eBook Gratuit

APPRENEZ SQL

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#sql

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: sql

It is an unofficial and free SQL ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official SQL.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec SQL

Remarques

SQL est un langage de requête structuré utilisé pour gérer les données dans un système de base de données relationnelle. Différents fournisseurs ont amélioré la langue et ont une variété de saveurs pour la langue.

NB: Cette balise fait explicitement référence au **standard SQL ISO / ANSI** ; pas à une mise en œuvre spécifique de cette norme.

Versions

Version	Nom court	la norme	Date de sortie
1986	SQL-86	ANSI X3.135-1986, ISO 9075: 1987	1986-01-01
1989	SQL-89	ANSI X3.135-1989, ISO / IEC 9075: 1989	1989-01-01
1992	SQL-92	ISO / IEC 9075: 1992	1992-01-01
1999	SQL: 1999	ISO / IEC 9075: 1999	1999-12-16
2003	SQL: 2003	ISO / IEC 9075: 2003	2003-12-15
2006	SQL: 2006	ISO / IEC 9075: 2006	2006-06-01
2008	SQL: 2008	ISO / IEC 9075: 2008	2008-07-15
2011	SQL: 2011	ISO / IEC 9075: 2011	2011-12-15
2016	SQL: 2016	ISO / IEC 9075: 2016	2016-12-01

Examples

Vue d'ensemble

Le langage de requête structuré (SQL) est un langage de programmation spécifique conçu pour gérer les données contenues dans un système de gestion de base de données relationnelle (SGBDR). Les langages de type SQL peuvent également être utilisés dans les systèmes de gestion de flux de données relationnels (RDSMS) ou dans les bases de données "non-SQL" (NoSQL).

SQL comprend 3 sous-langues principales:

1. Langage de définition de données (DDL): pour créer et modifier la structure de la base de données;
2. Langage de manipulation de données (DML): pour effectuer des opérations de lecture, d'insertion, de mise à jour et de suppression sur les données de la base de données;
3. Data Control Language (DCL): pour contrôler l'accès aux données stockées dans la base de données.

Article SQL sur Wikipedia

Les opérations DML principales sont Create, Read, Update et Delete (CRUD pour faire court) qui sont exécutées par les instructions `INSERT` , `SELECT` , `UPDATE` et `DELETE` .
Il existe également une instruction `MERGE` (récemment ajoutée) qui peut exécuter les trois opérations d'écriture (INSERT, UPDATE, DELETE).

Article du CRUD sur Wikipedia

De nombreuses bases de données SQL sont implémentées en tant que systèmes client / serveur. le terme "serveur SQL" décrit une telle base de données.
Dans le même temps, Microsoft crée une base de données nommée "SQL Server". Bien que cette base de données parle un langage SQL, les informations spécifiques à cette base de données ne figurent pas dans cette balise, mais appartiennent à la documentation de SQL Server .

Lire Démarrer avec SQL en ligne: https://riptutorial.com/fr/sql/topic/184/demarrer-avec-sql

Chapitre 2: Algèbre relationnelle

Examples

Vue d'ensemble

L'algèbre relationnelle n'est pas un langage *SQL* complet, mais plutôt un moyen d'acquérir une compréhension théorique du traitement relationnel. En tant que tel, il ne devrait pas faire référence à des entités physiques telles que des tables, des enregistrements et des champs. il devrait faire référence à des constructions abstraites telles que des relations, des tuples et des attributs. En disant cela, je n'utiliserai pas les termes académiques dans ce document et je m'en tiendrai aux termes profanes plus connus - tableaux, enregistrements et champs.

Quelques règles de l'algèbre relationnelle avant que nous commencions:

- Les opérateurs utilisés dans l'algèbre relationnelle travaillent sur des tables entières plutôt que sur des enregistrements individuels.
- Le résultat d'une expression relationnelle sera toujours une table (cela s'appelle la *propriété de fermeture*).

Tout au long de ce document, je ferai référence aux deux tableaux suivants:

Departments

ID	Dept
1	Production
2	Quality Control

People

ID	PersonName	StartYear	ManagerID	DepartmentID
1	Darren	2005		1
2	David	2006	1	1
3	Burt	2006	1	1
4	Sarah	2004		2
5	Fred	2008	4	2
6	Joanne	2005	4	2

SÉLECTIONNER

L'opérateur **select** renvoie un sous-ensemble de la table principale.
sélectionnez *<table>* ***où*** *<condition>*

Par exemple, examinez l'expression:

sélectionnez Personnes **où** DepartmentID = 2

Cela peut être écrit comme:

$$\sigma_{\text{DepartmentID} = 2}(\text{People})$$

Cela se traduira par une table dont les enregistrements comprennent tous les enregistrements de

la table *People* où la valeur *DepartmentID* est égale à 2:

ID	PersonName	StartYear	ManagerID	DepartmentID
4	Sarah	2004		2
5	Fred	2008	4	2
6	Joanne	2005	4	2

Les conditions peuvent également être jointes pour restreindre davantage l'expression:

sélectionnez Personnes **où** StartYear> 2005 **et** DepartmentID = 2

se traduira par le tableau suivant:

ID	PersonName	StartYear	ManagerID	DepartmentID
5	Fred	2008	4	2

PROJET

L'opérateur de **projet** renverra des valeurs de champ distinctes à partir d'une table.
projet *<table>* ***sur*** *<liste de champs>*

Par exemple, examinez l'expression suivante:
projet People **over** StartYear

Cela peut être écrit comme:

$$\Pi_{\text{StartYear}}\,(\text{People})$$

Cela se traduira par une table comprenant les valeurs distinctes contenues dans le champ *StartYear* de la table *People* .

StartYear
2005
2006
2004
2008

Les valeurs en double sont supprimées de la table résultante en raison de la *propriété de fermeture* créant une table relationnelle: tous les enregistrements d'une table relationnelle doivent être distincts.

Si la *liste de champs* comprend plusieurs champs, la table résultante est une version distincte de ces champs.
projet People **over** StartYear, DepartmentID retournera:

StartYear	DepartmentID
2005	1
2006	1
2004	2
2008	2
2005	2

Un enregistrement est supprimé en raison de la duplication de 2006 *StartYear* et 1 *DepartmentID* .

DONNANT

Les expressions relationnelles peuvent être chaînées en nommant les expressions individuelles à l'aide du mot-clé **donnant** ou en intégrant une expression dans une autre.

<expression de l'algèbre relationnelle> ***donnant*** *<nom de l'alias>*

Par exemple, considérez les expressions suivantes:
sélectionnez Personnes **où** DepartmentID = 2 **donne** A
projet A **sur** PersonName **donnant** B

Cela se traduira par la table B ci-dessous, la table A étant le résultat de la première expression.

A

ID	PersonName	StartYear	ManagerID	DepartmentID
4	Sarah	2004		2
5	Fred	2008	4	2
6	Joanne	2005	4	2

B

PersonName
Sarah
Fred
Joanne

La première expression est évaluée et le tableau résultant reçoit l'alias A. Cette table est ensuite utilisée dans la seconde expression pour donner la table finale avec un alias de B.

Une autre façon d'écrire cette expression consiste à remplacer le nom d'alias de la table dans la seconde expression par le texte entier de la première expression entre crochets:
projet (**sélectionnez** People **où** DepartmentID = 2) **sur** PersonName en **donnant** B

Ceci est appelé une *expression imbriquée* .

NATURAL JOIN

Une jointure naturelle assemble deux tables en utilisant un champ commun partagé entre les tables.

joindre *<table 1>* ***et*** *<table 2>* ***où*** *<champ 1> = <champ 2>*
en supposant que <champ 1> est dans <table 1> et que <champ 2> est dans <table 2>.

Par exemple, l'expression de jointure suivante joint les *personnes* et les *départements en* fonction des colonnes *DepartmentID* et *ID* des tables respectives:

rejoindre les personnes **et les** départements **où** DepartmentID = ID

ID	PersonName	StartYear	ManagerID	DepartmentID	Dept
1	Darren	2005		1	Production
2	David	2006	1	1	Production
3	Burt	2006	1	1	Production
4	Sarah	2004		2	Quality Control
5	Fred	2008	4	2	Quality Control
6	Joanne	2005	4	2	Quality Control

Notez que seul le *DepartmentID* de la table *People* est affiché et non l' *ID* de la table *Department* . Un seul des champs comparés doit être affiché, ce qui correspond généralement au nom de la première table de l'opération de jointure.

Bien que cela ne soit pas illustré dans cet exemple, il est possible que la réunion de tables génère deux champs ayant le même en-tête. Par exemple, si j'avais utilisé l'en-tête *Nom* pour identifier les champs *PersonName* et *Dept* (c'est-à-dire pour identifier le nom de la personne et le nom du service). Lorsque cette situation se présente, nous utilisons le nom de la table pour qualifier les noms de champs en utilisant la notation par points: *People.Name* et *Departments.Name*

La jointure combinée avec **select** et **project** peut être utilisée ensemble pour extraire des informations:

rejoindre les *personnes* ***et les*** *départements* ***où*** *DepartmentID = ID* ***donnant*** *A*
sélectionnez *A* ***où*** *StartYear = 2005* ***et*** *Dept = 'Production'* ***donnant*** *B*
projet *B* ***sur*** *PersonName en* ***donnant*** *C*

ou comme expression combinée:

projet *(* ***sélectionnez*** *(* ***joindre les*** *personnes* ***et les*** *départements* ***où*** *DepartmentID = ID)* ***où*** *StartYear = 2005* ***et*** *Dept = 'Production')* ***sur*** *PersonName en* ***donnant*** *C*

Cela se traduira par cette table:

PersonName
Darren

ALIAS

DIVISER

SYNDICAT

INTERSECTION

DIFFÉRENCE

UPDATE (: =)

FOIS

Lire Algèbre relationnelle en ligne: https://riptutorial.com/fr/sql/topic/7311/algebre-relationnelle

Chapitre 3: ALTER TABLE

Introduction

La commande ALTER dans SQL est utilisée pour modifier la colonne / contrainte dans une table

Syntaxe

- ALTER TABLE [nom_table] ADD [nom_colonne] [type de données]

Examples

Ajouter une ou plusieurs colonnes

```
ALTER TABLE Employees
ADD StartingDate date NOT NULL DEFAULT GetDate(),
    DateOfBirth date NULL
```

L'instruction ci-dessus ajouterait des colonnes nommées `StartingDate` qui ne peuvent pas être NULL avec la valeur par défaut comme date du jour et `DateOfBirth` qui peut être NULL dans la table Employees .

Drop Column

```
ALTER TABLE Employees
DROP COLUMN salary;
```

Cela supprimera non seulement les informations de cette colonne, mais supprimera le salaire de la colonne des employés de la table (la colonne n'existera plus).

Drop Constraint

```
ALTER TABLE Employees
DROP CONSTRAINT DefaultSalary
```

Cela supprime une contrainte appelée DefaultSalary à partir de la définition de la table Employés.

Remarque: - Assurez **-** vous que les contraintes de la colonne sont supprimées avant de supprimer une colonne.

Ajouter une contrainte

```
ALTER TABLE Employees
ADD CONSTRAINT DefaultSalary DEFAULT ((100)) FOR [Salary]
```

Cela ajoute une contrainte appelée DefaultSalary qui spécifie une valeur par défaut de 100 pour la colonne Salary.

Une contrainte peut être ajoutée au niveau de la table.

Types de contraintes

- Clé primaire - empêche un enregistrement en double dans la table
- Clé étrangère - pointe sur une clé primaire d'une autre table
- Not Null - empêche l'entrée de valeurs NULL dans une colonne
- Unique - identifie de manière unique chaque enregistrement de la table
- Default - spécifie une valeur par défaut
- Check - limite les plages de valeurs pouvant être placées dans une colonne

Pour en savoir plus sur les contraintes, consultez la documentation Oracle .

Alter Column

```
ALTER TABLE Employees
ALTER COLUMN StartingDate DATETIME NOT NULL DEFAULT (GETDATE())
```

Cette requête modifie le type de données de la colonne `StartingDate` et le fait passer de la simple `date` à la `datetime` et définit la valeur par défaut sur la date actuelle.

Ajouter une clé primaire

```
ALTER TABLE EMPLOYEES ADD pk_EmployeeID PRIMARY KEY (ID)
```

Cela ajoutera une clé primaire à la table Employés sur le champ `ID` . L'inclusion de plus d'un nom de colonne entre parenthèses avec ID créera une clé primaire composite. Lors de l'ajout de plusieurs colonnes, les noms des colonnes doivent être séparés par des virgules.

```
ALTER TABLE EMPLOYEES ADD pk_EmployeeID PRIMARY KEY (ID, FName)
```

Lire ALTER TABLE en ligne: https://riptutorial.com/fr/sql/topic/356/alter-table

Chapitre 4: application croisée, application extérieure

Examples

Notions de base CROSS APPLY et OUTER APPLY

Apply sera utilisé lorsque la valeur de la table fonctionne dans la bonne expression.

créer une table départementale pour contenir des informations sur les départements. Créez ensuite une table Employé contenant des informations sur les employés. Veuillez noter que chaque employé appartient à un département. Par conséquent, la table Employé a une intégrité référentielle avec la table Département.

La première requête sélectionne les données de la table Department et utilise CROSS APPLY pour évaluer la table Employee pour chaque enregistrement de la table Department. La deuxième requête rejoint simplement la table Department avec la table Employee et tous les enregistrements correspondants sont générés.

```
SELECT *
FROM Department D
CROSS APPLY (
    SELECT *
    FROM Employee E
    WHERE E.DepartmentID = D.DepartmentID
) A
GO
SELECT *
FROM Department D
INNER JOIN Employee E
  ON D.DepartmentID = E.DepartmentID
```

Si vous regardez les résultats qu'ils ont produits, c'est exactement le même résultat; En quoi diffère-t-il d'un JOIN et comment aide-t-il à écrire des requêtes plus efficaces?

La première requête du script n ° 2 sélectionne les données de la table Department et utilise OUTER APPLY pour évaluer la table Employee pour chaque enregistrement de la table Department. Pour les lignes pour lesquelles il n'y a pas de correspondance dans la table Employee, ces lignes contiennent des valeurs NULL, comme vous pouvez le voir dans les lignes 5 et 6. La deuxième requête utilise simplement LEFT OUTER JOIN entre la table Department et la table Employee. Comme prévu, la requête renvoie toutes les lignes de la table Department; même pour les lignes pour lesquelles il n'y a pas de correspondance dans la table Employee.

```
SELECT *
FROM Department D
OUTER APPLY (
    SELECT *
    FROM Employee E
```

```
    WHERE E.DepartmentID = D.DepartmentID
) A
GO
SELECT *
FROM Department D
LEFT OUTER JOIN Employee E
  ON D.DepartmentID = E.DepartmentID
GO
```

Même si les deux requêtes ci-dessus renvoient les mêmes informations, le plan d'exécution sera légèrement différent. Mais en termes de coûts, il n'y aura pas beaucoup de différence.

Maintenant vient le temps de voir où l'opérateur APPLY est vraiment nécessaire. Dans le script n ° 3, je crée une fonction table qui accepte DepartmentID comme paramètre et renvoie tous les employés appartenant à ce département. La requête suivante sélectionne les données de la table Department et utilise CROSS APPLY pour joindre la fonction créée. Il passe le DepartmentID pour chaque ligne de l'expression de la table externe (dans notre cas, la table Department) et évalue la fonction pour chaque ligne similaire à une sous-requête corrélée. La requête suivante utilise OUTER APPLY à la place de CROSS APPLY et, par conséquent, contrairement à CROSS APPLY qui n'a renvoyé que des données corrélées, OUTER APPLY renvoie également des données non corrélées, plaçant des valeurs NULL dans les colonnes manquantes.

```
CREATE FUNCTION dbo.fn_GetAllEmployeeOfADepartment (@DeptID AS int)
RETURNS TABLE
AS
  RETURN
  (
  SELECT
    *
  FROM Employee E
  WHERE E.DepartmentID = @DeptID
  )
GO
SELECT
  *
FROM Department D
CROSS APPLY dbo.fn_GetAllEmployeeOfADepartment(D.DepartmentID)
GO
SELECT
  *
FROM Department D
OUTER APPLY dbo.fn_GetAllEmployeeOfADepartment(D.DepartmentID)
GO
```

Alors maintenant, si vous vous demandez, pouvons-nous utiliser une simple jointure à la place des requêtes ci-dessus? Alors la réponse est NON, si vous remplacez CROSS / OUTER APPLY dans les requêtes ci-dessus par INNER JOIN / LEFT OUTER JOIN, spécifiez la clause ON (quelque chose comme 1 = 1) et lancez la requête, vous obtiendrez "l'identifiant multi-partie" D.DepartmentID "ne peut pas être lié". Erreur. En effet, avec JOINs, le contexte d'exécution de la requête externe est différent du contexte d'exécution de la fonction (ou d'une table dérivée) et vous ne pouvez pas lier une valeur / variable de la requête externe à la fonction en tant que paramètre. Par conséquent, l'opérateur APPLY est requis pour ces requêtes.

Lire application croisée, application extérieure en ligne:
https://riptutorial.com/fr/sql/topic/2516/application-croisee--application-exterieure

Chapitre 5: Base de données DROP ou DELETE

Syntaxe

- MSSQL Syntaxe:
- DROP DATABASE [IF EXISTS] {nom_base_de_données | database_snapshot_name} [, ...n] [;]
- La syntaxe MySQL:
- DROP {BASE DE DONNEES | SCHEMA} [IF EXISTS] nom_base

Remarques

`DROP DATABASE` est utilisé pour déposer une base de données à partir de SQL. Veillez à créer une sauvegarde de votre base de données avant de la supprimer pour éviter toute perte accidentelle d'informations.

Examples

Base de données DROP

La suppression de la base de données est une déclaration simple à une ligne. La base de données de suppression supprimera la base de données, assurez-vous donc toujours d'avoir une sauvegarde de la base de données si nécessaire.

Voici la commande permettant de supprimer la base de données des employés

```
DROP DATABASE [dbo].[Employees]
```

Lire Base de données DROP ou DELETE en ligne: https://riptutorial.com/fr/sql/topic/3974/base-de-donnees-drop-ou-delete

Chapitre 6: Blocs d'exécution

Examples

Utiliser BEGIN ... END

```
BEGIN
  UPDATE Employees SET PhoneNumber = '5551234567' WHERE Id = 1;
  UPDATE Employees SET Salary = 650 WHERE Id = 3;
END
```

Lire Blocs d'exécution en ligne: https://riptutorial.com/fr/sql/topic/1632/blocs-d-execution

Chapitre 7: CAS

Introduction

L'expression CASE est utilisée pour implémenter la logique if-then.

Syntaxe

- CASE input_expression
  QUAND comparer1 ALORS le résultat1
  [QUAND compare2 THEN result2] ...
  [AUTRE résultatX]
  FIN
- CAS
  QUAND condition1 THEN result1
  [QUAND condition2 THEN result2] ...
  [AUTRE résultatX]
  FIN

Remarques

L' *expression CASE simple* renvoie le premier résultat dont la valeur `compareX` est égale à l' *expression* `input_expression` .

L' *expression CASE recherchée* renvoie le premier résultat dont `conditionX` est true.

Examples

CASE recherché dans SELECT (correspond à une expression booléenne)

Le CASE *recherché* renvoie des résultats lorsqu'une expression *booléenne* est TRUE.

(Cela diffère du cas simple, qui ne peut que vérifier l'équivalence avec une entrée.)

```
SELECT Id, ItemId, Price,
  CASE WHEN Price < 10 THEN 'CHEAP'
       WHEN Price < 20 THEN 'AFFORDABLE'
       ELSE 'EXPENSIVE'
  END AS PriceRating
FROM ItemSales
```

Id	ID de l'article	Prix	Prix
1	100	34,5	COÛTEUX
2	145	2.3	PAS CHER

Id	ID de l'article	Prix	Prix
3	100	34,5	COÛTEUX
4	100	34,5	COÛTEUX
5	145	dix	ABORDABLE

Utilisez CASE to COUNT le nombre de lignes dans une colonne correspondent à une condition.

Cas d'utilisation

`CASE` peut être utilisé conjointement avec `SUM` pour renvoyer uniquement le nombre d'éléments correspondant à une condition prédéfinie. (Ceci est similaire à `COUNTIF` dans Excel.)

L'astuce consiste à renvoyer des résultats binaires indiquant les correspondances, ainsi les "1" renvoyés pour les entrées correspondantes peuvent être additionnés pour un compte du nombre total de correspondances.

Compte tenu de cette table `ItemSales` , supposons que vous souhaitiez connaître le nombre total d'éléments classés comme "chers":

Id	ID de l'article	Prix	Prix
1	100	34,5	COÛTEUX
2	145	2.3	PAS CHER
3	100	34,5	COÛTEUX
4	100	34,5	COÛTEUX
5	145	dix	ABORDABLE

Question

```
SELECT
    COUNT(Id) AS ItemsCount,
    SUM ( CASE
            WHEN PriceRating = 'Expensive' THEN 1
            ELSE 0
          END
        ) AS ExpensiveItemsCount
FROM ItemSales
```

Résultats:

ItemsCount	ExpensiveItemsCount
5	3

Alternative:

```
SELECT
    COUNT(Id) as ItemsCount,
    SUM (
        CASE PriceRating
            WHEN 'Expensive' THEN 1
            ELSE 0
        END
       ) AS ExpensiveItemsCount
FROM ItemSales
```

Shorthand CASE dans SELECT

La variante abrégée de `CASE` évalue une expression (généralement une colonne) par rapport à une série de valeurs. Cette variante est un peu plus courte et évite la répétition répétée de l'expression évaluée. La clause `ELSE` peut toujours être utilisée, bien que:

```
SELECT Id, ItemId, Price,
  CASE Price WHEN 5  THEN 'CHEAP'
             WHEN 15 THEN 'AFFORDABLE'
             ELSE         'EXPENSIVE'
  END as PriceRating
FROM ItemSales
```

Un mot d'avertissement. Il est important de réaliser que lors de l'utilisation de la variante courte, l'intégralité de l'instruction est évaluée à chaque `WHEN` . Par conséquent, la déclaration suivante:

```
SELECT
    CASE ABS(CHECKSUM(NEWID())) % 4
        WHEN 0 THEN 'Dr'
        WHEN 1 THEN 'Master'
        WHEN 2 THEN 'Mr'
        WHEN 3 THEN 'Mrs'
    END
```

peut produire un résultat `NULL` . C'est parce qu'à chaque `WHEN` `NEWID()` est appelée à nouveau avec un nouveau résultat. Équivalent à:

```
SELECT
    CASE
        WHEN ABS(CHECKSUM(NEWID())) % 4 = 0 THEN 'Dr'
        WHEN ABS(CHECKSUM(NEWID())) % 4 = 1 THEN 'Master'
        WHEN ABS(CHECKSUM(NEWID())) % 4 = 2 THEN 'Mr'
        WHEN ABS(CHECKSUM(NEWID())) % 4 = 3 THEN 'Mrs'
    END
```

Par conséquent, il peut manquer tous les cas `WHEN` et le résultat `NULL` .

CAS dans une clause ORDER BY

Nous pouvons utiliser 1,2,3 .. pour déterminer le type de commande:

```
SELECT * FROM DEPT
ORDER BY
CASE DEPARTMENT
      WHEN 'MARKETING' THEN  1
      WHEN 'SALES' THEN 2
      WHEN 'RESEARCH' THEN 3
      WHEN 'INNOVATION' THEN 4
      ELSE        5
      END,
      CITY
```

ID	RÉGION	VILLE	DÉPARTEMENT	EMPLOYEES_NUMBER
12	Nouvelle-Angleterre	Boston	COMMERCIALISATION	9
15	Ouest	San Francisco	COMMERCIALISATION	12
9	Midwest	Chicago	VENTES	8
14	Mid-Atlantique	New York	VENTES	12
5	Ouest	Los Angeles	RECHERCHE	11
dix	Mid-Atlantique	crême Philadelphia	RECHERCHE	13
4	Midwest	Chicago	INNOVATION	11
2	Midwest	Detroit	RESSOURCES HUMAINES	9

Utilisation de CASE dans UPDATE

échantillon sur les augmentations de prix:

```
UPDATE ItemPrice
SET Price = Price *
  CASE ItemId
    WHEN 1 THEN 1.05
    WHEN 2 THEN 1.10
    WHEN 3 THEN 1.15
    ELSE 1.00
  END
```

CASE utilise pour les valeurs NULL ordonnées en dernier

de cette manière, '0' représentant les valeurs connues est classé en premier, '1' représentant les valeurs NULL sont triés en fonction du dernier:

```
SELECT ID
      ,REGION
      ,CITY
      ,DEPARTMENT
      ,EMPLOYEES_NUMBER
  FROM DEPT
  ORDER BY
  CASE WHEN REGION IS NULL THEN 1
  ELSE 0
  END,
  REGION
```

ID	RÉGION	VILLE	DÉPARTEMENT	EMPLOYEES_NUMBER
dix	Mid-Atlantique	crême Philadelphia	RECHERCHE	13
14	Mid-Atlantique	New York	VENTES	12
9	Midwest	Chicago	VENTES	8
12	Nouvelle-Angleterre	Boston	COMMERCIALISATION	9
5	Ouest	Los Angeles	RECHERCHE	11
15	NUL	San Francisco	COMMERCIALISATION	12
4	NUL	Chicago	INNOVATION	11
2	NUL	Detroit	RESSOURCES HUMAINES	9

CASE dans la clause ORDER BY pour trier les enregistrements par la valeur la plus basse de 2 colonnes

Imaginez que vous ayez besoin de trier les enregistrements par la valeur la plus basse de l'une des deux colonnes. Certaines bases de données peuvent utiliser une fonction `MIN()` ou `LEAST()` non agrégée `LEAST()` pour cela (`... ORDER BY MIN(Date1, Date2)`), mais en SQL standard, vous devez utiliser une expression `CASE` .

L'expression `CASE` dans la requête ci-dessous examine les colonnes `Date1` et `Date2` , vérifie quelle colonne a la valeur la plus faible et trie les enregistrements en fonction de cette valeur.

Données d'échantillon

Id	Date1	Date2
1	2017-01-01	2017-01-31
2	2017-01-31	2017-01-03
3	2017-01-31	2017-01-02
4	2017-01-06	2017-01-31
5	2017-01-31	2017-01-05
6	2017-01-04	2017-01-31

Question

```
SELECT Id, Date1, Date2
FROM YourTable
ORDER BY CASE
           WHEN COALESCE(Date1, '1753-01-01') < COALESCE(Date2, '1753-01-01') THEN Date1
           ELSE Date2
         END
```

Résultats

Id	Date1	Date2
1	**2017-01-01**	2017-01-31
3	2017-01-31	**2017-01-02**
2	2017-01-31	**2017-01-03**
6	**2017-01-04**	2017-01-31
5	2017-01-31	**2017-01-05**
4	**2017-01-06**	2017-01-31

Explication

Comme vous voyez que la ligne avec `Id = 1` est la première, parce que `Date1` a l'enregistrement le plus bas de la table entière `2017-01-01` , la ligne où `Id = 3` est la seconde parce que `Date2` est égale à `2017-01-02` etc.

Nous avons donc trié les enregistrements de `2017-01-01` à `2017-01-01` en `2017-01-06` croissant et aucune attention sur laquelle une colonne `Date1` ou `Date2` sont ces valeurs.

