

Modèle Physique de données

SQL Server

CONTENU

Introduction	2
Créer une nouvelle Base	2
Créer vos tables	3
Créer une table	3
Types de données	3
Modifier une table	4
Contraintes CHECK, UNIQUE et clés	5
UNIQUE	5
CHECK	5
PRIMARY KEY	5
FOREIGN KEY	6
Suppression de contrainte	7
Ordre de création de tables et de clés	7
Insérer des enregistrements dans notre base de données	9
Créer des vues	10
Selectionner des données	11
Select	11
Alias : AS	11
Retour à la ligne	12
Ordonner les données : ORDER BY	12
Selection de ligne selon des valeurs : WHERE	13
AND / OR / NOT	13
LIKE	13
Effectuer des calculs	14
Grouper une requête selon les différentes valeurs d'une colonne : GROUP BY	14
Sous-requête	15
HAVING	15
Les jointures : JOIN	16

INNER JOIN 16

OUTER JOIN 16

 LEFT OUTER JOIN 16

 RIGHT OUTER JOIN 17

 FULL OUTER JOIN..... 18

 CROSS JOIN 18

Exemple de requête complexe 20



INTRODUCTION

Vous avez terminé de faire votre modèle logique de données (MLD), il est maintenant tant de créer votre base de données. Dans ce cours, nous utiliserons SQL Server Express et SQL Server Management Studio. On pourrait les comparer à Git et GitDesktop. Le premier apporte les fonctionnalités, le langage. L'autre apporte une interface graphique et une facilité d'utilisation. Le but de ce cours n'est pas de se reposer sur l'interface graphique mais de vous habituer à utiliser des requêtes SQL Server.

Après vous être connecté à votre serveur local, vous pourrez entrer dans le vif du sujet.

Dans ce cours, nous allons faire plusieurs requêtes SQL SERVER. Vous voudrez séparer vos fichiers de requêtes de la manière suivante :

- Un fichier pour la création des tables de votre base de données
- Un fichier pour l'insertion de données
- Un fichier pour les vues de vos données
- Un fichier pour la sélection de vos données

CREER UNE NOUVELLE BASE

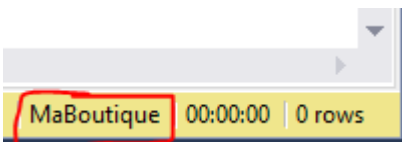
```
CREATE DATABASE MaBoutique;
```

Pour créer une nouvelle base de données par défaut, il suffit d'exécuter la ligne ci-dessus.

Pour plus d'informations sur les différentes options possibles : [suivez ce lien](#).

Après avoir créé votre base, vous pourrez passer sur vos fichiers de requêtes décrit précédemment.

Pour chaque fichier, il faudra vous assurez-vous que votre fichier de requêtes est bien sur votre base de données. Vous pouvez voir sur quelle base vous êtes en regardant en bas à droite de votre fenêtre SQL Server Management Studio :



```
USE MaBoutique;
```

Si vous n'êtes pas dessus vous pouvez exécuter

CREER VOS TABLES

CREER UNE TABLE

Vous pouvez maintenant créer vos tables et y préciser ses différentes colonnes. Pour rappel, dans un modèle physique de données, une table est l'équivalent d'une relation du MLD. Une table est constituée de colonnes qui sont les attributs du MLD. Chaque entrée de données est une ligne de votre table.

	Customer_id	Customer_name	Customer_address	Customer_age
1	1	Paul	Quelque part à Brunstatt	31
2	2	Ludovic	Mulhouse	21
3	3	Toto	Unknown	NULL

Figure 1 : table des clients avec 3 entrées.

Voici un exemple de création d'une table Customers :

```
CREATE TABLE Customers
(Customer_id INT NOT NULL IDENTITY,
Customer_name VARCHAR(50) NOT NULL,
Customer_address VARCHAR(255) NOT NULL,
Customer_age TINYINT);
```

Figure 2 : Création de table.

Pour chaque colonne, il faut obligatoirement donner un nom et un type de données. Il existe ensuite plusieurs arguments optionnels : ici NOT NULL rend obligatoire la saisie d'une valeur dans la colonne spécifiée lors de la saisie d'une nouvelle ligne. IDENTITY permet d'auto-incrémenter une valeur.

Pour voir toutes les possibilités à la création d'une table : [regardez ce lien](#).

TYPES DE DONNEES

Il existe de nombreux types de données. Ces types sont [répertoriés ici](#). Pour chaque colonne il faudra veiller à choisir le bon type.

Les types les plus courant sont :

- le CHAR(X) ou NCHAR(X) est une chaîne de caractères de longueur fixe.
- le VARCHAR(X) ou NVARCHAR(X) qui est une chaîne de caractères de longueur variable dont le nombre de caractères maximum dépend de X.
- le TEXT ou NTEXT permet de stocker de longs textes. Cependant il est amené à disparaître au profit de VARCHAR(max) ou NVARCHAR(max).
- le BIT qui permet d'avoir une valeur de 0 ou 1 ou NULL.
- la DATE qui est une date au format 'AAAA-MM-JJ'.
- les numériques qui sont des nombres entiers ou décimaux.

Les différents entiers sont notamment à choisir parmi les suivants selon les plages recherchées :

Type de données	Plage	Stockage
bigint	-2^{63} (-9 223 372 036 854 775 808) à $2^{63}-1$ (9 223 372 036 854 775 807)	Huit octets
int	-2^{31} (-2 147 483 648) à $2^{31}-1$ (2 147 483 647)	Quatre octets
smallint	-2^{15} (-32 768) à $2^{15}-1$ (32 767)	Deux octets
tinyint	0 à 255	Un octet

Pour information, CHAR et VARCHAR utilise les codes ASCII alors que NCHAR et NVARCHAR utilise l'Unicode (UTF-16). Vous pourrez représenter plus de caractères avec l'Unicode. Pour spécifier qu'une chaîne de caractère utilise l'UTF-16, il faudra également ajouter le N avant votre chaîne de caractère. Par exemple : N'Toto'.

Une donnée de type DATE peut avoir différents formats. On peut choisir à tout moment le format qui nous intéresse à l'affichage de la manière suivante :

```
SELECT FORMAT(Ma_Date, 'MMM/yyyy') ; -- Affiche Ma_Date sous la forme Nom_du_mois_sur_3_caractères/année
```

MODIFIER UNE TABLE

Il est possible d'ajouter une colonne supplémentaire à ma table en utilisant **ALTER TABLE [Nom_de_ma_table] ADD [Infos_colonne]** :

```
ALTER TABLE Customers
ADD Customer_email VARCHAR(200), Customer_birthdate VARCHAR(10);
```

Figure 3 : Ajout de colonnes dans une table.

Ici j'ajoute les colonnes Customer_email et Customer_birthdate à ma table Customers.

