

# SQL Tutorial



# Sommaire :

<b>SQL DATA TYPES &amp; COMMENTS</b>	<b>4</b>
Conclusion	8
<b>SQL OPERATORS &amp; SQL NULL FUNCTIONS</b>	<b>10</b>
<b>SQL OPERATORS :</b>	<b>10</b>
Opérateurs arithmétiques	10
Opérateurs au niveau du bit	11
Opérateurs de comparaison	12
Opérateurs composés :	14
Opérateurs logiques	14
SQL null functions :	18
fonction SQL ISNULL (), IFNULL () et COALESCE ()	18
isnull():	18
IFNULL():	19
COALESCE():	19
<b>SQL WHERE &amp; SQL AND OR NOT</b>	<b>22</b>
Les opérateurs SQL AND, OR et NOT	22
Conclusion	24
<b>SQL ORDER BY &amp; NULL VALUES</b>	<b>25</b>
<b>SQL SELECT TOP, SQL ANY, ALL -</b>	<b>29</b>
La clause SQL SELECT TOP	29
<b>SQL DATES &amp; EXISTS -</b>	<b>31</b>
<b>SQL &amp; MIN &amp; MAX, COUNT, AVG, SUM</b>	<b>45</b>
<b>SQL HAVING</b>	<b>47</b>
<b>GROUP BY</b>	<b>48</b>
<b>SQL LIKE &amp; UNION</b>	<b>50</b>
L'opérateur " LIKE "	50
<b>SQL WILDCARDS</b>	<b>53</b>
<b>SQL IN &amp; BETWEEN</b>	<b>55</b>
<b>SQL SELF JOIN &amp; FULL JOIN</b>	<b>56</b>
<b>SQL ALIAS &amp; JOINS</b>	<b>57</b>
<b>SQL CASE &amp; CREATE-AS SELECT</b>	<b>60</b>
<b>INNER JOIN &amp; LEFT JOIN &amp; RIGHT JOIN</b>	<b>64</b>

# SQL DATA TYPES & COMMENTS

@David

## SQL data types :

Les types de données SQL sont utilisés pour définir le type de données d'une colonne de la table. Voici quelques types de données couramment utilisés en SQL, avec des exemples :

- **INT** : Il s'agit d'un type de données qui stocke des valeurs entières. Par exemple, si vous voulez récupérer l'id des clients, utilisez **INT** pour stocker ces données. Exemple :**SELECT \* FROM customer WHERE customer\_id = 123;**

```
1 •  SELECT * FROM customer WHERE customer_id = 123;
2
3
```

< Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content: □

customer_id	store_id	first_name	last_name	email	address_id	active	create_date	last_update
123	2	SHANNON	FREEMAN	SHANNON.FREEMAN@sakilacustomer.org	127	1	2006-02-14 22:04:36	2006-02-15 04:57:20
*	HULL	HULL	HULL	HULL	HULL	HULL	HULL	HULL

Cette requête sélectionne toutes les colonnes de la table "customer" où l'ID de client est égal à 123. Vous pouvez remplacer "123" par n'importe quelle autre valeur d'ID de client pour rechercher des informations sur un client différent.

- **VARCHAR** : Il s'agit d'un type de données qui stocke des chaînes de caractères de longueur variable. Par exemple, si vous avez une colonne qui stocke les noms des films, vous pouvez utiliser **VARCHAR** trouver des films qui contiennent le mot "action" dans leur titre . Exemple :**SELECT \* FROM film WHERE title LIKE '%action%';**

```
1 •  SELECT * FROM film WHERE title LIKE '%action%';
2
3
```

< Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content: □

film_id	title	description	release_year	language_id	original_language_id	rental_duration	rental_rate	length	replacement
45	ATTRACTION NEWTON	A Astounding Panorama of a Composer And a F...	2006	1	HULL	5	4.99	83	14.99
287	ENTRAPMENT SATISFACTION	A Thoughtful Panorama of a Hunter And a Teac...	2006	1	HULL	5	0.99	176	19.99
763	SATISFACTION CONFIDENTIAL	A Lackluster Yarn of a Dentist And a Butler wh...	2006	1	HULL	3	4.99	75	26.99
881	TEMPLE ATTRACTION	A Action-Packed Saga of a Forensic Psychologis...	2006	1	HULL	5	4.99	71	13.99
*	HULL	HULL	HULL	HULL	HULL	HULL	HULL	HULL	HULL

Cette requête retournera tous les titres de films qui contiennent le mot "action", quels que soient les caractères qui le précèdent ou le suivent.

- **TEXT** : Le type de données **TEXT** en SQL est utilisé pour stocker des chaînes de caractères de longueur variable. Il peut stocker une grande quantité de texte, allant de quelques octets à plusieurs gigaoctets.

L'avantage de ce type de données est qu'il n'y a pas de limite de taille prédéfinie pour la longueur du texte à stocker, contrairement à d'autres types de données tels que "varchar" qui ont une limite de taille fixe. Cependant, le stockage de grandes quantités de données peut affecter les performances de la base de données.

Le type de données **TEXT** est souvent utilisé pour stocker des informations telles que des descriptions, des commentaires ou des notes dans une base de données. Il peut également être utilisé pour stocker des fichiers texte complets, tels que des documents ou des rapports.

- **DATE** : Il s'agit d'un type de données qui stocke des dates. Par exemple, si vous avez une colonne qui stocke la date de naissance des clients, vous pouvez utiliser DATE pour stocker ces données. [Exemple proposé par mathias plus bas !](#)
- **DECIMAL** : Il s'agit d'un type de données qui stocke des nombres décimaux précis. Par exemple, si vous avez une colonne qui stocke la durée des films, vous pouvez utiliser **DECIMAL** pour stocker ces données. Exemple :**SELECT \* FROM film WHERE length > 100.2;**

```
1 •  SELECT * FROM film WHERE length > 100.2;
2
3
```

film_id	title	description	release_year	language_id	original_language_id	rental_duration	rental_rate	length	replacement_cost
74	BIRCH ANTITRUST	A Fanciful Panorama of a Husband And a Pione...	2006	1	NULL	4	4.99	162	18.99
75	BIRD INDEPENDENCE	A Thrilling Documentary of a Car And a Student ...	2006	1	NULL	6	4.99	163	14.99
76	BIRDCAGE CASPER	A Fast-Paced Saga of a Frisbee And a Astronau...	2006	1	NULL	4	0.99	103	23.99
79	BLADE POLISH	A Thoughtful Character Study of a Frisbee And ...	2006	1	NULL	5	0.99	114	10.99
80	BLANKET BEVERLY	A Emotional Documentary of a Student And a Gi...	2006	1	NULL	7	2.99	148	21.99

Cette requête sélectionne toutes les colonnes de la table "film" où la durée du film (stockée dans la colonne "length") est supérieure à 100.2 minutes. Le nombre "100.2" est un exemple de valeur décimale pour la durée en minutes.

- **BOOLEAN** : Il s'agit d'un type de données qui stocke des valeurs booléennes (vrai ou faux). Par exemple, on va chercher si des comptes sont inactifs dans la base de données sakila . Exemple :**SELECT \* FROM customer WHERE active = 0;**

The screenshot shows a MySQL Workbench interface. At the top, a query window displays:

```
1 •  SELECT * FROM customer WHERE active = 0;
```

Below the query window is a result grid titled "Result Grid". The grid has the following columns:

	customer_id	store_id	first_name	last_name	email	address_id	active	create_date	last_update
▶	16	2	SANDRA	MARTIN	SANDRA.MARTIN@sakilacustomer.org	20	0	2006-02-14 22:04:36	2006-02-15 04:57:20
	64	2	JUDITH	COX	JUDITH.COX@sakilacustomer.org	68	0	2006-02-14 22:04:36	2006-02-15 04:57:20
	124	1	SHEILA	WELLS	SHEILA.WELLS@sakilacustomer.org	128	0	2006-02-14 22:04:36	2006-02-15 04:57:20
	169	2	ERICA	MATTHEWS	ERICA.MATTHEWS@sakilacustomer.org	173	0	2006-02-14 22:04:36	2006-02-15 04:57:20
	241	2	HEIDI	LARSON	HEIDI.LARSON@sakilacustomer.org	245	0	2006-02-14 22:04:36	2006-02-15 04:57:20
	271	1	PENNY	NEAL	PENNY.NEAL@sakilacustomer.org	276	0	2006-02-14 22:04:36	2006-02-15 04:57:20
	315	2	KENNETH	GOODEN	KENNETH.GOODEN@sakilacustomer.org	320	0	2006-02-14 22:04:37	2006-02-15 04:57:20
	368	1	HARRY	ARCE	HARRY.ARCE@sakilacustomer.org	373	0	2006-02-14 22:04:37	2006-02-15 04:57:20
	406	1	NATHAN	RUNYON	NATHAN.RUNYON@sakilacustomer.org	411	0	2006-02-14 22:04:37	2006-02-15 04:57:20
	446	2	THEODORE	CULP	THEODORE.CULP@sakilacustomer.org	451	0	2006-02-14 22:04:37	2006-02-15 04:57:20
	482	1	MAURICE	CRAWLEY	MAURICE.CRAWLEY@sakilacustomer.org	487	0	2006-02-14 22:04:37	2006-02-15 04:57:20

To

- **BLOB** : Il s'agit d'un type de données qui stocke de gros objets binaires tels que des images ou des fichiers. Par exemple, si vous avez une colonne qui stocke des images de film, vous pouvez utiliser **BLOB** pour stocker ces données.

<span style="color: red;">✖</span> 124 10:54:33 SELECT film_id, title, length, image FROM film WHERE image IS NOT NULL LIMIT 0, 1000	Error Code: 1054. Unknown column 'image' in field list'
--	---

- **CHAR** : Le type de données "char" en SQL est utilisé pour stocker des chaînes de caractères de longueur fixe. La longueur de la chaîne de caractères doit être spécifiée lors de la définition de la colonne qui utilise le type **CHAR**. Par exemple, "char(10)" permet de stocker une chaîne de caractères de longueur maximale de 10 caractères. Contrairement au type de données "varchar" qui permet de stocker des chaînes de caractères de longueur variable, le type **CHAR** alloue de l'espace pour la longueur maximale de la chaîne de caractères spécifiée, même si la chaîne de caractères stockée est plus courte.

Par exemple, si vous stockez une chaîne de caractères de 5 caractères dans une colonne "char(10)", 5 caractères seront stockés et les 5 caractères restants seront remplis avec des espaces. Le type "char" peut être utilisé pour stocker des chaînes de caractères qui ont une longueur fixe, telles que des codes postaux, des numéros de téléphone ou des identifiants d'utilisateur.

Un exemple : Supposons que vous souhaitez trouver tous les acteurs dont le nom commence par la lettre "A". Voici la requête que vous pouvez utiliser : **SELECT \* FROM actor WHERE first\_name LIKE 'A%';**

```

1 •  SELECT * FROM actor WHERE first_name LIKE 'A%';
2
3
4
```

**Result Grid** | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content: |

	actor_id	first_name	last_name	last_update
▶	29	ALEC	WAYNE	2006-02-15 04:34:33
	34	AUDREY	OLIVIER	2006-02-15 04:34:33
	49	ANNE	CRONYN	2006-02-15 04:34:33
	65	ANGELA	HUDSON	2006-02-15 04:34:33
	71	ADAM	GRANT	2006-02-15 04:34:33
	76	ANGELINA	ASTAIRE	2006-02-15 04:34:33
	125	ALBERT	NOLTE	2006-02-15 04:34:33
	132	ADAM	HOPPER	2006-02-15 04:34:33
	144	ANGELA	MONTGOMERY	2006-02-15 04:34:33

Cette requête sélectionne toutes les colonnes de la table "actor" où la valeur de la colonne "first\_name" commence par la lettre "A". La colonne "first\_name" est de type CHAR(45), ce qui signifie qu'elle peut stocker des chaînes de caractères jusqu'à 45 caractères de longueur fixe.

The screenshot shows the 'actor' table structure in MySQL Workbench. The columns are:

- actor\_id: SMALLINT
- first\_name: VARCHAR(45)
- last\_name: VARCHAR(45)
- last\_update: TIMESTAMP

- **FLOAT** : Le type de données "float" en SQL est un type de données numériques à virgule flottante. Il est utilisé pour stocker des nombres décimaux avec une précision variable. Les valeurs "float" sont stockées [sous forme de mantisse et d'exposant](#).

Le nombre de bits utilisés pour stocker les valeurs "float" dépend du système de gestion de base de données utilisé. En général, il est recommandé d'utiliser le type "float" pour les valeurs qui nécessitent une grande précision et une plage de valeurs très étendue.

Cependant, il est important de noter que les types de données "float" peuvent présenter des inexactitudes en raison de la façon dont les nombres décimaux sont stockés. Par conséquent, il est recommandé d'utiliser le type de données "numeric" ou "decimal" pour les calculs nécessitant une précision maximale.

En résumé, le type de données "float" est un type de données numériques à virgule flottante utilisé pour stocker des nombres décimaux avec une précision variable. Il est recommandé de l'utiliser pour les valeurs qui nécessitent une grande précision et une plage de valeurs très étendue, mais il peut présenter des inexactitudes en raison de la façon dont les nombres sont stockés.

Un exemple : on souhaite trouver tous les paiements dont le montant est supérieur à 8.5 euros. Voici la requête que vous pouvez utiliser : **SELECT \* FROM payment WHERE amount > 8.5 ;**

```

1 •   SELECT * FROM payment WHERE amount > 8.5;
2
3
4
```

	payment_id	customer_id	staff_id	rental_id	amount	payment_date	last_update
▶	5	1	2	1476	9.99	2005-06-15 21:08:46	2006-02-15 22:12:30
	44	2	2	9236	10.99	2005-07-30 13:47:43	2006-02-15 22:12:30
	62	3	1	1546	8.99	2005-06-16 01:34:05	2006-02-15 22:12:30
	69	3	2	7503	10.99	2005-07-27 20:23:12	2006-02-15 22:12:30
	81	3	1	13403	8.99	2005-08-19 22:18:07	2006-02-15 22:12:30

Cette requête sélectionne toutes les colonnes de la table "payment" où la valeur de la colonne "amount" est supérieure à 8.5 euros. La colonne "amount" est de type FLOAT, ce qui signifie qu'elle peut stocker des nombres à virgule flottante.

- **DOUBLE** : Le type de données Double en SQL est utilisé pour stocker des nombres décimaux avec une précision supérieure à celle du type de données Float. Double utilise 8 octets pour stocker chaque valeur décimale, ce qui signifie qu'il peut stocker des nombres plus grands avec une plus grande précision que le type de données Float.

En SQL, le type de données Double est souvent utilisé pour stocker des valeurs monétaires, des mesures scientifiques, des coordonnées géographiques et d'autres données qui nécessitent une grande précision décimale.

Il est important de noter que Double est un type de données à virgule flottante, ce qui signifie que les calculs impliquant des nombres à virgule flottante peuvent entraîner des erreurs d'arrondi ou de précision. Il est donc recommandé de faire preuve de prudence lors de l'utilisation de ce type de données pour des calculs critiques ou des comparaisons précises.