Lire CAS en ligne: https://riptutorial.com/fr/sql/topic/456/cas

Chapitre 8: CLAUSE EXISTE

Examples

CLAUSE EXISTE

Table client

Id	Prénom	Nom de famille
1	Ozgur	Ozturk
2	Youssef	Medi
3	Henri	Tai

Tableau de commande

Id	N ° de client	Montant
1	2	123,50
2	3	14.80

Obtenez tous les clients avec au moins une commande

```
SELECT * FROM Customer WHERE EXISTS (
    SELECT * FROM Order WHERE Order.CustomerId=Customer.Id
)
```

Résultat

Id	Prénom	Nom de famille
2	Youssef	Medi
3	Henri	Tai

Obtenez tous les clients sans commande

```
SELECT * FROM Customer WHERE NOT EXISTS (
    SELECT * FROM Order WHERE Order.CustomerId = Customer.Id
)
```

Résultat

Id	Prénom	Nom de famille
1	Ozgur	Ozturk

Objectif

`EXISTS` , `IN` et `JOIN` peuvent parfois être utilisés pour le même résultat, mais ils ne sont pas égaux:

- `EXISTS` doit être utilisé pour vérifier si une valeur existe dans une autre table
- `IN` doit être utilisé pour la liste statique
- `JOIN` doit être utilisé pour récupérer des données à partir d'autres tables

Lire CLAUSE EXISTE en ligne: https://riptutorial.com/fr/sql/topic/7933/clause-existe

Chapitre 9: Clause IN

Examples

Clause IN simple

Pour obtenir des enregistrements ayant l' **un** des `id` donnés

```
select *
from products
where id in (1,8,3)
```

La requête ci-dessus est égale à

```
select *
from products
where id = 1
   or id = 8
   or id = 3
```

Utilisation de la clause IN avec une sous-requête

```
SELECT *
FROM customers
WHERE id IN (
    SELECT DISTINCT customer_id
    FROM orders
);
```

Ce qui précède vous donnera tous les clients qui ont des commandes dans le système.

Lire Clause IN en ligne: https://riptutorial.com/fr/sql/topic/3169/clause-in

Chapitre 10: Clés étrangères

Examples

Créer une table avec une clé étrangère

Dans cet exemple, nous avons une table existante, `SuperHeros` .

Cette table contient un `ID` clé primaire.

Nous allons ajouter une nouvelle table afin de stocker les pouvoirs de chaque super héros:

```
CREATE TABLE HeroPowers
(
    ID int NOT NULL PRIMARY KEY,
    Name nvarchar(MAX) NOT NULL,
    HeroId int REFERENCES SuperHeros(ID)
)
```

La colonne `HeroId` est une **clé étrangère** à la table `SuperHeros` .

Clés étrangères expliquées

Les contraintes des clés étrangères garantissent l'intégrité des données en imposant que les valeurs d'une table doivent correspondre aux valeurs d'une autre table.

Voici un exemple de lieu où une clé étrangère est requise: Dans une université, un cours doit appartenir à un département. Le code pour ce scénario est le suivant:

```
CREATE TABLE Department (
    Dept_Code        CHAR (5)     PRIMARY KEY,
    Dept_Name        VARCHAR (20) UNIQUE
);
```

Insérer des valeurs avec l'instruction suivante:

```
INSERT INTO Department VALUES ('CS205', 'Computer Science');
```

Le tableau suivant contiendra les informations sur les sujets proposés par la branche informatique:

```
CREATE TABLE Programming_Courses (
    Dept_Code       CHAR(5),
    Prg_Code        CHAR(9) PRIMARY KEY,
    Prg_Name        VARCHAR (50) UNIQUE,
    FOREIGN KEY (Dept_Code) References Department(Dept_Code)
);
```

(Le type de données de la clé étrangère doit correspondre au type de données de la clé

référencée.)

La contrainte de clé étrangère sur la colonne `Dept_Code` autorise les valeurs uniquement si elles existent déjà dans la table référencée, `Department` . Cela signifie que si vous essayez d'insérer les valeurs suivantes:

```
INSERT INTO Programming_Courses Values ('CS300', 'FDB-DB001', 'Database Systems');
```

la base de données générera une erreur de violation de clé étrangère, car `CS300` n'existe pas dans la table `Department` . Mais lorsque vous essayez une valeur clé qui existe:

```
INSERT INTO Programming_Courses VALUES ('CS205', 'FDB-DB001', 'Database Systems');
INSERT INTO Programming_Courses VALUES ('CS205', 'DB2-DB002', 'Database Systems II');
```

alors la base de données autorise ces valeurs.

Quelques conseils pour l'utilisation de clés étrangères

- Une clé étrangère doit faire référence à une clé UNIQUE (ou PRIMARY) dans la table parent.
- La saisie d'une valeur NULL dans une colonne de clé étrangère ne génère pas d'erreur.
- Les contraintes de clé étrangère peuvent référencer des tables dans la même base de données.
- Les contraintes de clé étrangère peuvent faire référence à une autre colonne du même tableau (auto-référence).

Lire Clés étrangères en ligne: https://riptutorial.com/fr/sql/topic/1533/cles-etrangeres

Chapitre 11: Clés primaires

Syntaxe

- MySQL: CREATE TABLE Employees (ID int NOT NULL, PRIMARY KEY (Id), ...);
- Autres: CREATE TABLE Employees (Id int NOT NULL PRIMARY KEY, ...);

Examples

Créer une clé primaire

```
CREATE TABLE Employees (
    Id int NOT NULL,
    PRIMARY KEY (Id),
    ...
);
```

Cela créera la table Employees avec 'Id' comme clé primaire. La clé primaire peut être utilisée pour identifier de manière unique les lignes d'une table. Une seule clé primaire est autorisée par table.

Une clé peut également être composée d'un ou de plusieurs champs, appelés clé composite, avec la syntaxe suivante:

```
CREATE TABLE EMPLOYEE (
    e1_id INT,
    e2_id INT,
    PRIMARY KEY (e1_id, e2_id)
)
```

Utiliser l'incrément automatique

De nombreuses bases de données permettent d’incrémenter automatiquement la valeur de la clé primaire lorsqu’une nouvelle clé est ajoutée. Cela garantit que chaque clé est différente.

MySQL

```
CREATE TABLE Employees (
    Id int NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (Id)
);
```

PostgreSQL

```
CREATE TABLE Employees (
    Id SERIAL PRIMARY KEY
);
```

serveur SQL

```
CREATE TABLE Employees (
    Id int NOT NULL IDENTITY,
    PRIMARY KEY (Id)
);
```

SQLite

```
CREATE TABLE Employees (
    Id INTEGER PRIMARY KEY
);
```

Lire Clés primaires en ligne: https://riptutorial.com/fr/sql/topic/505/cles-primaires

Chapitre 12: COMMANDÉ PAR

Examples

Utilisez ORDER BY avec TOP pour renvoyer les x premières lignes en fonction de la valeur d'une colonne

Dans cet exemple, nous pouvons utiliser GROUP BY non seulement pour déterminer le *type* des lignes renvoyées, mais également quelles lignes *sont* renvoyées, car nous utilisons TOP pour limiter le jeu de résultats.

Disons que nous voulons renvoyer les 5 meilleurs utilisateurs de réputation depuis un site de questions / réponses peu connu.

Sans COMMANDE PAR

Cette requête renvoie les 5 premières lignes triées par défaut, qui dans ce cas est "Id", la première colonne de la table (même si ce n'est pas une colonne affichée dans les résultats).

```
SELECT TOP 5 DisplayName, Reputation
FROM Users
```

résultats...

Afficher un nom	Réputation
Communauté	1
Geoff Dalgas	12567
Jarrod Dixon	11739
Jeff Atwood	37628
Joel Spolsky	25784

Avec ORDER BY

```
SELECT TOP 5 DisplayName, Reputation
FROM Users
ORDER BY Reputation desc
```

résultats...

Afficher un nom	Réputation
JonSkeet	**865023**

Afficher un nom	Réputation
Darin Dimitrov	**661741**
BalusC	**650237**
Hans Passant	**625870**
Marc Gravell	**601636**

Remarques

Certaines versions de SQL (telles que MySQL) utilisent une clause `LIMIT` à la fin d'un `SELECT` , au lieu de `TOP` au début, par exemple:

```
SELECT DisplayName, Reputation
FROM Users
ORDER BY Reputation DESC
LIMIT 5
```

Tri par plusieurs colonnes

```
SELECT DisplayName, JoinDate, Reputation
FROM Users
ORDER BY JoinDate, Reputation
```

Afficher un nom	JoinDate	Réputation
Communauté	**2008-09-15**	**1**
Jeff Atwood	**2008-09-16**	**25784**
Joel Spolsky	2008-09-16	**37628**
Jarrod Dixon	**2008-10-03**	**11739**
Geoff Dalgas	2008-10-03	**12567**

Tri par numéro de colonne (au lieu de nom)

Vous pouvez utiliser le numéro d'une colonne (où la colonne la plus à gauche est «1») pour indiquer la colonne sur laquelle baser le tri, au lieu de décrire la colonne par son nom.

Pro: Si vous pensez qu'il est probable que vous modifieriez les noms de colonnes plus tard, cela ne briserait pas ce code.

Con: Cela réduira généralement la lisibilité de la requête (il est clair que «ORDER BY Reputation» signifie, alors que «ORDER BY 14» nécessite un comptage, probablement avec un doigt sur l'écran.)

Cette requête trie le résultat en fonction des informations dans la position relative de la colonne `3` partir de l'instruction select au lieu du nom de la colonne `Reputation` .

```
SELECT DisplayName, JoinDate, Reputation
FROM Users
ORDER BY 3
```

Afficher un nom	JoinDate	Réputation
Communauté	2008-09-15	**1**
Jarrod Dixon	2008-10-03	**11739**
Geoff Dalgas	2008-10-03	**12567**
Joel Spolsky	2008-09-16	**25784**
Jeff Atwood	2008-09-16	**37628**

Commande par alias

En raison de l'ordre de traitement des requêtes logiques, des alias peuvent être utilisés dans l'ordre.

```
SELECT DisplayName, JoinDate as jd, Reputation as rep
FROM Users
ORDER BY jd, rep
```

Et peut utiliser l'ordre relatif des colonnes dans l'instruction select. Considérez le même exemple que ci-dessus et, au lieu d'utiliser des alias, utilisez l'ordre relatif comme pour le nom d'affichage, il est 1, pour Jd il est 2 et ainsi de suite

```
SELECT DisplayName, JoinDate as jd, Reputation as rep
FROM Users
ORDER BY 2, 3
```

Ordre de tri personnalisé

Pour trier cette table `Employee` par département, vous utiliseriez `ORDER BY Department` . Toutefois, si vous souhaitez un ordre de tri différent de celui par ordre alphabétique, vous devez mapper les valeurs du `Department` en différentes valeurs qui sont triées correctement. cela peut être fait avec une expression CASE:

prénom	département
Hasan	IL
Yusuf	HEURE

prénom	département
Hillary	HEURE
Joe	IL
Joyeux	HEURE
Ken	Comptable

```
SELECT *
FROM Employee
ORDER BY CASE Department
         WHEN 'HR'         THEN 1
         WHEN 'Accountant' THEN 2
         ELSE                   3
         END;
```

prénom	département
Yusuf	**HEURE**
Hillary	**HEURE**
Joyeux	**HEURE**
Ken	**Comptable**
Hasan	**IL**
Joe	**IL**

Lire COMMANDÉ PAR en ligne: https://riptutorial.com/fr/sql/topic/620/commande-par

Chapitre 13: commentaires

Examples

Commentaires sur une seule ligne

Les commentaires sur une seule ligne sont précédés de `--` et vont jusqu'à la fin de la ligne:

```
SELECT *
FROM Employees -- this is a comment
WHERE FName = 'John'
```

Commentaires multilignes

Les commentaires de code multi-lignes sont enveloppés dans `/* ... */` :

```
/* This query
   returns all employees */
SELECT *
FROM Employees
```

Il est également possible d'insérer un tel commentaire au milieu d'une ligne:

```
SELECT /* all columns: */ *
FROM Employees
```

Lire commentaires en ligne: https://riptutorial.com/fr/sql/topic/1597/commentaires

Chapitre 14: CREATE Base de données

Syntaxe

- CREATE DATABASE nom_base;

Examples

CREATE Base de données

Une base de données est créée avec la commande SQL suivante:

```
CREATE DATABASE myDatabase;
```

Cela créerait une base de données vide nommée myDatabase où vous pouvez créer des tables.

Lire CREATE Base de données en ligne: https://riptutorial.com/fr/sql/topic/2744/create-base-de-donnees

Chapitre 15: CREER LA TABLE

Introduction

L'instruction CREATE TABLE est utilisée pour créer une nouvelle table dans la base de données. Une définition de table se compose d'une liste de colonnes, de leurs types et de toutes les contraintes d'intégrité.

Syntaxe

- CREATE TABLE nomTable ([ColumnName1] [type de données1] [, [ColumnName2] [type de données2] ...])

Paramètres

Paramètre	Détails
nom de la table	le nom de la table
colonnes	Contient une énumération de toutes les colonnes de la table. Voir **Créer une nouvelle table** pour plus de détails.

Remarques

Les noms de table doivent être uniques.

Examples

Créer une nouvelle table

Une table `Employees` base, contenant un identifiant, ainsi que le prénom et le nom de l'employé, ainsi que leur numéro de téléphone, peuvent être créés en utilisant

```
CREATE TABLE Employees(
    Id int identity(1,1) primary key not null,
    FName varchar(20) not null,
    LName varchar(20) not null,
    PhoneNumber varchar(10) not null
);
```

Cet exemple est spécifique à Transact-SQL

`CREATE TABLE` crée une nouvelle table dans la base de données, suivie du nom de la table,

`Employees`

Ceci est suivi de la liste des noms de colonnes et de leurs propriétés, telles que l'ID

```
Id int identity(1,1) not null
```

Valeur	Sens
`Id`	le nom de la colonne.
`int`	est le type de données.
`identity(1,1)`	indique que la colonne générera automatiquement des valeurs commençant à 1 et incrémentant de 1 pour chaque nouvelle ligne.
`primary key`	indique que toutes les valeurs de cette colonne auront des valeurs uniques
`not null`	indique que cette colonne ne peut pas avoir de valeurs nulles

Créer une table à partir de la sélection

Vous souhaiterez peut-être créer un duplicata d'une table:

```
CREATE TABLE ClonedEmployees AS SELECT * FROM Employees;
```

Vous pouvez utiliser l'une des autres fonctionnalités d'une instruction SELECT pour modifier les données avant de les transmettre à la nouvelle table. Les colonnes de la nouvelle table sont automatiquement créées en fonction des lignes sélectionnées.

```
CREATE TABLE ModifiedEmployees AS
SELECT Id, CONCAT(FName," ",LName) AS FullName FROM Employees
WHERE Id > 10;
```

Dupliquer une table

Pour dupliquer un tableau, procédez simplement comme suit:

```
CREATE TABLE newtable LIKE oldtable;
INSERT newtable SELECT * FROM oldtable;
```

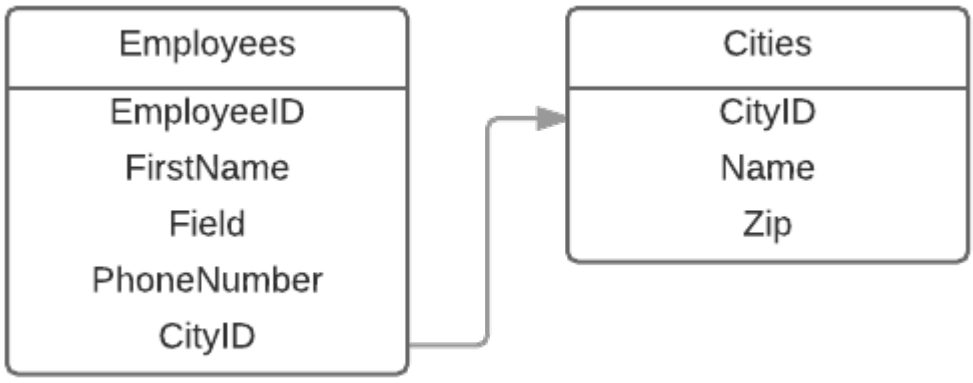
Créer une table avec clé étrangère

Vous trouverez ci-dessous la table `Employees` avec une référence à la table `Cities` .

```
CREATE TABLE Cities(
    CityID INT IDENTITY(1,1) NOT NULL,
    Name VARCHAR(20) NOT NULL,
    Zip VARCHAR(10) NOT NULL
);
```

```
CREATE TABLE Employees(
    EmployeeID INT IDENTITY (1,1) NOT NULL,
    FirstName VARCHAR(20) NOT NULL,
    LastName VARCHAR(20) NOT NULL,
    PhoneNumber VARCHAR(10) NOT NULL,
    CityID INT FOREIGN KEY REFERENCES Cities(CityID)
);
```

Vous pouvez trouver ici un diagramme de base de données.



La colonne `CityID` de la table `Employees` fera référence à la colonne `CityID` de la table `Cities` . Vous trouverez ci-dessous la syntaxe pour faire ceci.

```
CityID INT FOREIGN KEY REFERENCES Cities(CityID)
```

Valeur	Sens
`CityID`	Nom de la colonne
`int`	type de colonne
`FOREIGN KEY`	Fait la clé étrangère *(facultatif)*
`REFERENCES Cities(CityID)`	Fait la référence à la table `Cities` colonne `CityID`

Important: Vous ne pouvez pas faire référence à une table qui n'existe pas dans la base de données. Soyez la source pour faire d'abord la table `Cities` et ensuite la table `Employees` . Si vous le faites vice-versa, cela provoquera une erreur.

Créer une table temporaire ou en mémoire

PostgreSQL et SQLite

Pour créer une table temporaire locale à la session:

```
CREATE TEMP TABLE MyTable(...);
```

serveur SQL

Pour créer une table temporaire locale à la session:

```
CREATE TABLE #TempPhysical(...);
```

Pour créer une table temporaire visible par tous:

```
CREATE TABLE ##TempPhysicalVisibleToEveryone(...);
```

Pour créer une table en mémoire:

```
DECLARE @TempMemory TABLE(...);
```

Lire CREER LA TABLE en ligne: https://riptutorial.com/fr/sql/topic/348/creer-la-table

Chapitre 16: CREER UNE FONCTION

Syntaxe

- CREATE FUNCTION nom_fonction ([list_of_paramenters]) RETURNS return_data_type AS BEGIN function_body RETURN expression-scalaire END

Paramètres

Argument	La description
nom_fonction	le nom de la fonction
list_of_paramenters	paramètres que la fonction accepte
return_data_type	tapez cette fonction se répète. Certains types de données SQL
function_body	le code de fonction
expression scalaire	valeur scalaire renvoyée par la fonction

Remarques

CREATE FUNCTION crée une fonction définie par l'utilisateur qui peut être utilisée lors d'une requête SELECT, INSERT, UPDATE ou DELETE. Les fonctions peuvent être créées pour renvoyer une seule variable ou une seule table.

Examples

Créer une nouvelle fonction

```
CREATE FUNCTION FirstWord (@input varchar(1000))
RETURNS varchar(1000)
AS
BEGIN
    DECLARE @output varchar(1000)
    SET @output = SUBSTRING(@input, 0, CASE CHARINDEX(' ', @input)
        WHEN 0 THEN LEN(@input) + 1
        ELSE CHARINDEX(' ', @input)
    END)

    RETURN @output
END
```

Cet exemple crée une fonction nommée **FirstWord** , qui accepte un paramètre varchar et renvoie une autre valeur varchar.

Lire CREER UNE FONCTION en ligne: https://riptutorial.com/fr/sql/topic/2437/creer-une-fonction

Chapitre 17: CURSEUR SQL

Examples

Exemple de curseur qui interroge toutes les lignes par index pour chaque base de données

Ici, un curseur est utilisé pour parcourir toutes les bases de données.
De plus, un curseur de sql dynamique est utilisé pour interroger chaque base de données renvoyée par le premier curseur.
Ceci est pour démontrer l'étendue de connexion d'un curseur.

```
DECLARE @db_name nvarchar(255)
DECLARE @sql nvarchar(MAX)

DECLARE @schema nvarchar(255)
DECLARE @table nvarchar(255)
DECLARE @column nvarchar(255)




DECLARE db_cursor CURSOR FOR
SELECT name FROM sys.databases


OPEN db_cursor
FETCH NEXT FROM db_cursor INTO @db_name

WHILE @@FETCH_STATUS = 0
BEGIN
    SET @sql = 'SELECT * FROM ' + QUOTENAME(@db_name) + '.information_schema.columns'
    PRINT ''
    PRINT ''
    PRINT ''
    PRINT @sql
    -- EXECUTE(@sql)



    -- For each database

    DECLARE @sqlstatement nvarchar(4000)
    --move declare cursor into sql to be executed
    SET @sqlstatement = 'DECLARE  columns_cursor CURSOR FOR SELECT TABLE_SCHEMA, TABLE_NAME,
COLUMN_NAME FROM ' + QUOTENAME(@db_name) + '.information_schema.columns ORDER BY TABLE_SCHEMA,
TABLE_NAME, ORDINAL_POSITION'



    EXEC sp_executesql @sqlstatement


    OPEN columns_cursor
    FETCH NEXT FROM columns_cursor
```

```
    INTO @schema, @table, @column

    WHILE @@FETCH_STATUS = 0
    BEGIN
        PRINT @schema + '.' + @table + '.' + @column
        --EXEC asp_DoSomethingStoredProc @UserId

    FETCH NEXT FROM columns_cursor --have to fetch again within loop
    INTO @schema, @table, @column

    END
    CLOSE columns_cursor
    DEALLOCATE columns_cursor

    -- End for each database




    FETCH NEXT FROM db_cursor INTO @db_name
END

CLOSE db_cursor
DEALLOCATE db_cursor
```

Lire CURSEUR SQL en ligne: https://riptutorial.com/fr/sql/topic/8895/curseur-sql

Chapitre 18: Déclencheurs

Examples

CRÉER UN DÉCLENCHEUR

Cet exemple crée un déclencheur qui insère un enregistrement dans une seconde table (MyAudit) après l'insertion d'un enregistrement dans la table sur laquelle le déclencheur est défini (MyTable). Ici, le tableau "inséré" est une table spéciale utilisée par Microsoft SQL Server pour stocker les lignes affectées pendant les instructions INSERT et UPDATE; il existe également une table spéciale "supprimée" qui exécute la même fonction pour les instructions DELETE.

```
CREATE TRIGGER MyTrigger
    ON MyTable
    AFTER INSERT

AS

BEGIN
    -- insert audit record to MyAudit table
    INSERT INTO MyAudit(MyTableId, User)
    (SELECT MyTableId, CURRENT_USER FROM inserted)
END
```

Utiliser Trigger pour gérer une "Corbeille" pour les éléments supprimés

```
CREATE TRIGGER BooksDeleteTrigger
    ON MyBooksDB.Books
    AFTER DELETE
AS
  INSERT INTO BooksRecycleBin
    SELECT *
    FROM deleted;
GO
```

Lire Déclencheurs en ligne: https://riptutorial.com/fr/sql/topic/1432/declencheurs

Chapitre 19: Des synonymes

Examples

Créer un synonyme

```
CREATE SYNONYM EmployeeData
FOR MyDatabase.dbo.Employees
```

Lire Des synonymes en ligne: https://riptutorial.com/fr/sql/topic/2518/des-synonymes

Chapitre 20: Des vues

Examples

Des vues simples

Une vue peut filtrer certaines lignes de la table de base ou ne projeter que certaines colonnes:

```
CREATE VIEW new_employees_details AS
SELECT E.id, Fname, Salary, Hire_date
FROM Employees E
WHERE hire_date > date '2015-01-01';
```

Si vous sélectionnez la vue:

```
select * from new_employees_details
```

Id	FName	Un salaire	Date d'embauche
4	Johnathon	500	24-07-2016

Vues complexes

Une vue peut être une requête très complexe (agrégations, jointures, sous-requêtes, etc.). Veillez simplement à ajouter des noms de colonnes pour tout ce que vous sélectionnez:

```
Create VIEW dept_income AS
SELECT d.Name as DepartmentName, sum(e.salary) as TotalSalary
FROM Employees e
JOIN Departments d on e.DepartmentId = d.id
GROUP BY d.Name;
```

Maintenant, vous pouvez choisir parmi toutes les tables:

```
SELECT *
FROM dept_income;
```

Nom du département	Salaire total
HEURE	1900
Ventes	600

Lire Des vues en ligne: https://riptutorial.com/fr/sql/topic/766/des-vues

Chapitre 21: Design de table

Remarques

The Open University (1999) Systèmes de bases de données relationnelles: théorie relationnelle du bloc 2, Milton Keynes, The Open University.

Examples

Propriétés d'une table bien conçue.

Une véritable base de données relationnelle doit aller au-delà de la diffusion de données dans quelques tables et de l'écriture de certaines instructions SQL pour extraire ces données.
Au mieux, une structure de table mal conçue ralentira l'exécution des requêtes et pourrait empêcher la base de données de fonctionner comme prévu.

Une table de base de données ne doit pas être considérée comme une autre table; il doit suivre un ensemble de règles pour être considéré comme véritablement relationnel. Sur le plan académique, on parle de «relation» pour faire la distinction.

Les cinq règles d'une table relationnelle sont:

1. Chaque valeur est *atomique* ; la valeur dans chaque champ de chaque ligne doit être une valeur unique.
2. Chaque champ contient des valeurs du même type de données.
3. Chaque en-tête de champ porte un nom unique.
4. Chaque ligne de la table doit avoir au moins une valeur qui la rend unique parmi les autres enregistrements de la table.
5. L'ordre des lignes et des colonnes n'a aucune signification.

Un tableau conforme aux cinq règles:

Id	prénom	DOB	Directeur
1	Fred	11/02/1971	3
2	Fred	11/02/1971	3
3	poursuivre en justice	08/07/1975	2

- Règle 1: Chaque valeur est atomique. `Id` , `Name` , `DOB` et `Manager` ne contiennent qu'une seule valeur.
- Règle 2: `Id` ne contient que des entiers, `Name` contient du texte (on pourrait ajouter que c'est un texte de quatre caractères ou moins), `DOB` contient des dates d'un type valide et `Manager` contient des entiers (nous pourrions ajouter un champ Primary Key dans un manager) table).

- Règle 3: `Id` , `Name` , `DOB` et `Manager` sont des noms d'en-tête uniques dans la table.
- Règle 4: L'inclusion du champ `Id` garantit que chaque enregistrement est distinct de tout autre enregistrement dans la table.

Une table mal conçue:

Id	prénom	DOB	prénom
1	Fred	11/02/1971	3
1	Fred	11/02/1971	3
3	poursuivre en justice	Vendredi 18 juillet 1975	2, 1

- Règle 1: Le champ deuxième nom contient deux valeurs - 2 et 1.
- Règle 2: le champ DOB contient les dates et le texte.
- Règle 3: Il y a deux champs appelés "nom".
- Règle 4: Le premier et le deuxième enregistrement sont exactement les mêmes.
- Règle 5: Cette règle n'est pas cassée.

Lire Design de table en ligne: https://riptutorial.com/fr/sql/topic/2515/design-de-table

Chapitre 22: DROP Table

Remarques

DROP TABLE supprime la définition de la table du schéma avec les lignes, les index, les autorisations et les déclencheurs.

Examples

Simple goutte

```
Drop Table MyTable;
```

Vérifier l'existence avant de laisser tomber

MySQL 3.19

```
DROP TABLE IF EXISTS MyTable;
```

PostgreSQL 8.x

```
DROP TABLE IF EXISTS MyTable;
```

SQL Server 2005

```
If Exists(Select * From Information_Schema.Tables
          Where Table_Schema = 'dbo'
            And Table_Name = 'MyTable')
  Drop Table dbo.MyTable
```

SQLite 3.0

```
DROP TABLE IF EXISTS MyTable;
```

Lire DROP Table en ligne: https://riptutorial.com/fr/sql/topic/1832/drop-table

Chapitre 23: EFFACER

Introduction

L'instruction DELETE permet de supprimer des enregistrements d'une table.

Syntaxe

1. DELETE FROM *TableName* [*Condition* WHERE] [*Nombre de* LIMIT]

Examples

SUPPRIMER certaines lignes avec WHERE

Cela supprimera toutes les lignes correspondant aux critères `WHERE` .

```
DELETE FROM Employees
WHERE FName = 'John'
```

SUPPRIMER toutes les lignes

L'omission d'une clause `WHERE` supprime toutes les lignes d'une table.

```
DELETE FROM Employees
```

Voir la documentation TRUNCATE pour plus de détails sur la manière dont les performances TRUNCATE peuvent être améliorées, car elles ignorent les déclencheurs, les index et les journaux pour simplement supprimer les données.

