_PCT DEC(2,2) ) RETURNS DEC (8,2) AS BEGIN RETURN @SALARY \* (1 - @COMMISSION\_PCT); END;

Per eliminare una funzione scalare, si usa il comando DROP:

# DROP hr.guadagnonetto;

Le funzioni con valori di tabella sono simili a quelle con valori scalari, con la differenza che restituiscono una tabella come risultato. Rispetto alle funzioni con valori scalari, non hanno la parte di BEGIN-END (solo un RETURN seguito dalla query), né viene specificato il tipo di dato restituito (semplicemente un comando TABLE)

# CREATE FUNCTION hr.info\_dipendente( @EMPLOYEE\_ID DEC (6,0), @FIRST\_NAME DEC (20), @LAST\_NAME DEC (25), @SALARY DEC(8,2) ) RETURNS TABLE AS RETURN SELECT EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, SALARY FROM hr.employe;

Dopo averne creata una, si possono trovare le funzioni scalari nel menù a sinistra, sotto Database 🡪 Nome\_Database 🡪 Programmabilità 🡪 Funzioni 🡪 Funzioni a valori di tabella.

Si utilizzano solitamente nello statement FROM di una query (si possono inserire valori per i parametri inseriti) (si possono anche specificare particolari colonne/righe da restituire partendo dalla funzione):

# SELECT \* FROM hr.info\_dipendente()

Per modificare una funzione con valori di tabella:

# ALTER FUNCTION hr.info\_dipendente( @EMPLOYEE\_ID DEC (6,0), @FIRST\_NAME DEC (20), @LAST\_NAME DEC (25), @SALARY DEC(8,2) ) RETURNS TABLE AS RETURN SELECT EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, SALARY FROM hr.employe;

Per eliminare una funzione con valori di tabella, si usa il comando DROP:

# DROP hr.info\_dipendente;