## Conclusion

Ces types de données SQL sont utilisés pour définir le type de données qu'une colonne peut contenir. Il est important de choisir le bon type de données pour chaque colonne afin de garantir l'intégrité des données et de maximiser l'efficacité de la base de données.

Il existe également d'autres types de données SQL moins courants tels que les types géométriques, les types de données binaires, les types de données de tableau, etc.

## SQL comments :

Les commentaires SQL sont des annotations ou des notes qui peuvent être ajoutées à un script SQL pour fournir des informations supplémentaires sur le code. Les commentaires SQL ne sont pas exécutés par le système de gestion de base de données et sont ignorés lors de l'exécution du script.

Les commentaires SQL sont utilisés pour expliquer le but et la fonction du code SQL, pour documenter les changements apportés au code, pour indiquer les auteurs du code, pour fournir des instructions sur la façon d'utiliser le code, et pour ajouter toute autre information pertinente.

Les commentaires SQL peuvent être ajoutés en utilisant deux types de syntaxes. Les commentaires de ligne commencent par deux tirets (-), tandis que les commentaires de bloc commencent par /\* et se terminent par \*/. Il est important de noter que les commentaires de bloc ne peuvent pas être imbriqués.

```
1 •   SELECT * FROM payment WHERE amount > 8.5;
2     -- un type de commentaire
3     /* un deuxième
4       type de commentaire*/
```

Exemple :

# SQL OPERATORS & SQL NULL FUNCTIONS

@Anne-Sophie

## SQL OPERATORS :

Un opérateur est un symbole spécifiant une action exécutée sur une ou plusieurs expressions.

Il existe trois types d'expressions correspondant chacun à un type de données de SQL : **arithmétique, chaîne de caractère, date**. A chaque type correspondent des opérateurs et des fonctions spécifiques.

SQL autorise les mélanges de types dans les expressions et effectuera les conversions nécessaires : dans une expression mélangeant dates et chaînes de caractères, les chaînes de caractères seront converties en dates, dans une expression mélangeant nombres et chaînes de caractères, les chaînes de caractères seront converties en nombre.

Il existe 5 types d'opérateurs SQL:

- SQL Arithmetic Operators (opérateurs arithmétiques),
- SQL Bitwise Operators (opérateurs au niveau du bit),
- SQL Comparison Operators (opérateurs de comparaison),
- SQL Compound Operators (opérateurs composés),
- SQL Logical Operators (opérateurs logiques).

### Opérateurs arithmétiques

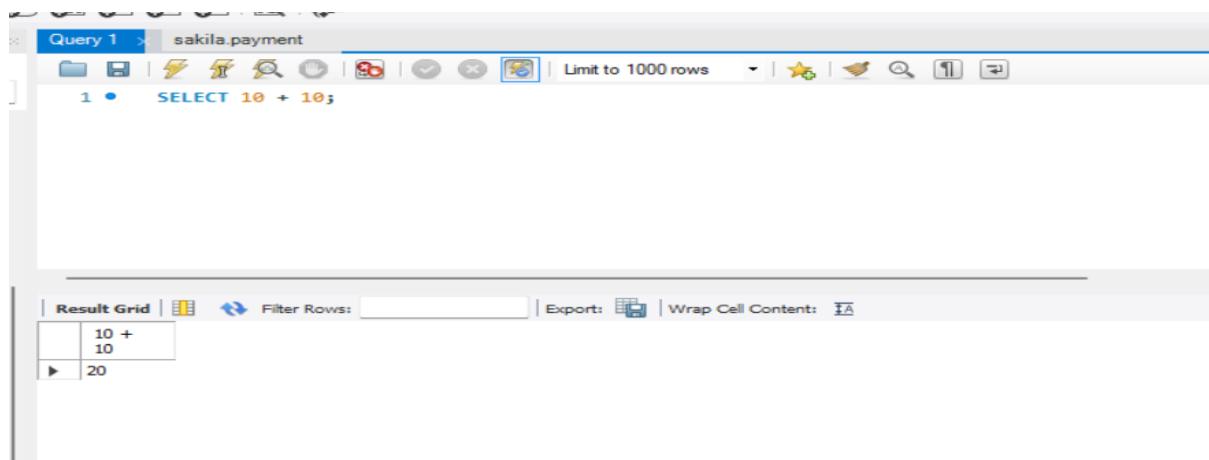
Les opérateurs arithmétiques exécutent des opérations mathématiques sur deux expressions d'un ou plusieurs types de données, à partir de la catégorie de type de données numérique.

Une expression arithmétique peut comporter plusieurs opérateurs. Dans ce cas, le résultat de l'expression peut varier selon l'ordre dans lequel sont effectuées les opérations. Les opérateurs de multiplication et de division sont prioritaires par rapport aux opérateurs d'addition et de soustraction. Des parenthèses peuvent être utilisées pour forcer l'évaluation de l'expression dans un ordre différent de celui découlant de la priorité des opérateurs.

Opérateur	Description
+	Addition

-	Soustraction
*	Multiplication
/	Division
%	Modulo

Exemple:



Exemple :

-> Donner, pour chaque commercial, son revenu (salaire + commission).

```
SELECT nom, salaire+comm
FROM emp
WHERE fonction = 'commercial';
```

### Opérateurs au niveau du bit

Les opérateurs au niveau du bit exécutent des manipulations de bits entre deux expressions de tout type de données de la catégorie entier.

Les opérateurs au niveau du bit permettent de convertir deux valeurs entières en bits binaires, d'effectuer l'opération AND (&), OR (|,^) ou NOT (~) sur chaque bit et de produire un résultat. Ils convertissent ensuite le résultat en entier.

Opérateur	Description
-----------	-------------

AND &	Le symbole & (Bitwise AND) compare chaque bit individuel dans une valeur avec son bit correspondant dans l'autre valeur.
OR   ou ^	Le   (Bitwise OR) effectue une opération OU logique au niveau du bit entre deux valeurs.
NOT ~	L'opérateur au niveau du bit ~ exécute une opération NOT logique au niveau du bit sur cette expression, en évaluant chaque bit. Si l' <i>expression</i> a la valeur 0, les bits du jeu de résultats prennent la valeur 1 ; sinon, le bit résultant est mis à 0. En d'autres termes, les uns sont changés en zéros et les zéros sont changés en uns.

### Opérateurs de comparaison

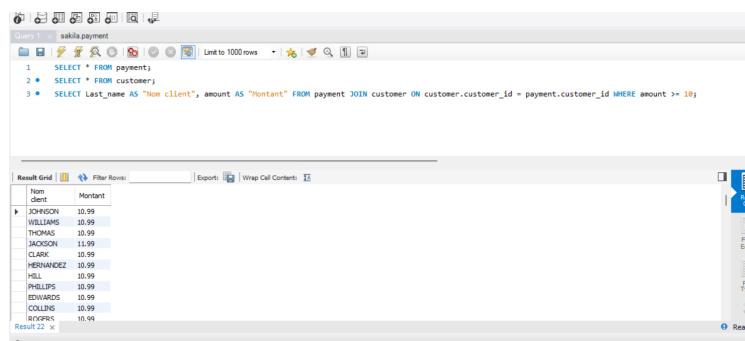
Un opérateur de comparaison est utilisé pour comparer deux valeurs et tester si elles sont identiques.

Opérateur	Description
!=	Pas égal à Le symbole != est utilisé pour filtrer les résultats qui ne correspondent pas à une certaine valeur.
>	Supérieur à) Le symbole > est utilisé pour filtrer les résultats lorsque la valeur d'une colonne est supérieure à la valeur interrogée.
!>	!> (Pas supérieur à) Le symbole !> est utilisé pour filtrer les résultats lorsque la valeur d'une colonne n'est pas supérieure à la valeur interrogée.

<	< (Inférieur à) Le symbole < est utilisé pour filtrer les résultats lorsque la valeur d'une colonne est inférieure à la valeur interrogée.
!<	!< (Pas moins de) Le symbole !< est utilisé pour filtrer les résultats lorsque la valeur d'une colonne n'est pas inférieure à la valeur interrogée.
>=	>= (supérieur ou égal à) Le symbole >= est utilisé pour filtrer les résultats lorsque la valeur d'une colonne est supérieure ou égale à la valeur interrogée.
<=	<= (inférieur ou égal à) Le symbole <= est utilisé pour filtrer les résultats lorsque la valeur d'une colonne est inférieure ou égale à la valeur interrogée.
<>	<> (différent de) Le symbole <> effectue exactement la même opération que le symbole != et est utilisé pour filtrer les résultats qui ne correspondent pas à une certaine valeur.

### Exemple :

Je souhaite connaître les noms des clients ainsi que le montant réglé par celui-ci, en ne prenant que les montants supérieur ou égal à 10:



The screenshot shows a MySQL Workbench interface with two panes. The top pane displays the SQL query:

```

1 • SELECT * FROM payment;
2 • SELECT * FROM customer;
3 • SELECT Last_name AS "Nom client", amount AS "Montant" FROM payment JOIN customer ON customer.customer_id = payment.customer_id WHERE amount >= 10;
  
```

The bottom pane shows the result grid with the following data:

Nom client	Montant
JOHNSON	10.99
WILLIAMS	10.99
THOMAS	10.99
JACKSON	11.99
CLARK	10.99
MARTINEZ	10.99
KING	10.99
PHILLIPS	10.99
EDWARDS	10.99
COLLINS	10.99
KYNGS	10.99

## Opérateurs composés :

Opérateur	Description
<code>+=</code>	<code>+=</code> (Ajouter égal) L'opérateur <code>+=</code> ajoutera une valeur à la valeur d'origine et stockera le résultat dans la valeur d'origine.  Cela peut également être utilisé sur les chaînes.
<code>-=</code>	<code>-=</code> (Soustraire égal) L'opérateur <code>-=</code> soustraira une valeur de la valeur d'origine et stockera le résultat dans la valeur d'origine.
<code>*=</code>	<code>*=</code> (multiplier est égal) L'opérateur <code>*=</code> multipliera une valeur par la valeur d'origine et stockera le résultat dans la valeur d'origine.
<code>/=</code>	<code>/=</code> (Diviser égal) L'opérateur <code>/=</code> divisera une valeur par la valeur d'origine et stockera le résultat dans la valeur d'origine
<code>%=</code>	<code>%=</code> (modulo égal) L'opérateur <code>%=</code> divisera une valeur par la valeur d'origine et stockera le reste dans la valeur d'origine.

## Opérateurs logiques

Les opérateurs logiques AND et OR peuvent être utilisés en complément de la commande WHERE pour combiner des conditions.

Les opérateurs logiques sont ceux qui renvoient vrai ou faux, comme l'opérateur AND, qui renvoie vrai lorsque les deux expressions sont rencontrées.

Opérateur	Description
<code>AND</code>	L'opérateur AND renvoie TRUE si toutes les conditions séparées par AND sont vraies.

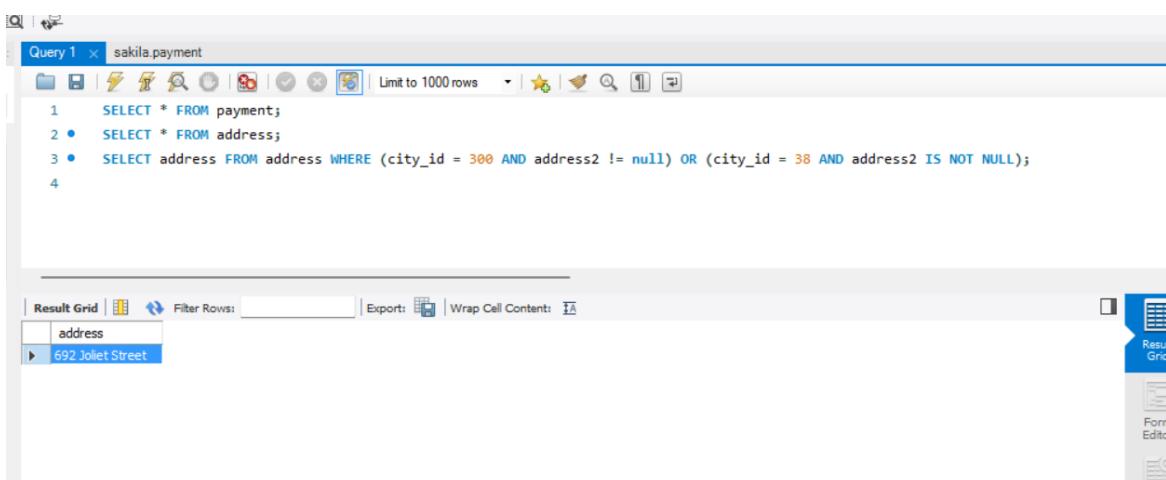
<b>OR</b>	L'opérateur OR renvoie TRUE si l'une des conditions séparées par OR est vraie.
<b>BETWEEN</b>	L'opérateur BETWEEN filtre votre requête pour ne renvoyer que les résultats qui correspondent à une plage spécifiée. (entre)
<b>ANY</b>	L'opérateur ANY renvoie TRUE si l'une des valeurs de la sous-requête répond à la condition spécifiée. (n'importe quel)
<b>EXISTS</b>	L'opérateur EXISTS est utilisé pour filtrer les données en recherchant la présence de tout enregistrement dans une sous-requête. (existe)
<b>IN</b>	L'opérateur IN inclut plusieurs valeurs définies dans la clause WHERE. (dans)
<b>LIKE</b>	L'opérateur LIKE recherche un modèle spécifié dans une colonne. (comme)
<b>NOT</b>	L'opérateur NOT renvoie des résultats si la ou les conditions ne sont pas vraies.
<b>SOME</b>	Renvoie le même résultat qu' ANY.
<b>ALL</b>	L'opérateur ALL renvoie TRUE si toutes les valeurs de la sous-requête remplissent la condition spécifiée
<b>IS NULL</b>	L'opérateur IS NULL est utilisé pour filtrer les résultats avec une valeur NULL.

Exemples :

#### AND/OR/IS NOT NULL:

Demande d'affichage des adresses ayant soit un city\_id = 300 mais une adresse2 différente de null, soit un city-id = 38 avec une adresse2 également non null.