Clause TRUNCATE

Utilisez cette option pour réinitialiser la table à la condition à laquelle elle a été créée. Cela supprime toutes les lignes et réinitialise des valeurs telles que l'incrémentation automatique. Il ne consigne pas non plus chaque suppression de ligne individuelle.

```
TRUNCATE TABLE Employees
```

SUPPRIMER certaines lignes en fonction de comparaisons avec d'autres tables

Il est possible de `DELETE` données d'une table si elles correspondent (ou ne correspondent pas) à certaines données dans d'autres tables.

Supposons que nous voulons `DELETE` données de la source une fois qu'elles sont chargées dans la

cible.

```
DELETE FROM Source
WHERE  EXISTS ( SELECT 1 -- specific value in SELECT doesn't matter
                FROM Target
                Where Source.ID = Target.ID )
```

Les implémentations les plus courantes des SGBDR (par exemple, MySQL, Oracle, PostgresSQL, Teradata) permettent de joindre des tables lors de `DELETE` ce qui permet une comparaison plus complexe dans une syntaxe compacte.

En ajoutant de la complexité au scénario d'origine, supposons que Aggregate est créé à partir de Target une fois par jour et ne contient pas le même ID mais contient la même date. Supposons également que nous souhaitons supprimer des données de la source *uniquement* après que l'agrégat ait été rempli pour la journée.

Sur MySQL, Oracle et Teradata, cela peut être fait en utilisant:

```
DELETE FROM Source
WHERE  Source.ID = TargetSchema.Target.ID
       AND TargetSchema.Target.Date = AggregateSchema.Aggregate.Date
```

Dans PostgreSQL, utilisez:

```
DELETE FROM Source
USING  TargetSchema.Target, AggregateSchema.Aggregate
WHERE  Source.ID = TargetSchema.Target.ID
       AND TargetSchema.Target.DataDate = AggregateSchema.Aggregate.AggDate
```

Cela se traduit essentiellement par des jointures internes entre source, cible et agrégat. La suppression est effectuée sur la source lorsque les mêmes identifiants existent dans la cible ET la date présente dans la cible pour ces identifiants existe également dans l'ensemble.

La même requête peut également être écrite (sur MySQL, Oracle, Teradata) en tant que:

```
DELETE Source
FROM   Source, TargetSchema.Target, AggregateSchema.Aggregate
WHERE  Source.ID = TargetSchema.Target.ID
       AND TargetSchema.Target.DataDate = AggregateSchema.Aggregate.AggDate
```

Les jointures explicites peuvent être mentionnées dans les instructions `Delete` de certaines implémentations de SGBDR (par exemple, Oracle, MySQL), mais ne sont pas prises en charge sur toutes les plates-formes (par exemple, Teradata ne les prend pas en charge).

Les comparaisons peuvent être conçues pour vérifier les scénarios de non-concordance au lieu de les faire correspondre à tous les styles de syntaxe (observez `NOT EXISTS` ci-dessous).

```
DELETE FROM Source
WHERE NOT EXISTS ( SELECT 1 -- specific value in SELECT doesn't matter
                FROM Target
                Where Source.ID = Target.ID )
```

Lire EFFACER en ligne: https://riptutorial.com/fr/sql/topic/1105/effacer

Chapitre 24: ESSAYEZ / CATCH

Remarques

TRY / CATCH est une construction de langage spécifique à T-SQL de MS SQL Server.

Il permet la gestion des erreurs dans T-SQL, similaire à celle observée dans le code .NET.

Examples

Transaction dans un TRY / CATCH

Cela annulera les deux insertions en raison d'un datetime non valide:

```
BEGIN TRANSACTION
BEGIN TRY
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, 'not a date', 1)
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    THROW
    ROLLBACK TRANSACTION
END CATCH
```

Cela va valider les deux insertions:

```
BEGIN TRANSACTION
BEGIN TRY
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    THROW
    ROLLBACK TRANSACTION
END CATCH
```

Lire ESSAYEZ / CATCH en ligne: https://riptutorial.com/fr/sql/topic/4420/essayez---catch

Chapitre 25: Exemples de bases de données et de tables

Examples

Base de données Auto Shop

Dans l'exemple suivant - Base de données pour un commerce automobile, nous avons une liste de départements, d'employés, de clients et de voitures client. Nous utilisons des clés étrangères pour créer des relations entre les différentes tables.

Exemple en direct: violon SQL

Relations entre les tables

- Chaque département peut avoir 0 employé ou plus
- Chaque employé peut avoir 0 ou 1 gestionnaire
- Chaque client peut avoir 0 ou plus de voitures

Départements

Id	prénom
1	HEURE
2	Ventes
3	Technologie

Instructions SQL pour créer la table:

```
CREATE TABLE Departments (
    Id INT NOT NULL AUTO_INCREMENT,
    Name VARCHAR(25) NOT NULL,
    PRIMARY KEY(Id)
);

INSERT INTO Departments
    ([Id], [Name])
VALUES
    (1, 'HR'),
    (2, 'Sales'),
    (3, 'Tech')
;
```

Des employés

Id	FName	LName	Numéro de téléphone	ManagerId	DépartementId	Un salaire	Date d'embauche
1	James	Forgeron	1234567890	NUL	1	1000	01-01-2002
2	John	Johnson	2468101214	1	1	400	23-03-2005
3	Michael	Williams	1357911131	1	2	600	12-05-2009
4	Johnathon	Forgeron	1212121212	2	1	500	24-07-2016

Instructions SQL pour créer la table:

```
CREATE TABLE Employees (
    Id INT NOT NULL AUTO_INCREMENT,
    FName VARCHAR(35) NOT NULL,
    LName VARCHAR(35) NOT NULL,
    PhoneNumber VARCHAR(11),
    ManagerId INT,
    DepartmentId INT NOT NULL,
    Salary INT NOT NULL,
    HireDate DATETIME NOT NULL,
    PRIMARY KEY(Id),
    FOREIGN KEY (ManagerId) REFERENCES Employees(Id),
    FOREIGN KEY (DepartmentId) REFERENCES Departments(Id)
);

INSERT INTO Employees
    ([Id], [FName], [LName], [PhoneNumber], [ManagerId], [DepartmentId], [Salary], [HireDate])
VALUES
    (1, 'James', 'Smith', 1234567890, NULL, 1, 1000, '01-01-2002'),
    (2, 'John', 'Johnson', 2468101214, '1', 1, 400, '23-03-2005'),
    (3, 'Michael', 'Williams', 1357911131, '1', 2, 600, '12-05-2009'),
    (4, 'Johnathon', 'Smith', 1212121212, '2', 1, 500, '24-07-2016')
;
```

Les clients

Id	FName	LName	Email	Numéro de téléphone	Contact préféré
1	William	Jones	william.jones@example.com	3347927472	TÉLÉPHONE
2	David	Meunier	dmiller@example.net	2137921892	EMAIL
3	Richard	Davis	richard0123@example.com	NUL	EMAIL

Instructions SQL pour créer la table:

```
CREATE TABLE Customers (
    Id INT NOT NULL AUTO_INCREMENT,
    FName VARCHAR(35) NOT NULL,
    LName VARCHAR(35) NOT NULL,
    Email varchar(100) NOT NULL,
    PhoneNumber VARCHAR(11),
    PreferredContact VARCHAR(5) NOT NULL,
    PRIMARY KEY(Id)
);

INSERT INTO Customers
    ([Id], [FName], [LName], [Email], [PhoneNumber], [PreferredContact])
VALUES
    (1, 'William', 'Jones', 'william.jones@example.com', '3347927472', 'PHONE'),
    (2, 'David', 'Miller', 'dmiller@example.net', '2137921892', 'EMAIL'),
    (3, 'Richard', 'Davis', 'richard0123@example.com', NULL, 'EMAIL')
;
```

Des voitures

Id	N ° de client	EmployeeId	Modèle	Statut	Coût total
1	1	2	Ford F-150	PRÊT	230
2	1	2	Ford F-150	PRÊT	200
3	2	1	Ford Mustang	ATTENDRE	100
4	3	3	Toyota Prius	TRAVAIL	1254

Instructions SQL pour créer la table:

```
CREATE TABLE Cars (
    Id INT NOT NULL AUTO_INCREMENT,
    CustomerId INT NOT NULL,
    EmployeeId INT NOT NULL,
    Model varchar(50) NOT NULL,
    Status varchar(25) NOT NULL,
    TotalCost INT NOT NULL,
    PRIMARY KEY(Id),
    FOREIGN KEY (CustomerId) REFERENCES Customers(Id),
    FOREIGN KEY (EmployeeId) REFERENCES Employees(Id)
);

INSERT INTO Cars
    ([Id], [CustomerId], [EmployeeId], [Model], [Status], [TotalCost])
VALUES
    ('1', '1', '2', 'Ford F-150', 'READY', '230'),
    ('2', '1', '2', 'Ford F-150', 'READY', '200'),
    ('3', '2', '1', 'Ford Mustang', 'WAITING', '100'),
    ('4', '3', '3', 'Toyota Prius', 'WORKING', '1254')
;
```

Base de données de la bibliothèque

Dans cet exemple de base de données pour une bibliothèque, nous avons des tables *Authors* , *Books* et *BooksAuthors* .

Exemple en direct: violon SQL

Les auteurs et les *livres* sont appelés **tables de base** , car ils contiennent une définition de colonne et des données pour les entités réelles du modèle relationnel. *BooksAuthors* est appelé la **table de relations** , car cette table définit la relation entre la table *Books* et *Authors* .

Relations entre les tables

- Chaque auteur peut avoir 1 ou plusieurs livres
- Chaque livre peut avoir un ou plusieurs auteurs

Auteurs

(*voir la table*)

Id	prénom	Pays
1	JD Salinger	Etats-Unis
2	F. Scott. Fitzgerald	Etats-Unis
3	Jane Austen	Royaume-Uni
4	Scott Hanselman	Etats-Unis
5	Jason N. Gaylord	Etats-Unis
6	Pranav Rastogi	Inde
7	Todd Miranda	Etats-Unis
8	Christian Wenz	Etats-Unis

SQL pour créer la table:

```
CREATE TABLE Authors (
    Id INT NOT NULL AUTO_INCREMENT,
    Name VARCHAR(70) NOT NULL,
    Country VARCHAR(100) NOT NULL,
    PRIMARY KEY(Id)
);
```

```
INSERT INTO Authors
    (Name, Country)
VALUES
    ('J.D. Salinger', 'USA'),
    ('F. Scott. Fitzgerald', 'USA'),
    ('Jane Austen', 'UK'),
    ('Scott Hanselman', 'USA'),
    ('Jason N. Gaylord', 'USA'),
    ('Pranav Rastogi', 'India'),
    ('Todd Miranda', 'USA'),
    ('Christian Wenz', 'USA')
;
```

Livres

(*voir la table*)

Id	Titre
1	Le receveur dans le seigle
2	Neuf histoires
3	Franny et Zooey
4	Gatsby le magnifique
5	Tender id la nuit
6	Fierté et préjugés
7	ASP.NET 4.5 professionnel en C # et VB

SQL pour créer la table:

```
CREATE TABLE Books (
    Id INT NOT NULL AUTO_INCREMENT,
    Title VARCHAR(50) NOT NULL,
    PRIMARY KEY(Id)
);

INSERT INTO Books
    (Id, Title)
VALUES
    (1, 'The Catcher in the Rye'),
    (2, 'Nine Stories'),
    (3, 'Franny and Zooey'),
    (4, 'The Great Gatsby'),
    (5, 'Tender id the Night'),
    (6, 'Pride and Prejudice'),
    (7, 'Professional ASP.NET 4.5 in C# and VB')
;
```

LivresAuteurs

(*voir la table*)

BookId	AuthorId
1	1
2	1
3	1
4	2
5	2
6	3
7	4
7	5
7	6
7	7
7	8

SQL pour créer la table:

```
CREATE TABLE BooksAuthors (
    AuthorId INT NOT NULL,
    BookId  INT NOT NULL,
    FOREIGN KEY (AuthorId) REFERENCES Authors(Id),
    FOREIGN KEY (BookId) REFERENCES Books(Id)
);

INSERT INTO BooksAuthors
    (BookId, AuthorId)
VALUES
    (1, 1),
    (2, 1),
    (3, 1),
    (4, 2),
    (5, 2),
    (6, 3),
    (7, 4),
    (7, 5),
    (7, 6),
    (7, 7),
    (7, 8)
;
```

Exemples

Voir tous les auteurs (voir l'exemple en direct):

```
SELECT * FROM Authors;
```

Afficher tous les titres de livres (voir l'exemple en direct):

```
SELECT * FROM Books;
```

Voir tous les livres et leurs auteurs (voir exemple en direct):

```
SELECT
  ba.AuthorId,
  a.Name AuthorName,
  ba.BookId,
  b.Title BookTitle
FROM BooksAuthors ba
  INNER JOIN Authors a ON a.id = ba.authorid
  INNER JOIN Books b ON b.id = ba.bookid
;
```

Table des pays

Dans cet exemple, nous avons une table **Pays** . Un tableau pour les pays a de nombreux usages, en particulier dans les applications financières impliquant des devises et des taux de change.

Exemple en direct: violon SQL

Certains logiciels de données de marché tels que Bloomberg et Reuters exigent que vous donniez à leur API un code de pays de 2 ou 3 caractères avec le code de devise. Par conséquent, cet exemple de tableau contient à la fois la colonne de code `ISO` 2 caractères et les colonnes de code `ISO3` 3 caractères.

Des pays

(*voir la table*)

Id	ISO	ISO3	ISONumérique	Nom du pays	Capitale	ContinentCode	Code de devise
1	AU	AUS	36	Australie	Canberra	OC	AUD
2	DE	DEU	276	Allemagne	Berlin	UE	EUR
2	DANS	INDIANA	356	Inde	New Delhi	COMME	INR

Id	ISO	ISO3	ISONumérique	Nom du pays	Capitale	ContinentCode	Code de devise
3	LA	LAO	418	Laos	Vientiane	COMME	LAK
4	NOUS	Etats-Unis	840	États Unis	Washington	N / A	USD
5	ZW	ZWE	716	Zimbabwe	Harare	UN F	ZWL

SQL pour créer la table:

```
CREATE TABLE Countries (
    Id INT NOT NULL AUTO_INCREMENT,
    ISO VARCHAR(2) NOT NULL,
    ISO3 VARCHAR(3) NOT NULL,
    ISONumeric INT NOT NULL,
    CountryName VARCHAR(64) NOT NULL,
    Capital VARCHAR(64) NOT NULL,
    ContinentCode VARCHAR(2) NOT NULL,
    CurrencyCode VARCHAR(3) NOT NULL,
    PRIMARY KEY(Id)
)
;

INSERT INTO Countries
    (ISO, ISO3, ISONumeric, CountryName, Capital, ContinentCode, CurrencyCode)
VALUES
    ('AU', 'AUS', 36, 'Australia', 'Canberra', 'OC', 'AUD'),
    ('DE', 'DEU', 276, 'Germany', 'Berlin', 'EU', 'EUR'),
    ('IN', 'IND', 356, 'India', 'New Delhi', 'AS', 'INR'),
    ('LA', 'LAO', 418, 'Laos', 'Vientiane', 'AS', 'LAK'),
    ('US', 'USA', 840, 'United States', 'Washington', 'NA', 'USD'),
    ('ZW', 'ZWE', 716, 'Zimbabwe', 'Harare', 'AF', 'ZWL')
;
```

Lire Exemples de bases de données et de tables en ligne:
https://riptutorial.com/fr/sql/topic/280/exemples-de-bases-de-donnees-et-de-tables

Chapitre 26: EXPLIQUEZ et DÉCRIVEZ

Examples

DESCRIBE nom_table;

`DESCRIBE` et `EXPLAIN` sont des synonymes. `DESCRIBE` sur un nom de table renvoie la définition des colonnes.

```
DESCRIBE tablename;
```

Résultat maximum:

```
COLUMN_NAME     COLUMN_TYPE     IS_NULLABLE     COLUMN_KEY     COLUMN_DEFAULT    EXTRA
id              int(11)         NO              PRI            0
auto_increment
test            varchar(255)    YES                            (null)
```

Vous voyez ici les noms de colonnes, suivis du type de colonnes. Il indique si `null` est autorisé dans la colonne et si la colonne utilise un index. la valeur par défaut est également affichée et si la table contient un comportement spécial tel qu'un `auto_increment` .

EXPLIQUEZ Sélectionnez la requête

Un `Explain` face d'une requête `select` vous montre comment la requête sera exécutée. De cette façon, vous pouvez voir si la requête utilise un index ou si vous pouvez optimiser votre requête en ajoutant un index.

Exemple de requête:

```
explain select * from user join data on user.test = data.fk_user;
```

Exemple de résultat:

```
id  select_type  table   type    possible_keys  key     key_len ref       rows  Extra
1   SIMPLE       user    index   test           test    5       (null)    1     Using where;
Using index
1   SIMPLE       data    ref     fk_user        fk_user 5       user.test 1     (null)
```

sur le `type` vous voyez si un index a été utilisé. Dans la colonne `possible_keys` vous voyez si le plan d'exécution peut choisir parmi différents index s'il n'en existe pas. `key` vous indique l'indice d'usage utilisé. `key_len` vous montre la taille en octets pour un élément d'index. Plus cette valeur est faible, plus il y a d'éléments d'index dans la même taille de mémoire et ils peuvent être traités plus rapidement. `rows` indique le nombre de lignes que la requête doit analyser, le plus bas possible.

Lire EXPLIQUEZ et DÉCRIVEZ en ligne: https://riptutorial.com/fr/sql/topic/2928/expliquez-et-

decrivez

Chapitre 27: Expressions de table communes

Syntaxe

- WITH QueryName [(ColumnName, ...)] AS (
  SELECT ...
)
  SELECT ... FROM QueryName ...;

- WITH RECURSIVE QueryName [(ColumnName, ...)] AS (
  SELECT ...
  UNION [TOUS]
  SELECT ... FROM QueryName ...
)
  SELECT ... FROM QueryName ...;

Remarques

Documentation officielle: clause WITH

Une expression de table commune est un jeu de résultats temporaire pouvant résulter d'une requête complexe. Il est défini à l'aide de la clause WITH. CTE améliore la lisibilité et est créé dans la mémoire plutôt que dans la base de données TempDB où la variable Table temporaire et Table est créée.

Concepts clés des expressions de table communes:

- Peut être utilisé pour casser des requêtes complexes, en particulier des jointures complexes et des sous-requêtes.
- Est un moyen d'encapsuler une définition de requête.
- Ne persister que jusqu'à l'exécution de la requête suivante.
- Une utilisation correcte peut entraîner des améliorations de la qualité / maintenabilité et de la rapidité du code.
- Peut être utilisé pour référencer la table résultante plusieurs fois dans la même instruction (élimine la duplication dans SQL).
- Peut remplacer une vue lorsque l'utilisation générale d'une vue n'est pas requise; en d'autres termes, vous n'avez pas besoin de stocker la définition dans les métadonnées.
- Sera exécuté lorsqu'il est appelé, pas lorsqu'il est défini. Si le CTE est utilisé plusieurs fois dans une requête, il sera exécuté plusieurs fois (éventuellement avec des résultats différents).

Examples

Requête temporaire

Celles-ci se comportent de la même manière que les sous-requêtes imbriquées mais avec une syntaxe différente.

```
WITH ReadyCars AS (
  SELECT *
  FROM Cars
  WHERE Status = 'READY'
)
SELECT ID, Model, TotalCost
FROM ReadyCars
ORDER BY TotalCost;
```

ID	Modèle	Coût total
1	Ford F-150	200
2	Ford F-150	230

Syntaxe de sous-requête équivalente

```
SELECT ID, Model, TotalCost
FROM (
  SELECT *
  FROM Cars
  WHERE Status = 'READY'
) AS ReadyCars
ORDER BY TotalCost
```

récursivement monter dans un arbre

```
WITH RECURSIVE ManagersOfJonathon AS (
    -- start with this row
    SELECT *
    FROM Employees
    WHERE ID = 4

    UNION ALL

    -- get manager(s) of all previously selected rows
    SELECT Employees.*
    FROM Employees
    JOIN ManagersOfJonathon
        ON Employees.ID = ManagersOfJonathon.ManagerID
)
SELECT * FROM ManagersOfJonathon;
```

Id	FName	LName	Numéro de téléphone	ManagerId	DépartementId
4	Johnathon	Forgeron	1212121212	**2**	1
2	John	Johnson	2468101214	**1**	1
1	James	Forgeron	1234567890	NUL	1

générer des valeurs

La plupart des bases de données ne permettent pas de générer une série de nombres pour une utilisation ponctuelle. Cependant, les expressions de table communes peuvent être utilisées avec la récursivité pour émuler ce type de fonction.

L'exemple suivant génère une expression de table commune appelée `Numbers` avec une colonne `i` comportant une ligne pour les numéros 1 à 5:

```
--Give a table name `Numbers" and a column `i` to hold the numbers
WITH Numbers(i) AS (
    --Starting number/index
    SELECT 1
    --Top-level UNION ALL operator required for recursion
    UNION ALL
    --Iteration expression:
    SELECT i + 1
    --Table expression we first declared used as source for recursion
    FROM Numbers
    --Clause to define the end of the recursion
    WHERE i < 5
)
--Use the generated table expression like a regular table
SELECT i FROM Numbers;
```

je
1
2
3
4
5

Cette méthode peut être utilisée avec n'importe quel intervalle numérique, ainsi que d'autres types de données.

énumérer récursivement un sous-arbre

```
WITH RECURSIVE ManagedByJames(Level, ID, FName, LName) AS (
    -- start with this row
    SELECT 1, ID, FName, LName
    FROM Employees
    WHERE ID = 1

    UNION ALL

    -- get employees that have any of the previously selected rows as manager
    SELECT ManagedByJames.Level + 1,
           Employees.ID,
```

```
          Employees.FName,
          Employees.LName
    FROM Employees
    JOIN ManagedByJames
        ON Employees.ManagerID = ManagedByJames.ID

    ORDER BY 1 DESC   -- depth-first search
)
SELECT * FROM ManagedByJames;
```

Niveau	ID	FName	LName
1	1	James	Forgeron
2	2	John	Johnson
3	4	Johnathon	Forgeron
2	3	Michael	Williams

Fonctionnalité Oracle CONNECT BY avec les CTE récursifs

La fonctionnalité CONNECT BY d'Oracle fournit de nombreuses fonctionnalités utiles et non triviales qui ne sont pas intégrées lors de l'utilisation de CTE récursifs standard SQL. Cet exemple réplique ces fonctionnalités (avec quelques ajouts par souci d'exhaustivité), en utilisant la syntaxe SQL Server. Il est très utile pour les développeurs Oracle de trouver de nombreuses fonctionnalités manquantes dans leurs requêtes hiérarchiques sur d'autres bases de données, mais il sert également à présenter ce qui peut être fait avec une requête hiérarchique en général.

```
  WITH tbl AS (
       SELECT id, name, parent_id
         FROM mytable)
     , tbl_hierarchy AS (
       /* Anchor */
       SELECT 1 AS "LEVEL"
            --, 1 AS CONNECT_BY_ISROOT
            --, 0 AS CONNECT_BY_ISBRANCH
            , CASE WHEN t.id IN (SELECT parent_id FROM tbl) THEN 0 ELSE 1 END AS
CONNECT_BY_ISLEAF
            , 0 AS CONNECT_BY_ISCYCLE
            , '/' + CAST(t.id   AS VARCHAR(MAX)) + '/' AS SYS_CONNECT_BY_PATH_id
            , '/' + CAST(t.name AS VARCHAR(MAX)) + '/' AS SYS_CONNECT_BY_PATH_name
            , t.id AS root_id
            , t.*
         FROM tbl t
        WHERE t.parent_id IS NULL                              -- START WITH parent_id IS NULL
       UNION ALL
       /* Recursive */
       SELECT th."LEVEL" + 1 AS "LEVEL"
            --, 0 AS CONNECT_BY_ISROOT
            --, CASE WHEN t.id IN (SELECT parent_id FROM tbl) THEN 1 ELSE 0 END AS
CONNECT_BY_ISBRANCH
            , CASE WHEN t.id IN (SELECT parent_id FROM tbl) THEN 0 ELSE 1 END AS
CONNECT_BY_ISLEAF
            , CASE WHEN th.SYS_CONNECT_BY_PATH_id LIKE '%/' + CAST(t.id AS VARCHAR(MAX)) +
```

```
'/%' THEN 1 ELSE 0 END AS CONNECT_BY_ISCYCLE
            , th.SYS_CONNECT_BY_PATH_id   + CAST(t.id   AS VARCHAR(MAX)) + '/' AS
SYS_CONNECT_BY_PATH_id
            , th.SYS_CONNECT_BY_PATH_name + CAST(t.name AS VARCHAR(MAX)) + '/' AS
SYS_CONNECT_BY_PATH_name
            , th.root_id
            , t.*
         FROM tbl t
              JOIN tbl_hierarchy th ON (th.id = t.parent_id) -- CONNECT BY PRIOR id =
parent_id
        WHERE th.CONNECT_BY_ISCYCLE = 0)                     -- NOCYCLE
SELECT th.*
     --, REPLICATE(' ', (th."LEVEL" - 1) * 3) + th.name AS tbl_hierarchy
  FROM tbl_hierarchy th
       JOIN tbl CONNECT_BY_ROOT ON (CONNECT_BY_ROOT.id = th.root_id)
 ORDER BY th.SYS_CONNECT_BY_PATH_name;                       -- ORDER SIBLINGS BY name
```

Fonctionnalités CONNECT BY démontrées ci-dessus, avec des explications:

- Clauses
  - CONNECT BY: Spécifie la relation qui définit la hiérarchie.
  - START WITH: Spécifie les nœuds racine.
  - ORDER SIBLINGS BY: Résultats des commandes correctement.
- Paramètres
  - NOCYCLE: Arrête le traitement d'une branche lorsqu'une boucle est détectée. Les hiérarchies valides sont des graphes acycliques dirigés et les références circulaires violent cette construction.
- Les opérateurs
  - PRIOR: Obtient les données du parent du noeud.
  - CONNECT_BY_ROOT: Obtient les données de la racine du noeud.
- Pseudocolonnes
  - NIVEAU: Indique la distance entre le nœud et sa racine.
  - CONNECT_BY_ISLEAF: Indique un nœud sans enfants.
  - CONNECT_BY_ISCYCLE: Indique un noeud avec une référence circulaire.
- Les fonctions
  - SYS_CONNECT_BY_PATH: Retourne une représentation aplatie / concaténée du chemin d'accès au noeud depuis sa racine.

Génération récursive de dates, étendue pour inclure la liste des équipes comme exemple

```
DECLARE @DateFrom DATETIME = '2016-06-01 06:00'
DECLARE @DateTo DATETIME = '2016-07-01 06:00'
DECLARE @IntervalDays INT = 7

-- Transition Sequence = Rest & Relax into Day Shift into Night Shift
-- RR (Rest & Relax) = 1
-- DS (Day Shift) = 2
-- NS (Night Shift) = 3

;WITH roster AS
(
   SELECT @DateFrom AS RosterStart, 1 AS TeamA, 2 AS TeamB, 3 AS TeamC
```

```
   UNION ALL
   SELECT DATEADD(d, @IntervalDays, RosterStart),
          CASE TeamA WHEN 1 THEN 2 WHEN 2 THEN 3 WHEN 3 THEN 1 END AS TeamA,
          CASE TeamB WHEN 1 THEN 2 WHEN 2 THEN 3 WHEN 3 THEN 1 END AS TeamB,
          CASE TeamC WHEN 1 THEN 2 WHEN 2 THEN 3 WHEN 3 THEN 1 END AS TeamC
   FROM roster WHERE RosterStart < DATEADD(d, -@IntervalDays, @DateTo)
)

SELECT RosterStart,
       ISNULL(LEAD(RosterStart) OVER (ORDER BY RosterStart), RosterStart + @IntervalDays) AS
RosterEnd,
       CASE TeamA WHEN 1 THEN 'RR' WHEN 2 THEN 'DS' WHEN 3 THEN 'NS' END AS TeamA,
       CASE TeamB WHEN 1 THEN 'RR' WHEN 2 THEN 'DS' WHEN 3 THEN 'NS' END AS TeamB,
       CASE TeamC WHEN 1 THEN 'RR' WHEN 2 THEN 'DS' WHEN 3 THEN 'NS' END AS TeamC
FROM roster
```

Résultat

Ie Pour TeamA est sur R & R, TeamB est sur Day Shift et TeamC est sur Night Shift.