On peut modifier une de mes colonnes également, pour changer son type et y ajouter plus d'options (NOT NULL) par exemple. Pour cela on utilise **ALTER COLUMN** en plus du **ALTER TABLE** :

```
ALTER TABLE Customers
ALTER COLUMN Customer_birthdate DATE NOT NULL;
```

Figure 4 : Modification de la colonne Customer_birdthdate dans la table Customers.

On peut également supprimer des colonnes en utilisant **DROP COLUMN** :

```
ALTER TABLE Customers
DROP COLUMN Customer_birthdate, Customer_email;
```

Figure 5 : Suppression d'une colonne dans une table.

Pour voir les différentes modifications possibles de table : [c'est ici](#).

CONTRAINTES CHECK, UNIQUE ET CLES

Les contraintes sont des règles que le SGBD applique automatiquement. Il existe plusieurs contraintes telles que UNIQUE, CHECK, PRIMARY KEY et FOREIGN KEY. On utilise le mot clé CONSTRAINT pour gérer ces contraintes lors de leur création ou de leur suppression.

UNIQUE

La contrainte UNIQUE, comme son nom l'indique, assure l'unicité des valeurs d'une colonne qui n'est pas une clé primaire. Si on essaie d'ajouter une valeur qui existe déjà dans la colonne visée par la contrainte, le SGBD lèvera une erreur car la contrainte a été violée et la ligne n'est pas ajoutée.

Contrairement à une clé primaire, la contrainte UNIQUE peut avoir une valeur NULL. Cependant il ne pourra y en avoir qu'une et une seule.

Les contraintes de type UNIQUE sont associées au symbole 'UQ'. Par exemple pour vérifier leur existence on peut utiliser :
 SELECT * FROM sys.sysobjects WHERE name = 'Nom_de_contrainte' AND xtype = 'UQ' ;

TODO : CONSTRAINT UQ_Customer_address UNIQUE (Address)

CHECK

La contrainte CHECK permet de limiter les valeurs qui peuvent être entrées dans la ou les colonnes ciblées par cette contrainte. De ce fait on peut limiter des valeurs de type CHAR ou VARCHAR à des mots spécifiques ou n'accepter que des valeurs numériques comprises dans une certaine fourchette pour des INT.

On peut appliquer plusieurs contraintes à une même colonne tout comme une contrainte peut cibler plusieurs colonnes.

Les contraintes de type CHECK sont associées au symbole 'C'. On peut donc les lister de la manière suivante :

SELECT * FROM sys.sysobjects WHERE xtype = 'C' ;

TODO :

ALTER TABLE Department

CONSTRAINT UQ_Department_Name CHECK (Name IN ('RESEARCH', 'ACCOUNTING', 'OPERATIONS')) ;

PRIMARY KEY

Une clé primaire d'une table permet d'identifier chaque enregistrement d'une table d'une base de données. Par exemple à partir d'un Customer_id = 2, je peux retrouver toutes les informations du client qui a son Customer_id = 2 : son nom, son adresse, son âge s'il a été indiqué, etc.

Une clé primaire est déclarée par PRIMARY KEY.

Une table ne peut avoir qu'une et une seule clé primaire. Cependant une clé primaire peut contenir plusieurs colonnes. Pour cela, lors de la déclaration de la clé primaire, il suffit de mettre entre parenthèses toutes les colonnes désirées.

Pour le moment, je n'ai pas défini la clé primaire de ma table Customers. Il est possible de la définir à la création de ma table ou par la suite en modifiant ma table.

A la création de ma table :

```
CREATE TABLE Customers
(Customer_id INT NOT NULL IDENTITY,
Customer_name VARCHAR(50) NOT NULL,
Customer_address VARCHAR(255) NOT NULL,
Customer_age TINYINT,
CONSTRAINT PK_Customers_Customer_id PRIMARY KEY (Customer_id));
```

Figure 6 : Clé primaire à la création d'une table.

En modifiant ma table avec **ALTER TABLE [Nom_de_ma_table] ADD CONSTRAINT :**

```
ALTER TABLE Customers
ADD CONSTRAINT PK_Customers_Customer_id PRIMARY KEY (Customer_id);
```

Figure 7 : Ajout d'une clé primaire en modification de table.

En fait, ici, j'ai à chaque fois ajouté ma clé primaire sous la forme d'une contrainte. L'avantage de ceci est de pouvoir la nommer comme je le désire.

Il est possible de déclarer une clé sans passer par le mot clé **CONSTRAINT**, cela créera de toute façon une contrainte de clé mais vous n'aurez aucun pouvoir sur son nom :

```
CREATE TABLE Primary_key_without_constraint
(Id INT IDENTITY PRIMARY KEY);
```

Ou alors en modifiant la table et en lui ajoutant uniquement la clé primaire :

```
ALTER TABLE Primary_key_without_constraint
ADD PRIMARY KEY (Id);
```

Si j'affiche le nom de toutes mes clés primaires, j'ai alors :

	name
1	PK_Customers_Customer_id
2	PK__Primary__3214EC0724B59EF4

Figure 8 : Affichage des clés primaires existantes.

La seconde clé primaire correspond à la clé primaire de Primary_key_without_constraint que je viens de créer.

FOREIGN KEY

Les clés étrangères font référence à des clés primaires de tables voisines. Par exemple, une facture est attribuée à 1 et 1 seul client. Donc dans la table d'une facture, j'aurais une clé étrangère vers la clé primaire du client concerné.

```
CREATE TABLE Invoices
(Invoice_id INT NOT NULL IDENTITY(100,100),
Invoice_total INT NOT NULL,
Invoice_customer_id int NOT NULL,
Invoice_date DATE,
CONSTRAINT PK_Invoices_id PRIMARY KEY (Invoice_id),
CONSTRAINT FK_Invoices_Invoice_client_id FOREIGN KEY (Invoice_customer_id)
REFERENCES Customers (Customer_id));
```

Figure 9 : Ajoute d'une clé étrangère à la création de la table.

Lorsque l'on utilise les mots clé FOREIGN KEY, il faudra ensuite donner la colonne de la table qui va devenir clé étrangère. Ensuite on donne la clé primaire de la table référencée. Ici ma **clé étrangère** sur la table **Invoices** est la colonne **Invoice_client_id**. Ma **référence** est dans la table **Customers** et c'est la colonne **Customer_id**.

Comme les clés primaires, les clés étrangères peuvent être ajoutés comme contrainte ou non et peuvent être mise en place lors de la création de la table ou via modification de la table.