La première condition n'étant pas vraie, aucune adresse ayant l'id 300 n'a une adresse2 non nulle, alors on me renvoie le résultat ayant la seconde condition pour vraie:



```

Query 1 x sakila.payment
1  SELECT * FROM payment;
2  •  SELECT * FROM address;
3  •  SELECT address FROM address WHERE (city_id = 300 AND address2 != null) OR (city_id = 38 AND address2 IS NOT NULL);
4

```

#### BETWEEN :

Demande d'affichage de toutes les colonnes qui concernent les montants compris entre 5 et 6 dans la table payment:

```

Query 1 | sakila.payment
1 • SELECT * FROM payment;
2 • SELECT * FROM address;
3 • SELECT * FROM payment WHERE amount BETWEEN 5 AND 6;

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content: | Fetch rows: | Result Grid | Form Editor | Field Types | Context Help | Schema
+-----+-----+-----+-----+-----+-----+-----+
| payment_id | customer_id | staff_id | rental_id | amount | payment_date | last_update |
+-----+-----+-----+-----+-----+-----+-----+
| 3 | 1 | 1 | 1185 | 5.99 | 2005-06-15 00:54:12 | 2006-02-15 22:12:30 |
| 10 | 1 | 2 | 4526 | 5.99 | 2005-07-08 03:17:05 | 2006-02-15 22:12:30 |
| 11 | 1 | 1 | 4611 | 5.99 | 2005-07-08 07:33:56 | 2006-02-15 22:12:30 |
| 32 | 1 | 1 | 15315 | 5.99 | 2005-08-22 20:03:46 | 2006-02-15 22:12:30 |
| 38 | 2 | 1 | 7376 | 5.99 | 2005-07-27 15:23:02 | 2006-02-15 22:12:30 |
| 39 | 2 | 2 | 7459 | 5.99 | 2005-07-27 18:40:20 | 2006-02-15 22:12:30 |
| 40 | 2 | 2 | 8230 | 5.99 | 2005-07-29 00:12:59 | 2006-02-15 22:12:30 |
| 42 | 2 | 2 | 8705 | 5.99 | 2005-07-29 17:14:29 | 2006-02-15 22:12:30 |
| 51 | 2 | 1 | 11087 | 5.99 | 2005-08-02 07:41:41 | 2006-02-15 22:12:30 |
| 57 | 2 | 2 | 14743 | 5.99 | 2005-08-21 22:41:56 | 2006-02-15 22:12:30 |
| 68 | 3 | 1 | 7096 | 5.99 | 2005-07-27 04:54:42 | 2006-02-15 22:12:30 |
+-----+-----+-----+-----+-----+-----+-----+

```

LIKE :

Demande d'afficher toutes les informations des acteurs ayant un film\_info commençant par 'c':

```

Query 1 | sakila.payment
1 • SELECT * FROM actor_info WHERE film_info LIKE 'C%';
2

Result Grid | Filter Rows: | Export: | Wrap Cell Content: | Result Grid | Form Editor | Field Types | Context Help | Schema
+-----+-----+-----+-----+
| actor_id | first_name | last_name | film_info |
+-----+-----+-----+-----+
| 74 | MILLA | KEITEL | Children: NOON PAPI; Classics: JERK PAYCHEC... |
| 97 | MEG | HAWKE | Children: GORGEOUS BINGO, MAKER GABLES, ... |
| 118 | CUBA | ALLEN | Children: MAKER GABLES, ROBBERS JOON, STR... |
| 124 | SCARLETT | BENING | Children: FULL FLATLINERS, INVASION CYCLON... |
| 148 | EMILY | DEE | Children: CHRISTMAS MOONSHINE, INVASION ... |
+-----+-----+-----+-----+

```

IN:

Affichage des noms ayant comme prénom Penelope :

The screenshot shows a MySQL Workbench interface. The top bar has tabs for 'Query 1' and 'sakila.payment'. Below the tabs is a toolbar with various icons. A dropdown menu shows 'Limit to 1000 rows'. The main area contains a query editor with the following code:

```
1 •  SELECT last_name FROM actor WHERE first_name IN ('PENELOPE');
2
3
```

Below the query editor is a 'Result Grid' table with one column labeled 'last\_name'. The data in the table is:

last_name
GUINNESS
PINKETT
CRONYN
MONROE

# SQL null functions :

## fonction SQL ISNULL (), IFNULL () et COALESCE ()

Faire:

```
SELECT ... WHERE MaColonne = NULL et cela ne marche pas !
```

NULL :

Cet élément du SQL n'est pas une valeur, puisque c'est justement l'absence de valeur. En fait, NULL est un marqueur comme NIL en programmation qui indique qu'au bout du pointeur il n'y a pas de ressource.

Par exemple, comparer une colonne à NULL n'a aucun sens. Pour traiter les marqueurs NULL il faut utiliser des prédictats spéciaux comme IS NULL, IS NOT NULL ou des fonctions particulières comme COALESCE ou NULLIF ...

### isnull():

Dans le langage SQL la fonction ISNULL() peut s'avérer utile pour traiter des résultats qui possèdent des données nulles. Attention toutefois, cette fonction s'utilise différemment selon le système de gestion de base de données :

- **MySQL** : la fonction ISNULL() prend un seul paramètre et permet de vérifier si une données est nulle
- **SQL Server** : la fonction ISNULL() prend 2 paramètres et sert à afficher la valeur du 2ème paramètre si le premier paramètre est null. Cette même fonctionnalité peut être effectuée par d'autres systèmes de gestion de base de données d'une manière différente :
  - MySQL : il faut utiliser la fonction IFNULL()
  - PostgreSQL : il faut utiliser la fonction COALESCE()
  - Oracle : il faut utiliser la fonction NVL()

Exemples:

-> Affichage des adresses, où address2 n'a pas de données :

Query 1 × sakila.payment

```
1 •  SELECT * FROM address WHERE ISNULL(address2) = 1;
2
```

address_id	address	address2	district	city_id	postal_code	phone	location	last_update
1	47 MySakila Drive	NULL	Alberta	300			BL0B	2014-09-25 22:30:27
2	28 MySQL Boulevard	NULL	QLD	576			BL0B	2014-09-25 22:30:09
3	23 Workhaven Lane	NULL	Alberta	300	14033335568		BL0B	2014-09-25 22:30:27
4	1411 Lillydale Drive	NULL	QLD	576	6172235589		BL0B	2014-09-25 22:30:09
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

### IFNULL():

La fonction MySQL **IFNULL()** vous permet de renvoyer une valeur alternative si une expression est NULL.

-> Si *expr1* vaut NULL, renvoie *expr2*, sinon renvoie *expr1*.

Exemple:

-> J'affiche la table address, en mettant les valeurs d' address2 null en "maj":

Query 1 × sakila.payment

```
1 •  select * from address;
2 •  select *,IFNULL(address2, 0) FROM address;
3 •  select *,IFNULL(address2, 'maj') FROM address;
```

address_id	address	address2	district	city_id	postal_code	phone	location	last_update	IFNULL(address2, 'maj')
1	47 MySakila Drive	NULL	Alberta	300			BL0B	2014-09-25 22:30:27	maj
2	28 MySQL Boulevard	NULL	QLD	576			BL0B	2014-09-25 22:30:09	maj
3	23 Workhaven Lane	NULL	Alberta	300	14033335568		BL0B	2014-09-25 22:30:27	maj
4	1411 Lillydale Drive	NULL	QLD	576	6172235589		BL0B	2014-09-25 22:30:09	maj
5	1913 Hanoi Way	Nagasaki	463	35200	28303384290		BL0B	2014-09-25 22:31:53	
6	1121 Loja Avenue	California	449	17886	838635286649		BL0B	2014-09-25 22:34:01	
7	692 Joliet Street	Attika	38	83579	448477190408		BL0B	2014-09-25 22:31:07	
8	1566 Ingle Manor	Mandalay	349	53561	705814003527		BL0B	2014-09-25 22:32:18	
9	53 Idfu Parkway	Nantou	361	42399	10655648674		BL0B	2014-09-25 22:33:16	
10	1795 Santiago de Compostela Way	Texas	295	18743	860452626434		BL0B	2014-09-25 22:33:55	

### COALESCE():

Évalue les arguments dans l'ordre et renvoie la valeur actuelle de la première expression qui n'est initialement pas évaluée à NULL.

Par exemple:

```
SELECT COALESCE(NULL, NULL, 'third_value', 'fourth_value');
```

-> renvoie la troisième valeur car la troisième valeur est la première valeur qui n'est pas nulle.

Si tous les arguments sont NULL, COALESCE renvoie NULL. Au moins une des valeurs nulles doit être un type NULL.

L'expression COALESCE est un raccourci syntaxique de l'expression CASE. Autrement dit, le code COALESCE(expression1,...n) est réécrit par l'optimiseur de requête sous la forme de l'expression CASE suivante :

```
CASE
WHEN (expression1 IS NOT NULL) THEN expression1
WHEN (expression2 IS NOT NULL) THEN expression2
...
ELSE expressionN
END
```

Pour faire simple:

Avec MySQL la valeur NULL est particulière et a parfois un comportement inattendu.

MySQL écarte automatiquement les valeurs NULL avant de faire une comparaison.

Et c'est là qu'intervient la fonction COALESCE qui permet de remplacer NULL par une valeur donnée.

Il est possible de remplacer à la volée, donc dans une requête, un marqueur NULL par une valeur conventionnelle. Par exemple, zéro, chaîne vide, etc. La particularité de cette fonction est d'accepter autant de paramètres que l'on veut (sans limitation théorique).

Elle renvoie la première expression évaluable (donc ne contenant pas de marqueur NULL) dans l'ordre positionnel de l'écriture de gauche à droite.

Exemple:

Affichage des valeurs de la colonne address2 avec valeurs null qui seront remplacées par la valeur de la colonne district.

The screenshot shows a MySQL Workbench interface with a query editor and a results grid. The query is:

```
1 •  SELECT coalesce(address2,district) FROM address;
```

The results grid displays the output of the query:

coalesce(address2,district)
Alberta
QLD
Alberta
QLD

Exemple:

-> Affichage des valeurs null de la colonne address2, remplacée par "à renseigner".

The screenshot shows a MySQL Workbench interface with a query editor and a results grid. The query is:

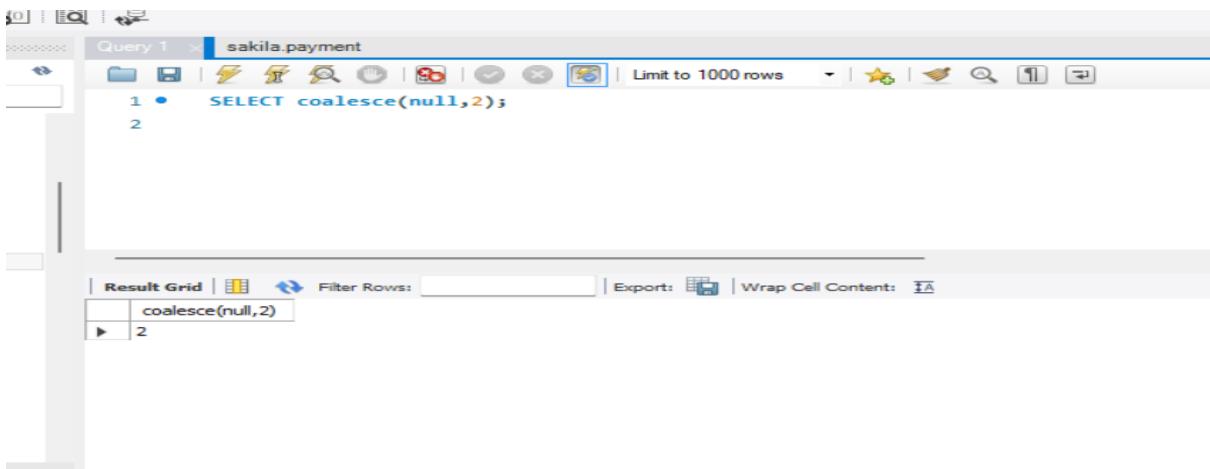
```
1 •  SELECT address, coalesce(address2,"A renseigner") FROM address;
```

The results grid displays the output of the query:

address	coalesce(address2,"A renseigner")
47 MySakila Drive	A renseigner
28 MySQL Boulevard	A renseigner
23 Workhaven Lane	A renseigner
1411 Lillydale Drive	A renseigner
1913 Hanoi Way	
1121 Loja Avenue	
692 Joliet Street	
1566 Inglel Manor	

Autres exemples:

-> Première valeur null donc renvoie 2:

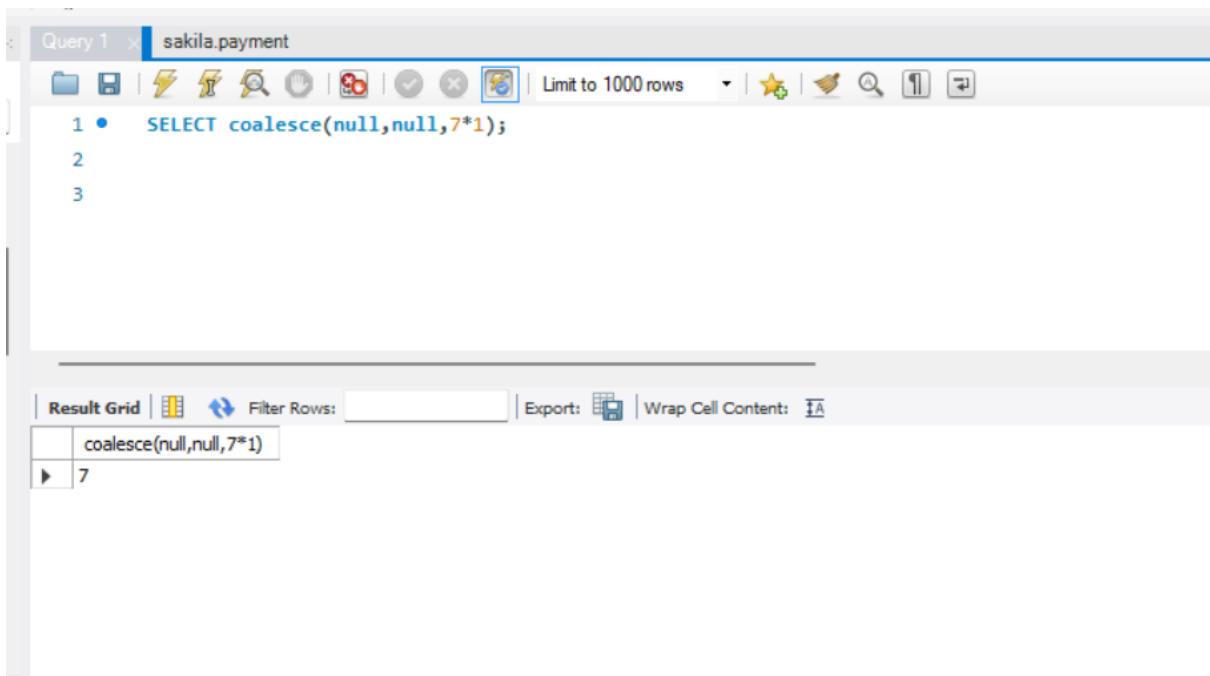


The screenshot shows a MySQL Workbench interface with a query editor titled "Query 1" and a database "sakila.payment". The query is:

```
1 •    SELECT coalesce(null,2);  
2
```

The result grid shows one row with the value 2.

-> Première et deuxième valeurs null donc renvoie 7\*1 :



The screenshot shows a MySQL Workbench interface with a query editor titled "Query 1" and a database "sakila.payment". The query is:

```
1 •    SELECT coalesce(null,null,7*1);  
2  
3
```

The result grid shows one row with the value 7.

## SQL WHERE & SQL AND OR NOT

[@Faouzilha](#)

### Les opérateurs SQL AND, OR et NOT

La **WHERE** clause peut être combinée avec les opérateurs **AND**, **OR** et **NOT**.

Les opérateurs **AND** et **OR** sont utilisés pour filtrer les enregistrements en fonction de plusieurs conditions :

L'**AND** opérateur affiche un enregistrement si toutes les conditions séparées par **AND** sont VRAIES.

L' **OR** opérateur affiche un enregistrement si l'une des conditions séparées par **OR** est VRAIE.