Results | Messages

	RosterStart	RosterEnd	TeamA	TeamB	TeamC
1	2016-06-01 06:00:00.000	2016-06-08 06:00:00.000	RR	DS	NS
2	2016-06-08 06:00:00.000	2016-06-15 06:00:00.000	DS	NS	RR
3	2016-06-15 06:00:00.000	2016-06-22 06:00:00.000	NS	RR	DS
4	2016-06-22 06:00:00.000	2016-06-29 06:00:00.000	RR	DS	NS
5	2016-06-29 06:00:00.000	2016-07-06 06:00:00.000	DS	NS	RR

Refactoring d'une requête pour utiliser les expressions de table communes

Supposons que nous voulons obtenir toutes les catégories de produits dont les ventes totales sont supérieures à 20.

Voici une requête sans expressions de table communes:

```
SELECT category.description, sum(product.price) as total_sales
FROM sale
LEFT JOIN product on sale.product_id = product.id
LEFT JOIN category on product.category_id = category.id
GROUP BY category.id, category.description
HAVING sum(product.price) > 20
```

Et une requête équivalente utilisant les expressions de table communes:

```
WITH all_sales AS (
  SELECT product.price, category.id as category_id, category.description as
category_description
  FROM sale
  LEFT JOIN product on sale.product_id = product.id
  LEFT JOIN category on product.category_id = category.id
)
, sales_by_category AS (
  SELECT category_description, sum(price) as total_sales
  FROM all_sales
```

```
  GROUP BY category_id, category_description
)
SELECT * from sales_by_category WHERE total_sales > 20
```

Exemple d'un SQL complexe avec une expression de table commune

Supposons que nous voulons interroger les "produits les moins chers" des "catégories supérieures".

Voici un exemple de requête utilisant les expressions de table communes

```
-- all_sales: just a simple SELECT with all the needed JOINS
WITH all_sales AS (
  SELECT
  product.price as product_price,
  category.id as category_id,
  category.description as category_description
  FROM sale
  LEFT JOIN product on sale.product_id = product.id
  LEFT JOIN category on product.category_id = category.id
)
-- Group by category
, sales_by_category AS (
  SELECT category_id, category_description,
  sum(product_price) as total_sales
  FROM all_sales
  GROUP BY category_id, category_description
)
-- Filtering total_sales > 20
, top_categories AS (
  SELECT * from sales_by_category WHERE total_sales > 20
)
-- all_products: just a simple SELECT with all the needed JOINS
, all_products AS (
  SELECT
  product.id as product_id,
  product.description as product_description,
  product.price as product_price,
  category.id as category_id,
  category.description as category_description
  FROM product
  LEFT JOIN category on product.category_id = category.id
)
-- Order by product price
, cheapest_products AS (
  SELECT * from all_products
  ORDER by product_price ASC
)
-- Simple inner join
, cheapest_products_from_top_categories AS (
  SELECT product_description, product_price
  FROM cheapest_products
  INNER JOIN top_categories ON cheapest_products.category_id = top_categories.category_id
)
--The main SELECT
SELECT * from cheapest_products_from_top_categories
```

Lire Expressions de table communes en ligne: https://riptutorial.com/fr/sql/topic/747/expressions-

de-table-communes

Chapitre 28: Filtrer les résultats en utilisant WHERE et HAVING

Syntaxe

- SELECT nom_colonne
  FROM nom_table
  WHERE valeur d'opérateur nom_colonne
- SELECT nom_colonne, agrégat_fonction (nom_colonne)
  FROM nom_table
  GROUP BY nom_colonne
  Valeur d'opérateur agrégat_fonction (nom_colonne)

Examples

La clause WHERE ne renvoie que les lignes correspondant à ses critères

Steam a une section de jeux de moins de 10 $ sur sa page de magasin. Quelque part au cœur de leurs systèmes, il y a probablement une requête qui ressemble à:

```
SELECT *
FROM Items
WHERE Price < 10
```

Utilisez IN pour renvoyer des lignes avec une valeur contenue dans une liste

Cet exemple utilise la table Car de l'exemple de bases de données.

```
SELECT *
FROM Cars
WHERE TotalCost IN (100, 200, 300)
```

Cette requête renvoie la voiture n ° 2 qui coûte 200 et la voiture n ° 3 qui coûte 100. Notez que cela équivaut à utiliser plusieurs clauses avec `OR` , par exemple:

```
SELECT *
FROM Cars
WHERE TotalCost = 100 OR TotalCost = 200 OR TotalCost = 300
```

Utilisez LIKE pour trouver des chaînes et des sous-chaînes correspondantes

Voir la documentation complète sur l'opérateur LIKE .

Cet exemple utilise la table Employees de l'exemple de base de données.

```
SELECT *
FROM Employees
WHERE FName LIKE 'John'
```

Cette requête ne renverra que l'employé n ° 1 dont le prénom correspond à «John» exactement.

```
SELECT *
FROM Employees
WHERE FName like 'John%'
```

Ajouter `%` vous permet de rechercher une sous-chaîne:

- `John%` - renverra tout employé dont le nom commence par «John», suivi de toute quantité de caractères
- `%John` - renverra tout employé dont le nom se termine par «John», par n'importe quel nombre de caractères
- `%John%` - renverra tout employé dont le nom contient «John» n'importe où dans la valeur

Dans ce cas, la requête renvoie Employee # 2 dont le nom est "John" ainsi que Employee # 4 dont le nom est "Johnathon".

Clause WHERE avec des valeurs NULL / NOT NULL

```
SELECT *
FROM Employees
WHERE ManagerId IS NULL
```

Cette instruction renvoie tous les enregistrements d' employé dont la valeur de la colonne `ManagerId` est `NULL` .

Le résultat sera:

```
Id    FName    LName    PhoneNumber    ManagerId    DepartmentId
1     James    Smith    1234567890     NULL         1
```

```
SELECT *
FROM Employees
WHERE ManagerId IS NOT NULL
```

Cette instruction retournera tous les enregistrements d' employé dont la valeur de `ManagerId` n'est *pas* `NULL` .

Le résultat sera:

```
Id    FName       LName     PhoneNumber    ManagerId    DepartmentId
2     John        Johnson   2468101214     1            1
3     Michael     Williams  1357911131     1            2
4     Johnathon   Smith     1212121212     2            1
```

Remarque: La même requête ne renverra pas de résultats si vous modifiez la clause `WHERE ManagerId = NULL` en `WHERE ManagerId = NULL` ou `WHERE ManagerId <> NULL` .

Utilisez HAVING avec des fonctions d'agrégat

Contrairement à la clause `WHERE` , `HAVING` peut être utilisé avec des fonctions d'agrégat.

> Une fonction d'agrégat est une fonction où les valeurs de plusieurs lignes sont regroupées en tant que données d'entrée sur certains critères pour former une valeur unique de signification ou de mesure plus significative (Wikipedia).

Les fonctions d'agrégation courantes incluent `COUNT()` , `SUM()` , `MIN()` et `MAX()` .

Cet exemple utilise la table Car de l'exemple de bases de données.

```
SELECT CustomerId, COUNT(Id) AS [Number of Cars]
FROM Cars
GROUP BY CustomerId
HAVING COUNT(Id) > 1
```

Cette requête renvoie le nombre `CustomerId` et `Number of Cars` de tout client disposant de plusieurs voitures. Dans ce cas, le seul client qui possède plus d'une voiture est le client n ° 1.

Les résultats ressembleront à:

N ° de client	Nombre de voitures
1	2

Utilisez entre pour filtrer les résultats

Les exemples suivants utilisent les exemples de bases de données Ventes d'articles et Clients .

> Remarque: l'opérateur BETWEEN *est* inclus.

Utilisation de l'opérateur BETWEEN avec des nombres:

```
SELECT * From ItemSales
WHERE Quantity BETWEEN 10 AND 17
```

Cette requête renvoie tous les enregistrements `ItemSales` dont la quantité est supérieure ou égale à 10 et inférieure ou égale à 17. Les résultats seront les suivants:

Id	Date de vente	ID de l'article	Quantité	Prix
1	2013-07-01	100	dix	34,5
4	2013-07-23	100	15	34,5

Id	Date de vente	ID de l'article	Quantité	Prix
5	2013-07-24	145	dix	34,5

Utilisation de l'opérateur BETWEEN avec les valeurs de date:

```
SELECT * From ItemSales
WHERE SaleDate BETWEEN '2013-07-11' AND '2013-05-24'
```

Cette requête renverra tous `ItemSales` enregistrements avec un `SaleDate` qui est supérieur ou égal à 11 Juillet 2013 et inférieure ou égale au 24 mai 2013.

Id	Date de vente	ID de l'article	Quantité	Prix
3	2013-07-11	100	20	34,5
4	2013-07-23	100	15	34,5
5	2013-07-24	145	dix	34,5

> Lorsque vous comparez des valeurs de date / heure au lieu de dates, vous devrez peut-être convertir les valeurs de date / heure en valeurs de date ou ajouter ou soustraire 24 heures pour obtenir les résultats corrects.

Utilisation de l'opérateur BETWEEN avec des valeurs de texte:

```
SELECT Id, FName, LName FROM Customers
WHERE LName BETWEEN 'D' AND 'L';
```

Exemple en direct: violon SQL

Cette requête renvoie tous les clients dont le nom est alphabétiquement compris entre les lettres "D" et "L". Dans ce cas, les clients n ° 1 et n ° 3 seront renvoyés. Le client n ° 2, dont le nom commence par un «M», ne sera pas inclus.

Id	FName	LName
1	William	Jones
3	Richard	Davis

Égalité

```
SELECT * FROM Employees
```

Cette instruction retournera toutes les lignes de la table `Employees` .

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
1    James     Smith    1234567890    NULL        1               1000    01-01-2002    01-01-
2002    01-01-2002
2    John      Johnson  2468101214    1           1               400     23-03-2005    23-03-
2005    01-01-2002
3    Michael   Williams 1357911131    1           2               600     12-05-2009    12-05-
2009    NULL
4    Johnathon Smith    1212121212    2           1               500     24-07-2016    24-07-
2016    01-01-2002
```

L'utilisation d'un `WHERE` à la fin de votre `SELECT` vous permet de limiter les lignes renvoyées à une condition. Dans ce cas, il existe une correspondance exacte avec le signe `=` :

```
SELECT * FROM Employees WHERE DepartmentId = 1
```

Ne renverra que les lignes où `DepartmentId` est égal à `1` :

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
1    James     Smith    1234567890    NULL        1               1000    01-01-2002    01-01-
2002    01-01-2002
2    John      Johnson  2468101214    1           1               400     23-03-2005    23-03-
2005    01-01-2002
4    Johnathon Smith    1212121212    2           1               500     24-07-2016    24-07-
2016    01-01-2002
```

ET et OU

Vous pouvez également combiner plusieurs opérateurs pour créer des conditions `WHERE` plus complexes. Les exemples suivants utilisent la table `Employees` :

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
1    James     Smith    1234567890    NULL        1               1000    01-01-2002    01-01-
2002    01-01-2002
2    John      Johnson  2468101214    1           1               400     23-03-2005    23-03-
2005    01-01-2002
3    Michael   Williams 1357911131    1           2               600     12-05-2009    12-05-
2009    NULL
4    Johnathon Smith    1212121212    2           1               500     24-07-2016    24-07-
2016    01-01-2002
```

ET

```
SELECT * FROM Employees WHERE DepartmentId = 1 AND ManagerId = 1
```

Reviendra:

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
2    John      Johnson  2468101214    1           1               400     23-03-2005    23-03-
```

```
2005    01-01-2002
```

OU

```
SELECT * FROM Employees WHERE DepartmentId = 2 OR ManagerId = 2
```

Reviendra:

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
3    Michael   Williams 1357911131    1           2               600     12-05-2009    12-05-
2009    NULL
4    Johnathon Smith    1212121212    2           1               500     24-07-2016    24-07-
2016    01-01-2002
```

Utilisez HAVING pour vérifier plusieurs conditions dans un groupe

Tableau des commandes

N ° de client	ProductId	Quantité	Prix
1	2	5	100
1	3	2	200
1	4	1	500
2	1	4	50
3	5	6	700

Pour vérifier les clients qui ont commandé les deux - ProductID 2 et 3, HAVING peut être utilisé

```
select customerId
from orders
where productID in (2,3)
group by customerId
having count(distinct productID) = 2
```

Valeur de retour:

N ° de client
1

La requête sélectionne uniquement les enregistrements avec les ID de produit dans les questions et la clause HAVING recherche les groupes ayant 2 ID de produit et pas un seul.

Une autre possibilité serait

```
select customerId
from orders
group by customerId
having sum(case when productID = 2 then 1 else 0 end) > 0
   and sum(case when productID = 3 then 1 else 0 end) > 0
```

Cette requête sélectionne uniquement les groupes ayant au moins un enregistrement avec productID 2 et au moins un avec productID 3.

Où EXISTE

Sélectionne les enregistrements dans `TableName` qui ont des enregistrements correspondants dans `TableName1` .

```
SELECT * FROM TableName t WHERE EXISTS (
    SELECT 1 FROM TableName1 t1 where t.Id = t1.Id)
```

Lire Filtrer les résultats en utilisant WHERE et HAVING en ligne:
https://riptutorial.com/fr/sql/topic/636/filtrer-les-resultats-en-utilisant-where-et-having

Chapitre 29: Fonctions (agrégées)

Syntaxe

- Fonction (expression [*DISTINCT*]) -DISTINCT est un paramètre facultatif
- AVG (expression [ALL | DISTINCT])
- COUNT ({[ALL | DISTINCT] expression] | *})
- GROUPING (<expression_colonne>)
- MAX (expression [ALL | DISTINCT])
- MIN (expression [ALL | DISTINCT])
- SUM ([ALL | DISTINCT] expression)
- VAR (expression [ALL | DISTINCT])
  OVER ([partition_by_clause] order_by_clause)
- VARP (expression [ALL | DISTINCT])
  OVER ([partition_by_clause] order_by_clause
- STDEV (expression [ALL | DISTINCT])
  OVER ([partition_by_clause] order_by_clause)
- STDEVP (expression [ALL | DISTINCT])
  OVER ([partition_by_clause] order_by_clause)

Remarques

Dans la gestion de base de données, une fonction d'agrégat est une fonction où les valeurs de plusieurs lignes sont regroupées en tant qu'entrée sur certains critères pour former une valeur unique de signification ou de mesure plus significative telle qu'un ensemble, un sac ou une liste.

```
MIN         returns the smallest value in a given column
MAX         returns the largest value in a given column
SUM         returns the sum of the numeric values in a given column
AVG         returns the average value of a given column
COUNT       returns the total number of values in a given column
COUNT(*)    returns the number of rows in a table
GROUPING    Is a column or an expression that contains a column in a GROUP BY clause.
STDEV       returns the statistical standard deviation of all values in the specified
expression.
STDEVP      returns the statistical standard deviation for the population for all values in the
specified expression.
VAR         returns the statistical variance of all values in the specified expression. may be
followed by the OVER clause.
VARP        returns the statistical variance for the population for all values in the specified
expression.
```

> Les fonctions d'agrégat sont utilisées pour calculer une "colonne de données numériques renvoyée" de votre `SELECT` . Ils résument essentiellement les résultats d'une colonne particulière de données sélectionnées. - SQLCourse2.com

Toutes les fonctions d'agrégation ignorent les valeurs NULL.

Examples

SOMME

`Sum` fonction somme additionne la valeur de toutes les lignes du groupe. Si la clause group by est omise, puis somme toutes les lignes.

```
select sum(salary) TotalSalary
from employees;
```

Salaire total
2500

```
select DepartmentId, sum(salary) TotalSalary
from employees
group by DepartmentId;
```

DépartementId	Salaire total
1	2000
2	500

Agrégation conditionnelle

Tableau des paiements

Client	Type de paiement	Montant
Peter	Crédit	100
Peter	Crédit	300
John	Crédit	1000
John	Débit	500

```
select customer,
       sum(case when payment_type = 'credit' then amount else 0 end) as credit,
       sum(case when payment_type = 'debit' then amount else 0 end) as debit
from payments
group by customer
```

Résultat:

Client	Crédit	Débit
Peter	400	0
John	1000	500

```
select customer,
       sum(case when payment_type = 'credit' then 1 else 0 end) as credit_transaction_count,
       sum(case when payment_type = 'debit' then 1 else 0 end) as debit_transaction_count
from payments
group by customer
```

Résultat:

Client	credit_transaction_count	debit_transaction_count
Peter	2	0
John	1	1

AVG ()

La fonction d'agrégation AVG () renvoie la moyenne d'une expression donnée, généralement des valeurs numériques dans une colonne. Supposons que nous ayons un tableau contenant le calcul annuel de la population dans les villes du monde entier. Les enregistrements pour New York City ressemblent à ceux ci-dessous:

EXEMPLE DE TABLE

Nom de Ville	population	an
La ville de New York	8 550 405	2015
La ville de New York	...	...
La ville de New York	8 000 906	2005

Pour sélectionner la population moyenne de la ville de New York, États-Unis, à partir d'un tableau contenant les noms de villes, les mesures de population et les années de mesure pour les dix dernières années:

QUESTION

```
select city_name, AVG(population) avg_population
from city_population
where city_name = 'NEW YORK CITY';
```

Notez que l'année de mesure est absente de la requête, car la population est moyennée au fil du temps.

RÉSULTATS

Nom de Ville	avg_population
La ville de New York	8 250 754

> Remarque: La fonction AVG () convertit les valeurs en types numériques. Ceci est particulièrement important lorsque vous travaillez avec des dates.

Concaténation de liste

Crédit partiel à cette réponse SO.

La concaténation de liste agrège une colonne ou une expression en combinant les valeurs en une seule chaîne pour chaque groupe. Une chaîne pour délimiter chaque valeur (vide ou une virgule si elle est omise) et l'ordre des valeurs dans le résultat peuvent être spécifiés. Bien qu'il ne fasse pas partie du standard SQL, chaque grand fournisseur de bases de données relationnelles le prend en charge à sa manière.

MySQL

```
SELECT ColumnA
     , GROUP_CONCAT(ColumnB ORDER BY ColumnB SEPARATOR ',') AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

Oracle et DB2

```
SELECT ColumnA
     , LISTAGG(ColumnB, ',') WITHIN GROUP (ORDER BY ColumnB) AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

PostgreSQL

```
SELECT ColumnA
     , STRING_AGG(ColumnB, ',' ORDER BY ColumnB) AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
```

```
 ORDER BY ColumnA;
```

serveur SQL

SQL Server 2016 et versions antérieures

(CTE inclus pour encourager le principe DRY)

```
  WITH CTE_TableName AS (
       SELECT ColumnA, ColumnB
         FROM TableName)
SELECT t0.ColumnA
     , STUFF((
       SELECT ',' + t1.ColumnB
         FROM CTE_TableName t1
        WHERE t1.ColumnA = t0.ColumnA
        ORDER BY t1.ColumnB
          FOR XML PATH('')), 1, 1, '') AS ColumnBs
  FROM CTE_TableName t0
 GROUP BY t0.ColumnA
 ORDER BY ColumnA;
```

SQL Server 2017 et SQL Azure

```
SELECT ColumnA
     , STRING_AGG(ColumnB, ',') WITHIN GROUP (ORDER BY ColumnB) AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

SQLite

sans commander:

```
SELECT ColumnA
     , GROUP_CONCAT(ColumnB, ',') AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

la commande nécessite une sous-requête ou CTE:

```
  WITH CTE_TableName AS (
       SELECT ColumnA, ColumnB
         FROM TableName
        ORDER BY ColumnA, ColumnB)
SELECT ColumnA
     , GROUP_CONCAT(ColumnB, ',') AS ColumnBs
  FROM CTE_TableName
```

```
GROUP BY ColumnA
ORDER BY ColumnA;
```

Compter

Vous pouvez compter le nombre de lignes:

```
SELECT count(*) TotalRows
FROM employees;
```

TotalRows
4

Ou comptez les employés par département:

```
SELECT DepartmentId, count(*) NumEmployees
FROM employees
GROUP BY DepartmentId;
```

DépartementId	NumEmployees
1	3
2	1

Vous pouvez compter sur une colonne / expression avec l'effet qui ne compte pas les valeurs `NULL` :

```
SELECT count(ManagerId) mgr
FROM EMPLOYEES;
```

mgr
3

(Il y a une colonne managerID valeur nulle)

Vous pouvez également utiliser **DISTINCT** dans une autre fonction telle que **COUNT** pour ne trouver que les membres **DISTINCT** de l'ensemble pour effectuer l'opération.

Par exemple:

```
SELECT COUNT(ContinentCode) AllCount
,      COUNT(DISTINCT ContinentCode) SingleCount
FROM Countries;
```

Renverra des valeurs différentes. Le compte *SingleCount* ne comptera qu'une fois les continents,

tandis que le compte *AllCount* comportera des doublons.

ContinentCode
OC
UE
COMME
N / A
N / A
UN F
UN F

AllCount: 7 SingleCount: 5

Max

Trouvez la valeur maximale de la colonne:

```
select max(age) from employee;
```

L'exemple ci-dessus renverra la valeur la plus élevée pour l' `age` de la colonne de la table des `employee` .

Syntaxe:

```
SELECT MAX(column_name) FROM table_name;
```

Min

Trouvez la plus petite valeur de colonne:

```
 select min(age) from employee;
```

L'exemple ci-dessus renvoie la plus petite valeur pour l' `age` de la colonne de la table des `employee` .

Syntaxe:

```
 SELECT MIN(column_name) FROM table_name;
```

Lire Fonctions (agrégées) en ligne: https://riptutorial.com/fr/sql/topic/1002/fonctions--agregees-

Chapitre 30: Fonctions (analytique)

Introduction

Vous utilisez des fonctions analytiques pour déterminer des valeurs basées sur des groupes de valeurs. Par exemple, vous pouvez utiliser ce type de fonction pour déterminer les totaux, les pourcentages ou le résultat supérieur d'un groupe.

Syntaxe

1. FIRST_VALUE (expression_ scalaire) OVER ([partition_by_clause] order_by_clause [rows_range_clause])
2. LAST_VALUE (expression_ scalaire) OVER ([partition_by_clause] order_by_clause [rows_range_clause])
3. LAG (scalar_expression [, offset] [, default]) OVER ([partition_by_clause] order_by_clause)
4. LEAD (scalar_expression [, offset], [default]) OVER ([partition_by_clause] order_by_clause)
5. PERCENT_RANK () OVER ([partition_by_clause] order_by_clause)
6. CUME_DIST () OVER ([partition_by_clause] order_by_clause)
7. PERCENTILE_DISC (numeric_literal) WITHIN GROUP (ORDER BY order_by_expression [ASC | DESC]) OVER ([<partition_by_clause>])
8. PERCENTILE_CONT (numeric_literal) WITHIN GROUP (ORDER BY order_by_expression [ASC | DESC]) OVER ([<partition_by_clause>])

Examples

FIRST_VALUE

Vous utilisez la fonction `FIRST_VALUE` pour déterminer la première valeur d'un jeu de résultats ordonné, que vous identifiez à l'aide d'une expression scalaire.

```
SELECT StateProvinceID, Name, TaxRate,
       FIRST_VALUE(StateProvinceID)
        OVER(ORDER BY TaxRate ASC) AS FirstValue
FROM SalesTaxRate;
```

Dans cet exemple, la fonction `FIRST_VALUE` est utilisée pour renvoyer l' `ID` de l'état ou de la province ayant le taux de taxe le plus bas. La clause `OVER` est utilisée pour ordonner les taux de taxe afin d'obtenir le taux le plus bas.

StateProvinceID	prénom	Taux d'imposition	Première valeur
74	Taxe de vente de l'État de l'Utah	5.00	74
36	Taxe de vente de l'état du	6,75	74

StateProvinceID	prénom	Taux d'imposition	Première valeur
	Minnesota		
30	Massachusetts State Tax Tax	7.00	74
1	TPS canadienne	7.00	74
57	TPS canadienne	7.00	74
63	TPS canadienne	7.00	74

LAST_VALUE

La fonction `LAST_VALUE` fournit la dernière valeur d'un ensemble de résultats ordonnés, que vous spécifiez à l'aide d'une expression scalaire.

```
SELECT TerritoryID, StartDate, BusinessentityID,
       LAST_VALUE(BusinessentityID)
        OVER(ORDER BY TerritoryID) AS LastValue
FROM SalesTerritoryHistory;
```

Cet exemple utilise la fonction `LAST_VALUE` pour renvoyer la dernière valeur de chaque jeu de lignes dans les valeurs `LAST_VALUE` .

TerritoireID	Date de début	BusinessentityID	LastValue
1	2005-07-01 00.00.00.000	280	283
1	2006-11-01 00.00.00.000	284	283
1	2005-07-01 00.00.00.000	283	283
2	2007-01-01 00.00.00.000	277	275
2	2005-07-01 00.00.00.000	275	275
3	2007-01-01 00.00.00.000	275	277

LAG et LEAD

La fonction `LAG` fournit des données sur les lignes avant la ligne en cours dans le même jeu de résultats. Par exemple, dans une `SELECT` , vous pouvez comparer les valeurs de la ligne actuelle avec les valeurs d'une ligne précédente.

Vous utilisez une expression scalaire pour spécifier les valeurs à comparer. Le paramètre offset est le nombre de lignes avant la ligne en cours qui sera utilisé dans la comparaison. Si vous ne spécifiez pas le nombre de lignes, la valeur par défaut d'une ligne est utilisée.

Le paramètre par défaut spécifie la valeur à renvoyer lorsque l'expression à l'offset a une valeur `NULL` . Si vous ne spécifiez pas de valeur, la valeur `NULL` est renvoyée.

La fonction `LEAD` fournit des données sur les lignes après la ligne en cours dans le jeu de lignes. Par exemple, dans une `SELECT` , vous pouvez comparer les valeurs de la ligne en cours avec les valeurs de la ligne suivante.

Vous spécifiez les valeurs à comparer en utilisant une expression scalaire. Le paramètre offset est le nombre de lignes après la ligne en cours à utiliser dans la comparaison.

Vous spécifiez la valeur à renvoyer lorsque l'expression à l'offset a une valeur `NULL` à l'aide du paramètre par défaut. Si vous ne spécifiez pas ces paramètres, la valeur par défaut d'une ligne est utilisée et la valeur `NULL` est renvoyée.

```
SELECT BusinessEntityID, SalesYTD,
       LEAD(SalesYTD, 1, 0) OVER(ORDER BY BusinessEntityID) AS "Lead value",
       LAG(SalesYTD, 1, 0) OVER(ORDER BY BusinessEntityID) AS "Lag value"
FROM SalesPerson;
```

Cet exemple utilise les fonctions LEAD et LAG pour comparer les valeurs de vente de chaque employé à ce jour avec celles des employés répertoriés ci-dessus et ci-dessous, les enregistrements étant classés en fonction de la colonne BusinessEntityID.

BusinessEntityID	SalesYTD	Valeur de plomb	Valeur de retard
274	559697.5639	3763178.1787	0,0000
275	3763178.1787	4251368.5497	559697.5639
276	4251368.5497	3189418.3662	3763178.1787
277	3189418.3662	1453719.4653	4251368.5497
278	1453719.4653	2315185.6110	3189418.3662
279	2315185.6110	1352577.1325	1453719.4653

PERCENT_RANK et CUME_DIST

La fonction `PERCENT_RANK` calcule le classement d'une ligne par rapport à l'ensemble de lignes. Le pourcentage est basé sur le nombre de lignes du groupe dont la valeur est inférieure à la ligne actuelle.

La première valeur du jeu de résultats a toujours un classement en pourcentage de zéro. La valeur de la plus haute ou de la dernière valeur du jeu est toujours une.

La fonction `CUME_DIST` calcule la position relative d'une valeur spécifiée dans un groupe de valeurs

en déterminant le pourcentage de valeurs inférieures ou égales à cette valeur. Cela s'appelle la distribution cumulative.