SUPPRESSION DE CONTRAINTE

Tout comme une colonne d'une table, je peux choisir de supprimer une contrainte choisie. Pour cela on passe par ALTER TABLE [Table_a_modifier] DROP CONSTRAINT [Nom_de_la_contrainte] :

```
ALTER TABLE Primary_key_without_constraint
DROP CONSTRAINT PK__Primary___3214EC0724B59EF4;
```

Figure 10 : Suppression de la clé primaire de la table Primary_key_without_constraint.

Ici je supprime la clé étrangère créée précédemment.

Et je peux en profiter pour supprimer la table aussi :

```
DROP TABLE Primary_key_without_constraint;
```

Figure 11 : Suppression de la table Primary_key_without_constraint.

ORDRE DE CREATION DE TABLES ET DE CLES

Lors de la création de vos tables, vous risquez d'avoir plusieurs problèmes. La première étant qu'on ne peut pas créer une table qui existe déjà, il faudra la supprimer avant. De même, on ne peut supprimer une table que si elle existe déjà.

Il est cependant possible de créer des nouvelles tables que si elles n'existent pas déjà. Pour cela on peut utiliser l'instruction suivante :

```
IF NOT EXISTS (SELECT * FROM sys.sysobjects where name = 'Nom_table_a_creer' AND xtype = 'U' );
BEGIN
    [CREATION DE LA TABLE Nom_table_a_creer]
END
```

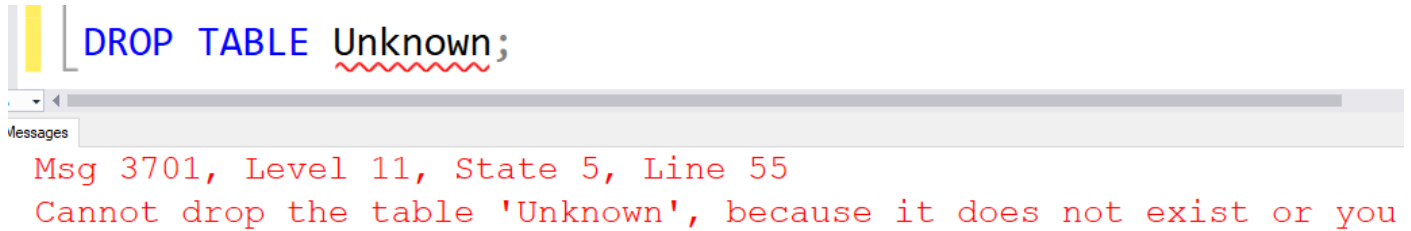



Figure 12 : Suppression de table qui n'existe pas.

Une clé étrangère doit obligatoirement référencer une clé primaire. Il faut donc d'abord créer la clé primaire référencée puis créer la clé étrangère. Donc dans le cas où vous créez vos clés à la création de vos tables, il faudra également bien respecter cet ordre. Dans les exemples précédents, on a bien créé la table Customers et sa clé primaire avant d'ajouter la clé étrangère de la table Invoices.

Il en va de même pour la suppression des clés et des tables. Il faudra d'abord supprimer les clés étrangères puis les clés primaires. Ou bien supprimer les tables avec clés étrangères avant de supprimer les tables qui sont référencées par ces clés étrangères.

INSERER DES ENREGISTREMENTS DANS NOTRE BASE DE DONNEES

Pour ajouter des enregistrement on utilise **INSERT INTO [Nom_de_ma_table] (colonne_désirée1, colonne_désirée2, colonne_désirée3,) VALUES (valeur_colonne_désirée1, valeur_colonne_désirée2, valeur_colonne_désirée3, ...)** :

```
INSERT INTO Customers (Customer_name, Customer_address, Customer_age)
VALUES ('Bourgeois', 'Somewhere', 31), ('Toto', 'Anywhere', 12);
```

Figure 13 : Insertion de 2 lignes dans la table Customers.

On peut ajouter plusieurs enregistrements à la fois. Ici 2 lignes ont été ajoutées.

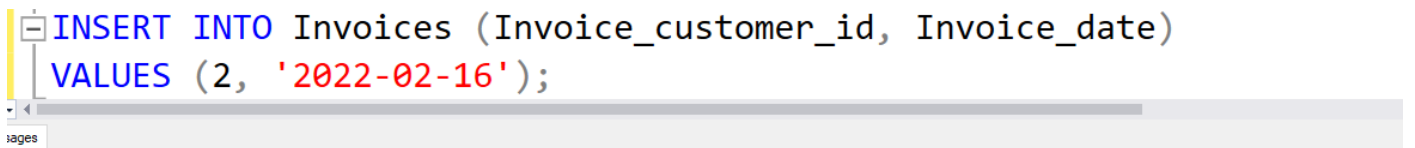
On peut ignorer les colonnes qui ne sont pas obligatoires :

```
INSERT INTO Invoices (Invoice_total, Invoice_customer_id)
VALUES (1000, 1), (50, 1), (200, 2);
```

Figure 14 : Insertion de lignes dans la table Invoices sans spécifier de date.

Cependant, il faudra veiller à ce que tous les champs obligatoires reçoivent bien une valeur ou que la valeur d'une clé étrangère existe déjà dans la table référencée.

Tentative d'ajout d'une ligne sans avoir spécifié de valeur pour Invoice_total qui est obligatoire :

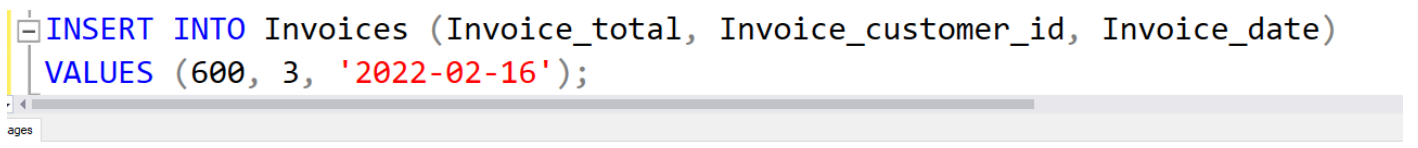


```
INSERT INTO Invoices (Invoice_customer_id, Invoice_date)
VALUES (2, '2022-02-16');
```

Msg 515, Level 16, State 2, Line 51
Cannot insert the value NULL into column 'Invoice_total', table 'MaBoutique'

Figure 15 : Insertion de ligne avec champ obligatoire manquant.

Tentative d'ajout d'une ligne dans Invoices alors qu'aucun Client n'existe avec un Customer_id de 3 :



```
INSERT INTO Invoices (Invoice_total, Invoice_customer_id, Invoice_date)
VALUES (600, 3, '2022-02-16');
```

Msg 547, Level 16, State 0, Line 48
The INSERT statement conflicted with the FOREIGN KEY constraint "FK_Invoices_Invoice"

Figure 16 : Insertion d'une ligne avec une valeur de clé étrangère qui n'existe pas dans la table référencée.