L' **NOT** opérateur affiche un enregistrement si la ou les conditions ne sont **PAS VRAIES**.

Exemple :

Opérateurs **where not and not**

sélection de tous le champs de "rating" où le rating **n'est pas** "PG-13" et pas "NC-17"

Select **title, rating** from film **where not rating='PG-13' and not rating='NC-17'**

	film_id	title	description	release_year	language_id	original_language_id	rental_dura	rental_rate	length	replacement_c	rating
▶	1	ACADEMY DINOSAUR	A Epic Drama of a Feminist...	2006	1	NULL	6	0.99	86	20.99	PG
	2	ACE GOLDFINGER	A Astounding Epistle of a ...	2006	1	NULL	3	4.99	48	12.99	G
	3	ADAPTATION HOLES	A Astounding Reflection o...	2006	1	NULL	7	2.99	50	18.99	NC-17
	4	AFFAIR PREJUDICE	A Fanciful Documentary of...	2006	1	NULL	5	2.99	117	26.99	G
	5	AFRICAN EGG	A Fast-Paced Documentar...	2006	1	NULL	6	2.99	130	22.99	G
	6	AGENT TRUMAN	A Intrepid Panorama of a ...	2006	1	NULL	3	2.99	169	17.99	PG
	7	AIRPLANE SIERRA	A Touching Saga of a Hun...	2006	1	NULL	6	4.99	62	28.99	PG-13
	...	...	...	...	...	...	...	...	...	...	...
	film 49	film 54	×								

Résultat :

Result Grid		
	title	rating
▶	ACADEMY DINOSAUR	PG
	ACE GOLDFINGER	G
	AFFAIR PREJUDICE	G
	AFRICAN EGG	G
	AGENT TRUMAN	PG
	AIRPORT POLLOCK	R
	ALAMO VIDEOTAPE	G
	ALASKA PHANTOM	PG
	ALT FORFVFR	PG
	film 49	film 53
	film 53	×

Opérateurs **where and or**

select length, rental\_duration from film **where length < '90' and rental\_duration < 4 or rental\_duration > 6**

Résultat :

	length	rental_dura
▶	48	3
	50	7
	82	3
	74	3
	86	3
	179	7
	127	7
	--	--
	film 49	film 57
	film 57	×

Les opérateurs suivants peuvent être utilisés dans la **WHERE** clause

## Opérateurs de comparaison

Opérateur	Description
<	moins que
>	plus grand que
<=	inférieur ou égal à
>=	Plus grand ou égal à
=	égal
<>ou!=	inégal

Sélection de tous les film **ou** le champs replacement\_cost est = à 12.99 **et** film\_id inférieur à 100 dans la table film

select\* from film **where** replacement\_cost=12.99 **and** film\_id<100

	film_id	title	description	release_year	language_id	original_language_id	rental_duration	rental_rate	length	replacement_cost
▶	2	ACE GOLDFINGER	A Astounding Epistle of a ...	2006	1	NULL	3	4.99	48	12.99
	27	ANONYMOUS HUMAN	A Amazing Reflection of a ...	2006	1	NULL	7	0.99	179	12.99
	36	ARGONAUTS TOWN	A Emotional Epistle of a Fo...	2006	1	NULL	7	0.99	127	12.99
	72	BILL OTHERS	A Stunning Saga of a Mad ...	2006	1	NULL	6	2.99	93	12.99
	78	BLACKOUT PRIVATE	A Intrepid Yarn of a Pastr...	2006	1	NULL	7	2.99	85	12.99
*	98	BRIGHT ENCOUNTERS	A Fateful Yarn of a Lumbe...	2006	1	NULL	4	4.99	73	12.99
	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Sélection de la colonne titre dont le nom commence par "F" avec like

select title, film\_id, rating from film **where** title like "F%"

	title	film_id	rating
▶	FACTORY DRAGON	299	PG-13
	FALCON VOLUME	300	PG-13
	FAMILY SWEET	301	R
	FANTASIA PARK	302	G
	FANTASY TROOPERS	303	PG-13
	FARGO GANDHI	304	G
	FATAL HAUNTED	305	PG
	FEATHERS METAL	306	PG-13
	FFI LOWSHTP AI IMN	307	NC-17

## Conclusion

La clause SQL WHERE est utilisée pour restreindre le nombre de lignes affectées par une requête SELECT, UPDATE ou DELETE.

La condition WHERE dans SQL peut être utilisée conjointement avec des opérateurs logiques tels que AND et OR, des opérateurs de comparaison tels que ,= etc.

Lorsqu'il est utilisé avec l'opérateur logique ET, tous les critères doivent être remplis.

Lorsqu'il est utilisé avec l'opérateur logique OR, tous les critères doivent être satisfait.

Le mot clé IN est utilisé pour sélectionner les lignes correspondant à une liste de valeurs.

# SQL ORDER BY & NULL VALUES

@Romain

## LA REQUÊTE SQL ORDER BY example

La requête SQL ORDER BY permet de trier les lignes dans le résultat d'une requête SQL.  
Il est possible de pouvoir trier les données sur une ou plusieurs colonnes en utilisant soit la commande:

- ORDER BY DESC → classement par ordre descendant.
- ORDER BY ASC → classement par ordre ascendant.

La requête SQL ORDER BY : permet de pouvoir ordonner la table '**actor**' de la base de données Sakila en fonction de la colonne choisie : **actor\_id**, **first-name** ou **last\_name**

La commande **SELECT** retourne les valeurs enregistrées dans le tableau ' actor'

Présentation de la syntaxe SQL ORDER BY

```
-- SELECT * FROM sakila.actor;
-- SELECT*FROM sakila.actor
ORDER BY first_name , last_name;  (Dans l'exemple, si on rajoute last_name , on trie celle-ci par rapport aux résultats de la colonne first_name )
```

La table actor d'origine sans la requête SQL ORDER BY

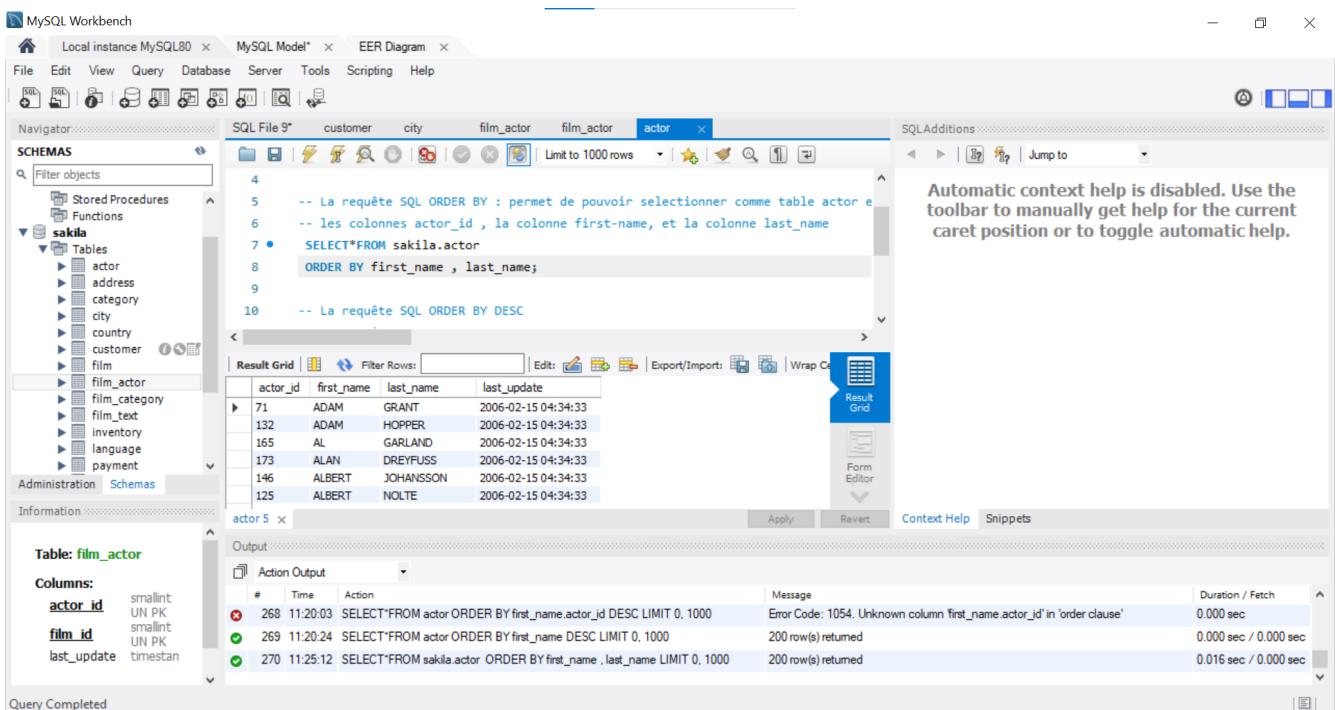
actor_id	first_name	last_name	last_update
1	PENELope	GUINNESS	2006-02-15 04:34:33
2	NICK	WAHLBERG	2006-02-15 04:34:33
3	ED	CHASE	2006-02-15 04:34:33
4	JENNIFER	DAVIS	2006-02-15 04:34:33
5	JOHNNY	LOLLOBRIGIDA	2006-02-15 04:34:33
6	BETTE	NICHOLSON	2006-02-15 04:34:33

## LA REQUÊTE SQL ORDER BY DESC

Présentation de la syntaxe SQL ORDER BY

-- La requête SQL ORDER BY  
 -- La requête SQL ORDER BY permet de trier les champs de ligne rangés par ordre alphabétique **le first\_name** avec leur **id\_actor** dans la colonne **first\_name**

-- SELECT\*FROM actor  
 -- ORDER BY first\_name ,actor\_id;



The screenshot shows the MySQL Workbench interface. In the SQL Editor tab, the following SQL code is written:

```

-- La requête SQL ORDER BY : permet de pouvoir sélectionner comme table actor e
-- les colonnes actor_id , la colonne first-name , et la colonne last-name
SELECT*FROM sakila.actor
ORDER BY first_name , last_name;

-- La requête SQL ORDER BY DESC

```

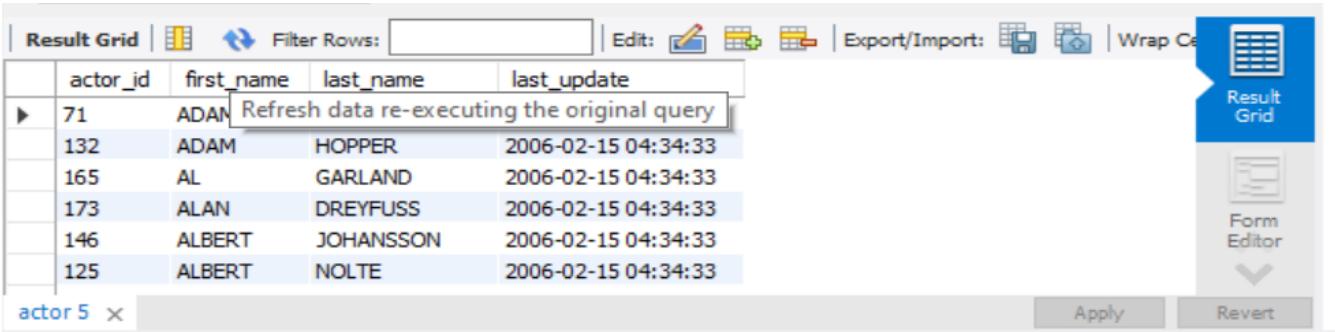
The Result Grid shows the following data:

actor_id	first_name	last_name	last_update
71	ADAM	GRANT	2006-02-15 04:34:33
132	ADAM	HOPPER	2006-02-15 04:34:33
165	AL	GARLAND	2006-02-15 04:34:33
173	ALAN	DREYFUSS	2006-02-15 04:34:33
146	ALBERT	JOHANSSON	2006-02-15 04:34:33
125	ALBERT	NOLTE	2006-02-15 04:34:33

The Output pane shows the following log entries:

#	Time	Action	Message	Duration / Fetch
268	11:20:03	SELECT*FROM actor ORDER BY first_name.actor_id DESC LIMIT 0, 1000	Error Code: 1054. Unknown column 'first_name.actor_id' in 'order clause'	0.000 sec
269	11:20:24	SELECT*FROM actor ORDER BY first_name DESC LIMIT 0, 1000	200 row(s) returned	0.000 sec / 0.000 sec
270	11:25:12	SELECT*FROM sakila.actor ORDER BY first_name , last_name LIMIT 0, 1000	200 row(s) returned	0.016 sec / 0.000 sec

Le résultat obtenu après avoir lancer la requête SQL ORDER BY



The screenshot shows the MySQL Workbench Result Grid. The data is identical to the one shown in the previous screenshot:

actor_id	first_name	last_name	last_update
71	ADAM	GRANT	2006-02-15 04:34:33
132	ADAM	HOPPER	2006-02-15 04:34:33
165	AL	GARLAND	2006-02-15 04:34:33
173	ALAN	DREYFUSS	2006-02-15 04:34:33
146	ALBERT	JOHANSSON	2006-02-15 04:34:33
125	ALBERT	NOLTE	2006-02-15 04:34:33

LA REQUÊTE SQL ORDER BY ASC

Présentation de la syntaxe SQL ORDER BY ASC

-- La requête SQL ORDER BY ASC

-- La requête SQL ORDER BY ASC  
-- **SELECT first\_name , last\_name**  
-- **FROM customer**  
-- **order by last\_name ASC;**

Le résultat obtenu après avoir lancer la requête SQL ORDER BY ASC

The screenshot shows the MySQL Workbench interface. In the top-left pane, the 'Schemas' tree is visible, showing tables like actor, address, category, city, country, and customer. The 'customer' table is selected. In the main query editor window, the following SQL code is written:

```
-- ORDER BY first_name ,actor_id DESC;
-- La requête SQL ORDER BY ASC
-- SELECT first_name , last_name
-- FROM customer
-- order by last_name ASC;
```

Below the code, the 'Result Grid' shows the following data:

first_name	last_name
RAFAEL	ABNEY
NATHANIEL	ADAM
KATHLEEN	ADAMS
DIANA	ALEXANDER
GORDON	ALLARD
SHIRLEY	ALLEN

The 'Output' pane at the bottom shows the execution log:

#	Time	Action	Message	Duration / Fetch
310	17:04:15	SELECT first_name ,last_name FROM actor order by last_name ASC LIMIT 0, 1000	200 row(s) returned	0.000 sec / 0.000 sec
311	17:10:21	SELECT * FROM sakila.film_actor LIMIT 0, 1000	1000 row(s) returned	0.000 sec / 0.000 sec
312	17:10:23	SELECTfirst_name ,last_name FROM customer order by last_name ASC LIMIT ...	599 row(s) returned	0.000 sec / 0.000 sec

This is a zoomed-in view of the 'Result Grid' from the MySQL Workbench interface. It displays the same six rows of data from the 'customer' table, ordered by last name:

first_name	last_name
RAFAEL	ABNEY
NATHANIEL	ADAM
KATHLEEN	ADAMS
DIANA	ALEXANDER
GORDON	ALLARD
SHIRLEY	ALLEN

## LA REQUÊTE SQL NULL Values

### Présentation de la syntaxe SQL NULL Values

La table address d'origine sans la requête SQL NULL Values

The screenshot shows the MySQL Workbench interface. On the left, the schema browser displays the 'customer' table with columns: customer\_id, store\_id, first\_name, last\_name, email, address\_id, and active. The 'address\_id' column is highlighted. The main area shows a result grid for the 'customer' table with 5 rows of data. Below the grid, the query history shows three queries related to the 'address' table:

- 325 10:01:47 SELECT \* FROM address WHERE address2 IS NULL LIMIT 0, 1000
- 326 10:01:50 SELECT \* FROM sakila.customer LIMIT 0, 1000
- 327 10:01:50 SELECT \* FROM address WHERE address2 IS NULL LIMIT 0, 1000

The status bar at the bottom indicates a temperature of 17°C, clear skies, and the date/time 10/03/2023 10:01.

```
1 •   SELECT * FROM address WHERE address2 IS NULL ;
2   -- SELECT * FROM address ;
3
```

The screenshot shows the MySQL Workbench interface with the 'address' table selected. The result grid displays 4 rows of data, each with a different address and its corresponding address2 value. The 'address2' column contains several NULL values. The status bar at the bottom indicates a temperature of 17°C, clear skies, and the date/time 10/03/2023 10:01.

address_id	address	address2	district	city_id	postal_code	phone	location	last_update
1	47 MySakila Drive	NULL	Alberta	300			BLOB	2014-09-25 22:30:2
2	28 MySQL Boulevard	NULL	QLD	576			BLOB	2014-09-25 22:30:0
3	23 Workhaven Lane	NULL	Alberta	300		14033335568	BLOB	2014-09-25 22:30:2
4	1411 Lillydale Drive	NULL	QLD	576		6172235589	BLOB	2014-09-25 22:30:0

# SQL SELECT TOP, SQL ANY, ALL -

@Anthony

## La clause SQL SELECT TOP

La SELECT TOP est utilisée pour spécifier le nombre d'enregistrements à retourner.