```
SELECT BusinessEntityID, JobTitle, SickLeaveHours,
PERCENT_RANK() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours DESC)
       AS "Percent Rank",
CUME_DIST() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours DESC)
       AS "Cumulative Distribution"
FROM Employee;
```

Dans cet exemple, vous utilisez une clause `ORDER` pour partitionner ou grouper les lignes extraites par l' `SELECT` en fonction des titres de travail des employés, les résultats de chaque groupe étant triés en fonction du nombre d'heures de congé de maladie utilisées par les employés.

BusinessEntityID	Profession	SickLeaveHours	Rang en pourcentage	Distribution cumulative
267	Spécialiste d'application	57	0	0,25
268	Spécialiste d'application	56	0.333333333333333	0,75
269	Spécialiste d'application	56	0.333333333333333	0,75
272	Spécialiste d'application	55	1	1
262	Assistante du responsable financier Cheif	48	0	1
239	Spécialiste des avantages	45	0	1
252	Acheteur	50	0	0.11111111111111111
251	Acheteur	49	0,125	0.333333333333333
256	Acheteur	49	0,125	0.333333333333333
253	Acheteur	48	0.375	0.555555555555555
254	Acheteur	48	0.375	0.555555555555555

La fonction `PERCENT_RANK` classe les entrées dans chaque groupe. Pour chaque entrée, il renvoie le pourcentage d'entrées du même groupe qui ont des valeurs inférieures.

La fonction `CUME_DIST` est similaire, sauf qu'elle renvoie le pourcentage de valeurs inférieures ou égales à la valeur actuelle.

PERCENTILE_DISC et PERCENTILE_CONT

La fonction `PERCENTILE_DISC` répertorie la valeur de la première entrée où la distribution cumulative est supérieure au centile que vous fournissez à l'aide du paramètre `numeric_literal` .

Les valeurs sont regroupées par jeu de lignes ou partition, comme spécifié par la clause `WITHIN GROUP` .

La fonction `PERCENTILE_CONT` est similaire à la fonction `PERCENTILE_DISC` , mais renvoie la moyenne de la somme de la première entrée correspondante et de l'entrée suivante.

```
SELECT BusinessEntityID, JobTitle, SickLeaveHours,
       CUME_DIST() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours ASC)
       AS "Cumulative Distribution",
       PERCENTILE_DISC(0.5) WITHIN GROUP(ORDER BY SickLeaveHours)
          OVER(PARTITION BY JobTitle) AS "Percentile Discreet"
FROM Employee;
```

Pour rechercher la valeur exacte de la ligne qui correspond ou dépasse le 0,5 centile, vous transmettez le percentile en tant que littéral numérique dans la fonction `PERCENTILE_DISC` . La colonne Percentile Discreet d'un jeu de résultats répertorie la valeur de la ligne à laquelle la distribution cumulative est supérieure au centile spécifié.

BusinessEntityID	Profession	SickLeaveHours	Distribution cumulative	Centile discret
272	Spécialiste d'application	55	0,25	**56**
268	Spécialiste d'application	56	0,75	**56**
269	Spécialiste d'application	56	0,75	**56**
267	Spécialiste d'application	57	1	**56**

Pour baser le calcul sur un ensemble de valeurs, vous utilisez la fonction `PERCENTILE_CONT` . La colonne "Percentile Continuous" dans les résultats indique la valeur moyenne de la somme de la valeur du résultat et de la valeur correspondante la plus élevée suivante.

```
SELECT BusinessEntityID, JobTitle, SickLeaveHours,
       CUME_DIST() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours ASC)
       AS "Cumulative Distribution",
       PERCENTILE_DISC(0.5) WITHIN GROUP(ORDER BY SickLeaveHours)
          OVER(PARTITION BY JobTitle) AS "Percentile Discreet",
       PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY SickLeaveHours)
          OVER(PARTITION BY JobTitle) AS "Percentile Continuous"
FROM Employee;
```

BusinessEntityID	Profession	SickLeaveHours	Distribution cumulative	Centile discret	Percentile continu
272	Spécialiste d'application	55	0,25	56	**56**
268	Spécialiste d'application	56	0,75	56	**56**
269	Spécialiste d'application	56	0,75	56	**56**
267	Spécialiste d'application	57	1	56	**56**

Lire Fonctions (analytique) en ligne: https://riptutorial.com/fr/sql/topic/8811/fonctions--analytique-

Chapitre 31: Fonctions (Scalar / Single Row)

Introduction

SQL fournit plusieurs fonctions scalaires intégrées. Chaque fonction scalaire prend une valeur en entrée et renvoie une valeur en sortie pour chaque ligne d'un jeu de résultats.

Vous utilisez des fonctions scalaires partout où une expression est autorisée dans une instruction T-SQL.

Syntaxe

- CAST (expression AS data_type [(length)])
- CONVERT (type_données [(longueur)], expression [, style])
- PARSE (string_value AS data_type [culture USING])
- DATENAME (datepart, date)
- AVOIR UN RENDEZ-VOUS ( )
- DATEDIFF (datepart, date de début, date de fin)
- DATEADD (datepart, numéro, date)
- CHOISIR (index, val_1, val_2 [, val_n])
- IIF (expression_booléenne, valeur_ true, valeur_file)
- SIGN (expression_numérique)
- POWER (expression_flottant, y)

Remarques

Les fonctions scalaires ou à une seule ligne permettent d’exploiter chaque ligne de données du jeu de résultats, par opposition aux fonctions d’agrégation qui opèrent sur l’ensemble de résultats complet.

Il existe dix types de fonctions scalaires.

1. Les fonctions de configuration fournissent des informations sur la configuration de l'instance SQL en cours.
2. Les fonctions de conversion convertissent les données dans le type de données approprié pour une opération donnée. Par exemple, ces types de fonctions peuvent reformater les informations en convertissant une chaîne en date ou en nombre pour permettre la comparaison de deux types différents.
3. Les fonctions de date et d'heure manipulent des champs contenant des valeurs de date et d'heure. Ils peuvent renvoyer des valeurs numériques, de date ou de chaîne. Par exemple, vous pouvez utiliser une fonction pour récupérer le jour actuel de la semaine ou de l'année ou pour ne récupérer que l'année de la date.

Les valeurs renvoyées par les fonctions de date et d'heure dépendent de la date et de l'heure définies pour le système d'exploitation de l'ordinateur exécutant l'instance SQL.

4. Fonction logique qui effectue des opérations à l'aide d'opérateurs logiques. Il évalue un ensemble de conditions et renvoie un seul résultat.
5. Les fonctions mathématiques effectuent des opérations mathématiques ou des calculs sur des expressions numériques. Ce type de fonction renvoie une valeur numérique unique.
6. Les fonctions de métadonnées récupèrent des informations sur une base de données spécifiée, telles que son nom et ses objets de base de données.
7. Les fonctions de sécurité fournissent des informations que vous pouvez utiliser pour gérer la sécurité d'une base de données, telles que des informations sur les utilisateurs et les rôles de la base de données.
8. Les fonctions de chaîne effectuent des opérations sur les valeurs de chaîne et renvoient des valeurs numériques ou de chaîne.

À l'aide des fonctions de chaîne, vous pouvez, par exemple, combiner des données, extraire une sous-chaîne, comparer des chaînes ou convertir une chaîne en caractères majuscules ou minuscules.

9. Les fonctions système effectuent des opérations et renvoient des informations sur les valeurs, les objets et les paramètres de l'instance SQL en cours.
10. Les fonctions statistiques du système fournissent diverses statistiques sur l'instance SQL en cours, par exemple pour que vous puissiez surveiller les niveaux de performances actuels du système.

Examples

Modifications de personnage

Les fonctions de modification des caractères incluent la conversion de caractères en majuscules ou en minuscules, la conversion de nombres en nombres formatés, la manipulation de caractères, etc.

La fonction `lower(char)` convertit le paramètre de caractère donné en caractères minuscules.

```
SELECT customer_id, lower(customer_last_name) FROM customer;
```

retournerait le nom de famille du client changé de "SMITH" à "smith".

Date et l'heure

Dans SQL, vous utilisez des types de données de date et heure pour stocker les informations de calendrier. Ces types de données incluent l'heure, la date, smalldatetime, datetime, datetime2 et datetimeoffset. Chaque type de données a un format spécifique.

Type de données	Format
temps	hh: mm: ss [.nnnnnnn]
rendez-vous amoureux	AAAA-MM-JJ

Type de données	Format
petit temps	AAAA-MM-JJ hh: mm: ss
datetime	AAAA-MM-JJ hh: mm: ss [.nnn]
datetime2	AAAA-MM-JJ hh: mm: ss [.nnnnnnn]
datetimeoffset	AAAA-MM-JJ hh: mm: ss [.nnnnnnn] [+/-] hh: mm

La fonction `DATENAME` renvoie le nom ou la valeur d'une partie spécifique de la date.

```
SELECT DATENAME (weekday,'2017-01-14') as Datename
```

Nom de données
samedi

Vous utilisez la fonction `GETDATE` pour déterminer la date et l'heure actuelles de l'ordinateur exécutant l'instance SQL en cours. Cette fonction n'inclut pas la différence de fuseau horaire.

```
SELECT GETDATE() as Systemdate
```

Date du système
2017-01-14 11: 11: 47.7230728

La fonction `DATEDIFF` renvoie la différence entre deux dates.

Dans la syntaxe, datepart est le paramètre qui spécifie la partie de la date que vous souhaitez utiliser pour calculer la différence. La date peut être l'année, le mois, la semaine, le jour, l'heure, la minute, la seconde ou la milliseconde. Vous spécifiez ensuite la date de début dans le paramètre startdate et la date de fin dans le paramètre enddate pour lequel vous souhaitez trouver la différence.

```
SELECT SalesOrderID, DATEDIFF(day, OrderDate, ShipDate)
AS 'Processing time'
FROM Sales.SalesOrderHeader
```

SalesOrderID	Temps de traitement
43659	7
43660	7
43661	7

SalesOrderID	Temps de traitement
43662	7

La fonction `DATEADD` vous permet d'ajouter un intervalle à une partie d'une date spécifique.

```
SELECT DATEADD (day, 20, '2017-01-14') AS Added20MoreDays
```

Ajouté20MoisDays
2017-02-03 00: 00: 00.000

Fonction de configuration et de conversion

Un exemple de fonction de configuration dans SQL est la fonction `@@SERVERNAME` . Cette fonction fournit le nom du serveur local exécutant SQL.

```
SELECT @@SERVERNAME AS 'Server'
```

Serveur
SQL064

En SQL, la plupart des conversions de données ont lieu implicitement, sans intervention de l'utilisateur.

Pour effectuer des conversions qui ne peuvent être exécutées implicitement, vous pouvez utiliser les fonctions `CAST` ou `CONVERT` .

La syntaxe de la fonction `CAST` est plus simple que la syntaxe de la fonction `CONVERT` , mais elle est limitée dans ce qu'elle peut faire.

Ici, nous utilisons les fonctions `CAST` et `CONVERT` pour convertir le type de données datetime en type de données `varchar` .

La fonction `CAST` utilise toujours le paramètre de style par défaut. Par exemple, il représentera les dates et les heures en utilisant le format AAAA-MM-JJ.

La fonction `CONVERT` utilise le style de date et d'heure que vous spécifiez. Dans ce cas, 3 spécifie le format de date jj / mm / aa.

```
USE AdventureWorks2012
GO
SELECT FirstName + ' ' + LastName + ' was hired on ' +
       CAST(HireDate AS varchar(20)) AS 'Cast',
       FirstName + ' ' + LastName + ' was hired on ' +
       CONVERT(varchar, HireDate, 3) AS 'Convert'
FROM Person.Person AS p
```

```
JOIN HumanResources.Employee AS e
ON p.BusinessEntityID = e.BusinessEntityID
GO
```

Jeter	Convertir
David Hamiltion a été embauché le 2003-02-04	David Hamiltion a été embauché le 04/02/03

Un autre exemple de fonction de conversion est la fonction `PARSE` . Cette fonction convertit une chaîne en un type de données spécifié.

Dans la syntaxe de la fonction, vous spécifiez la chaîne à convertir, le mot-clé `AS` , puis le type de données requis. Vous pouvez également spécifier la culture dans laquelle la valeur de chaîne doit être formatée. Si vous ne le spécifiez pas, la langue de la session est utilisée.

Si la valeur de la chaîne ne peut pas être convertie en format numérique, date ou heure, cela entraînera une erreur. Vous devrez ensuite utiliser `CAST` ou `CONVERT` pour la conversion.

```
SELECT PARSE('Monday, 13 August 2012' AS datetime2 USING 'en-US') AS 'Date in English'
```

Date en anglais
2012-08-13 00: 00: 00.0000000

Fonction logique et mathématique

SQL a deux fonctions logiques: `CHOOSE` et `IIF` .

La fonction `CHOOSE` renvoie un élément d'une liste de valeurs, en fonction de sa position dans la liste. Cette position est spécifiée par l'index.

Dans la syntaxe, le paramètre index spécifie l'élément et est un nombre entier ou un entier. Le paramètre val_1… val_n identifie la liste de valeurs.

```
SELECT CHOOSE(2, 'Human Resources', 'Sales', 'Admin', 'Marketing' ) AS Result;
```

Résultat
Ventes

Dans cet exemple, vous utilisez la fonction `CHOOSE` pour renvoyer la deuxième entrée dans une liste de départements.

La fonction `IIF` renvoie l'une des deux valeurs, en fonction d'une condition particulière. Si la

condition est vraie, elle retournera la valeur vraie. Sinon, il retournera une valeur fausse.

Dans la syntaxe, le paramètre boolean_expression spécifie l'expression booléenne. Le paramètre true_value spécifie la valeur à renvoyer si la valeur boolean_expression est définie sur true et que le paramètre false_value spécifie la valeur à renvoyer si la valeur boolean_expression est définie sur false.

```
SELECT BusinessEntityID, SalesYTD,
       IIF(SalesYTD > 200000, 'Bonus', 'No Bonus') AS 'Bonus?'
FROM Sales.SalesPerson
GO
```

BusinessEntityID	SalesYTD	Prime?
274	559697.5639	Prime
275	3763178.1787	Prime
285	172524.4512	Pas de bonus

Dans cet exemple, vous utilisez la fonction IIF pour renvoyer l'une des deux valeurs. Si le chiffre d'affaires annuel d'un commercial est supérieur à 200 000, cette personne sera éligible à un bonus. Des valeurs inférieures à 200 000 signifient que les employés ne sont pas admissibles aux primes.

SQL inclut plusieurs fonctions mathématiques que vous pouvez utiliser pour effectuer des calculs sur des valeurs d'entrée et renvoyer des résultats numériques.

Un exemple est la fonction `SIGN` , qui renvoie une valeur indiquant le signe d'une expression. La valeur -1 indique une expression négative, la valeur +1 indique une expression positive et 0 indique zéro.

```
SELECT SIGN(-20) AS 'Sign'
```

Signe
-1

Dans l'exemple, l'entrée est un nombre négatif, donc le volet Résultats répertorie le résultat -1.

Une autre fonction mathématique est la fonction `POWER` . Cette fonction fournit la valeur d'une expression portée à une puissance spécifiée.

Dans la syntaxe, le paramètre float_expression spécifie l'expression et le paramètre y spécifie la puissance à laquelle vous souhaitez augmenter l'expression.

```
SELECT POWER(50, 3) AS Result
```

Résultat
125000

Lire Fonctions (Scalar / Single Row) en ligne: https://riptutorial.com/fr/sql/topic/6898/fonctions--scalar---single-row-

Chapitre 32: Fonctions de chaîne

Introduction

Les fonctions de chaîne effectuent des opérations sur les valeurs de chaîne et renvoient des valeurs numériques ou de chaîne.

À l'aide des fonctions de chaîne, vous pouvez, par exemple, combiner des données, extraire une sous-chaîne, comparer des chaînes ou convertir une chaîne en caractères majuscules ou minuscules.

Syntaxe

- CONCAT (string_value1, string_value2 [, string_valueN])
- LTRIM (expression_caractère)
- RTRIM (expression_caractère)
- SUBSTRING (expression, début, longueur)
- ASCII (expression_caractère)
- REPLICATE (expression_chaîne, expression_entier)
- REVERSE (expression_chaîne)
- UPPER (expression_caractère)
- TRIM ([caractères FROM] chaîne)
- STRING_SPLIT (chaîne, séparateur)
- STUFF (expression_caractère, début, longueur, remplacement_expression)
- REPLACE (expression_chaîne, string_pattern, string_replacement)

Remarques

Référence des fonctions de chaîne pour Transact-SQL / Microsoft

Référence des fonctions de chaîne pour MySQL

Référence des fonctions de chaîne pour PostgreSQL

Examples

Couper les espaces vides

Trim est utilisé pour supprimer l'espace d'écriture au début ou à la fin de la sélection

En MSSQL, il n'y a pas un seul `TRIM()`

```
SELECT LTRIM('  Hello  ') --returns 'Hello  '
SELECT RTRIM('  Hello  ') --returns '  Hello'
SELECT LTRIM(RTRIM('  Hello  ')) --returns 'Hello'
```

MySql et Oracle

```
SELECT TRIM('  Hello  ') --returns 'Hello'
```

Enchaîner

En (standard ANSI / ISO) SQL, l'opérateur pour la concaténation de chaînes est `||` . Cette syntaxe est prise en charge par toutes les bases de données principales, à l'exception de SQL Server:

```
SELECT 'Hello' || 'World' || '!'; --returns HelloWorld!
```

De nombreuses bases de données prennent en charge une fonction `CONCAT` pour joindre des chaînes:

```
SELECT CONCAT('Hello', 'World'); --returns 'HelloWorld'
```

Certaines bases de données prennent en charge `CONCAT` pour joindre plus de deux chaînes (Oracle non):

```
SELECT CONCAT('Hello', 'World', '!'); --returns 'HelloWorld!'
```

Dans certaines bases de données, les types non-chaîne doivent être convertis ou convertis:

```
SELECT CONCAT('Foo', CAST(42 AS VARCHAR(5)), 'Bar'); --returns 'Foo42Bar'
```

Certaines bases de données (par exemple, Oracle) effectuent des conversions implicites sans perte. Par exemple, un `CONCAT` sur un `CLOB` et `NCLOB` produit un `NCLOB` . Un `CONCAT` sur un nombre et un `varchar2` donne un `varchar2` , etc.:

```
SELECT CONCAT(CONCAT('Foo', 42), 'Bar') FROM dual; --returns Foo42Bar
```

Certaines bases de données peuvent utiliser l'opérateur non standard `+` (mais dans la plupart des cas, `+` ne fonctionne que pour les nombres):

```
SELECT 'Foo' + CAST(42 AS VARCHAR(5)) + 'Bar';
```

Sur SQL Server <2012, où `CONCAT` n'est pas pris en charge, `+` est le seul moyen de joindre des chaînes.

Minuscules supérieure

```
SELECT UPPER('HelloWorld') --returns 'HELLOWORLD'
SELECT LOWER('HelloWorld') --returns 'helloworld'
```

Substring

La syntaxe est la suivante: `SUBSTRING ( string_expression, start, length )` . Notez que les chaînes SQL sont indexées en 1.

```
SELECT SUBSTRING('Hello', 1, 2) --returns 'He'
SELECT SUBSTRING('Hello', 3, 3) --returns 'llo'
```

Ceci est souvent utilisé avec la fonction `LEN()` pour obtenir les `n` derniers caractères d'une chaîne de longueur inconnue.

```
DECLARE @str1 VARCHAR(10) = 'Hello', @str2 VARCHAR(10) = 'FooBarBaz';
SELECT SUBSTRING(@str1, LEN(@str1) - 2, 3) --returns 'llo'
SELECT SUBSTRING(@str2, LEN(@str2) - 2, 3) --returns 'Baz'
```

Divisé

Divise une expression de chaîne à l'aide d'un séparateur de caractères. Notez que `STRING_SPLIT()` est une fonction table.

```
SELECT value FROM STRING_SPLIT('Lorem ipsum dolor sit amet.', ' ');
```

Résultat:

```
value
-----
Lorem
ipsum
dolor
sit
amet.
```

Des trucs

Farcir une chaîne dans une autre, en remplaçant 0 ou plusieurs caractères à une certaine position.

Remarque: la position de `start` est indexée sur 1 (vous commencez à indexer à 1 et non à 0).

Syntaxe:

```
STUFF ( character_expression , start , length , replaceWith_expression )
```

Exemple:

```
SELECT STUFF('FooBarBaz', 4, 3, 'Hello') --returns 'FooHelloBaz'
```

Longueur

serveur SQL

Le LEN ne compte pas l'espace de fuite.

```
SELECT LEN('Hello') -- returns 5

SELECT LEN('Hello '); -- returns 5
```

La DATALENGTH compte l'espace de fuite.

```
SELECT DATALENGTH('Hello') -- returns 5

SELECT DATALENGTH('Hello '); -- returns 6
```

Il convient toutefois de noter que DATALENGTH renvoie la longueur de la représentation sous-jacente des octets de la chaîne, qui dépend, entre autres, du jeu de caractères utilisé pour stocker la chaîne.

```
DECLARE @str varchar(100) = 'Hello ' --varchar is usually an ASCII string, occupying 1 byte
per char
SELECT DATALENGTH(@str) -- returns 6

DECLARE @nstr nvarchar(100) = 'Hello ' --nvarchar is a unicode string, occupying 2 bytes per
char
SELECT DATALENGTH(@nstr) -- returns 12
```

Oracle

Syntaxe: Longueur (char)

Exemples:

```
SELECT Length('Bible') FROM dual; --Returns 5
SELECT Length('righteousness') FROM dual; --Returns 13
SELECT Length(NULL) FROM dual; --Returns NULL
```

Voir aussi: LongueurB, LongueurC, Longueur2, Longueur4

Remplacer

Syntaxe:

`REPLACE(` Chaîne à rechercher , Chaîne à rechercher et remplacer , Chaîne à placer dans la chaîne d'origine `)`

Exemple:

```
SELECT REPLACE( 'Peter Steve Tom', 'Steve', 'Billy' ) --Return Values: Peter Billy Tom
```

GAUCHE DROITE

La syntaxe est la suivante:
LEFT (expression de chaîne, entier)
DROITE (expression de chaîne, entier)

```
SELECT LEFT('Hello',2)  --return He
SELECT RIGHT('Hello',2) --return lo
```

Oracle SQL n'a pas de fonctions GAUCHE et DROITE. Ils peuvent être émulés avec SUBSTR et LONGUEUR.
SUBSTR (expression de chaîne, 1, entier)
SUBSTR (expression_chaîne, longueur (expression_chaîne) -intérieur + 1, entier)

```
SELECT SUBSTR('Hello',1,2)  --return He
SELECT SUBSTR('Hello',LENGTH('Hello')-2+1,2) --return lo
```

SENS INVERSE

La syntaxe est la suivante: REVERSE (expression de chaîne)

```
SELECT REVERSE('Hello') --returns olleH
```

REPRODUIRE

La fonction `REPLICATE` concatène une chaîne avec elle-même un nombre de fois spécifié.

La syntaxe est la suivante: REPLICATE (expression de chaîne, entier)

```
SELECT REPLICATE ('Hello',4) --returns 'HelloHelloHelloHello'
```

REGEXP

MySQL 3.19

Vérifie si une chaîne correspond à une expression régulière (définie par une autre chaîne).

```
SELECT 'bedded' REGEXP '[a-f]' -- returns True

SELECT 'beam' REGEXP '[a-f]' -- returns False
```

Remplacer la fonction dans sql Sélectionner et mettre à jour la requête

La fonction Remplacer dans SQL est utilisée pour mettre à jour le contenu d'une chaîne. L'appel de fonction est REPLACE () pour MySQL, Oracle et SQL Server.

La syntaxe de la fonction Remplacer est la suivante:

```
REPLACE (str, find, repl)
```

L'exemple suivant remplace les occurrences de `South` par `Southern` dans la table Employees:

Prénom	Adresse
James	New York du Sud
John	Boston Sud
Michael	Sud de san diego

Select Statement:

Si nous appliquons la fonction Remplacer suivante:

```
SELECT
    FirstName,
    REPLACE (Address, 'South', 'Southern') Address
FROM Employees
ORDER BY FirstName
```

Résultat:

Prénom	Adresse
James	Sud de New York
John	Sud de Boston
Michael	Sud de san diego

Déclaration de mise à jour:

Nous pouvons utiliser une fonction de remplacement pour apporter des modifications permanentes à notre tableau en suivant l'approche suivante.