CREER DES VUES

Les vues sont des tables virtuelles dont le contenu est défini par une requête. Une vue peut contenir des données provenant de plusieurs tables de votre base de données. Les vues permettent d'avoir des perceptions adaptées à chaque utilisateur de votre base de données mais aussi de sécuriser votre base en autorisant vos utilisateurs à accéder aux données uniquement via ces vues sans leur accorder un accès direct aux tables.

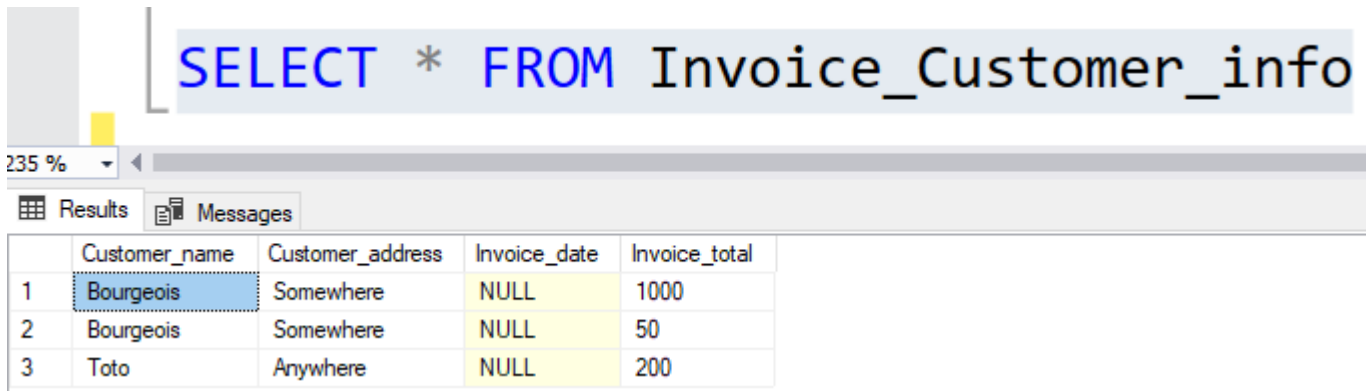
Pour créer une vue, il suffit d'utiliser **CREATE VIEW [Nom_de_la_vue] AS [Requête_de_sélection]**.

```
CREATE VIEW Invoice_Customer_info
AS
```

```
SELECT C.Customer_name, C.Customer_address, I.Invoice_date, I.Invoice_total
FROM Customers AS C INNER JOIN Invoices AS I
ON C.Customer_id = I.Invoice_customer_id
```

Figure 17 : Création d'une nouvelle vue.

Une fois la vue créée, on peut l'afficher facilement avec un **SELECT * FROM [Nom_de_la_vue]** :



The screenshot shows a SQL query editor with the query `SELECT * FROM Invoice_Customer_info` entered. Below the query, there are tabs for 'Results' and 'Messages'. The 'Results' tab is active, displaying a table with the following data:

	Customer_name	Customer_address	Invoice_date	Invoice_total
1	Bourgeois	Somewhere	NULL	1000
2	Bourgeois	Somewhere	NULL	50
3	Toto	Anywhere	NULL	200

Figure 18 : Sélection de tous les champs de la vue créée.

Maintenant que vous savez créer une vue, il est temps de passer à la sélection de données.

SELECTIONNER DES DONNEES

La sélection de données permet de retrouver et d'afficher des lignes de votre base de données d'une ou plusieurs tables. La sélection utilise le mot clé **SELECT**. Vous pouvez trouver [une syntaxe simplifiée et des exemples de SELECT ici](#).

SELECT

Pour un SELECT, il faudra spécifier les colonnes à afficher et la table visée :

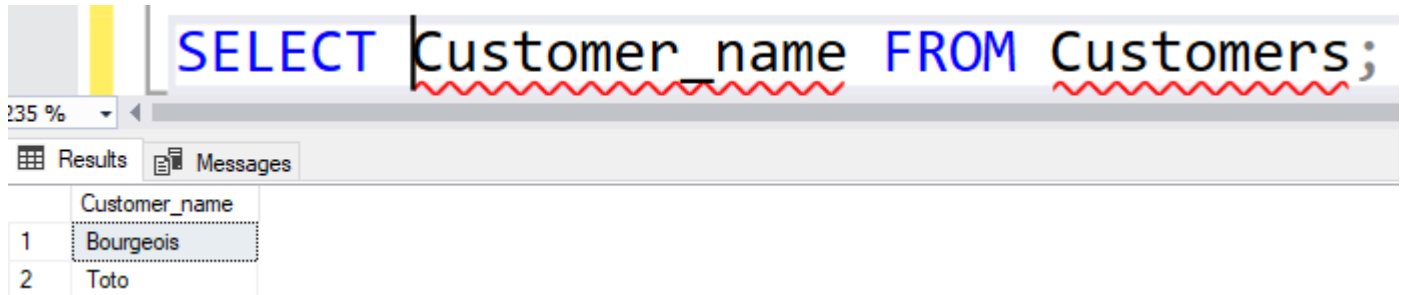


Figure 19 : Sélection de toutes les valeurs de la colonne Customer_name venant de la table Customers.

Il est possible de spécifier au niveau des colonnes de quelle table vient la colonne. Cependant il est obligatoire de spécifier le FROM [Nom_de_la_table].

```
SELECT Customer_name FROM Customers;
SELECT Customers.Customer_name FROM Customers;
```

Figure 20 : Effet identique pour les deux SELECT.

On peut utiliser le symbole * pour spécifier que l'on souhaite voir la totalité des colonnes trouvées dans la table. Cela se fait uniquement à titre d'exemple dans ce cours et pour déboguer la base de données. **Mais ce symbole ne doit jamais figurer dans une requête utilisée par une application !**

ALIAS : AS

Il est possible d'utiliser des alias pour les noms de vos tables et de vos colonnes en utilisant **AS [Nom_alias]**. Cela permet de ne pas avoir à réécrire le nom complet de vos tables :

```
SELECT C.Customer_name FROM Customers AS C;
```

Figure 21 : Utilisation de l'alias C pour ne pas avoir besoin d'écrire "Customers".

Cela changera le nom de la colonne à l'affichage :

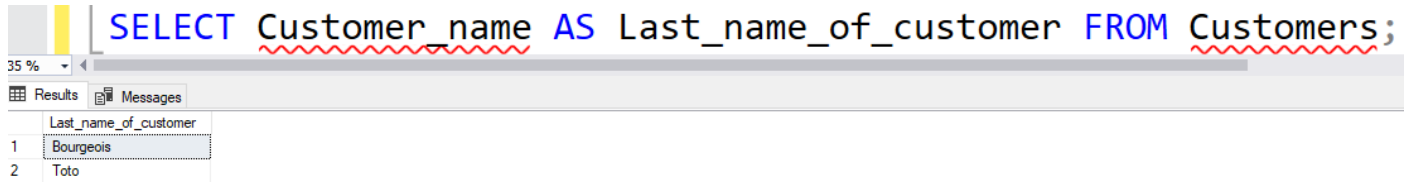


Figure 22 : Ajout d'un alias sur le nom de la colonne Customer_name.