La SELECT TOP est utile sur les grandes tables avec des milliers d'enregistrements. Le renvoi d'un grand nombre d'enregistrements peut avoir un impact sur les performances.

**Remarque :** Tous les systèmes de base de données ne prennent pas en charge la SELECT TOP clause. MySQL prend en charge la LIMITE permettant de sélectionner un nombre limité d'enregistrements, tandis qu'Oracle utilise `and . FETCH FIRST n ROWS ONLY ROWNUM`

```
--  
3      -- SELECT TOP  
4      -- ON REMPLACE LA COMMANDE SELECT TOP EN FIXANT UNE LIMITE A 5 ID  
5 •  SELECT actor_id  
6      from  actor  
7      limit 5;
```

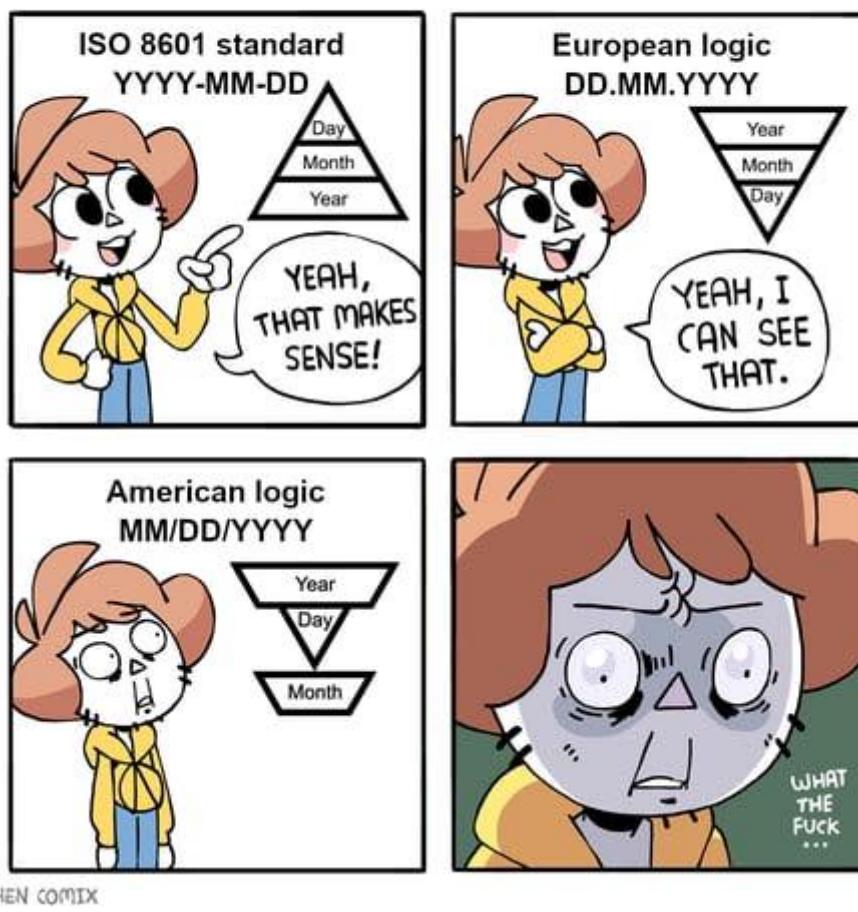
The screenshot shows the MySQL Workbench interface with a result grid. The grid has one column labeled 'actor\_id'. The data rows are 58, 92, 182, 118, and 145. The grid includes standard database navigation buttons like 'First', 'Last', 'Previous', 'Next', and 'Last' at the top, and 'Apply' at the bottom right.

actor_id
58
92
182
118
145

# SQL DATES & EXISTS -

@Mathias

## Date Formats



### *Table de type Dates :*

Il existe quatre types de table MySQL permettant de stocker une date dans une base de données.

[DATE](#), [DATETIME](#), [TIMESTAMP](#), [YEAR](#)

Chacun de ces types de table utilise un format de date spécifique :

- [DATE](#) => le format sera : [YYYY-MM-DD](#)
- [DATETIME](#) => le format sera : [YYYY-MM-DD HH:MI:SS](#)
- [TIMESTAMP](#) => le format sera : [YYYY-MM-DD HH:MI:SS](#)

- **YEAR** => le format sera : YYYY ou YY

Le **DATETIME** et le **TIMESTAMP** utilisent le même format, leurs différences se situent au niveau du stockage et de l'intervalle de valeur disponible.

#### DATETIME :

Le **DATETIME** contient une date dans son format spécifique sur 8 octets et son intervalle va de '1000-01-01' à '9999-12-31'.

#### TIMESTAMP :

Le **TIMESTAMP** contient, comme son nom l'indique, un **TIMESTAMP** qui est une représentation en nombre du nombre de seconde écoulés depuis le '01/01/1970 00:00:00' et sa limite haute est définie au "19/01/2038 03:14:07".

Il est stocké sur 4 octets et prend donc moins de place que le **DATETIME**.

Recherche avec **WHERE** :

Si l'on fait une recherche en utilisant une date précise, il faut s'assurer que l'on respecte le format de la table ciblé.

## *Exemples :*

#### DATETIME :

La table "**last\_update**" est de type **DATETIME**, en recherchant '2006-02-15', je n'ai aucun résultat car je n'ai pas respecté le format qui est YYYY-MM-DD HH:MI:SS.

The screenshot shows the MySQL Workbench interface. The SQL Editor tab is active, displaying the following query:

```
1 •  Select * FROM rental WHERE last_update = '2006-02-15'
2
```

The Result Grid tab is below, showing a single row with all columns set to NULL:

	rental_id	rental_date	inventory_id	customer_id	return_date	staff_id	last_update
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL

La table "**rental\_date**" est également de type DATETIME, en recherchant "2005-05-24 22:53:30", je trouve un résultat car j'ai respecté le format de la table.

The screenshot shows the MySQL Workbench interface. At the top, there's a toolbar with various icons. Below it is a SQL editor window containing the following code:

```

1 •  Select * FROM rental WHERE rental_date = '2005-05-24 22:53:30'
2

```

Below the editor is a results grid titled "Result Grid". It has columns for rental\_id, rental\_date, inventory\_id, customer\_id, return\_date, staff\_id, and last\_update. The data row is:

	rental_id	rental_date	inventory_id	customer_id	return_date	staff_id	last_update
▶	1	2005-05-24 22:53:30	367	130	2005-05-26 22:04:30	1	2006-02-15 21:30:53
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL

#### TIMESTAMP :

Les colonnes "`last_update`" présentent dans la plupart des tables du schéma Sakila sont toutes de type **TIMESTAMP**, cela est dû aux particularités du **TIMESTAMP**, ces colonnes ayant pour but de noter à quelle date la base de données a été mise à jour, elles sont donc compris dans l'intervalle "1970/2038", aucune raison donc d'utiliser un type plus lourd comme le **DATETIME**.

Lorsque l'on fait une requête en cherchant une date en particulier, on doit tout de même respecter le format "**YYYY-MM-DD HH:MI:SS**".

Si je recherche la date "**2006-02-15 05:02:19**" en nombre de secondes "**1139976139**", je n'aurais aucun résultat car la requête SQL comparera la valeur affichée et non celle stockée

The screenshot shows the MySQL Workbench interface. At the top, there's a toolbar with various icons. Below it is a SQL editor window containing the following code:

```

1 •  Select * FROM language where last_update = 1139976139
2

```

Below the editor is a results grid titled "Result Grid". It has columns for language\_id, name, and last\_update. The data row is:

	language_id	name	last_update
*	NULL	NULL	NULL

En revanche, en recherchant la date avec le bon format ("**2006-02-15 05:02:19**"), j'ai des résultats :

SQL File 6\*

```
1 •  Select * FROM language where last_update = "2006-02-15 05:02:19"
2
```

Result Grid | Filter Rows: [ ] | Edit: [ ] | Export/Import: [ ]

	language_id	name	last_update
▶	1	English	2006-02-15 05:02:19
	2	Italian	2006-02-15 05:02:19
	3	Japanese	2006-02-15 05:02:19
	4	Mandarin	2006-02-15 05:02:19
	5	French	2006-02-15 05:02:19
*	6	German	2006-02-15 05:02:19
	NULL	NULL	NULL

## DATE :

N'ayant pas de DATE dans la base de données “Sakila”, j'ai créé une table avec une colonne de type DATE.

SQL File 7\*

```
1      SELECT achatdate FROM achat
```

Result Grid | Filter Rows: [ ]

achatdate
▶ 2006-02-15
2007-03-15
2007-05-15
2008-08-10

Maintenant, je recherche la date du “2006-02-15”, j'obtiens un résultat.

SQL File 7\* x

The screenshot shows the MySQL Workbench interface with a query editor window titled "SQL File 7\*". The query is:

```
1   SELECT achatdate FROM achat WHERE achatdate = "2006-02-15"
```

The results pane below shows a single row in a grid:

	achatdate
▶	2006-02-15

Mais si je rajoute l'heure derrière, cela ne marche plus ("2006-02-15 10:50:30") :

SQL File 7\* x

The screenshot shows the MySQL Workbench interface with a query editor window titled "SQL File 7\*". The query is:

```
1   SELECT achatdate FROM achat WHERE achatdate = "2006-02-15 10:50:30"
```

The results pane below is empty, indicating no rows were found.

YEAR :

La table film contient une table "release\_year" qui est de type Year, pour rechercher une date précise dans cette table, je dois donc saisir uniquement l'année que je recherche.

SQL File 6\*

1 • Select \* FROM film

2

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell C

	film_id	title	description	release_year	language_id
	12	ALASKA PHANTOM	A Fanciful Saga of a Hunter And a Pastry Chef ...	2006	1
	13	ALI FOREVER	A Action-Packed Drama of a Dentist And a Croc...	2006	1
	14	ALICE FANTASIA	A Emotional Drama of a A Shark And a Databas...	2006	1
	15	ALIEN CENTER	A Brilliant Drama of a Cat And a Mad Scientist w...	2006	1
	16	ALLEY EVOLUTION	A Fast-Paced Drama of a Robot And a Composer...	2006	1
	17	ALONE TRIP	A Fast-Paced Character Study of a Composer A...	2006	1
	18	ALTER VICTORY	A Thoughtful Drama of a Composer And a Femi...	2006	1

En recherchant '2006' dans "release\_year", j'obtiens donc la liste de tous les films sortis en 2006 présents dans la base de données.

SQL File 6\*

1 • Select \* FROM film where release\_year = '2006'

2

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell C

	film_id	title	description	release_year
▶	1	ACADEMY DINOSAUR	A Epic Drama of a Feminist And a Mad Scientist ...	2006
	2	ACE GOLDFINGER	A Astounding Epistle of a Database Administrat...	2006
	3	ADAPTATION HOLES	A Astounding Reflection of a Lumberjack And a ...	2006
	4	AFFAIR PREJUDICE	A Fanciful Documentary of a Frisbee And a Lum...	2006

## Fonction des Dates :

**ADDDATE, DATE\_ADD** : Permet de rajouter un intervalle de temps à une date.  
**ADDDATE(date, INTERVAL valeur unitétemporel)**

```
5 • SELECT ADDDATE(last_update, INTERVAL 10 DAY) FROM address
```

Result Grid		Filter Rows:	Export:	Wrap Cell Content:
ADDDATE(last_update, INTERVAL 10 DAY)				
▶	2014-10-05 22:30:27			
	2014-10-05 22:30:09			
	2014-10-05 22:30:27			

**ADDTIME** : Permet de rajouter un intervalle de temps à une date en respectant le format date.

ADDTIME (date, valeurtemporel)

```
5 • SELECT ADDDATE(last_update, "10:50:03"), last_update FROM address
```

Result Grid		Filter Rows:	Export:	Wrap Cell Content:
ADDDATE(last_update, "10:50:03")				
▶	2014-10-05 22:30:27	2014-09-25 22:30:27		
	2014-10-05 22:30:09	2014-09-25 22:30:09		
	2014-10-05 22:30:27	2014-09-25 22:30:27		
	2014-10-05 22:30:09	2014-09-25 22:30:09		

**CURDATE, CURRENT\_DATE, CURRENT\_TIME, CURRENT\_TIMESTAMP, LOCALTIME, LOCALTIMESTAMP, NOW, CURTIME, SYSDATE:**

Renvoie la date actuelle dans un format qui dépend de la fonction utilisée.