```
Update Employees
Set city = (Address, 'South', 'Southern');
```

Une approche plus commune consiste à utiliser ceci en conjonction avec une clause WHERE comme ceci:

```
Update Employees
Set Address = (Address, 'South', 'Southern')
Where Address LIKE 'South%';
```

PARSENAME

BASE DE DONNEES : SQL Server

La fonction **PARSENAME** renvoie la partie spécifique de la chaîne donnée (nom d'objet). Le nom de l'objet peut contenir une chaîne comme le nom de l'objet, le nom du propriétaire, le nom de la base de données et le nom du serveur.

Plus de détails `MSDN: PARSENAME`

Syntaxe

```
PARSENAME('NameOfStringToParse',PartIndex)
```

Exemple

Pour obtenir le nom de l'objet, utilisez l'index de la partie `1`

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',1)  // returns `ObjectName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',1)     // returns `Student`
```

Pour obtenir le nom du schéma, utilisez l'index du composant `2`

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',2)  // returns `SchemaName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',2)     // returns `school`
```

Pour obtenir le nom de la base de données, utilisez l'index de la pièce `3`

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',3) // returns `DatabaseName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',3)    // returns `SchoolDatabase`
```

Pour obtenir le nom du serveur, utilisez l'index de la pièce `4`

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',4)  // returns `ServerName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',4)     // returns `[1012-1111]`
```

PARSENAME retournera null si la partie spécifiée n'est pas présente dans la chaîne de nom d'objet donnée

INSTR

Retourne l'index de la première occurrence d'une sous-chaîne (zéro si non trouvé)

Syntaxe: INSTR (chaîne, sous-chaîne)

```
SELECT INSTR('FooBarBar', 'Bar') -- return 4
SELECT INSTR('FooBarBar', 'Xar') -- return 0
```

Lire Fonctions de chaîne en ligne: https://riptutorial.com/fr/sql/topic/1120/fonctions-de-chaine

Chapitre 33: Fonctions de fenêtre

Examples

Ajout du nombre total de lignes sélectionnées à chaque ligne

```
SELECT your_columns, COUNT(*) OVER() as Ttl_Rows FROM your_data_set
```

id	prénom	Ttl_Rows
1	Exemple	5
2	foo	5
3	bar	5
4	baz	5
5	quux	5

Au lieu d'utiliser deux requêtes pour obtenir un compte, vous pouvez utiliser un agrégat comme fonction de fenêtre et utiliser l'ensemble de résultats complet comme fenêtre.
Cela peut être utilisé comme base pour un calcul ultérieur sans la complexité des auto-jointures supplémentaires.

Configuration d'un indicateur si d'autres lignes ont une propriété commune

Disons que j'ai ces données:

Articles de table

id	prénom	marque
1	Exemple	unique_tag
2	foo	simple
42	bar	simple
3	baz	Bonjour
51	quux	monde

Je voudrais obtenir toutes ces lignes et savoir si un tag est utilisé par d'autres lignes

```
SELECT id, name, tag, COUNT(*) OVER (PARTITION BY tag) > 1 AS flag FROM items
```

Le résultat sera:

id	prénom	marque	drapeau
1	Exemple	unique_tag	faux
2	foo	simple	vrai
42	bar	simple	vrai
3	baz	Bonjour	faux
51	quux	monde	faux

Si votre base de données n'a pas OVER et PARTITION, vous pouvez l'utiliser pour produire le même résultat:

```
SELECT id, name, tag, (SELECT COUNT(tag) FROM items B WHERE tag = A.tag) > 1 AS flag FROM
items A
```

Obtenir un total cumulé

Compte tenu de ces données:

rendez-vous amoureux	montant
2016-03-12	200
2016-03-11	-50
2016-03-14	100
2016-03-15	100
2016-03-10	-250

```
SELECT date, amount, SUM(amount) OVER (ORDER BY date ASC) AS running
FROM operations
ORDER BY date ASC
```

te donnera

rendez-vous amoureux	montant	fonctionnement
2016-03-10	-250	-250
2016-03-11	-50	-300
2016-03-12	200	-100

rendez-vous amoureux	montant	fonctionnement
2016-03-14	100	0
2016-03-15	100	-100

Obtenir les N lignes les plus récentes sur plusieurs regroupements

Compte tenu de ces données

Identifiant d'utilisateur	Date d'achèvement
1	2016-07-20
1	2016-07-21
2	2016-07-20
2	2016-07-21
2	2016-07-22

```
;with CTE as
(SELECT *,
        ROW_NUMBER() OVER (PARTITION BY User_ID
                           ORDER BY Completion_Date DESC) Row_Num
FROM    Data)
SELECT * FORM CTE WHERE Row_Num <= n
```

En utilisant n = 1, vous obtenez la dernière ligne par `user_id` :

Identifiant d'utilisateur	Date d'achèvement	Row_Num
1	2016-07-21	1
2	2016-07-22	1

Recherche d'enregistrements "hors séquence" à l'aide de la fonction LAG ()

Compte tenu de ces exemples de données:

ID	STATUT	STATUS_TIME	STATUS_BY
1	UN	2016-09-28-19.47.52.501398	USER_1
3	UN	2016-09-28-19.47.52.501511	USER_2
1	TROIS	2016-09-28-19.47.52.501517	USER_3

ID	STATUT	STATUS_TIME	STATUS_BY
3	DEUX	2016-09-28-19.47.52.501521	USER_2
3	TROIS	2016-09-28-19.47.52.501524	USER_4

Les éléments identifiés par `ID` valeurs `ID` doivent passer de `STATUS` 'ONE' à 'TWO' à 'THREE' dans l'ordre, sans ignorer les statuts. Le problème est de trouver des utilisateurs (`STATUS_BY`) qui violent la règle et passent de «UN» immédiatement à «TROIS».

La fonction analytique `LAG()` aide à résoudre le problème en renvoyant pour chaque ligne la valeur de la ligne précédente:

```
SELECT * FROM (
 SELECT
  t.*,
  LAG(status) OVER (PARTITION BY id ORDER BY status_time) AS prev_status
  FROM test t
) t1 WHERE status = 'THREE' AND prev_status != 'TWO'
```

Si votre base de données n'a pas de LAG (), vous pouvez l'utiliser pour produire le même résultat:

```
SELECT A.id, A.status, B.status as prev_status, A.status_time, B.status_time as
prev_status_time
FROM Data A, Data B
WHERE A.id = B.id
AND   B.status_time = (SELECT MAX(status_time) FROM Data where status_time < A.status_time and
id = A.id)
AND   A.status = 'THREE' AND NOT B.status = 'TWO'
```

Lire Fonctions de fenêtre en ligne: https://riptutorial.com/fr/sql/topic/647/fonctions-de-fenetre

Chapitre 34: FUSIONNER

Introduction

MERGE (souvent aussi appelé UPSERT pour "update or insert") permet d'insérer de nouvelles lignes ou, si une ligne existe déjà, de mettre à jour la ligne existante. Le but est d'exécuter l'ensemble des opérations de manière atomique (pour garantir la cohérence des données) et d'éviter toute surcharge de communication pour plusieurs instructions SQL dans un système client / serveur.

Examples

MERGE pour faire la cible Match Source

```
MERGE INTO targetTable t
    USING sourceTable s
        ON t.PKID = s.PKID
    WHEN MATCHED AND NOT EXISTS (
            SELECT s.ColumnA, s.ColumnB, s.ColumnC
            INTERSECT
            SELECT t.ColumnA, t.ColumnB, s.ColumnC
            )
        THEN UPDATE SET
            t.ColumnA = s.ColumnA
            ,t.ColumnB = s.ColumnB
            ,t.ColumnC = s.ColumnC
    WHEN NOT MATCHED BY TARGET
        THEN INSERT (PKID, ColumnA, ColumnB, ColumnC)
        VALUES (s.PKID, s.ColumnA, s.ColumnB, s.ColumnC)
    WHEN NOT MATCHED BY SOURCE
        THEN DELETE
    ;
```

Remarque: La partie `AND NOT EXISTS` empêche la mise à jour des enregistrements qui n'ont pas changé. L'utilisation de la construction `INTERSECT` permet de comparer les colonnes nullables sans traitement spécial.

MySQL: compter les utilisateurs par nom

Supposons que nous voulons savoir combien d'utilisateurs ont le même nom. Laissez-nous créer des `users` table comme suit:

```
create table users(
    id int primary key auto_increment,
    name varchar(8),
    count int,
    unique key name(name)
);
```

Maintenant, nous venons de découvrir un nouvel utilisateur nommé Joe et aimerions le prendre en compte. Pour ce faire, nous devons déterminer s'il existe une ligne existante avec son nom et, le cas échéant, la mettre à jour pour incrémenter le nombre; d'autre part, s'il n'y a pas de ligne existante, nous devrions la créer.

MySQL utilise la syntaxe suivante: insert… on duplicate key update… . Dans ce cas:

```
insert into users(name, count)
       values ('Joe', 1)
       on duplicate key update count=count+1;
```

PostgreSQL: compter les utilisateurs par nom

Supposons que nous voulons savoir combien d'utilisateurs ont le même nom. Laissez-nous créer des `users` table comme suit:

```
create table users(
    id serial,
    name varchar(8) unique,
    count int
);
```

Maintenant, nous venons de découvrir un nouvel utilisateur nommé Joe et aimerions le prendre en compte. Pour ce faire, nous devons déterminer s'il existe une ligne existante avec son nom et, le cas échéant, la mettre à jour pour incrémenter le nombre; d'autre part, s'il n'y a pas de ligne existante, nous devrions la créer.

PostgreSQL utilise la syntaxe suivante: insert… on conflict… met à jour… . Dans ce cas:

```
insert into users(name, count)
    values('Joe', 1)
    on conflict (name) do update set count = users.count + 1;
```

Lire FUSIONNER en ligne: https://riptutorial.com/fr/sql/topic/1470/fusionner

Chapitre 35: GRANT et REVOKE

Syntaxe

- GRANT [privilège1] [, [privilège2] ...] ON [table] TO [grantee1] [, [grantee2] ...] [AVEC GRANT OPTION]
- REVOKE [privilège1] [, [privilège2] ...] ON [table] FROM [grantee1] [, [grantee2] ...]

Remarques

Accorder des autorisations aux utilisateurs. Si l' `WITH GRANT OPTION` est spécifiée, le bénéficiaire bénéficie également du privilège d'accorder l'autorisation ou de révoquer les autorisations précédemment accordées.

Examples

Accorder / révoquer des privilèges

```
GRANT SELECT, UPDATE
ON Employees
TO User1, User2;
```

Accordez à l'utilisateur `User1` et `User2` autorisation d'effectuer les opérations `SELECT` et `UPDATE` sur les `Employees` table.

```
REVOKE SELECT, UPDATE
ON Employees
FROM User1, User2;
```

Révoquer de `User1` et `User2` l'autorisation d'effectuer des opérations `SELECT` et `UPDATE` sur les employés de la table.

Lire GRANT et REVOKE en ligne: https://riptutorial.com/fr/sql/topic/5574/grant-et-revoke

Chapitre 36: Identifiant

Introduction

Cette rubrique concerne les identificateurs, c'est-à-dire les règles de syntaxe pour les noms de tables, de colonnes et d'autres objets de base de données.

Le cas échéant, les exemples doivent couvrir les variations utilisées par différentes implémentations SQL ou identifier l'implémentation SQL de l'exemple.

Examples

Identifiants non cotés

Les identificateurs non cotés peuvent utiliser des lettres (`a` - `z`), des chiffres (`0` - `9`) et un trait de soulignement (`_`) et doivent commencer par une lettre.

Selon l'implémentation SQL et / ou les paramètres de la base de données, d'autres caractères peuvent être autorisés, certains même comme premier caractère, par exemple

- MS SQL: `@` , `$` , `#` et autres lettres Unicode *(source)*
- MySQL: `$` *(source)*
- Oracle: `$` , `#` et autres lettres du jeu de caractères de la base de données *(source)*
- PostgreSQL: `$` et autres lettres Unicode *(source)*

Les identificateurs non cotés sont insensibles à la casse. La manière dont cela est géré dépend beaucoup de l'implémentation SQL:

- MS SQL: la conservation de la casse, la sensibilité définie par le jeu de caractères de la base de données, peut donc être sensible à la casse.
- MySQL: Préserver la casse, la sensibilité dépend des paramètres de la base de données et du système de fichiers sous-jacent.
- Oracle: converti en majuscule, puis géré comme un identifiant cité.
- PostgreSQL: converti en minuscule, puis géré comme un identifiant cité.
- SQLite: préservation de la casse; insensibilité à la casse uniquement pour les caractères ASCII.

Lire Identifiant en ligne: https://riptutorial.com/fr/sql/topic/9677/identifiant

Chapitre 37: Index

Introduction

Les index sont une structure de données contenant des pointeurs vers le contenu d'une table organisée dans un ordre spécifique, pour aider la base de données à optimiser les requêtes. Ils sont similaires à l'index du livre, où les pages (lignes du tableau) sont indexées par leur numéro de page.

Plusieurs types d'index existent et peuvent être créés sur une table. Lorsqu'un index existe sur les colonnes utilisées dans la clause WHERE, la clause JOIN ou la clause ORDER BY d'une requête, il peut améliorer sensiblement les performances des requêtes.

Remarques

Les index permettent d'accélérer les requêtes de lecture en triant les lignes d'une table en fonction d'une colonne.

L'effet d'un index n'est pas perceptible pour les petites bases de données telles que l'exemple, mais s'il existe un grand nombre de lignes, cela peut grandement améliorer les performances. Au lieu de vérifier chaque ligne de la table, le serveur peut effectuer une recherche binaire sur l'index.

Le compromis pour la création d'un index est la vitesse d'écriture et la taille de la base de données. Stocker l'index prend de la place. De plus, chaque fois qu'une INSERT est effectuée ou que la colonne est mise à jour, l'index doit être mis à jour. Ce n'est pas une opération aussi onéreuse que l'analyse de la table entière sur une requête SELECT, mais il faut toujours garder cela à l'esprit.

Examples

Créer un index

```
CREATE INDEX ix_cars_employee_id ON Cars (EmployeeId);
```

Cela créera un index pour la colonne *EmployeeId* dans la table *Cars* . Cet index améliorera la vitesse des requêtes demandant au serveur de trier ou de sélectionner par valeurs dans *EmployeeId* , telles que les suivantes:

```
SELECT * FROM Cars WHERE EmployeeId = 1
```

L'index peut contenir plus d'une colonne, comme dans la suite;

```
CREATE INDEX ix_cars_e_c_o_ids ON Cars (EmployeeId, CarId, OwnerId);
```

Dans ce cas, l'index serait utile pour les requêtes demandant de trier ou de sélectionner toutes les colonnes incluses, si l'ensemble des conditions est ordonné de la même manière. Cela signifie que lors de la récupération des données, il peut trouver les lignes à récupérer à l'aide de l'index, au lieu de parcourir la table complète.

Par exemple, le cas suivant utiliserait le deuxième index;

```
SELECT * FROM Cars WHERE EmployeeId = 1 Order by CarId DESC
```

Si l'ordre diffère, l'index n'a pas les mêmes avantages que dans la suite;

```
SELECT * FROM Cars WHERE OwnerId = 17 Order by CarId DESC
```

L'index n'est pas aussi utile car la base de données doit récupérer l'index complet, sur toutes les valeurs de EmployeeId et de CarID, afin de déterminer quels éléments ont `OwnerId = 17` .

(L'index peut toujours être utilisé; il se peut que l'optimiseur de requêtes trouve que récupérer l'index et filtrer sur `OwnerId` , puis récupérer uniquement les lignes nécessaires est plus rapide que de récupérer la table complète, surtout si la table est grande.)

Index clusterisés, uniques et triés

Les index peuvent avoir plusieurs caractéristiques qui peuvent être définies lors de la création ou en modifiant les index existants.

```
CREATE CLUSTERED INDEX ix_clust_employee_id ON Employees(EmployeeId, Email);
```

L'instruction SQL ci-dessus crée un nouvel index clusterisé sur Employees. Les index clusterisés sont des index qui déterminent la structure réelle de la table. la table elle-même est triée pour correspondre à la structure de l'index. Cela signifie qu'il peut y avoir au plus un index cluster sur une table. Si un index clusterisé existe déjà sur la table, l'instruction ci-dessus échouera. (Les tables sans index clusterisés sont également appelées heaps.)

```
CREATE UNIQUE INDEX uq_customers_email ON Customers(Email);
```

Cela créera un index unique pour la colonne *Email* dans la table *Customers* . Cet index, associé à l'accélération des requêtes comme un index normal, forcera également chaque adresse e-mail de cette colonne à être unique. Si une ligne est insérée ou mise à jour avec une valeur de *messagerie* non unique, l'insertion ou la mise à jour échouera par défaut.

```
CREATE UNIQUE INDEX ix_eid_desc ON Customers(EmployeeID);
```

Cela crée un index sur les clients qui crée également une contrainte de table que l'ID employé doit être unique. (Cela échouera si la colonne n'est pas unique actuellement - dans ce cas, s'il y a des employés qui partagent un ID.)

```
CREATE INDEX ix_eid_desc ON Customers(EmployeeID Desc);
```

Cela crée un index trié par ordre décroissant. Par défaut, les index (dans MSSQL Server, au moins) sont ascendants, mais cela peut être modifié.

Insérer avec un index unique

```
UPDATE Customers SET Email = "richard0123@example.com" WHERE id = 1;
```

Cela échouera si un index unique est défini dans la colonne *Email* des *clients* . Cependant, un autre comportement peut être défini pour ce cas:

```
UPDATE Customers SET Email = "richard0123@example.com" WHERE id = 1 ON DUPLICATE KEY;
```

SAP ASE: index de chute

Cette commande supprime l'index dans la table. Cela fonctionne sur le serveur `SAP ASE` .

Syntaxe:

```
DROP INDEX [table name].[index name]
```

Exemple:

```
DROP INDEX Cars.index_1
```

Index trié

Si vous utilisez un index trié comme vous le feriez, l' `SELECT` ne ferait pas de tri supplémentaire lors de l'extraction.

```
CREATE INDEX ix_scoreboard_score ON scoreboard (score DESC);
```

Lorsque vous exécutez la requête

```
SELECT * FROM scoreboard ORDER BY score DESC;
```

Le système de base de données ne ferait pas de tri supplémentaire, car il peut effectuer une recherche d'index dans cet ordre.

Suppression d'un index ou désactivation et reconstruction

```
DROP INDEX ix_cars_employee_id ON Cars;
```

Nous pouvons utiliser la commande `DROP` pour supprimer notre index. Dans cet exemple , nous `DROP` l'index appelé *ix_cars_employee_id* sur les *voitures* de table.

Cela supprime entièrement l'index, et si l'index est en cluster, supprimera tout cluster. Il ne peut

pas être reconstruit sans recréer l'index, qui peut être lent et coûteux en calculs. En alternative, l'index peut être désactivé:

```
ALTER INDEX ix_cars_employee_id ON Cars DISABLE;
```

Cela permet à la table de conserver la structure, ainsi que les métadonnées relatives à l'index.

Critiquement, cela conserve les statistiques d'index, de sorte qu'il est possible d'évaluer facilement le changement. Si cela est justifié, l'index peut ensuite être reconstruit, au lieu d'être recréé complètement;

```
ALTER INDEX ix_cars_employee_id ON Cars REBUILD;
```

Index unique permettant NULLS

```
CREATE UNIQUE INDEX idx_license_id
   ON Person(DrivingLicenseID) WHERE DrivingLicenseID IS NOT NULL
GO
```

Ce schéma permet une relation 0..1 - les personnes peuvent avoir zéro ou un permis de conduire et chaque licence ne peut appartenir qu'à une seule personne

Reconstruire l'index

Au fil du temps, les index B-Tree peuvent être fragmentés en raison de la mise à jour / suppression / insertion de données. Dans la terminologie SQLServer, nous pouvons avoir interne (page d'index à moitié vide) et externe (l'ordre des pages logiques ne correspond pas à l'ordre physique). La reconstruction de l'index est très similaire à la suppression et à la recréation.

Nous pouvons reconstruire un index avec

```
ALTER INDEX index_name REBUILD;
```

Par défaut, l'index de reconstruction est une opération hors ligne qui verrouille la table et empêche la création de DML, mais de nombreux SGBDR permettent la reconstruction en ligne. En outre, certains fournisseurs de `COALESCE` données proposent des alternatives à la reconstruction d'index, telles que `REORGANIZE` (SQLServer) ou `COALESCE` / `SHRINK SPACE` (Oracle).

Index clusterisé

Lorsque vous utilisez un index clusterisé, les lignes de la table sont triées en fonction de la colonne à laquelle l'index cluster est appliqué. Par conséquent, il ne peut y avoir qu'un seul index cluster sur la table car vous ne pouvez pas commander la table par deux colonnes différentes.

En règle générale, il est préférable d'utiliser l'index clusterisé lors des lectures sur les tables de données volumineuses. Le principal avantage de l'index clusterisé réside dans l'écriture sur la table et la réorganisation des données (recours).

Exemple de création d'un index cluster sur une table Employees sur la colonne Employee_Surname:

```
CREATE CLUSTERED INDEX ix_employees_name ON Employees(Employee_Surname);
```

Index non clusterisé

Les index non clusterisés sont stockés séparément de la table. Chaque index de cette structure contient un pointeur vers la ligne de la table qu'il représente.

Ces pointeurs sont appelés localisateurs de lignes. La structure du localisateur de lignes dépend de si les pages de données sont stockées dans un segment de mémoire ou une table en cluster. Pour un tas, un localisateur de lignes est un pointeur sur la ligne. Pour une table en cluster, le localisateur de lignes est la clé d'index en cluster.

Exemple de création d'un index non clusterisé sur la table Employees et la colonne Employee_Surname:

```
CREATE NONCLUSTERED INDEX ix_employees_name ON Employees(Employee_Surname);
```

Il peut y avoir plusieurs index non clusterisés sur la table. Les opérations de lecture sont généralement plus lentes avec les index non clusterisés qu'avec les index clusterisés, car vous devez d'abord indexer et indexer la table. Il n'y a pas de restrictions dans les opérations d'écriture cependant.

Index partiel ou filtré

SQL Server et SQLite permettent de créer des index contenant non seulement un sous-ensemble de colonnes, mais également un sous-ensemble de lignes.

Considérez un nombre croissant d'ordres avec `order_state_id` égal à terminé (2) et un nombre stable de commandes avec `order_state_id equal` à started (1).

Si votre entreprise utilise des requêtes comme celle-ci:

```
SELECT id, comment
  FROM orders
 WHERE order_state_id =  1
   AND product_id = @some_value;
```

L’indexation partielle vous permet de limiter l’index, en incluant uniquement les ordres inachevés:

```
CREATE INDEX Started_Orders
          ON orders(product_id)
       WHERE order_state_id = 1;
```

Cet index sera plus petit qu'un index non filtré, ce qui économise de l'espace et réduit les coûts de mise à jour de l'index.

Lire Index en ligne: https://riptutorial.com/fr/sql/topic/344/index

Chapitre 38: Injection SQL

Introduction

L'injection SQL est une tentative d'accéder aux tables de base de données d'un site Web en injectant du code SQL dans un champ de formulaire. Si un serveur Web ne protège pas contre les attaques par injection SQL, un pirate peut amener la base de données à exécuter le code SQL supplémentaire. En exécutant leur propre code SQL, les pirates peuvent mettre à niveau l'accès à leur compte, consulter les informations privées de quelqu'un d'autre ou apporter d'autres modifications à la base de données.

Examples

Échantillon d'injection SQL

En supposant que l'appel du gestionnaire de connexion de votre application Web ressemble à ceci:

```
https://somepage.com/ajax/login.ashx?username=admin&password=123
```

Maintenant, dans login.ashx, vous lisez ces valeurs:

```
strUserName = getHttpsRequestParameterString("username");
strPassword = getHttpsRequestParameterString("password");
```

et interrogez votre base de données pour déterminer si un utilisateur avec ce mot de passe existe.

Donc, vous construisez une chaîne de requête SQL:

```
txtSQL = "SELECT * FROM Users WHERE username = '" + strUserName + "' AND password = '"+
strPassword +"'";
```

Cela fonctionnera si le nom d'utilisateur et le mot de passe ne contiennent pas de devis.

Cependant, si l'un des paramètres contient un devis, le SQL qui est envoyé à la base de données ressemblera à ceci:

```
-- strUserName = "d'Alambert";
txtSQL = "SELECT * FROM Users WHERE username = 'd'Alambert' AND password = '123'";
```

Cela entraînera une erreur de syntaxe, car la citation après le `d` dans `d'Alambert` termine la chaîne SQL.

Vous pouvez corriger cela en échappant aux guillemets dans le nom d'utilisateur et le mot de passe, par exemple:

```
strUserName = strUserName.Replace("'", "''");
strPassword = strPassword.Replace("'", "''");
```

Cependant, il est plus approprié d'utiliser les paramètres:

```
cmd.CommandText = "SELECT * FROM Users WHERE username = @username AND password = @password";

cmd.Parameters.Add("@username", strUserName);
cmd.Parameters.Add("@password", strPassword);
```

Si vous n'utilisez pas de paramètres et que vous oubliez de remplacer quote par une seule des valeurs, un utilisateur malveillant (aka hacker) peut l'utiliser pour exécuter des commandes SQL sur votre base de données.

Par exemple, si un attaquant est mauvais, il / elle définira le mot de passe pour

```
lol'; DROP DATABASE master; --
```

et alors le SQL ressemblera à ceci:

```
"SELECT * FROM Users WHERE username = 'somebody' AND password = 'lol'; DROP DATABASE master; --'";
```

Malheureusement pour vous, il s'agit d'un code SQL valide, et la base de données l'exécutera!

Ce type d'exploitation s'appelle une injection SQL.

Un utilisateur malveillant peut faire bien d'autres choses, comme voler l'adresse électronique de chaque utilisateur, voler le mot de passe de chacun, voler des numéros de carte de crédit, voler une quantité de données dans votre base de données, etc.

C'est pourquoi vous devez toujours échapper à vos ficelles.
Et le fait que vous oubliez invariablement de le faire tôt ou tard est exactement la raison pour laquelle vous devez utiliser des paramètres. Parce que si vous utilisez des paramètres, votre framework de langage de programmation fera tout ce qui est nécessaire pour vous échapper.

échantillon d'injection simple

Si l'instruction SQL est construite comme ceci:

```
SQL = "SELECT * FROM Users WHERE username = '" + user + "' AND password ='" + pw + "'";
db.execute(SQL);
```

Un pirate pourrait alors récupérer vos données en donnant un mot de passe comme `pw' or '1'='1` ; l'instruction SQL qui en résultera sera:

```
SELECT * FROM Users WHERE username = 'somebody' AND password ='pw' or '1'='1'
```

Celui-ci passera la vérification du mot de passe pour toutes les lignes de la table `Users` car `'1'='1'`

est toujours vrai.

Pour éviter cela, utilisez les paramètres SQL:

```
SQL = "SELECT * FROM Users WHERE username = ? AND password = ?";
db.execute(SQL, [user, pw]);
```

Lire Injection SQL en ligne: https://riptutorial.com/fr/sql/topic/3517/injection-sql

Chapitre 39: INSÉRER

Syntaxe

- INSERT INTO nom_table (column1, column2, column3, ...) VALUES (valeur1, valeur2, valeur3, ...);
- INSERT INTO nom_table (column1, column2 ...) SELECT value1, value2 ... from other_table

Examples

Insérer une nouvelle ligne

```
INSERT INTO Customers
VALUES ('Zack', 'Smith', 'zack@example.com', '7049989942', 'EMAIL');
```

Cette instruction va insérer une nouvelle ligne dans la table `Customers` . Notez qu'une valeur n'a pas été spécifiée pour la colonne `Id` , car elle sera ajoutée automatiquement. Cependant, toutes les autres valeurs de colonne doivent être spécifiées.

Insérer uniquement des colonnes spécifiées

```
INSERT INTO Customers (FName, LName, Email, PreferredContact)
VALUES ('Zack', 'Smith', 'zack@example.com', 'EMAIL');
```

Cette instruction va insérer une nouvelle ligne dans la table `Customers` . Les données seront uniquement insérées dans les colonnes spécifiées - notez qu'aucune valeur n'a été fournie pour la colonne `PhoneNumber` . Notez toutefois que toutes les colonnes marquées comme `not null` doivent être incluses.

INSERER des données d'une autre table en utilisant SELECT

```
INSERT INTO Customers (FName, LName, PhoneNumber)
SELECT FName, LName, PhoneNumber FROM Employees
```

Cet exemple insérera tous les employés dans la table Customers . Comme les deux tables ont des champs différents et que vous ne voulez pas déplacer tous les champs, vous devez définir les champs à insérer et les champs à sélectionner. Les noms de champs en corrélation n'ont pas besoin d'être appelés de la même manière, mais doivent être du même type. Cet exemple suppose que le champ ID a un ensemble de spécifications d'identité et s'incrémentera automatiquement.

Si vous avez deux tables qui ont exactement les mêmes noms de champs et que vous voulez simplement déplacer tous les enregistrements, vous pouvez utiliser:

```
INSERT INTO Table1
SELECT * FROM Table2
```

Insérer plusieurs lignes à la fois

Plusieurs lignes peuvent être insérées avec une seule commande d'insertion:

```
INSERT INTO tbl_name (field1, field2, field3)

VALUES (1,2,3), (4,5,6), (7,8,9);
```

Pour l'insertion simultanée de grandes quantités de données (insertion groupée), des fonctionnalités et des recommandations spécifiques au SGBD existent.

MySQL - LOAD DATA INFILE

MSSQL - INSERT EN VRAC

Lire INSÉRER en ligne: https://riptutorial.com/fr/sql/topic/465/inserer

Chapitre 40: JOINDRE

Introduction

JOIN est une méthode permettant de combiner des informations provenant de deux tables. Le résultat est un ensemble de colonnes assemblées provenant des deux tables, défini par le type de jointure (INNER / OUTER / CROSS et LEFT / RIGHT / FULL, expliqué ci-dessous) et les critères de jointure (la relation entre les lignes des deux tables).

Une table peut être jointe à elle-même ou à toute autre table. Si des informations provenant de plus de deux tables doivent être consultées, plusieurs jointures peuvent être spécifiées dans une clause FROM.

Syntaxe

- `[ { INNER | { { LEFT | RIGHT | FULL } [ OUTER ] } } ] JOIN`

Remarques

Comme leur nom l'indique, les jointures permettent d'interroger les données de plusieurs tables de manière conjointe, les lignes affichant des colonnes provenant de plusieurs tables.

Examples

Jointure interne explicite de base

Une jointure de base (également appelée "jointure interne") interroge les données de deux tables, leur relation étant définie dans une clause de `join` .

L'exemple suivant sélectionne les prénoms des employés (FName) dans la table Employees et le nom du département pour lequel ils travaillent (Nom) dans la table Departments:

```
SELECT Employees.FName, Departments.Name
FROM   Employees
JOIN   Departments
ON Employees.DepartmentId = Departments.Id
```

Cela renverrait les éléments suivants de la base de données exemple :

Employees.FName	Départements.Nom
James	HEURE
John	HEURE
Richard	Ventes

Jointure implicite

Les jointures peuvent également être effectuées en ayant plusieurs tables dans la clause `from` , séparées par des virgules `,` et en définissant la relation entre elles dans la clause `where` . Cette technique s'appelle une jointure implicite (car elle ne contient pas de clause de `join`).

Tous les SGBDR le supportent, mais la syntaxe est généralement déconseillée. Les raisons pour lesquelles il est déconseillé d'utiliser cette syntaxe sont les suivantes:

- Il est possible d'obtenir des jointures croisées accidentelles qui renvoient des résultats incorrects, surtout si vous avez beaucoup de jointures dans la requête.
- Si vous envisagez une jointure croisée, la syntaxe n'est pas claire (écrivez plutôt CROSS JOIN), et quelqu'un est susceptible de le modifier lors de la maintenance.

L'exemple suivant sélectionne les prénoms des employés et le nom des départements pour lesquels ils travaillent:

```
SELECT e.FName, d.Name
FROM   Employee e, Departments d
WHERE  e.DeptartmentId = d.Id
```

Cela renverrait les éléments suivants de la base de données exemple :

e.FName	d.Name
James	HEURE
John	HEURE
Richard	Ventes

Jointure externe gauche

Une jointure externe gauche (également appelée jointure gauche ou jointure externe) est une jointure qui garantit que toutes les lignes de la table de gauche sont représentées. Si aucune ligne correspondante de la table de droite n'existe, ses champs correspondants sont `NULL` .

L'exemple suivant sélectionne tous les départements et le prénom des employés qui travaillent dans ce service. Les départements sans employés sont toujours renvoyés dans les résultats, mais auront NULL pour le nom de l'employé:

```
SELECT          Departments.Name, Employees.FName
FROM            Departments
LEFT OUTER JOIN Employees
ON              Departments.Id = Employees.DepartmentId
```

Cela renverrait les éléments suivants de la base de données exemple :

Départements.Nom	Employees.FName
HEURE	James
HEURE	John
HEURE	Johnathon
Ventes	Michael
Technologie	NUL

Alors, comment ça marche?

Il y a deux tables dans la clause FROM:

Id	FName	LName	Numéro de téléphone	ManagerId	DépartementId	Un salaire	Date d'embauche
1	James	Forgeron	1234567890	NUL	1	1000	01-01-2002
2	John	Johnson	2468101214	1	1	400	23-03-2005
3	Michael	Williams	1357911131	1	2	600	12-05-2009
4	Johnathon	Forgeron	1212121212	2	1	500	24-07-2016

et

Id	prénom
1	HEURE
2	Ventes
3	Technologie

Tout d'abord, un produit *cartésien* est créé à partir des deux tables donnant une table intermédiaire.
Les enregistrements qui répondent aux critères de jointure (*Departments.Id* = *Employees.DepartmentId*) sont surlignés en gras. ceux-ci sont passés à l'étape suivante de la requête.

Comme il s'agit d'une jointure externe LEFT, tous les enregistrements sont renvoyés du côté gauche de la jointure (Departments), tandis que tous les enregistrements du côté DROIT reçoivent un marqueur NULL s'ils ne correspondent pas aux critères de jointure. Dans le tableau ci-dessous, vous verrez **Tech** avec `NULL`

Id	prénom	Id	FName	LName	Numéro de téléphone	ManagerId	DépartementId	Un salaire
1	**HEURE**	**1**	**James**	**Forgeron**	**1234567890**	**NUL**	**1**	**1000**
1	**HEURE**	**2**	**John**	**Johnson**	**2468101214**	**1**	**1**	**400**
1	HEURE	3	Michael	Williams	1357911131	1	2	600
1	**HEURE**	**4**	**Johnathon**	**Forgeron**	**1212121212**	**2**	**1**	**500**
2	Ventes	1	James	Forgeron	1234567890	NUL	1	1000
2	Ventes	2	John	Johnson	2468101214	1	1	400
2	**Ventes**	**3**	**Michael**	**Williams**	**1357911131**	**1**	**2**	**600**
2	Ventes	4	Johnathon	Forgeron	1212121212	2	1	500
3	Technologie	1	James	Forgeron	1234567890	NUL	1	1000
3	Technologie	2	John	Johnson	2468101214	1	1	400
3	Technologie	3	Michael	Williams	1357911131	1	2	600
3	Technologie	4	Johnathon	Forgeron	1212121212	2	1	500

Enfin, chaque expression utilisée dans la clause **SELECT** est évaluée pour renvoyer notre table finale:

Départements.Nom	Employees.FName
HEURE	James
HEURE	John
Ventes	Richard
Technologie	NUL

Self Join

Une table peut être jointe à elle-même, avec différentes lignes correspondant à une condition. Dans ce cas d'utilisation, des alias doivent être utilisés pour distinguer les deux occurrences de la table.

Dans l'exemple ci-dessous, pour chaque employé dans la table Employees de la base de données exemple , un enregistrement contenant le prénom de l'employé et le prénom correspondant du responsable de l'employé est renvoyé. Les managers étant également des

employés, la table est jointe à elle-même:

```
SELECT
    e.FName AS "Employee",
    m.FName AS "Manager"
FROM
    Employees e
JOIN
    Employees m
    ON e.ManagerId = m.Id
```

Cette requête renvoie les données suivantes:

Employé	Directeur
John	James
Michael	James
Johnathon	John

Alors, comment ça marche?

La table d'origine contient ces enregistrements:

Id	FName	LName	Numéro de téléphone	ManagerId	DépartementId	Un salaire	Date d'embauche
1	James	Forgeron	1234567890	NUL	1	1000	01-01-2002
2	John	Johnson	2468101214	1	1	400	23-03-2005
3	Michael	Williams	1357911131	1	2	600	12-05-2009
4	Johnathon	Forgeron	1212121212	2	1	500	24-07-2016

La première action consiste à créer un produit *cartésien* de tous les enregistrements des tables utilisées dans la clause **FROM** . Dans ce cas, c'est la table Employees à deux reprises, de sorte que la table intermédiaire ressemblera à ceci (j'ai supprimé tous les champs non utilisés dans cet exemple):

e.Id	e.FName	e.ManagerId	milieu	m.FName	mManagerId
1	James	NUL	1	James	NUL
1	James	NUL	2	John	1
1	James	NUL	3	Michael	1

e.Id	e.FName	e.ManagerId	milieu	m.FName	mManagerId
1	James	NUL	4	Johnathon	2
2	John	1	1	James	NUL
2	John	1	2	John	1
2	John	1	3	Michael	1
2	John	1	4	Johnathon	2
3	Michael	1	1	James	NUL
3	Michael	1	2	John	1
3	Michael	1	3	Michael	1
3	Michael	1	4	Johnathon	2
4	Johnathon	2	1	James	NUL
4	Johnathon	2	2	John	1
4	Johnathon	2	3	Michael	1
4	Johnathon	2	4	Johnathon	2

L'action suivante consiste à ne conserver que les enregistrements répondant aux critères **JOIN** , de sorte que tous les enregistrements pour lesquels la table `e ManagerId` est égale à l' `Id` table `m ManagerId` l'alias:

e.Id	e.FName	e.ManagerId	milieu	m.FName	mManagerId
2	John	1	1	James	NUL
3	Michael	1	1	James	NUL
4	Johnathon	2	2	John	1

Ensuite, chaque expression utilisée dans la clause **SELECT** est évaluée pour renvoyer cette table:

e.FName	m.FName
John	James
Michael	James
Johnathon	John

Enfin, les noms de colonne `e.FName` et `m.FName` sont remplacés par leurs noms de colonne d'alias, attribués à l'opérateur AS :

Employé	Directeur
John	James
Michael	James
Johnathon	John

CROSS JOIN

La jointure croisée fait un produit cartésien des deux membres. Un produit cartésien signifie que chaque ligne d'une table est combinée avec chaque ligne de la seconde table de la jointure. Par exemple, si `TABLEA` a 20 lignes et que `TABLEB` a 20 lignes, le résultat serait `20*20 = 400` lignes de sortie.

Utilisation de la base de données exemple

```
SELECT d.Name, e.FName
FROM   Departments d
CROSS JOIN Employees e;
```

Qui retourne:

d.Name	e.FName
HEURE	James
HEURE	John
HEURE	Michael
HEURE	Johnathon
Ventes	James
Ventes	John
Ventes	Michael
Ventes	Johnathon
Technologie	James
Technologie	John
Technologie	Michael

d.Name	e.FName
Technologie	Johnathon

Il est recommandé d'écrire une jointure CROSS JOIN explicite si vous voulez faire une jointure cartésienne, pour souligner que c'est ce que vous voulez.

Rejoindre une sous-requête

La jointure d'une sous-requête est souvent utilisée lorsque vous souhaitez obtenir des données agrégées à partir d'une table enfant / détails et les afficher avec les enregistrements de la table parent / header. Par exemple, vous pouvez souhaiter obtenir un nombre d'enregistrements enfants, une moyenne de certaines colonnes numériques dans les enregistrements enfants ou la ligne supérieure ou inférieure en fonction d'une date ou d'un champ numérique. Cet exemple utilise des alias, ce qui facilite la lecture des requêtes lorsque plusieurs tables sont impliquées. Voici à quoi ressemble une jointure de sous-requête assez typique. Dans ce cas, nous récupérons toutes les lignes des commandes d'achat de la table parente et ne récupérons que la première ligne pour chaque enregistrement parent de la table enfant PurchaseOrderLineItems.

```
SELECT po.Id, po.PODate, po.VendorName, po.Status, item.ItemNo,
  item.Description, item.Cost, item.Price
FROM PurchaseOrders po
LEFT JOIN
     (
       SELECT l.PurchaseOrderId, l.ItemNo, l.Description, l.Cost, l.Price, Min(l.id) as Id
       FROM PurchaseOrderLineItems l
       GROUP BY l.PurchaseOrderId, l.ItemNo, l.Description, l.Cost, l.Price
     ) AS item ON item.PurchaseOrderId = po.Id
```

CROSS APPLY & LATERAL JOIN

Un type très intéressant de JOIN est le LATERAL JOIN (nouveau dans PostgreSQL 9.3+), qui est également connu sous le nom de CROSS APPLY / OUTER APPLY dans SQL-Server & Oracle.

L'idée de base est qu'une fonction de table (ou sous-requête en ligne) soit appliquée à chaque ligne que vous joignez.

Cela permet, par exemple, de ne joindre que la première entrée correspondante dans une autre table.
La différence entre une jointure normale et une jointure latérale réside dans le fait que vous pouvez utiliser une colonne que vous avez précédemment jointe **à la sous** - **requête et** que vous avez "CROSS APPLY".

Syntaxe:

PostgreSQL 9.3+

> à gauche | droit | interne JOIN **LATERAL**

Serveur SQL:

CROSS | APPLICATION **EXTERNE**

`INNER JOIN LATERAL` est identique à `CROSS APPLY`
et `LEFT JOIN LATERAL` est identique à `OUTER APPLY`

Exemple d'utilisation (PostgreSQL 9.3+):

```
SELECT * FROM T_Contacts

--LEFT JOIN T_MAP_Contacts_Ref_OrganisationalUnit ON MAP_CTCOU_CT_UID = T_Contacts.CT_UID AND
MAP_CTCOU_SoftDeleteStatus = 1
--WHERE T_MAP_Contacts_Ref_OrganisationalUnit.MAP_CTCOU_UID IS NULL -- 989


LEFT JOIN LATERAL
(
    SELECT
         --MAP_CTCOU_UID
         MAP_CTCOU_CT_UID
        ,MAP_CTCOU_COU_UID
        ,MAP_CTCOU_DateFrom
        ,MAP_CTCOU_DateTo
   FROM T_MAP_Contacts_Ref_OrganisationalUnit
   WHERE MAP_CTCOU_SoftDeleteStatus = 1
   AND MAP_CTCOU_CT_UID = T_Contacts.CT_UID

    /*
    AND
    (
        (__in_DateFrom <= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateTo)
        AND
        (__in_DateTo >= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateFrom)
    )
    */
   ORDER BY MAP_CTCOU_DateFrom
   LIMIT 1
) AS FirstOE
```

Et pour SQL-Server

```
SELECT * FROM T_Contacts

--LEFT JOIN T_MAP_Contacts_Ref_OrganisationalUnit ON MAP_CTCOU_CT_UID = T_Contacts.CT_UID AND
MAP_CTCOU_SoftDeleteStatus = 1
--WHERE T_MAP_Contacts_Ref_OrganisationalUnit.MAP_CTCOU_UID IS NULL -- 989

-- CROSS APPLY -- = INNER JOIN
OUTER APPLY    -- = LEFT JOIN
(
    SELECT TOP 1
         --MAP_CTCOU_UID
         MAP_CTCOU_CT_UID
        ,MAP_CTCOU_COU_UID
        ,MAP_CTCOU_DateFrom
        ,MAP_CTCOU_DateTo
   FROM T_MAP_Contacts_Ref_OrganisationalUnit
```

```
   WHERE MAP_CTCOU_SoftDeleteStatus = 1
   AND MAP_CTCOU_CT_UID = T_Contacts.CT_UID

    /*
    AND
    (
        (@in_DateFrom <= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateTo)
        AND
        (@in_DateTo >= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateFrom)
    )
    */
   ORDER BY MAP_CTCOU_DateFrom
) AS FirstOE
```

FULL JOIN

Un type de JOIN moins connu est le FULL JOIN.
(Remarque: FULL JOIN n'est pas pris en charge par MySQL en 2016)

Un FULL OUTER JOIN renvoie toutes les lignes de la table de gauche et toutes les lignes de la table de droite.

S'il y a des lignes dans la table de gauche qui n'ont pas de correspondance dans la table de droite ou s'il existe des lignes dans la table de droite qui n'ont pas de correspondance dans la table de gauche, ces lignes seront également répertoriées.

Exemple 1 :

```
SELECT * FROM Table1

FULL JOIN Table2
     ON 1 = 2
```

Exemple 2:

```
SELECT
     COALESCE(T_Budget.Year, tYear.Year) AS RPT_BudgetInYear
    ,COALESCE(T_Budget.Value, 0.0) AS RPT_Value
FROM T_Budget

FULL JOIN tfu_RPT_All_CreateYearInterval(@budget_year_from, @budget_year_to) AS tYear
      ON tYear.Year = T_Budget.Year
```

Notez que si vous utilisez des suppressions logicielles, vous devrez vérifier à nouveau l'état de suppression logicielle dans la clause WHERE (car FULL JOIN se comporte comme un UNION);
Il est facile de négliger ce petit fait, puisque vous mettez AP_SoftDeleteStatus = 1 dans la clause de jointure.

De plus, si vous effectuez un FULL JOIN, vous devrez généralement autoriser NULL dans la clause WHERE; Si vous oubliez d'autoriser NULL sur une valeur, cela aura les mêmes effets qu'une jointure INNER, ce que vous ne voulez pas si vous faites un FULL JOIN.

Exemple:

```
SELECT
     T_AccountPlan.AP_UID
    ,T_AccountPlan.AP_Code
    ,T_AccountPlan.AP_Lang_EN
    ,T_BudgetPositions.BUP_Budget
    ,T_BudgetPositions.BUP_UID
    ,T_BudgetPositions.BUP_Jahr
FROM T_BudgetPositions

FULL JOIN T_AccountPlan
    ON T_AccountPlan.AP_UID = T_BudgetPositions.BUP_AP_UID
    AND T_AccountPlan.AP_SoftDeleteStatus = 1

WHERE (1=1)
AND (T_BudgetPositions.BUP_SoftDeleteStatus = 1 OR T_BudgetPositions.BUP_SoftDeleteStatus IS
NULL)
AND (T_AccountPlan.AP_SoftDeleteStatus = 1 OR T_AccountPlan.AP_SoftDeleteStatus IS NULL)
```

JOIN Récursives

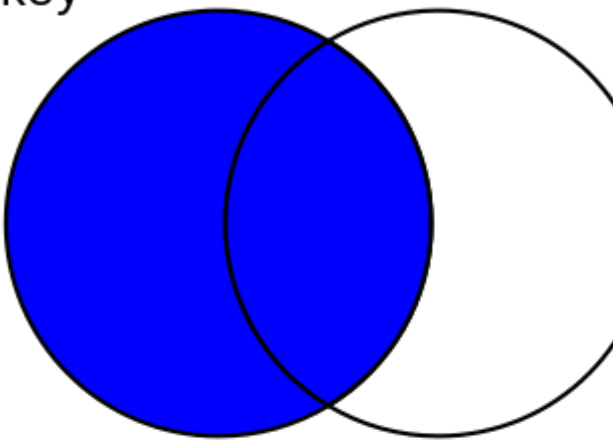
Les jointures récursives sont souvent utilisées pour obtenir des données parent-enfant. En SQL, ils sont implémentés avec des expressions de table communes récursives, par exemple:

```
WITH RECURSIVE MyDescendants AS (
    SELECT Name
    FROM People
    WHERE Name = 'John Doe'

    UNION ALL

    SELECT People.Name
    FROM People
    JOIN MyDescendants ON People.Name = MyDescendants.Parent
)
SELECT * FROM MyDescendants;
```

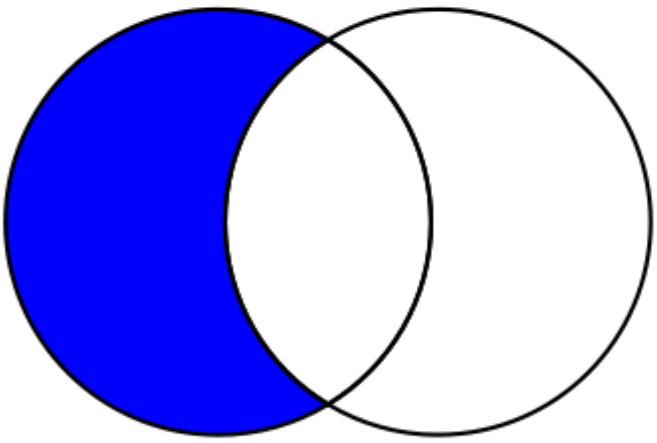
Différences entre les jointures intérieures / extérieures

SQL a différents types de jointure pour spécifier si des lignes correspondantes (non) sont incluses dans le résultat: `INNER JOIN` , `LEFT OUTER JOIN` , `RIGHT OUTER JOIN` et `FULL OUTER JOIN` (les mots-clés `INNER` et `OUTER` sont facultatifs). La figure ci-dessous souligne les différences entre ces types de jointures: la zone bleue représente les résultats renvoyés par la jointure et la zone blanche représente les résultats que la jointure ne renverra pas.

```
SELECT <fields>
FROM TableA  A
LEFT JOIN TableB  B
ON A.key = B.key
```



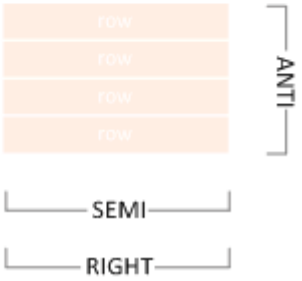
```
SELECT <fields>
FROM TableA  A
LEFT JOIN TableB  B
ON A.key = B.key
WHERE B.key IS NULL
```



ici .

Right Anti Semi Join

Inclut les lignes droites qui ne correspondent **pas** aux lignes de gauche.




```
SELECT * FROM B WHERE Y NOT IN (SELECT X FROM A);

Y
-------
Tim
Vincent
```

Comme vous pouvez le voir, il n'y a pas de syntaxe NOT IN dédiée pour la jointure anti-semi gauche et droite - nous obtenons l'effet simplement en changeant les positions de la table dans le texte SQL.

Cross Join

Un produit cartésien de tous laissé avec toutes les bonnes lignes.

```
SELECT * FROM A CROSS JOIN B;

X      Y
```

```
-----  -------
Amy    Lisa
John   Lisa
Lisa   Lisa
Marco  Lisa
Phil   Lisa
Amy    Marco
John   Marco
Lisa   Marco
Marco  Marco
Phil   Marco
Amy    Phil
John   Phil
Lisa   Phil
Marco  Phil
Phil   Phil
Amy    Tim
John   Tim
Lisa   Tim
Marco  Tim
Phil   Tim
Amy    Vincent
John   Vincent
Lisa   Vincent
Marco  Vincent
Phil   Vincent
```

La jointure croisée équivaut à une jointure interne avec une condition de jointure qui correspond toujours, de sorte que la requête suivante aurait renvoyé le même résultat:

```
SELECT * FROM A JOIN B ON 1 = 1;
```

Self-Join

Cela dénote simplement une table se joignant à elle-même. Une auto-jointure peut être l'un des types de jointure décrits ci-dessus. Par exemple, ceci est une auto-jointure interne:

```
SELECT * FROM A A1 JOIN A A2 ON LEN(A1.X) < LEN(A2.X);

X     X
----  -----
Amy   John
Amy   Lisa
Amy   Marco
John  Marco
Lisa  Marco
Phil  Marco
Amy   Phil
```

Lire JOINDRE en ligne: https://riptutorial.com/fr/sql/topic/261/joindre

Chapitre 41: METTRE À JOUR

Syntaxe

- *Table de mise à jour*
  SET *nom_colonne* = *valeur* , *nom_colonne2* = *valeur_2* , ..., *nom_colonne_n* = *valeur_n*
  WHERE *condition* (*opérateur logique* condition_n)

Examples

Mise à jour de toutes les lignes

Cet exemple utilise la table Cars de l'exemple de base de données.

```
UPDATE Cars
SET Status = 'READY'
```

Cette instruction définit la colonne 'status' de toutes les lignes de la table 'Cars' sur "READY" car elle ne contient pas de clause `WHERE` pour filtrer le jeu de lignes.

Mise à jour des lignes spécifiées

Cet exemple utilise la table Cars de l'exemple de base de données.

```
UPDATE
    Cars
SET
    Status = 'READY'
WHERE
    Id = 4
```

Cette instruction définira le statut de la ligne de 'Cars' avec l'ID 4 sur "READY".

`WHERE` clause `WHERE` contient une expression logique qui est évaluée pour chaque ligne. Si une ligne remplit les critères, sa valeur est mise à jour. Sinon, une ligne reste inchangée.

Modification des valeurs existantes

Cet exemple utilise la table Cars de l'exemple de base de données.

```
UPDATE Cars
SET TotalCost = TotalCost + 100
WHERE Id = 3 or Id = 4
```

Les opérations de mise à jour peuvent inclure des valeurs actuelles dans la ligne mise à jour. Dans cet exemple simple, `TotalCost` est incrémenté de 100 pour deux lignes:

- Le TotalCost de la voiture n ° 3 passe de 100 à 200
- Le TotalCost de la voiture n ° 4 est passé de 1254 à 1354

La nouvelle valeur d'une colonne peut être dérivée de sa valeur précédente ou de toute autre valeur de colonne dans la même table ou une table jointe.

MISE À JOUR avec des données d'une autre table

Les exemples ci-dessous `PhoneNumber` un `PhoneNumber` de `PhoneNumber` pour tout employé qui est également `Customer` et pour lequel aucun numéro de téléphone n'est actuellement défini dans la table `Employees` .

(Ces exemples utilisent les tables Employees et Customers des exemples de bases de données.)

SQL standard

Mettre à jour en utilisant une sous-requête corrélée:

```
UPDATE
    Employees
SET PhoneNumber =
    (SELECT
         c.PhoneNumber
     FROM
         Customers c
     WHERE
         c.FName = Employees.FName
         AND c.LName = Employees.LName)
WHERE Employees.PhoneNumber IS NULL
```

SQL: 2003

Mettre à jour en utilisant `MERGE` :

```
MERGE INTO
    Employees e
USING
    Customers c
ON
    e.FName = c.Fname
    AND e.LName = c.LName
    AND e.PhoneNumber IS NULL
WHEN MATCHED THEN
   UPDATE
      SET PhoneNumber = c.PhoneNumber
```

serveur SQL

Mettre à jour avec `INNER JOIN` :

```
UPDATE
    Employees
SET
    PhoneNumber = c.PhoneNumber
FROM
    Employees e
INNER JOIN Customers c
        ON e.FName = c.FName
        AND e.LName = c.LName
WHERE
    PhoneNumber IS NULL
```

Capture des enregistrements mis à jour

Parfois, on veut capturer les enregistrements qui viennent d'être mis à jour.

```
CREATE TABLE #TempUpdated(ID INT)

Update TableName SET Col1 = 42
    OUTPUT inserted.ID INTO #TempUpdated
    WHERE Id > 50
```

Lire METTRE À JOUR en ligne: https://riptutorial.com/fr/sql/topic/321/mettre-a-jour

Chapitre 42: Nettoyer le code en SQL

Introduction

Comment écrire de bonnes requêtes SQL lisibles et des exemples de bonnes pratiques.

Examples

Formatage et orthographe des mots-clés et des noms

Noms de table / colonne

`CamelCase` et `snake_case` sont deux méthodes courantes de formatage des noms de table / colonne:

```
SELECT FirstName, LastName
FROM Employees
WHERE Salary > 500;
```

```
SELECT first_name, last_name
FROM employees
WHERE salary > 500;
```

Les noms doivent décrire ce qui est stocké dans leur objet. Cela implique que les noms de colonnes doivent généralement être singuliers. Si les noms de table doivent utiliser le singulier ou le pluriel est une question très discutée , mais dans la pratique, il est plus courant d'utiliser des noms de table pluriels.

L'ajout de préfixes ou de suffixes comme `tbl` ou `col` réduit la lisibilité, alors évitez-les. Cependant, ils sont parfois utilisés pour éviter les conflits avec les mots-clés SQL et sont souvent utilisés avec des déclencheurs et des index (dont les noms ne sont généralement pas mentionnés dans les requêtes).

Mots clés

Les mots-clés SQL ne sont pas sensibles à la casse. Cependant, il est courant de les écrire en majuscule.

SELECT *

`SELECT *` renvoie toutes les colonnes dans le même ordre qu'elles sont définies dans la table.

Lorsque vous utilisez `SELECT *` , les données renvoyées par une requête peuvent changer chaque fois que la définition de la table change. Cela augmente le risque que différentes versions de votre

application ou de votre base de données soient incompatibles.

De plus, la lecture de plus de colonnes que nécessaire peut augmenter la quantité d'E / S disque et réseau.

Vous devez donc toujours spécifier explicitement la ou les colonnes que vous souhaitez récupérer:

```
--SELECT *                                  don't
  SELECT ID, FName, LName, PhoneNumber  -- do
  FROM Emplopees;
```

(Lorsque vous effectuez des requêtes interactives, ces considérations ne s'appliquent pas.)

Cependant, `SELECT *` ne fait pas mal dans la sous-requête d'un opérateur EXISTS, car EXISTS ignore de toute façon les données réelles (il vérifie uniquement si au moins une ligne a été trouvée). Pour la même raison, il n'est pas utile de lister des colonnes spécifiques pour EXISTS, de sorte que `SELECT *` a plus de sens:

```
-- list departments where nobody was hired recently
SELECT ID,
       Name
FROM Departments
WHERE NOT EXISTS (SELECT *
                  FROM Employees
                  WHERE DepartmentID = Departments.ID
                    AND HireDate >= '2015-01-01');
```

En retrait

Il n'y a pas de norme largement acceptée. Tout le monde s'accorde sur le fait que tout faire en une seule ligne est mauvais:

```
SELECT d.Name, COUNT(*) AS Employees FROM Departments AS d JOIN Employees AS e ON d.ID =
e.DepartmentID WHERE d.Name != 'HR' HAVING COUNT(*) > 10 ORDER BY COUNT(*) DESC;
```

Au minimum, mettez chaque clause dans une nouvelle ligne et séparez les lignes si elles deviennent trop longues sinon:

```
SELECT d.Name,
       COUNT(*) AS Employees
FROM Departments AS d
JOIN Employees AS e ON d.ID = e.DepartmentID
WHERE d.Name != 'HR'
HAVING COUNT(*) > 10
ORDER BY COUNT(*) DESC;
```

Parfois, tout ce qui suit le mot-clé SQL introduisant une clause est en retrait sur la même colonne:

```
SELECT   d.Name,
```

```
        COUNT(*) AS Employees
FROM    Departments AS d
JOIN    Employees AS e ON d.ID = e.DepartmentID
WHERE   d.Name != 'HR'
HAVING  COUNT(*) > 10
ORDER BY COUNT(*) DESC;
```

(Cela peut également être fait en alignant correctement les mots-clés SQL.)

Un autre style courant consiste à placer des mots-clés importants sur leurs propres lignes:

```
SELECT
    d.Name,
    COUNT(*) AS Employees
FROM
    Departments AS d
JOIN
    Employees AS e
    ON d.ID = e.DepartmentID
WHERE
    d.Name != 'HR'
HAVING
    COUNT(*) > 10
ORDER BY
    COUNT(*) DESC;
```

L'alignement vertical de plusieurs expressions similaires améliore la lisibilité:

```
SELECT Model,
       EmployeeID
FROM Cars
WHERE CustomerID = 42
  AND Status     = 'READY';
```

L'utilisation de plusieurs lignes rend plus difficile l'intégration de commandes SQL dans d'autres langages de programmation. Cependant, de nombreux langages ont un mécanisme pour les chaînes multi-lignes, par exemple `@"..."` en C #, `"""..."""` en Python ou `R"(...)"` en C ++.

Joint

Les jointures explicites doivent toujours être utilisées. les jointures implicites ont plusieurs problèmes:

- La condition de jointure se trouve quelque part dans la clause WHERE, mélangée à toute autre condition de filtre. Cela rend plus difficile de voir quelles tables sont jointes et comment.
- En raison de ce qui précède, le risque d'erreurs est plus élevé et il est plus probable qu'elles soient retrouvées plus tard.
- En SQL standard, les jointures explicites sont le seul moyen d'utiliser des jointures externes

:

```
SELECT d.Name,
       e.Fname || e.LName AS EmpName
FROM      Departments AS d
LEFT JOIN Employees   AS e ON d.ID = e.DepartmentID;
```

- Les jointures explicites permettent d'utiliser la clause USING:

```
SELECT RecipeID,
       Recipes.Name,
       COUNT(*) AS NumberOfIngredients
FROM      Recipes
LEFT JOIN Ingredients USING (RecipeID);
```

(Cela nécessite que les deux tables utilisent le même nom de colonne.
USING supprime automatiquement la colonne dupliquée du résultat, par exemple, la jointure dans cette requête renvoie une seule colonne `RecipeID` .)

Lire Nettoyer le code en SQL en ligne: https://riptutorial.com/fr/sql/topic/9843/nettoyer-le-code-en-sql

Chapitre 43: NUL

Introduction

`NULL` dans SQL, ainsi que la programmation en général, signifie littéralement "rien". En SQL, il est plus facile de comprendre "l'absence de valeur".

Il est important de le distinguer des valeurs apparemment vides, telles que la chaîne vide `''` ou le nombre `0` , dont aucun n'est réellement `NULL` .

Il est également important de faire attention à ne pas inclure `NULL` entre guillemets, comme `'NULL'` , qui est autorisé dans les colonnes qui acceptent le texte, mais n'est pas `NULL` et peut provoquer des erreurs et des ensembles de données incorrects.

Examples

Filtrage pour NULL dans les requêtes

La syntaxe de filtrage pour `NULL` (c'est-à-dire l'absence de valeur) dans les blocs `WHERE` est légèrement différente du filtrage pour des valeurs spécifiques.