RETOUR A LA LIGNE

Jusqu'à présent, les SELECT se sont fait sur des requêtes simples et on peut les lire assez facilement sur une seule ligne. Cependant, il est plus facile de lire une requête dont les éléments sont espacés par des sauts de ligne. On comprend ainsi mieux chaque étape de la requête.

```
SELECT C.Customer_name AS The_name,
       C.Customer_address AS The_address,
       C.Customer_id AS The_id
FROM Customers AS C;
```

Figure 23 : Requête avec sauts de lignes pour bien voir les colonnes à afficher et la table visée.

ORDONNER LES DONNEES : ORDER BY

Lors d'un SELECT, les lignes s'affichent dans l'ordre croissant selon la clé primaire de vos données. Ajoutons deux nouvelles personnes avec des noms déjà existants dans notre table Customers :

```
INSERT INTO Customers (Customer_name, Customer_address, Customer_age)
VALUES
('Toto', 'MULHOUSE', NULL),
('Bourgeois', 'Brunstatt', 22);
```

Figure 24 : Insertion de 2 clés supplémentaires.

Avec le SELECT précédent, les données sont affichées ainsi :

	The_name	The_address	The_id
1	Bourgeois	Somewhere	1
2	Toto	Anywhere	2
3	Toto	MULHOUSE	3
4	Bourgeois	Brunstatt	4

Figure 25 : Nouvel affichage avec ordre basique.

Vous pouvez choisir de changer cet affichage avec **ORDER BY [Nom_colonne] [Type_ordre]**. Cela permet d'afficher les données selon l'ordre croissant (**ASC**) ou décroissant (**DESC**) sur une ou plusieurs colonnes :

```

SELECT C.Customer_name AS The_name,
       C.Customer_address AS The_address,
       C.Customer_id AS The_id
FROM Customers AS C
ORDER BY The_name DESC, Customer_age ASC;

```

Figure 26 : Sélection des lignes selon l'ordre décroissant du nom du client puis l'âge croissant du client.

	The_name	The_address	The_id
1	Toto	MULHOUSE	3
2	Toto	Anywhere	2
3	Bourgeois	Brunstatt	4
4	Bourgeois	Somewhere	1

Figure 27 : Résultat du SELECT précédent.

SELECTION DE LIGNE SELON DES VALEURS : WHERE

Souvent, il est nécessaire d'apporter des spécifications sur les valeurs recherchées plutôt que de toujours prendre toutes les lignes d'une table. C'est là qu'intervient le **WHERE [Condition]** :

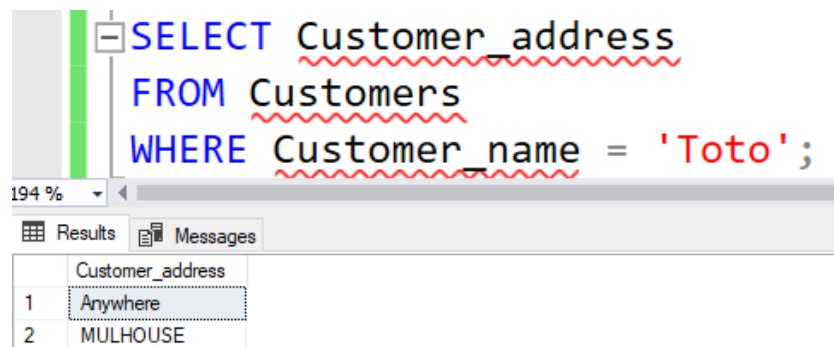


Figure 28 : Sélection des adresses des clients dont le nom du client est "Toto".

AND / OR / NOT

Comme en algèbre de Bool, il est possible de faire des conditions combinées. On peut utiliser AND comme un et logique, OR comme un ou logique et NOT comme un non logique.

LIKE

Permet de spécifier un modèle de chaîne de caractère à respecter pour qu'une condition soit validée. On pourrait associer le fonctionnement d'un like comme le fonctionnement d'une expression régulière.

Un LIKE peut contenir plusieurs caractères spéciaux. Le '%' est un peu comme un joker qui permet de remplacer n'importe quels caractères aucune ou plusieurs fois, le '_' remplace un seul caractère quel qu'il soit, les caractères compris entre '[' et ']' sont des caractères possibles que l'on attend à cet emplacement donc 'm[ao]t' peut correspondre avec 'mat' et 'mot'.

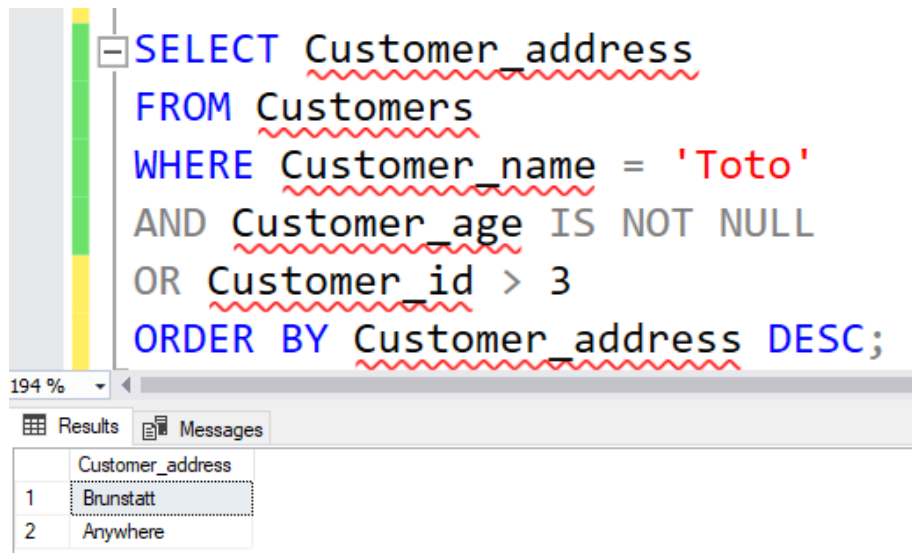


Figure 29 : Utilisation de WHERE avec de multiples conditions.