**CURDATE/CURRENT\_DATE** => "YYYY-MM-DD"

**CURTIME/CURRENT\_TIME** => "HH:MI:SS"

**CURRENT\_TIMESTAMP/LOCALTIME/LOCALTIMESTAMP/NOW/SYSDATE** => "YYYY-MM-DD HH:MI:SS"

**DATE** : Renvoie la date dans une chaîne de caractères .

```
5 • SELECT DATE(last_update), last_update FROM address
```

Result Grid		Filter Rows:	Export:	Wrap Cell Content:
DATE(last_update)				
▶	2014-09-25	2014-09-25 22:30:27		
	2014-09-25	2014-09-25 22:30:09		
	2014-09-25	2014-09-25 22:30:27		
	2014-09-25	2014-09-25 22:30:09		

**DATEDIFF** : Compare deux dates et renvoie le nombre de jours entre les deux.

```
5 • SELECT DATEDIFF(DATE(last_update),DATE(create_date)) FROM customer
```

Result Grid		Filter Rows:	Export:	Wrap Cell Content:
DATEDIFF(DATE(last_update),DATE(create_date))				
1				

**DATE\_FORMAT, DAY, MONTH, YEAR, MINUTE, HOUR, SECOND, MICROSECOND, EXTRACT** : Change le format d'écriture d'une date.

5 • `SELECT DATE_FORMAT(last_update, "%Y"), last_update FROM customer`

Result Grid	
Filter Rows:	
Export:	
DATE_FORMAT(last_update, "%Y")	last_update
2006	2006-02-15 04:57:20
2006	2006-02-15 04:57:20
2006	2006-02-15 04:57:20
2006	2006-02-15 04:57:20
2006	2006-02-15 04:57:20

5 • `SELECT DAY(last_update), last_update FROM customer`

Result Grid	
Filter Rows:	
Export:	
DAY(last_update)	last_update
15	2006-02-15 04:57:20
15	2006-02-15 04:57:20
15	2006-02-15 04:57:20

5 • `SELECT EXTRACT(HOUR FROM last_update), last_update FROM customer`

Result Grid	
Filter Rows:	
Export:	
EXTRACT(HOUR FROM last_update)	last_update
4	2006-02-15 04:57:20
4	2006-02-15 04:57:20
4	2006-02-15 04:57:20

**DATE\_SUB, SUBDATE** : Soustrait un interval de temps à une date :

5 • `SELECT DATE_SUB(last_update, INTERVAL 10 DAY), last_update FROM customer`

Result Grid	
Filter Rows:	
Export:	
DATE_SUB(last_update, INTERVAL 10 DAY)	last_update
2006-02-05 04:57:20	2006-02-15 04:57:20
2006-02-05 04:57:20	2006-02-15 04:57:20
2006-02-05 04:57:20	2006-02-15 04:57:20

**DAYNAME** : Renvoie le nom du jour de la semaine d'une date :

5 • `SELECT DAYNAME(last_update), last_update FROM customer`

Result Grid		Filter Rows:	Export:	Wrap Cell Content
	DAYNAME(last_update)	last_update		
▶	Wednesday	2006-02-15 04:57:20		
	Wednesday	2006-02-15 04:57:20		
	Wednesday	2006-02-15 04:57:20		

**DAYOFMONTH, DAYOFWEEK, WEEKDAY, DAYOFTIME** : Renvoie le numéro du jour par rapport à l'intervalle de temps donné :

5 • `SELECT DAYOFYEAR(last_update), last_update FROM customer`

Result Grid		Filter Rows:	Export:	Wrap Cell Content:
	DAYOFYEAR(last_update)	last_update		
▶	46	2006-02-15 04:57:20		
	46	2006-02-15 04:57:20		
	46	2006-02-15 04:57:20		

**FROM\_DAYS** : Renvoie une date à partir d'un nombre de jours, en comptant à partir de l'an 0 :

6 • `SELECT FROM_DAYS(650009);`

Result Grid		Filter Rows:
	FROM_DAYS(650009)	
▶	1779-08-31	

**LAST\_DAY** : Renvoie le dernier du mois d'une date :

5 • `SELECT LAST_DAY(last_update), last_update FROM customer`

Result Grid		Filter Rows:	Export:	Wrap Cell Content:
	LAST_DAY(last_update)	last_update		
▶	2006-02-28	2006-02-15 04:57:20		
	2006-02-28	2006-02-15 04:57:20		
	2006-02-28	2006-02-15 04:57:20		
	2006-02-28	2006-02-15 04:57:20		

**MAKEDATE** : Renvoie une date à partir d'une année et d'un nombre de jours :

5 • `SELECT MAKEDATE(2017, 3);`

Result Grid	
<input type="button" value="Export"/>	Filter Rows:
MAKEDATE(2017, 3)	2017-01-03

**MAKETIME** : Renvoie une valeur de temps à partir d'une heure, de minutes et de secondes :

5 • `SELECT MAKETIME(11, 35, 4);`

Result Grid	
<input type="button" value="Export"/>	Filter Rows:
MAKETIME(11, 35, 4)	11:35:04

**MONTHNAME** : Renvoie le nom du mois d'une date donné :

5 • `SELECT MONTHNAME(last_update), last_update FROM address;`

Result Grid	
MONTHNAME(last_update)	last_update
September	2014-09-25 22:30:27
September	2014-09-25 22:30:09
September	2014-09-25 22:30:27

**PERIOD\_ADD, PERIOD\_DIFF** : Ajoute ou soustrait un nombre de mois à une période en mois :

6 • `SELECT PERIOD_DIFF(201710, 201703);`

Result Grid	
<input type="button" value="Export"/>	Filter Rows:
PERIOD_DIFF(201710, 201703)	7

**QUARTER** : Renvoie le trimestre de l'année d'une date donnée :

5 • `SELECT QUARTER(last_update), last_update FROM address`

QUARTER(last_update)	last_update
3	2014-09-25 22:30:27
3	2014-09-25 22:30:09
3	2014-09-25 22:30:27

**SEC\_TO\_TIME** : Renvoie une valeur de temps à partir d'un nombre de secondes :

5 • `SELECT SEC_TO_TIME(1);`

SEC_TO_TIME(1)
00:00:01

**STR\_TO\_DATE** : Renvoie une date à partir d'une string :

5 • `SELECT STR_TO_DATE("August 10 2017", "%M %d %Y");`

STR_TO_DATE("August 10 2017", "%M %d %Y")
2017-08-10

**SUBTIME, TIMEDIFF** : Soustrait une valeur de temps à une date :

5 • `SELECT SUBTIME(last_update, "5:10:30"), last_update FROM address`

SUBTIME(last_update, "5:10:30")	last_update
2014-09-25 17:19:57	2014-09-25 22:30:27
2014-09-25 17:19:39	2014-09-25 22:30:09
2014-09-25 17:19:57	2014-09-25 22:30:27

**TIME** : Renvoie la valeur temps d'une date :

5 • `SELECT TIME(last_update), last_update FROM address`

TIME(last_update)	last_update
22:30:27	2014-09-25 22:30:27
22:30:09	2014-09-25 22:30:09
22:30:27	2014-09-25 22:30:27

**TIME\_FORMAT** : Change le format d'une valeur de temps :

```
5 • SELECT TIME_FORMAT("19:30:10", "%H %i %s");
```

Result Grid		Filter Rows:	Export:	Wrap Cell Content:
TIME_FORMAT("19:30:10", "%H %i %s")				
19 30 10				

**TIME\_TO\_SEC** : Convertit une valeur de temps en seconde :

```
5 • SELECT TIME_TO_SEC(TIME(last_update)), last_update FROM address;
```

Result Grid		Filter Rows:	Export:	Wrap Cell Content:
TIME_TO_SEC(TIME(last_update))	last_update			
81027	2014-09-25 22:30:27			
81009	2014-09-25 22:30:09			
81027	2014-09-25 22:30:27			

**TIMESTAMP** : Renvoie une date à partir des arguments :

```
5 • SELECT TIMESTAMP("2017-07-23", "13:10:11");
```

Result Grid		Filter Rows:	Export:	Wrap Cell Content:
TIMESTAMP("2017-07-23", "13:10:11")				
2017-07-23 13:10:11				

**TO\_DAYS** : Renvoie le nombre de jours entre une date donnée et l'an 0 :

```
5 • SELECT TO_DAYS(last_update), last_update FROM address;
```

Result Grid		Filter Rows:	Export:	Wrap Cell Content:
TO_DAYS(last_update)	last_update			
735866	2014-09-25 22:30:27			
735866	2014-09-25 22:30:09			
735866	2014-09-25 22:30:27			

**WEEK, WEEKOFYEAR** : Renvoie le numéro de la semaine dans l'année d'une date donnée :

```
5 • SELECT WEEK(last_update), last_update FROM address;
```

Result Grid		Filter Rows:	Export:	Wrap Cell Content:
WEEK(last_update)	last_update			
38	2014-09-25 22:30:27			
38	2014-09-25 22:30:09			
38	2014-09-25 22:30:27			
38	2014-09-25 22:30:09			

**YEARWEEK** : Renvoie l'année et le numéro de la semaine dans l'année d'une date donnée :

```
5 •   SELECT YEARWEEK(last_update), last_update FROM address
```

The screenshot shows the MySQL Workbench interface with a result grid. The grid has two columns: 'YEARWEEK(last\_update)' and 'last\_update'. There are three rows of data:

YEARWEEK(last_update)	last_update
201438	2014-09-25 22:30:27
201438	2014-09-25 22:30:09
201438	2014-09-25 22:30:27

## Commande SQL **Exists** :

La commande SQL **EXISTS** permet de vérifier dans une sous-requête l'existence de lignes dans une table.

Au niveau de la syntaxe, on l'utilisera sur un **WHERE** :

**SELECT \* FROM language**

**WHERE EXISTS** (Sous requête)

## Exemples :

Dans la base de données "Sakila", mettons que je veuille voir les langues des films disponibles dans la base de donnée, la requête suivante renverra les langues dans lesquelles au moins un film est disponible :

The screenshot shows the MySQL Workbench interface with a SQL editor and a result grid. The SQL query is:

```
1  SELECT * FROM language
2  WHERE EXISTS (SELECT * FROM film
3                  WHERE film.language_id = language.language_id)
4
```

The result grid shows the following data:

	language_id	name	last_update
▶	1	English	2006-02-15 05:02:19
*	NULL	NULL	NULL

Comme on peut le voir, je n'ai qu'un résultat car tous les films ne sont disponibles qu'en anglais dans la base de données.

```

UPDATE film
SET language_id = 2
WHERE film_id = 8

```

Je modifie un film dans la base de donnée pour changer la langue d'un film.

The screenshot shows the MySQL Workbench interface. The top tab bar has 'SQL File 7\*' and 'SQL File 8\*'. The toolbar includes icons for file operations, search, and database navigation. A dropdown menu says 'Limit to 1000 rows'. Below the toolbar is a query editor window containing:

```

1   SELECT * FROM language
2   WHERE EXISTS (SELECT * FROM film WHERE film.language_id = language.language_id)
3

```

Below the query editor is the 'Result Grid' pane, which displays the results of the query. The grid has columns: language\_id, name, and last\_update. The data is:

	language_id	name	last_update
▶	1	English	2006-02-15 05:02:19
	2	Italian	2006-02-15 05:02:19
*	HULL	NULL	NULL

Comme on peut le voir, il y a maintenant 2 résultats à ma requête, conséquence directe d'avoir la langue d'un des films de la base de données, il y a maintenant plusieurs langues qui existent dans la table "film".

Et si je souhaite connaître les langues qui n'ont pas de films disponibles dans la base de données ?

Et bien, la requête suivante répondra à cette question :

The screenshot shows the MySQL Workbench interface. The top tab bar has 'SQL File 7\*' and 'SQL File 8\*'. The toolbar includes icons for file operations, search, and database navigation. A dropdown menu says 'Limit to 1000 rows'. Below the toolbar is a query editor window containing:

```

1   SELECT * FROM language
2   WHERE NOT EXISTS (SELECT * FROM film WHERE film.language_id = language.language_id)
3

```

Below the query editor is the 'Result Grid' pane, which displays the results of the query. The grid has columns: language\_id, name, and last\_update. The data is:

	language_id	name	last_update
▶	3	Japanese	2006-02-15 05:02:19
	4	Mandarin	2006-02-15 05:02:19
	5	French	2006-02-15 05:02:19
*	6	German	2006-02-15 05:02:19
*	HULL	NULL	NULL

Cette requête me renvoie toutes les langues dont l'id n'existe pas dans la table film, par conséquent, je connais désormais les langues non disponibles dans les films.

# SQL & MIN & MAX, COUNT, AVG, SUM

@Angelika

Les fonctions d'agrégation MIN (), MAX (), COUNT (), AVG(), SUM () : ce sont des opérations statistiques sur un ensemble d'enregistrements  
Les principales fonctions sont les suivantes :

- **MIN()** pour récupérer la valeur minimum de la même manière que MAX()

Elle respecte la syntaxe suivante:

```
SELECT MIN(nom_colonne) FROM table
```

Exemple: Pour repérer la valeur minimale de la colonne longueur ("length") de la table film

```
select min(length) from film
```

	min(length)
▶	46

- **MAX()** pour récupérer la valeur maximum d'une colonne sur un ensemble de lignes. Cela s'applique à la fois pour des données numériques ou alphanumériques.

Elle respecte la syntaxe suivante:

```
SELECT MAX(nom_colonne) FROM table
```

Lorsque cette fonctionnalité est utilisée en association avec la commande GROUP BY, la requête peut ressembler à l'exemple ci-dessous:

```
SELECT colonne1, MAX(colonne2)
FROM table
GROUP BY colonne1
```

Exemple: Pour repérer la valeur maximale de la colonne longueur ("length") de la table film

```
select max(length) from film
```

	max(length)
▶	185

- **COUNT()** pour compter le nombre d'enregistrements sur une table ou une colonne distincte.  
Pour connaître le nombre de lignes totales dans une table, il suffit d'effectuer la requête SQL suivante : `SELECT COUNT(*) FROM table`

Elle respecte la syntaxe suivante:

```
SELECT COUNT(nom_colonne) FROM table
```

Exemple: Pour compter le nombre de titres (“title”) de la table “film”

```
select count(title) from film
```

	count(title)
▶	1000

- **AVG()** pour calculer la moyenne sur un ensemble d'enregistrements.

Elle respecte la syntaxe suivante:

```
SELECT AVG(nom_colonne) FROM nom_table
```

Exemple: Pour avoir la moyenne des montants (“amount”) de la table “payment”

```
select avg(amount) from payment
```

	avg(amount)
▶	4.201356

- **SUM()** pour calculer la somme sur un ensemble d'enregistrements. Cette fonction ne fonctionne que sur des colonnes de types numériques (INT, FLOAT ...) et n'additionne pas les valeurs NULL.

Elle respecte la syntaxe suivante:

```
SELECT SUM(nom_colonne)  
FROM table
```

Exemple: Pour calculer la somme de la colonne des montants (“amount”) de la table “payment”

```
select sum(amount) from payment
```

	sum(amount)
▶	67406.56

## SQL HAVING

La condition **HAVING** en SQL est presque similaire à **WHERE** à la seule différence que HAVING permet de filtrer en utilisant des fonctions telles que **SUM()**, **COUNT()**, **AVG()**, **MIN()** ou **MAX()**.

L'utilisation de **HAVING** s'utilise de la manière suivante :

```
SELECT colonne1, SUM(colonne2)
      FROM nom_table
      GROUP BY colonne1
      HAVING fonction(colonne2) operateur valeur
```

Cela permet donc de SÉLECTIONNER les colonnes DE la table “**nom\_table**” en GROUPANT les lignes qui ont des valeurs identiques sur la colonne “**colonne1**” et que la condition de **HAVING** soit respectée.

La commande Having est quasiment tout le temps utilisée avec la commande **GROUP BY** bien que ce ne soit pas obligatoire.

id	client	tarif	date_achat
1	Pierre	102	2012-10-23
2	Simon	47	2012-10-27
3	Marie	18	2012-11-05
4	Marie	20	2012-11-14
5	Pierre	160	2012-12-03

Si dans cette table on souhaite récupérer la liste des clients qui ont commandé plus de 40€, toute commandes confondu alors il est possible d'utiliser la requête suivante :

```
SELECT client, SUM(tarif)
      FROM achat
      GROUP BY client
      HAVING SUM(tarif) > 40
```

client	SUM(tarif)
Pierre	262
Simon	47

La cliente "Marie" a cumulée 38€ d'achat (un achat de 18€ et un autre de 20€) ce qui est inférieur à la limite de 40€ imposée par **HAVING**. En conséquence, cette ligne n'est pas affichée dans le résultat.

## GROUP BY

La commande **GROUP BY** est utilisée en SQL pour grouper plusieurs résultats et utiliser une fonction de totaux sur un groupe de résultat. Sur une table qui contient toutes les ventes d'un magasin, il est par exemple possible de regrouper les ventes par clients identiques et d'obtenir le coût total des achats pour chaque client.

De façon générale, la commande **GROUP BY** s'utilise de la façon suivant:

```
SELECT colonne1, fonction(colonne2)
      FROM table
      GROUP BY colonne1
```

Cette commande doit toujours s'utiliser après la commande **WHERE** et avant la commande **HAVING**.

Prenons en considération une table "**achat**" qui résume les ventes d'une boutique :

id	client	tarif	date
1	Pierre	102	2012-10-23
2	Simon	47	2012-10-27
3	Marie	18	2012-11-05
4	Marie	20	2012-11-14
5	Pierre	160	2012-12-03

Ce tableau contient une colonne qui sert d'identifiant pour chaque ligne, une autre qui contient le nom du client, le coût de la vente et la date d'achat.

Pour obtenir le coût total de chaque client en regroupant les commandes des mêmes clients, il faut utiliser la requête suivante :

```
SELECT client, SUM(tarif)
      FROM achat
      GROUP BY client
```

La fonction **SUM()** permet d'additionner la valeur de chaque tarif pour un même client. Le résultat sera donc le suivant :

client	SUM(tarif)
Pierre	262
Simon	47
Marie	38

La manière simple de comprendre le **GROUP BY** c'est tout simplement d'assimiler qu'il va éviter de présenter plusieurs fois les mêmes lignes. C'est une méthode pour éviter les doublons.

Juste à titre informatif, voici ce qu'on obtient de la requête sans utiliser **GROUP BY**.

La requête est donc :

```
SELECT client, SUM(tarif)
      FROM achat
```

Voici donc le résultat :

client	SUM(tarif)
Pierre	262
Simon	47
Marie	38
Marie	38
Pierre	262

# SQL LIKE & UNION

@Lisa

## L'opérateur “ LIKE ”

L'opérateur **LIKE** est utilisé dans la clause **WHERE** des requêtes SQL. Ce mot-clé permet d'effectuer une recherche sur un modèle particulier. Il est par exemple possible de rechercher les enregistrements dont la valeur d'une colonne commence par telle ou telle lettre. Les modèles de recherche sont multiples.

Sa syntaxe est la suivante :

```
1 •      SELECT * FROM film WHERE title LIKE "F%"
```

Cela renvoi :

	film_id	title	description	release_year	language_id	or
▶	299	FACTORY DRAGON	A Action-Packed Saga of a Teacher And a Frisb...	2006	1	NUL
	300	FALCON VOLUME	A Fateful Saga of a Sumo Wrestler And a Hunte...	2006	1	NUL
	301	FAMILY SWEET	A Epic Documentary of a Teacher And a Bov wh...	2006	1	NUL

Ici nous avons utiliser le modèle “a%” mais il en existe bien d'autres :

- **LIKE “%a”**
  - Ceci trouve les mots finissant par “a” (ex : Félinbra).
- **LIKE “a%”**
  - Ceci trouve les mots commençant par “a” (ex : Amande).
- **LIKE “%a%”**
  - Ceci trouve les mots contenant la lettre “a” (ex : Rigolade).
- **LIKE “pr%o”**
  - Ceci trouve les mots qui commencent par “pr” et finissent par ‘o’ (ex : Promo).
- **LIKE “a\_c”**
  - Ceci trouve les mots qui commencent par “a” et fini par “c” avec le “\_” remplacer par une lettre (ex : Aac, Azc,etc...). Le caractère “\_” (underscore) peut être remplacé par n'importe quel caractère, mais un seul caractère uniquement (alors que le symbole pourcentage “%” peut être remplacé par un nombre incalculable de caractères .
- **LIKE “\_a%”**
  - Ceci trouve les mots qui possèdent la lettre “a” en 2e position (ex : Mana, Banane).
- **LIKE “a\_\_%”**
  - Ceci trouve les mots qui commencent par “a” et qui possède minimum 3 caractères (ex : Ana, Any).
- **LIKE “[azk]%”**

- Ceci trouve les mots commençant par les lettres dans les crochets (ex : Amandine, Zekrom, Koala).
- **LIKE “[a-c]%"**
  - Ceci trouve les mot commençant par les lettres de “a” à “c” (ex : Amande, Banane, Clémentine).
- **LIKE “[^abc]%"**
  - Ceci trouve tous les mots sauf ceux qui commencent par les lettres mise après le “^” (ex : Zidane, François, Kayla). L'icône “^” sert à exclure.

## L'opérateur “**UNION**”

L'opérateur **UNION** est utilisé pour combiner les résultats de deux déclarations **SELECT** ou plus.

- Chaque instruction **SELECT** doit avoir le même nombre de colonnes,
- Les colonnes doivent également avoir des types de données similaires,
- Les colonnes de chaque instruction **SELECT** doivent également être dans le même ordre

Il y a un effet équivalent aux **DISTINCT**, c'est-à-dire que l'opérateur enlève naturellement les doublons durant la fusion, si ce n'est pas le résultat escompté il faudra dans ce cas là utiliser l'opérateur **UNION ALL**.

Exemple:

La table du magasin n°1 s'appelle “magasin1\_client” et contient les données suivantes :

prenom	nom	ville	date_naissance	total_achat
Léon	Dupuis	Paris	1983-03-06	135
Marie	Bernard	Paris	1993-07-03	75
Sophie	Dupond	Marseille	1986-02-22	27
Marcel	Martin	Paris	1976-11-24	39

La table du magasin n°2 s'appelle “magasin2\_client” et contient les données suivantes :

prenom	nom	ville	date_naissance	total_achat
Marion	Leroy	Lyon	1982-10-27	285
Paul	Moreau	Lyon	1976-04-19	133
Marie	Bernard	Paris	1993-07-03	75
Marcel	Martin	Paris	1976-11-24	39

```
SELECT * FROM magasin1_client
```

```
UNION
```

```
SELECT * FROM magasin2_client
```

Ce qui donne :<sup>1</sup>

prenom	nom	ville	date_naissance	total_achat
Léon	Dupuis	Paris	1983-03-06	135
Marie	Bernard	Paris	1993-07-03	75
Sophie	Dupond	Marseille	1986-02-22	27
Marcel	Martin	Paris	1976-11-24	39
Marie	Leroy	Lyon	1982-10-27	285
Paul	Moreay	Lyon	1976-04-19	133

Pour UNION ALL cela donnerait cette syntaxe avec ce résultat :

```
SELECT * FROM magasin1_client
```

```
UNION ALL
```

```
SELECT * FROM magasin2_client
```

prenom	nom	ville	date_naissance	total_achat
Léon	Dupuis	Paris	1983-03-06	135
Marie	Bernard	Paris	1993-07-03	75
Sophie	Dupond	Marseille	1986-02-22	27
Marcel	Martin	Paris	1976-11-24	39
Marion	Leroy	Lyon	1982-10-27	285
Paul	Moreau	Lyon	1976-04-19	133
Marie	Bernard	Paris	1993-07-03	75
Marcel	Martin	Paris	1976-11-24	39

Nous voyons bien un doublon avec la méthode UNION ALL

# SQL WILDCARDS

@Zuzanna

```
SELECT * FROM film  
WHERE title LIKE 'z%';
```

Sélectionne toutes les colonnes de la table film de la base de donnée sakila où le title commence par un z.

```
SELECT * FROM film  
WHERE title LIKE '%a';
```

Sélectionne toutes les colonnes de la table film de la base de donnée sakila où le title fini par un a.

```
SELECT * FROM film  
WHERE title LIKE '_a%';
```

Sélectionne toutes les colonnes de la table film de la base de donnée sakila où le deuxième caractère de title commence par un a. Le \_ représente un seul caractère “joker”.

```
SELECT * FROM film  
WHERE title LIKE '[bsp]%;'
```

Sélectionne toutes les colonnes de la table films où la première lettre de la colonne title est soit un b soit un s soit un p.

(fonctionne sur W3school mais pas sur le workbench)

```
SELECT * FROM film  
WHERE (title LIKE 'a%'  
      OR title LIKE 'c%'  
      OR title LIKE 's%');
```

Sélectionne toutes les colonnes de la table Customers où la première lettre de la colonne City est soit un a soit un c soit un s.

## Wildcard Characters in SQL Server

Symbol	Description	Example
%	Represents zero or more characters <b>Représente zéro ou plusieurs caractères</b>	bl% finds bl, black, blue, and blob
-	Represents a single character <b>Représente un seul caractère</b>	h_t finds hot, hat, and hit
[ ]	Represents any single character within the brackets <b>Représente n'importe quel caractère entre crochets</b>	h[oa]t finds hot and hat, but not hit
^	Represents any character not in the brackets <b>Représente tout caractère qui n'est pas entre parenthèses</b>	h[^oa]t finds hit, but not hot and hat
-	Represents any single character within the specified range <b>Représente n'importe quel caractère unique dans la plage spécifiée</b>	c[a-b]t finds cat and cbt

# SQL IN & BETWEEN

@Marie-Claire

Le mot **IN**

```
SELECT * FROM actor  
WHERE last_name IN ('WILLIAMS', 'WINSLET', 'DAVIS');
```

Sélectionne toutes les colonnes (actor\_id, first\_name, last\_name, last\_update) de la table acteur de la base de donnée sakila où last\_name prend la valeur de WILLIAMS , WINSLET ou DAVIS

équivalent/raccourci de

```
SELECT * FROM actor  
  
WHERE last_name = "DAVIS" OR last_name = "WINSLET" OR last_name = "WILLIAMS";
```

le mot **BETWEEN**

exemple pour les nombres:

```
SELECT * FROM payment  
  
WHERE amount BETWEEN 2 AND 4;
```

Sélectionne toutes les colonnes (payment\_id, customer\_id, staff\_id, rental\_id, amount, payment\_date, last\_update) de la table payment de la base de donnée sakila dont la valeur de amount est comprise entre 2 et 4

exemple pour les string

```
SELECT * FROM film_text  
  
WHERE title BETWEEN "j" AND "m";
```

ou

```
SELECT * FROM film_text  
  
WHERE title BETWEEN "J" AND "M";
```

Sélectionne toutes les colonnes (film\_id, title, et description) de la table film\_text de la base de donnée sakila dont le titre est compris entre J' inclus et M exclus par ordre alphabétique.

# SQL SELF JOIN & FULL JOIN

@Bessem

## SELF JOIN

un **SELF JOIN** correspond à une jointure d'une table avec elle-même. Ce type de requête n'est pas si commun mais très pratique dans le cas où une table lie des informations avec des enregistrements de la même table.

```
SELECT * from rental AS R1, rental AS R2  
WHERE R1.staff_id = R2.staff_id
```

un **SELF JOIN** correspond à une jointure d'une table avec elle-même.

```
SELECT p1.customer_id, p1.staff_id, p2.customer_id, p2.staff_id  
FROM payment AS p1  
LEFT OUTER JOIN payment AS p2 ON p2.customer_id = p1.customer_id
```

## FULL JOIN

La commande **FULL JOIN** (ou **FULL OUTER JOIN**) permet de faire une jointure entre 2 tables. L'utilisation de cette commande permet de combiner les résultats des 2 tables, les associer entre eux grâce à une condition et remplir avec des valeurs NULL si la condition n'est pas respectée.

```
SELECT film_id, title,  
FROM film  
FULL JOIN film_actor ON film.film_id = film_actor.film_id;
```

```
SELECT *  
FROM customer  
FULL JOIN payment ON customer.customer_id = payment.customer_id;
```

```
SELECT * FROM customer AS C  
LEFT JOIN payment AS P ON C.customer_id = P.customer_id  
UNION ALL  
SELECT * FROM customer AS C  
RIGHT JOIN payment AS P ON C.customer_id = P.customer_id  
WHERE C.customer_id IS NULL OR P.customer_id IS NULL;
```

# SQL ALIAS & JOINS

@Laurie

## → ALIAS

Permet de renommer le nom d'une colonne dans les résultats d'une requête SQL. C'est pratique pour avoir un nom facilement identifiable dans une application qui doit ensuite exploiter les résultats d'une recherche.

### Comment l'utiliser ?

Pour cela il suffit d'utiliser AS

### Exemple :

Dans cet exemple on sélectionne la colonne 'title' de la table 'film' que l'on renomme 'titre' avec AS

The screenshot shows a database interface with a query editor and a result grid. The query editor contains the following SQL code:

```
1 •  SELECT title AS titre FROM film;
```

The result grid displays the following data:

titre
ACADEMY DINOSAUR
ACE GOLDFINGER
ADAPTATION HOLES
AFFAIR PREJUDICE
AFRICAN EGG
AGENT TRUMAN
AIRPLANE SIERRA
AIRPORT POLLOCK
ALABAMA DEVIL
ALADDIN CALENDAR

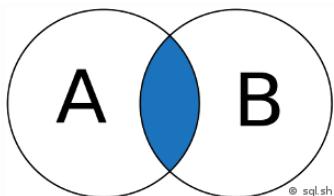
On the right side of the interface, there is a sidebar with three icons: 'Result Grid' (selected), 'Form Editor', and 'Field Types'.

## → JOINS

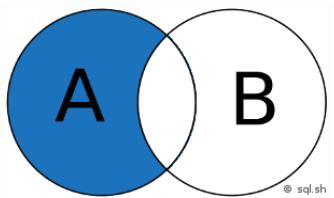
Les jointures permettent d'associer plusieurs tables dans une même requête. Cela permet d'obtenir des résultats qui combinent les données de plusieurs tables.

→ Il existe plusieurs types de jointure :

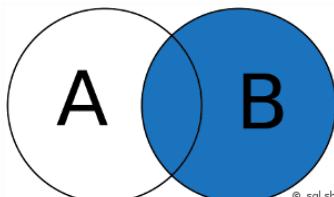
- ◆ **INNER JOIN** : Retourne les valeurs dont la condition est vraie dans les 2 tables.



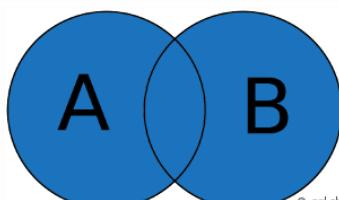
- ◆ **LEFT JOIN** : Retourne toutes les valeurs de la table gauche même si la condition n'est pas vérifiée dans l'autre table.