EFFECTUER DES CALCULS

Il est possible d'effectuer des calculs sur une colonne pour avoir de nouveaux résultats :

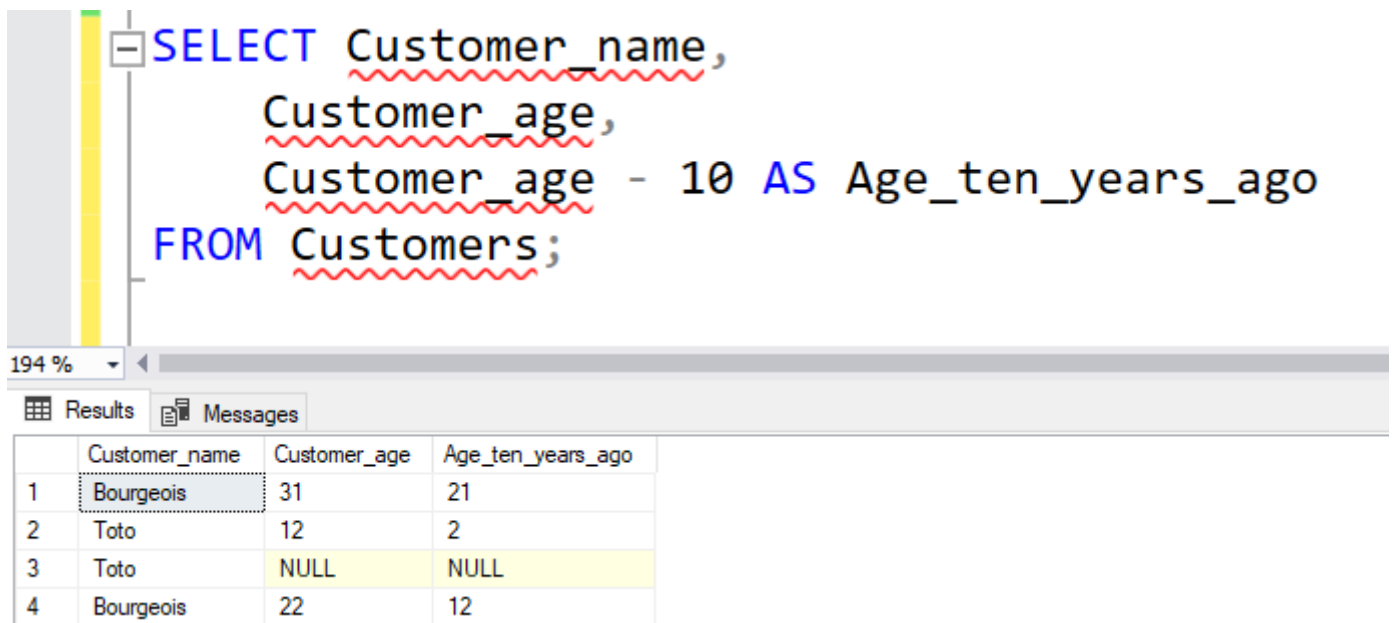


Figure 30 : Calcul de l'âge des clients il y a 10 ans.

Il est possible de faire plusieurs types de calculs différents : [des basiques](#) et [des plus compliqués](#).

GROUPER UNE REQUETE SELON LES DIFFERENTES VALEURS D'UNE COLONNE : [GROUP BY](#)

Parfois, on voudra récupérer le résultat de notre requête en réunissant nos lignes dans une seule ligne selon les valeurs uniques pour une colonne. On dit que l'on fait une agrégation des lignes. Pour cela, on utilisera **GROUP BY [Nom_colonne]** :

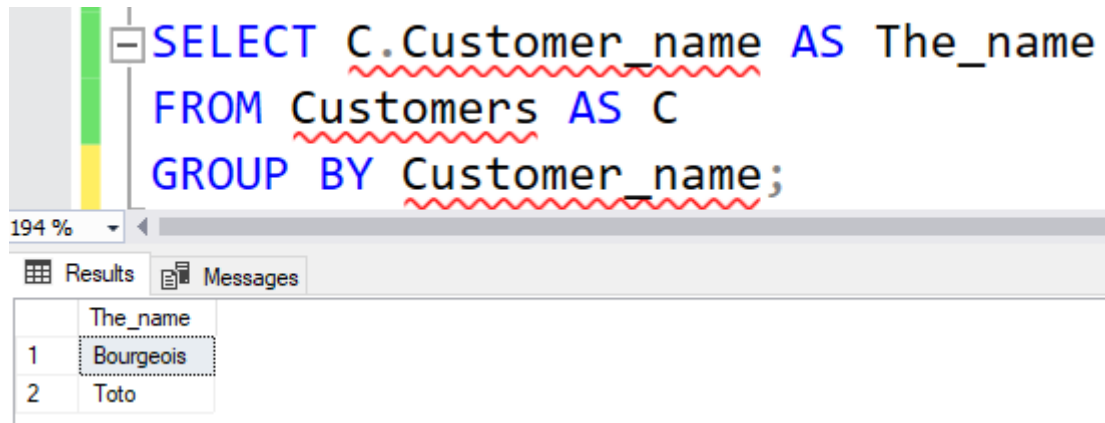


Figure 31 : GROUP BY sur les noms de clients.

Un GROUP BY peut se faire sur plusieurs colonnes à la fois. Il y a un bon exemple dans la documentation officielle où on regroupe par pays et région des ventes pour calculer la somme des ventes dans ces régions.

Les GROUP BY sont notamment utiles pour des calculs de sommes ou de moyennes. Mais aussi un maximum, un minimum ou encore compter le nombre de lignes dans chaque groupe.

SOUS-REQUETE

Une sous-requête est une requête utilisée à l'intérieure d'une autre requête. Généralement utilisées dans une condition au niveau d'un WHERE pour comparer des valeurs précises dans notre requête principale :

```
SELECT Customer_name, Customer_address
FROM Customers
WHERE Customer_id IN (SELECT Customer_id FROM Invoices);
```

HAVING

Indique un critère de recherche pour un groupe ou une fonction d'agrégation. Il permet de de spécifier par exemple que l'on désire sélectionner uniquement les groupes dont le nombre de lignes (COUNT(*)) est supérieur à 2 :

```
Select Customer_name
FROM Customers
GROUP BY Customer_name
HAVING COUNT(*) > 2;
```


LES JOINTURES : [JOIN](#)

Les jointures font partie des SELECT mais méritent un chapitre à elles seules. Le principe d'une jointure est, comme son nom l'indique, de lier ensemble des tables. Une jointure va lier entre elles les ligne d'une table A avec les lignes d'une table B. Ces liens se font à partir d'une correspondance de valeurs entre une colonne de la table A et d'une colonne de la table B (Excepté pour la jointure de type CROSS JOIN). De manière générale, cela se fait à partir couples clé primaire/clé étrangère entre les tables A et B. Mais une jointure peut tout à fait se faire sur d'autres colonnes que les clés des tables. Une jointure peut se faire sur une seule et même table mais il faudra utiliser des alias pour bien différencier les parties du résultat de la jointure. Enfin, on peut enchaîner plusieurs jointures les unes derrière les autres.