- ◆ **RIGHT JOIN** : Retourne toutes les valeurs de la table droite même si la condition n'est pas vérifiée dans l'autre table.



- ◆ **FULL JOIN** : Retourne les valeurs dont la condition est vraie dans au moins une des deux tables.



```
25 -- JOIN
26
27 -- JE DEMANDE DE RETOURNER TT LES NOMS ET LEURS ID DANS LA TABLE actor
28 -- grace a la clef commune aux tables actor et film_actor
29 • SELECT actor.last_name, actor.actor_id
30 FROM actor
31 JOIN film_actor
32 ON actor.actor_id = film_actor.actor_id;
33
34
```

The screenshot shows the MySQL Workbench interface with a query editor and a result grid. The query editor contains the code above. The result grid shows the output of the query:

last_name	actor_id
AKROYD	58

Result 8 ×

```
21 • SELECT first_name AS "Prénom",
22      last_name AS "Nom de famille"
23      FROM actor;
24
25
26
27
```

The screenshot shows the MySQL Workbench interface with a query editor and a result grid. The query editor contains the code above. The result grid shows the output of the query:

Prénom	Nom de famille
PENELOPE	GUINNESS
NICK	WAHLBERG
ED	CHASE
JENNIFER	DAVIS
JOHNNY	LOLLOBRIGIDA
BETTE	NICHOLSON
GRACE	MOSTEL
MATTHEW	JOHANSSON
JOE	SWANK
CHRISTIAN	GABLE
ZERO	CAGE
MONIQUE	SEAGROVE

actor 6 ×

Output

# SQL CASE & CREATE-AS SELECT

@Benoît

La commande **CASE** passe par des conditions et renvoie une valeur lorsqu'une condition est remplie (exactement comme une instruction **if else**). Ainsi, une fois qu'une condition est vraie, elle arrête de lire et renvoie le résultat. Si aucune condition n'est vraie, elle renvoie la valeur dans la commande **ELSE**.

S'il n'y a pas de clause **ELSE** et qu'aucune condition n'est vraie, elle renvoie **NULL**.

- Afficher un message selon une condition

Il est possible d'effectuer une requête qui va afficher un message personnalisé en fonction de la valeur de la marge.

```
1 •  SELECT film_id, title, description, release_year,
2   CASE
3     WHEN rental_duration>4 THEN 'Retard'
4     ELSE 'Tout va bien'
5   END
6   FROM film;
7
```

Result Grid   Filter Rows:  Search   Export: Fetch rows:

film_id	title	description	release_year	CASE WHEN rental_duration>4 THEN 'Retard' ELSE 'Tout va bien'
2	Alice Goldfringen	A Astounding Episite of a Database Administrat...	2006	Tout va bien
3	ADAPTATION HOLES	A Astounding Reflection of a Lumberjack And a...	2006	Retard
4	AFFAIR PREJUDICE	A Fanciful Documentary of a Frisbee And a Lum...	2006	Retard
5	AFRICAN EGG	A Fast-Paced Documentary of a Pastry Chef An...	2006	Retard
6	AGENT TRUMAN	A Intrepid Panorama of a Robot And a Boy who...	2006	Tout va bien
7	AIRPLANE SIERRA	A Touching Saga of a Hunter And a Butler who...	2006	Retard
8	AIRPORT POLLOCK	A Epic Tale of a Moose And a Girl who must Co...	2006	Retard
9	ALABAMA DEVIL	A Thoughtful Panorama of a Database Administ...	2006	Tout va bien

- Élaborer une série de conditions booléennes pour déterminer un résultat

```
1 •  SELECT film_id, title, description, release_year,
2   CASE
3     WHEN rental_duration=4 THEN 'Ok'
4     WHEN rental_duration=6 THEN 'Retard'
5     WHEN rental_duration=8 THEN 'Pénalités'
6   END
7   FROM film;
8
```

Result Grid   Filter Rows:  Search   Export: Fetch rows:

film_id	title	description	release_year	CASE WHEN rental_duration=4 THEN 'Ok' WHEN rental_duration=6 THEN 'Retard' WHEN rental_duration=8 THEN 'Pénalités'
14	ALICE FANTASIA	A Emotional Drama of a A Shark And a Databas...	2006	Retard
15	ALIEN CENTER	A Brilliant Drama of a Cat And a Mad Scientist w...	2006	NULL
16	ALLEY EVOLUTION	A Fast-Paced Drama of a Robot And a Compos...	2006	Retard
17	ALONE TRIP	A Fast-Paced Character Study of a Composer A...	2006	NULL
18	ALTER VICTORY	A Thoughtful Drama of a Composer And a Femili...	2006	Retard
19	AMADEUS HOLY	A Emotional Display of a Pioneer And a Technic...	2006	Retard
20	AMELIE HELLFIGHT...	A Boring Drama of a Woman And a Squirrel wh...	2006	Ok
21	AMERICAN CIRCUS	A Insightful Drama of a Girl And a Astronaut wh...	2006	NULL

- Afficher un prix unitaire différent selon une condition

```

1 •  SELECT film_id, title, rental_rate,
2   CASE
3     WHEN rental_duration=6 THEN rental_rate * 2
4     WHEN rental_duration=8 THEN rental_rate * 10
5   END
6   FROM film;
7

```

100% 36:1

**Result Grid** Filter Rows: Search Export: Fetch rows:

film_id	title	rental_rate	CASE WHEN rental_duration=6 THEN rental_rate...
5	AFRICAN EGG	2.99	5.98
6	AGENT TRUMAN	2.99	NULL
7	AIRPLANE SIERRA	4.99	9.98
8	AIRPORT POLLOCK	4.99	9.98
9	ALABAMA DEVIL	2.99	NULL
10	ALADDIN CALENDAR	4.99	9.98
11	ALAMO VIDEOTAPE	0.99	1.98
12	ALASKA PHANTOM	0.99	1.98
13	ALL FOREVER	4.99	NULL

- Comparer un champ à une valeur donnée

```

1 •  SELECT film_id, title, rental_duration,
2   CASE rental_duration
3     WHEN 3 THEN 'OK'
4     WHEN 6 THEN 'Retard'
5     WHEN 8 THEN 'Gros retard'
6   END
7   FROM film;
8

```

100% 39:1

**Result Grid** Filter Rows: Search Export: Fetch rows:

film_id	title	rental_duration	CASE rental_duration WHEN 3 THEN 'OK'
2	ACE GOLDFINGER	3	OK
3	ADAPTATION HOLES	7	NULL
4	AFFAIR PREJUDICE	5	NULL
5	AFRICAN EGG	6	Retard
6	AGENT TRUMAN	3	OK
7	AIRPLANE SIERRA	6	Retard
8	AIRPORT POLLOCK	6	Retard
9	ALABAMA DEVIL	2	OK

## CREATE TABLE

La commande CREATE TABLE permet de créer une table en SQL.

La syntaxe générale pour créer une table est la suivante :

```
CREATE TABLE nom_de_la_table
(
    colonne1 type_donnees,
    colonne2 type_donnees,
    colonne3 type_donnees,
    colonne4 type_donnees
)
```

Le mot-clé “type\_donnees” sera à remplacer par un mot-clé pour définir le type de données (INT, DATE, TEXT ...). Pour chaque colonne, il est également possible de définir des options telles que (liste non-exhaustive):

- **NOT NULL** : empêche d'enregistrer une valeur nulle pour une colonne.
- **DEFAULT** : attribuer une valeur par défaut si aucune données n'est indiquée pour cette colonne lors de l'ajout d'une ligne dans la table.
- **PRIMARY KEY** : indiquer si cette colonne est considérée comme clé primaire pour un index.

```
40   --
41   -- Table structure for table `actor`
42   --
43
44 • ⊞ CREATE TABLE actor (
45     actor_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
46     first_name VARCHAR(45) NOT NULL,
47     last_name VARCHAR(45) NOT NULL,
48     last_update TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
49     PRIMARY KEY (actor_id),
50     KEY idx_actor_last_name (last_name)
51 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
52
53 --
```

- **actor\_id** : id unique avec un type SMALLINT qui est utilisé comme clé primaire et qui n'est pas nulle
- **first\_name** : colonne prénom avec un type chaîne de caractères limité à 45 caractères et qui n'est pas nulle
- **last\_name** : idem mais pour le nom
- **last\_update**
- **PRIMARY KEY raccorde la clef primaire à la colonne actor\_id**
- **KEY**

## CREATE TABLE - SELECT AS

Créer une table à l'aide d'une autre table

La nouvelle table obtient les mêmes définitions de colonne. Toutes les colonnes ou des colonnes spécifiques peuvent être sélectionnées.

Si vous créez une nouvelle table à l'aide d'une table existante, la nouvelle table sera remplie avec les valeurs existantes de l'ancienne table.

The screenshot shows the MySQL Workbench interface. On the left, the 'SCHEMAS' tree view is open, showing various database objects like film\_category, film\_text, inventory, language, payment, rental, staff, store, and a newly created table 'Table\_Name'. The 'Columns' node under 'Table\_Name' is expanded, showing columns title and description. On the right, the SQL editor contains the following code:

```
1 • CREATE TABLE Table_Name AS SELECT title, description FROM film;
```

Le SQL suivant crée une nouvelle table appelée "BackUpFilm" (qui est une copie de la table "film") :

The screenshot shows the MySQL Workbench interface. On the left, the 'SCHEMAS' tree view is open, showing the 'BackUpFilm' table. The 'Columns' node under 'BackUpFilm' is expanded, listing columns: film\_id, title, description, release\_year, language\_id, original\_langu..., rental\_duration, rental\_rate, and length. On the right, the SQL editor contains the following code:

```
1 • CREATE TABLE BackUpFilm AS SELECT * FROM film;
```

# INNER JOIN & LEFT JOIN & RIGHT JOIN

@James

Avec **INNER JOIN**, ON RETOURNE LES ENREGISTREMENTS DE 2 TABLES QUI CORRESPONDENT À CHACUNES DES 2 TABLES.

**SELECT**

```
customer.customer_id,  
first_name,  
last_name,  
email,  
amount,  
payment_date
```

**FROM** customer

**INNER JOIN** payment **ON** payment.customer\_id = customer.customer\_id;

```
10 -- INNER JOIN & LEFT JOIN & RIGHT JOIN  
11 -- AVEC INNER JOIN , ON RETOURNE LES ENREGISTREMENTS DE 2 TABLES QUI CORRESPONDENT A CHACUNES DES 2 TABLES.  
12 • SELECT  
13   customer.customer_id,  
14   first_name,  
15   last_name,  
16   email,  
17   amount,  
18   payment_date  
19   FROM customer  
20   INNER JOIN payment ON payment.customer_id = customer.customer_id
```

The screenshot shows a database query results grid. The query joins the 'customer' table with the 'payment' table based on their primary keys. The resulting grid has columns: customer\_id, first\_name, last\_name, email, amount, and payment\_date. There are 26 rows returned, all corresponding to a single customer record (customer\_id 1). The data shows multiple payments made by 'MARY SMITH' at different dates and amounts.

customer_id	first_name	last_name	email	amount	payment_date
1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	2.99	2005-05-25 11:30:37
1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	0.99	2005-05-28 10:35:23
1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	5.99	2005-06-15 00:54:12
1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	0.99	2005-06-15 18:02:53
1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	9.99	2005-06-15 21:08:46
1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	4.99	2005-06-16 15:18:57
1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	4.99	2005-06-18 08:41:48
1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	0.99	2005-06-18 13:33:59
1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	3.99	2005-06-21 06:24:45
1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	5.99	2005-07-08 03:17:05

```

25
26    -- AVEC RIGHT JOIN , ON PRENDS DES ELEMENTS DE LA TABLE DE DROITE (actor)
27    -- ET ON RAJOUTE DES ELEMENTS DE LA TABLE DE GAUCHE (customer)
28
29    -- SELECT c.customer_id,
30    --     c.first_name,
31    --     c.last_name,
32    --     a.actor_id,
33    --     a.first_name,
34    --     a.last_name
35    -- FROM customer c
36    -- RIGHT JOIN actor a
37    -- ON c.last_name = a.last_name;
38

```

The screenshot shows a database query results window. At the top, there is a code editor with the SQL query. Below it is a result grid table with the following columns: customer\_id, first\_name, last\_name, email, amount, and payment\_date. The table contains 10 rows of data for a customer named MARY SMITH. The sidebar on the right has three tabs: 'Result Grid' (which is selected), 'Form Editor', and 'Field Types'.

	customer_id	first_name	last_name	email	amount	payment_date
▶	1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	2.99	2005-05-25 11:30:37
	1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	0.99	2005-05-28 10:35:23
	1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	5.99	2005-06-15 00:54:12
	1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	0.99	2005-06-15 18:02:53
	1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	9.99	2005-06-15 21:08:46
	1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	4.99	2005-06-16 15:18:57
	1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	4.99	2005-06-18 08:41:48
	1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	0.99	2005-06-18 13:33:59
	1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	3.99	2005-06-21 06:24:45
◀	MARY	SMITH	MARY.SMITH@sakilacustomer.org	5.00	2005-07-08 02:17:05	

Avec **RIGHT JOIN**, on prends des éléments de la table de droite (actor) et on rajoute des éléments de la table de gauche (customer)

```

SELECT c.customer_id,
       c.first_name,
       c.last_name,
       a.actor_id,
       a.first_name,
       a.last_name

```

```

FROM customer c
RIGHT JOIN actor a
ON c.last_name = a.last_name;

```

The screenshot shows the MySQL Workbench interface. In the top panel, there is a query editor window with the following SQL code:

```
41 -- AVEC LEFT JOIN , ON PRENDS DES ELEMENTS DE LA TABLE DE GAUCHE (actor)
42 -- ET ON RAJOUTE DES ELEMENTS DE LA TABLE DE DROITE (customer)
43
44 • SELECT
45   film.film_id,
46   film.title,
47   inventory.inventory_id
48
49   FROM film
50   LEFT JOIN inventory ON inventory.film_id = film.film_id;
51
52
53
54
```

Below the query editor is a result grid window displaying the output of the query. The grid has three columns: film\_id, title, and inventory\_id. The data is as follows:

film_id	title	inventory_id
1	ACADEMY DINOSAUR	1
1	ACADEMY DINOSAUR	2
1	ACADEMY DINOSAUR	3
1	ACADEMY DINOSAUR	4
1	ACADEMY DINOSAUR	5
1	ACADEMY DINOSAUR	6
1	ACADEMY DINOSAUR	7
1	ACADEMY DINOSAUR	8
2	ACE GOLDFINGER	9
2	ACE GOLDFINGER	10
2	ACE GOLDFINGER	11
3	ADAPTATION HOLES	12

On the right side of the interface, there is a vertical toolbar with icons for Result Grid, Form Editor, and Field Types.

AVEC LEFT JOIN , ON PRENDS DES ÉLÉMENTS DE LA TABLE DE GAUCHE (film)  
ET ON RAJOUTE DES ÉLÉMENTS DE LA TABLE DE DROITE (inventory)

```
SELECT film.film_id,
       film.title,
       inventory.inventory_id

  FROM film
LEFT JOIN inventory ON inventory.film_id = film.film_id;
```