Il existe cinq jointures différentes :

- INNER JOIN
- LEFT OUTER JOIN
- RIGHT OUTER JOIN
- FULL OUTER JOIN
- CROSS JOIN

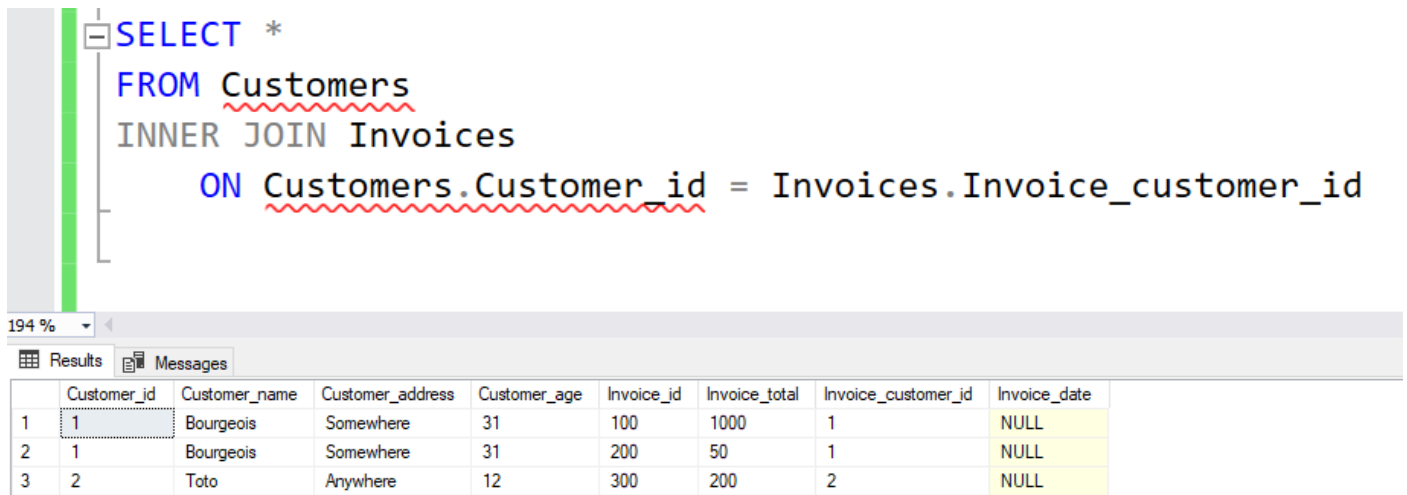
Même si je vais essayer de vous expliquer la différence entre chacune de ces jointures, je vous conseille de regarder ce [lien pour en comprendre rapidement les différences](#).

Une jointure se fait en suivant le schéma suivant après FROM [Nom_de_tableA] :

[Type_jointure] JOIN [Nom_tableB] ON [Nom_de_tableA].[Colonne_tableA] = [Nom_de_tableB].[Colonne_tableB]

INNER JOIN

La jointure interne entre deux tables est l'intersection de ces deux tables. Cela signifie que la jointure ne va récupérer dans la table A que les lignes qui ont un équivalent dans la table B et inversement.



```
SELECT *
FROM Customers
INNER JOIN Invoices
ON Customers.Customer_id = Invoices.Invoice_customer_id
```

	Customer_id	Customer_name	Customer_address	Customer_age	Invoice_id	Invoice_total	Invoice_customer_id	Invoice_date
1	1	Bourgeois	Somewhere	31	100	1000	1	NULL
2	1	Bourgeois	Somewhere	31	200	50	1	NULL
3	2	Toto	Anywhere	12	300	200	2	NULL

Figure 32 : INNER JOIN entre Customers et Invoices.

OUTER JOIN

Les jointures externes vont faire une union entre les tables. Elles vont récupérer toutes les lignes dans l'une des tables, dans l'autre table ou dans les deux tables concernées par la jointure et ce qu'il y ait ou non des équivalences entre elles.

LEFT OUTER JOIN

La jointure gauche affichera toutes les lignes de la table A même s'il n'y a aucune correspondance dans la table B. S'il n'y a aucune valeur dans la table B, les colonnes seront remplies par la valeur NULL.

```
SELECT *
FROM Customers
LEFT OUTER JOIN Invoices
ON Customers.Customer_id = Invoices.Invoice_customer_id
```

	Customer_id	Customer_name	Customer_address	Customer_age	Invoice_id	Invoice_total	Invoice_customer_id	Invoice_date
1	1	Bourgeois	Somewhere	31	100	1000	1	NULL
2	1	Bourgeois	Somewhere	31	200	50	1	NULL
3	2	Toto	Anywhere	12	300	200	2	NULL
4	3	Toto	MULHOUSE	NULL	NULL	NULL	NULL	NULL
5	4	Bourgeois	Brunstatt	22	NULL	NULL	NULL	NULL

Figure 33 : LEFT JOIN entre Customers et Invoices.

On voit bien dans cet exemple que les lignes 4 et 5 n'ont aucune donnée sur des colonnes de la table Invoices.

Il est possible d'écrire LEFT JOIN plutôt que d'écrire LEFT OUTER JOIN.

RIGHT OUTER JOIN

La jointure droite fonctionne de la même manière qu'une jointure gauche à la différence qu'elle va récupérer toutes les lignes de la table B qu'elles aient ou non des occurrences dans la table A.

```
SELECT *
FROM Customers
RIGHT OUTER JOIN Invoices
ON Customers.Customer_id = Invoices.Invoice_customer_id;
```

	Customer_id	Customer_name	Customer_address	Customer_age	Invoice_id	Invoice_total	Invoice_customer_id	Invoice_date
1	1	Bourgeois	Somewhere	31	100	1000	1	NULL
2	1	Bourgeois	Somewhere	31	200	50	1	NULL
3	2	Toto	Anywhere	12	300	200	2	NULL

Figure 34 : RIGHT JOINT entre Customers et Invoices.

Dans cet exemple, la jointure externe droite est identique à la jointure interne car la colonne Invoice_customer_id dans la table Invoices est obligatoire. Donc pour toute facture il y a un client. Cependant on pourrait inverser l'ordre des tables dans la requête pour avoir le même résultat qu'avec une jointure externe gauche :

```

SELECT Customers.*, Invoices.*
FROM Invoices
RIGHT OUTER JOIN Customers
ON Customers.Customer_id = Invoices.Invoice_customer_id;

```

194 %

Results Messages

	Customer_id	Customer_name	Customer_address	Customer_age	Invoice_id	Invoice_total	Invoice_customer_id	Invoice_date
1	1	Bourgeois	Somewhere	31	100	1000	1	NULL
2	1	Bourgeois	Somewhere	31	200	50	1	NULL
3	2	Toto	Anywhere	12	300	200	2	NULL
4	3	Toto	MULHOUSE	NULL	NULL	NULL	NULL	NULL
5	4	Bourgeois	Brunstatt	22	NULL	NULL	NULL	NULL

Figure 35 : RIGHT JOIN entre Invoices et Customers.

Comme la jointure gauche, on pourrait écrire uniquement RIGHT JOIN plutôt que RIGHT OUTER JOIN pour avoir le même résultat.

FULL OUTER JOIN

La jointure externe totale va prendre toutes les lignes des deux tables pour toutes les afficher. Qu'il y ait des correspondances ou non. Dans notre cas, cela donnera le même résultat que la première jointure externe gauche puisque qu'une facture ne peut être définie sans identifiant de client :

```

SELECT Customers.*, Invoices.*
FROM Invoices
FULL OUTER JOIN Customers
ON Customers.Customer_id = Invoices.Invoice_customer_id;

```

194 %

Results Messages

	Customer_id	Customer_name	Customer_address	Customer_age	Invoice_id	Invoice_total	Invoice_customer_id	Invoice_date
1	1	Bourgeois	Somewhere	31	100	1000	1	NULL
2	1	Bourgeois	Somewhere	31	200	50	1	NULL
3	2	Toto	Anywhere	12	300	200	2	NULL
4	3	Toto	MULHOUSE	NULL	NULL	NULL	NULL	NULL
5	4	Bourgeois	Brunstatt	22	NULL	NULL	NULL	NULL

Figure 36 : FULL OUTER JOIN entre Invoices et Customers.

Encore une fois, on pourrait écrire FULL JOIN plutôt que FULL OUTER JOIN.

CROSS JOIN

La jointure croisée va créer toutes les combinaisons possibles entre les deux tables. Dans notre exemple de clients et facture, cela n'a pas de logique mais nous pouvons tout de même le faire :

```

SELECT Customers.*, Invoices.*
FROM Invoices
CROSS JOIN Customers
ORDER BY Customers.Customer_age ASC, Invoices.Invoice_id;

```

194 %

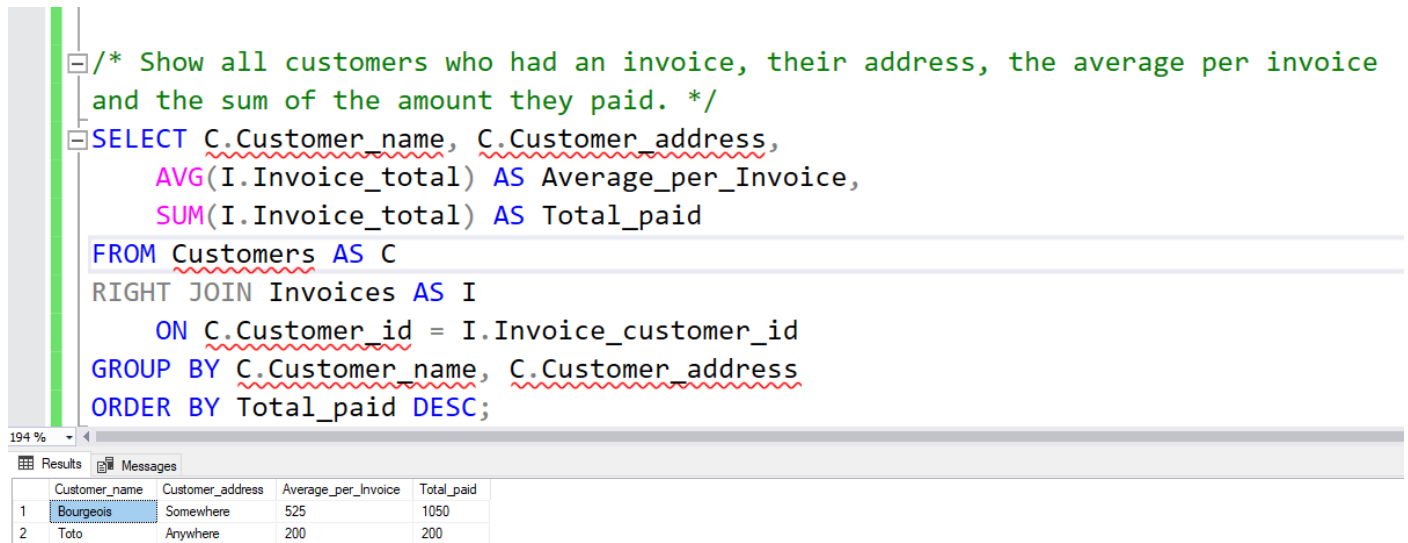
Results Messages

	Customer_id	Customer_name	Customer_address	Customer_age	Invoice_id	Invoice_total	Invoice_customer_id	Invoice_date
1	3	Toto	MULHOUSE	NULL	100	1000	1	NULL
2	3	Toto	MULHOUSE	NULL	200	50	1	NULL
3	3	Toto	MULHOUSE	NULL	300	200	2	NULL
4	2	Toto	Anywhere	12	100	1000	1	NULL
5	2	Toto	Anywhere	12	200	50	1	NULL
6	2	Toto	Anywhere	12	300	200	2	NULL
7	4	Bourgeois	Brunstatt	22	100	1000	1	NULL
8	4	Bourgeois	Brunstatt	22	200	50	1	NULL
9	4	Bourgeois	Brunstatt	22	300	200	2	NULL
10	1	Bourgeois	Somewhere	31	100	1000	1	NULL
11	1	Bourgeois	Somewhere	31	200	50	1	NULL
12	1	Bourgeois	Somewhere	31	300	200	2	NULL

Figure 37 : CROSS JOIN entre Invoices et Customers.

EXEMPLE DE REQUETE COMPLEXE

Voici une requête qui mélange un peu tout ce qui a été vu dans la section SELECT :



The screenshot shows a SQL query editor with a query that selects customer information and invoice statistics. The query is as follows:

```

/* Show all customers who had an invoice, their address, the average per invoice
and the sum of the amount they paid. */
SELECT C.Customer_name, C.Customer_address,
       AVG(I.Invoice_total) AS Average_per_Invoice,
       SUM(I.Invoice_total) AS Total_paid
FROM Customers AS C
RIGHT JOIN Invoices AS I
      ON C.Customer_id = I.Invoice_customer_id
GROUP BY C.Customer_name, C.Customer_address
ORDER BY Total_paid DESC;

```

Below the query, the results are displayed in a table with the following columns: Customer_name, Customer_address, Average_per_Invoice, and Total_paid. The results are ordered by Total_paid in descending order.

	Customer_name	Customer_address	Average_per_Invoice	Total_paid
1	Bourgeois	Somewhere	525	1050
2	Toto	Anywhere	200	200

Figure 38 : exemple de requête.

--- FIN DU DOCUMENT ---