```
SELECT * FROM Employees WHERE ManagerId IS NULL ;
SELECT * FROM Employees WHERE ManagerId IS NOT NULL ;
```

Notez que parce que `NULL` n'est égal à rien, même pas à lui-même, l'utilisation des opérateurs d'égalité `= NULL` ou `<> NULL` (ou `!= NULL`) donnera toujours la valeur de vérité de `UNKNOWN` qui sera rejetée par `WHERE` .

`WHERE` filtre toutes les lignes dont la condition est `FALSE` ou `UKNOWN` et ne conserve que les lignes dont la condition est `TRUE` .

Colonnes nullables dans les tableaux

Lors de la création de tables, il est possible de déclarer une colonne comme nullable ou non nullable.

```
CREATE TABLE MyTable
(
    MyCol1 INT NOT NULL, -- non-nullable
    MyCol2 INT NULL      -- nullable
) ;
```

Par défaut, chaque colonne (à l'exception de celles dans la contrainte de clé primaire) est nullable sauf si nous définissons explicitement la contrainte `NOT NULL` .

Si vous tentez d'attribuer `NULL` à une colonne non nullable, cela entraînera une erreur.

```
INSERT INTO MyTable (MyCol1, MyCol2) VALUES (1, NULL) ;  -- works fine

INSERT INTO MyTable (MyCol1, MyCol2) VALUES (NULL, 2) ;
        -- cannot insert
        -- the value NULL into column 'MyCol1', table 'MyTable';
        -- column does not allow nulls. INSERT fails.
```

Mise à jour des champs sur NULL

Définir un champ sur `NULL` fonctionne exactement comme avec toute autre valeur:

```
UPDATE Employees
SET ManagerId = NULL
WHERE Id = 4
```

Insertion de lignes avec des champs NULL

Par exemple, insérer un employé sans numéro de téléphone et aucun responsable dans le tableau exemple Employees :

```
INSERT INTO Employees
    (Id, FName, LName, PhoneNumber, ManagerId, DepartmentId, Salary, HireDate)
VALUES
    (5, 'Jane', 'Doe', NULL, NULL, 2, 800, '2016-07-22') ;
```

Lire NUL en ligne: https://riptutorial.com/fr/sql/topic/3421/nul

Chapitre 44: Numéro de ligne

Syntaxe

- ROW_NUMBER ()
- OVER ([PARTITION BY value_expression, ... [n]] order_by_clause)

Examples

Numéros de lignes sans partitions

Inclure un numéro de ligne selon l'ordre spécifié.

```
SELECT
  ROW_NUMBER() OVER(ORDER BY Fname ASC) AS RowNumber,
  Fname,
  LName
FROM Employees
```

Numéros de lignes avec partitions

Utilise un critère de partition pour regrouper la numérotation des lignes en fonction de celle-ci.

```
SELECT
  ROW_NUMBER() OVER(PARTITION BY DepartmentId ORDER BY DepartmentId ASC) AS RowNumber,
  DepartmentId, Fname, LName
FROM Employees
```

Supprimer tout sauf le dernier enregistrement (1 à plusieurs tableaux)

```
WITH cte AS (
  SELECT ProjectID,
         ROW_NUMBER() OVER (PARTITION BY ProjectID ORDER BY InsertDate DESC) AS rn
  FROM ProjectNotes
)
DELETE FROM cte WHERE rn > 1;
```

Lire Numéro de ligne en ligne: https://riptutorial.com/fr/sql/topic/1977/numero-de-ligne

Chapitre 45: Opérateur LIKE

Syntaxe

- **Wild Card avec%:** SELECT * FROM [table] WHERE [nom_colonne] Comme '% Value%'

  Wild Card avec _: SELECT * FROM [table] WHERE [nom_colonne] Comme 'V_n%'

  Wild Card avec [charlist]: SELECT * FROM [table] WHERE [nom_colonne] Comme 'V [abc] n%'

Remarques

La condition LIKE dans la clause WHERE est utilisée pour rechercher des valeurs de colonne qui correspondent au modèle donné. Les motifs sont formés à l'aide des deux caractères génériques suivants

- % (Symbole de pourcentage) - Utilisé pour représenter zéro ou plusieurs caractères
- _ (Underscore) - Utilisé pour représenter un seul caractère

Examples

Match à motif ouvert

Le caractère générique ％ ajouté au début ou à la fin (ou les deux) d'une chaîne autorise la correspondance de 0 ou plus de tout caractère avant le début ou après la fin du motif.

L'utilisation de '%' au milieu permettra de faire correspondre au moins 0 caractère entre les deux parties du motif.

Nous allons utiliser cette table Employees:

Id	FName	LName	Numéro de téléphone	ManagerId	DépartementId	Un salaire	Date d'embauche
1	John	Johnson	2468101214	1	1	400	23-03-2005
2	Sophie	Amudsen	2479100211	1	1	400	11-01-2010
3	Ronny	Forgeron	2462544026	2	1	600	06-08-2015
4	Jon	Sanchez	2454124602	1	1	400	23-03-2005
5	Hilde	Knag	2468021911	2	1	800	01-01-2000

L'instruction suivante correspond à tous les enregistrements ayant FName **contenant la** chaîne

'on' de la table Employees.

```
SELECT * FROM Employees WHERE FName LIKE '%on%';
```

Id	FName	LName	Numéro de téléphone	ManagerId	DépartementId	Un salaire	Date d'embauche
3	R **on** ny	Forgeron	2462544026	2	1	600	06-08-2015
4	J **sur**	Sanchez	2454124602	1	1	400	23-03-2005

L'instruction suivante correspond à tous les enregistrements ayant PhoneNumber **commençant par la** chaîne '246' par Employees.

```
SELECT * FROM Employees WHERE PhoneNumber LIKE '246%';
```

Id	FName	LName	Numéro de téléphone	ManagerId	DépartementId	Un salaire	Date d'embauche
1	John	Johnson	**246** 8101214	1	1	400	23-03-2005
3	Ronny	Forgeron	**246** 2544026	2	1	600	06-08-2015
5	Hilde	Knag	**246** 8021911	2	1	800	01-01-2000

L'instruction suivante correspond à tous les enregistrements ayant PhoneNumber se **terminant par la** chaîne '11' de Employees.

```
SELECT * FROM Employees WHERE PhoneNumber LIKE '%11'
```

Id	FName	LName	Numéro de téléphone	ManagerId	DépartementId	Un salaire	Date d'embauche
2	Sophie	Amudsen	24791002 **11**	1	1	400	11-01-2010
5	Hilde	Knag	24680219 **11**	2	1	800	01-01-2000

Tous les enregistrements où le **3ème caractère de** Fname est 'n' de Employees.

```
SELECT * FROM Employees WHERE FName LIKE '__n%';
```

(deux traits de soulignement sont utilisés avant "n" pour sauter les 2 premiers caractères)

Id	FName	LName	Numéro de téléphone	ManagerId	DépartementId	Un salaire	Date d'embauche
3	Ronny	Forgeron	2462544026	2	1	600	06-08-2015
4	Jon	Sanchez	2454124602	1	1	400	23-03-2005

Match de personnage unique

Pour élargir les sélections d'une instruction de langage de requête structurée (SQL-SELECT), vous pouvez utiliser des caractères génériques, le signe de pourcentage (%) et le trait de soulignement (_).

Le caractère _ (trait de soulignement) peut être utilisé comme caractère générique pour n'importe quel caractère d'un motif.

Trouvez tous les employés dont le nom F commence par 'j' et se termine par 'n' et comporte exactement 3 caractères dans Fname.

```
SELECT * FROM Employees WHERE FName LIKE 'j_n'
```

_ (trait de soulignement) peut également être utilisé plusieurs fois comme joker pour correspondre à des motifs.

Par exemple, ce modèle correspondrait à "jon", "jan", "jen", etc.

Ces noms ne seront pas affichés "jn", "john", "jordan", "justin", "jason", "julian", "jillian", "joann" car dans notre requête un trait de soulignement est utilisé et il peut sauter un caractère, le résultat doit donc être de 3 caractères Fname.

Par exemple, ce modèle correspondrait à "LaSt", "LoSt", "HaLt", etc.

```
SELECT * FROM Employees WHERE FName LIKE '_A_T'
```

Match par plage ou ensemble

Faites correspondre n'importe quel caractère dans la plage spécifiée (par exemple: `[af]`) ou set (par exemple: `[abcdef]`).

Ce modèle de plage correspondrait à "gary" mais pas à "mary":

```
SELECT * FROM Employees WHERE FName LIKE '[a-g]ary'
```

Ce pattern correspondrait à "mary" mais pas à "gary":

```
SELECT * FROM Employees WHERE Fname LIKE '[lmnop]ary'
```

La plage ou l'ensemble peut également être annulé en ajoutant le curseur ^ avant l'intervalle ou le jeu:

Ce modèle d'intervalle *ne* correspondrait *pas* à "gary" mais correspondrait à "mary":

```
SELECT * FROM Employees WHERE FName LIKE '[^a-g]ary'
```

Ce pattern *ne* correspondrait *pas* à "mary" mais correspondrait à "gary":

```
SELECT * FROM Employees WHERE Fname LIKE '[^lmnop]ary'
```

Correspondre TOUT contre TOUS

Correspond à n'importe lequel:
Doit correspondre à au moins une chaîne. Dans cet exemple, le type de produit doit être «électronique», «livres» ou «vidéo».

```
SELECT *
FROM   purchase_table
WHERE  product_type LIKE ANY ('electronics', 'books', 'video');
```

Match all (doit répondre à toutes les exigences).
Dans cet exemple, les termes «royaume-uni» *et* «Londres» *et* «route de l’est» (y compris les variantes) doivent être appariés.

```
SELECT *
FROM   customer_table
WHERE  full_address LIKE ALL ('%united kingdom%', '%london%', '%eastern road%');
```

Sélection négative:
Utilisez ALL pour exclure tous les éléments.
Cet exemple fournit tous les résultats lorsque le type de produit n'est pas «électronique» et non «livres» et non «vidéo».

```
SELECT *
FROM   customer_table
WHERE  product_type NOT LIKE ALL ('electronics', 'books', 'video');
```

Rechercher une gamme de caractères

L'instruction suivante correspond à tous les enregistrements ayant FName qui commence par une lettre de A à F à partir de la table Employees .

```
SELECT * FROM Employees WHERE FName LIKE '[A-F]%'
```

Instruction ESCAPE dans la requête LIKE

Si vous implémentez une recherche textuelle comme `LIKE` -query, vous le faites généralement comme ceci:

```
SELECT *
FROM T_Whatever
WHERE SomeField LIKE CONCAT('%', @in_SearchText, '%')
```

Cependant, (à part le fait que vous ne devriez pas nécessairement utiliser `LIKE` lorsque vous pouvez utiliser la recherche en texte intégral), cela crée un problème lorsque quelqu'un saisit un texte comme "50%" ou "a_b".

Donc, au lieu de passer en recherche de texte intégral, vous pouvez résoudre ce problème en utilisant l'instruction `LIKE` -escape:

```
SELECT *
FROM T_Whatever
WHERE SomeField LIKE CONCAT('%', @in_SearchText, '%') ESCAPE '\'
```

Cela signifie que `\` sera maintenant traité comme un caractère ESCAPE. Cela signifie que vous pouvez maintenant simplement ajouter `\` à chaque caractère de la chaîne recherchée, et les résultats commenceront à être corrects, même lorsque l'utilisateur entre un caractère spécial tel que `%` ou _ .

par exemple

```
string stringToSearch = "abc_def 50%";
string newString = "";
foreach(char c in stringToSearch)
     newString += @"\" + c;

sqlCmd.Parameters.Add("@in_SearchText", newString);
// instead of sqlCmd.Parameters.Add("@in_SearchText", stringToSearch);
```

Remarque: L'algorithme ci-dessus est à des fins de démonstration uniquement. Cela ne fonctionnera pas dans les cas où 1 graphème est composé de plusieurs caractères (utf-8). par exemple `string stringToSearch = "Les Mise\u0301rables";` Vous devrez le faire pour chaque graphème, pas pour chaque caractère. Vous ne devez pas utiliser l'algorithme ci-dessus si vous traitez avec des langues asiatiques / asiatiques / asiatiques du sud. Ou plutôt, si vous voulez un code correct pour commencer, vous devez simplement le faire pour chaque graphe.

Voir aussi ReverseString, une question d'interview C #

Caractères génériques

les caractères génériques sont utilisés avec l'opérateur SQL LIKE. Les caractères génériques SQL sont utilisés pour rechercher des données dans une table.

Les caractères génériques dans SQL sont:%, _, [charlist], [^ charlist]

% - Un substitut à zéro ou plusieurs caractères

```
   Eg:  //selects all customers with a City starting with "Lo"
        SELECT * FROM Customers
        WHERE City LIKE 'Lo%';

       //selects all customers with a City containing the pattern "es"
      SELECT * FROM Customers
       WHERE City LIKE '%es%';
```

_ - Un substitut à un seul caractère

```
Eg://selects all customers with a City starting with any character, followed by "erlin"
SELECT * FROM Customers
WHERE City LIKE '_erlin';
```

[charlist] - Ensembles et plages de caractères à associer

```
Eg://selects all customers with a City starting with "a", "d", or "l"
SELECT * FROM Customers
WHERE City LIKE '[adl]%';

//selects all customers with a City starting with "a", "d", or "l"
SELECT * FROM Customers
WHERE City LIKE '[a-c]%';
```

[^ charlist] - Correspond uniquement à un caractère non spécifié entre crochets

```
Eg://selects all customers with a City starting with a character that is not "a", "p", or "l"
SELECT * FROM Customers
WHERE City LIKE '[^apl]%';

or

SELECT * FROM Customers
WHERE City NOT LIKE '[apl]%' and city like '_%';
```

Lire Opérateur LIKE en ligne: https://riptutorial.com/fr/sql/topic/860/operateur-like

Chapitre 46: Opérateurs ET & OU

Syntaxe

1. SELECT * FROM table WHERE (condition1) AND (condition2);
2. SELECT * FROM table WHERE (condition1) OU (condition2);

Examples

ET OU Exemple

Avoir une table

prénom	Âge	Ville
Bob	dix	Paris
Tapis	20	Berlin
Marie	24	Prague

```
select Name from table where Age>10 AND City='Prague'
```

Donne

prénom
Marie

```
select Name from table where Age=10 OR City='Prague'
```

Donne

prénom
Bob
Marie

Lire Opérateurs ET & OU en ligne: https://riptutorial.com/fr/sql/topic/1386/operateurs-et--amp--ou

Chapitre 47: Ordre d'exécution

Examples

Ordre logique du traitement des requêtes en SQL

```
/*(8)*/  SELECT /*9*/ DISTINCT /*11*/ TOP
/*(1)*/  FROM
/*(3)*/         JOIN
/*(2)*/        ON
/*(4)*/  WHERE
/*(5)*/  GROUP BY
/*(6)*/  WITH {CUBE | ROLLUP}
/*(7)*/  HAVING
/*(10)*/ ORDER BY
/*(11)*/ LIMIT
```

L'ordre dans lequel une requête est traitée et la description de chaque section.

VT signifie «Virtual Table» et montre comment différentes données sont générées lors du traitement de la requête.

1. FROM: Un produit cartésien (jointure croisée) est exécuté entre les deux premières tables de la clause FROM et, par conséquent, la table virtuelle VT1 est générée.

2. ON: Le filtre ON est appliqué à VT1. Seules les lignes pour lesquelles le est TRUE sont insérées dans VT2.

3. OUTER (join): si un joint OUTER JOIN est spécifié (par opposition à un CROSS JOIN ou un INNER JOIN), les lignes de la ou des tables conservées pour lesquelles une correspondance n'a pas été trouvée sont ajoutées aux lignes de VT2 en tant que lignes externes, générant VT3. Si plus de deux tables apparaissent dans la clause FROM, les étapes 1 à 3 sont appliquées de manière répétée entre le résultat de la dernière jointure et la table suivante de la clause FROM jusqu'à ce que toutes les tables soient traitées.

4. WHERE: Le filtre WHERE est appliqué à VT3. Seules les lignes pour lesquelles la valeur est TRUE sont insérées dans VT4.

5. GROUP BY: les lignes de VT4 sont organisées en groupes en fonction de la liste de colonnes spécifiée dans la clause GROUP BY. VT5 est généré.

6. CUBE | ROLLUP: Des supergroupes (groupes de groupes) sont ajoutés aux lignes de VT5, générant VT6.

7. HAVING: Le filtre HAVING est appliqué à VT6. Seuls les groupes pour lesquels le est TRUE sont insérés dans VT7.

8. SELECT: la liste SELECT est traitée, générant VT8.

9. DISTINCT: les lignes dupliquées sont supprimées de VT8. VT9 est généré.

10. ORDER BY: Les lignes de VT9 sont triées en fonction de la liste de colonnes spécifiée dans la clause ORDER BY. Un curseur est généré (VC10).

11. TOP: Le nombre ou le pourcentage de lignes spécifié est sélectionné depuis le début de VC10. La table VT11 est générée et renvoyée à l'appelant. LIMIT a la même fonctionnalité que TOP dans certains dialectes SQL tels que Postgres et Netezza.

Lire Ordre d'exécution en ligne: https://riptutorial.com/fr/sql/topic/3671/ordre-d-execution

Chapitre 48: PAR GROUPE

Introduction

Les résultats d'une requête SELECT peuvent être regroupés par une ou plusieurs colonnes à l'aide de l'instruction `GROUP BY` : tous les résultats ayant la même valeur dans les colonnes groupées sont regroupés. Cela génère une table de résultats partiels, au lieu d'un résultat. GROUP BY peut être utilisé conjointement avec les fonctions d'agrégation à l'aide de l'instruction `HAVING` pour définir la manière dont les colonnes non groupées sont agrégées.

Syntaxe

- PAR GROUPE {
  expression de colonne
  | ROLLUP (<group_by_expression> [, ... n])
  | CUBE (<expression_de_groupe> [, ... n])
  | GROUPING SETS ([, ... n])
  | () calcule le total général
  } [, ... n]

- <group_by_expression> :: =
  expression de colonne
  | (expression de colonne [, ... n])

- <grouping_set> :: =
  () calcule le total général
  | <grouping_set_item>
  | (<grouping_set_item> [, ... n])

- <grouping_set_item> :: =
  <group_by_expression>
  | ROLLUP (<group_by_expression> [, ... n])
  | CUBE (<expression_de_groupe> [, ... n])

Examples

UTILISEZ GROUP BY pour COUNT le nombre de lignes pour chaque entrée unique dans une colonne donnée

Supposons que vous souhaitiez générer des comptes ou des sous-totaux pour une valeur donnée dans une colonne.

Compte tenu de ce tableau, "Westerosians":

prénom	GreatHouseAllegience
Arya	Rigide
Cercei	Lannister
Myrcella	Lannister
Yara	Greyjoy
Catelyn	Rigide
Sansa	Rigide

Sans GROUP BY, COUNT retournera simplement un nombre total de lignes:

```
SELECT Count(*) Number_of_Westerosians
FROM Westerosians
```

résultats...

Number_of_Westerosians
6

Mais en ajoutant GROUP BY, nous pouvons COMPTER les utilisateurs pour chaque valeur dans une colonne donnée, pour retourner le nombre de personnes dans une Grande Maison donnée, par exemple:

```
SELECT GreatHouseAllegience House, Count(*) Number_of_Westerosians
FROM Westerosians
GROUP BY GreatHouseAllegience
```

résultats...

Maison	Number_of_Westerosians
Rigide	3
Greyjoy	1
Lannister	2

Il est courant de combiner GROUP BY avec ORDER BY pour trier les résultats par catégorie la plus grande ou la plus petite:

```
SELECT GreatHouseAllegience House, Count(*) Number_of_Westerosians
FROM Westerosians
GROUP BY GreatHouseAllegience
```

```
ORDER BY Number_of_Westerosians Desc
```

résultats...

Maison	Number_of_Westerosians
Rigide	3
Lannister	2
Greyjoy	1

Filtrez les résultats de GROUP BY en utilisant une clause HAVING

Une clause HAVING filtre les résultats d'une expression GROUP BY. Remarque: Les exemples suivants utilisent la base de données exemple de la bibliothèque .

Exemples:

Renvoie tous les auteurs ayant écrit plus d'un livre (exemple en direct).

```
SELECT
  a.Id,
  a.Name,
  COUNT(*) BooksWritten
FROM BooksAuthors ba
  INNER JOIN Authors a ON a.id = ba.authorid
GROUP BY
  a.Id,
  a.Name
HAVING COUNT(*) > 1    -- equals to HAVING BooksWritten > 1
;
```

Retourne tous les livres qui ont plus de trois auteurs (exemple live).

```
SELECT
  b.Id,
  b.Title,
  COUNT(*) NumberOfAuthors
FROM BooksAuthors ba
  INNER JOIN Books b ON b.id = ba.bookid
GROUP BY
  b.Id,
  b.Title
HAVING COUNT(*) > 3    -- equals to HAVING NumberOfAuthors > 3
;
```

Exemple de GROUP BY de base

Cela pourrait être plus facile si vous considérez GROUP BY comme "pour chacun" pour des raisons d'explication. La requête ci-dessous:

```
SELECT EmpID, SUM (MonthlySalary)
FROM Employee
GROUP BY EmpID
```

Est en train de dire:

"Donnez-moi la somme de MonthlySalary's **pour chaque** EmpID"

Donc, si votre table ressemblait à ceci:

```
+-----+-------------+
|EmpID|MonthlySalary|
+-----+-------------+
|1    |200          |
+-----+-------------+
|2    |300          |
+-----+-------------+
```

Résultat:

```
+-+---+
|1|200|
+-+---+
|2|300|
+-+---+
```

Sum ne semble rien faire car la somme d'un nombre est ce nombre. Par contre si ça ressemblait à ça:

```
+-----+-------------+
|EmpID|MonthlySalary|
+-----+-------------+
|1    |200          |
+-----+-------------+
|1    |300          |
+-----+-------------+
|2    |300          |
+-----+-------------+
```

Résultat:

```
+-+---+
|1|500|
+-+---+
|2|300|
+-+---+
```

Alors ce serait parce qu'il y a deux EmpID 1 à résumer ensemble.

Agrégation ROLAP (Data Mining)

La description

Le standard SQL fournit deux opérateurs d'agrégat supplémentaires. Ceux-ci utilisent la valeur polymorphe "ALL" pour désigner l'ensemble de toutes les valeurs qu'un attribut peut prendre. Les deux opérateurs sont:

- `with data cube` qu'il fournit toutes les combinaisons possibles aux attributs d'argument de la clause.
- `with roll up` qu'il fournit les agrégats obtenus en considérant les attributs dans l'ordre de gauche à droite par rapport à la façon dont ils sont listés dans l'argument de la clause.

Versions standard SQL qui prennent en charge ces fonctionnalités: 1999,2003,2006,2008,2011.

Exemples

Considérez ce tableau:

Aliments	Marque	Montant total
Pâtes	Marque1	100
Pâtes	Marque2	250
Pizza	Marque2	300

Avec cube

```
select Food,Brand,Total_amount
from Table
group by Food,Brand,Total_amount with cube
```

Aliments	Marque	Montant total
Pâtes	Marque1	100
Pâtes	Marque2	250
Pâtes	TOUT	350
Pizza	Marque2	300
Pizza	TOUT	300
TOUT	Marque1	100
TOUT	Marque2	550
TOUT	TOUT	650

Avec roll up

```
select Food,Brand,Total_amount
from Table
group by Food,Brand,Total_amount with roll up
```

Aliments	Marque	Montant total
Pâtes	Marque1	100
Pâtes	Marque2	250
Pizza	Marque2	300
Pâtes	TOUT	350
Pizza	TOUT	300
TOUT	TOUT	650

Lire PAR GROUPE en ligne: https://riptutorial.com/fr/sql/topic/627/par-groupe

Chapitre 49: Procédures stockées

Remarques

Les procédures stockées sont des instructions SQL stockées dans la base de données qui peuvent être exécutées ou appelées dans des requêtes. L'utilisation d'une procédure stockée permet d'encapsuler une logique complexe ou fréquemment utilisée et améliore les performances des requêtes en utilisant des plans de requête mis en cache. Ils peuvent renvoyer n'importe quelle valeur qu'une requête standard peut renvoyer.

D'autres avantages par rapport aux expressions SQL dynamiques sont répertoriés sur Wikipeida .

Examples

Créer et appeler une procédure stockée

Les procédures stockées peuvent être créées via une interface graphique de gestion de base de données (exemple SQL Server) ou via une instruction SQL comme suit:

```
-- Define a name and parameters
CREATE PROCEDURE Northwind.getEmployee
    @LastName nvarchar(50),
    @FirstName nvarchar(50)
AS

-- Define the query to be run
SELECT FirstName, LastName, Department
FROM Northwind.vEmployeeDepartment
WHERE FirstName = @FirstName AND LastName = @LastName
AND EndDate IS NULL;
```

Appeler la procédure:

```
EXECUTE Northwind.getEmployee N'Ackerman', N'Pilar';

-- Or
EXEC Northwind.getEmployee @LastName = N'Ackerman', @FirstName = N'Pilar';
GO

-- Or
EXECUTE Northwind.getEmployee @FirstName = N'Pilar', @LastName = N'Ackerman';
GO
```

Lire Procédures stockées en ligne: https://riptutorial.com/fr/sql/topic/1701/procedures-stockees

Chapitre 50: Recherche de doublons sur un sous-ensemble de colonne avec détails

Remarques

- Pour sélectionner des lignes sans doublons, remplacez la clause WHERE par "RowCnt = 1"
- Pour sélectionner une ligne de chaque ensemble, utilisez Rank () au lieu de Sum () et modifiez la clause WHERE externe pour sélectionner des lignes avec Rank () = 1

Examples

Etudiants avec même nom et date de naissance

```
WITH CTE (StudentId, Fname, LName, DOB, RowCnt)
as (
SELECT StudentId, FirstName, LastName, DateOfBirth as DOB, SUM(1) OVER (Partition By
FirstName, LastName, DateOfBirth) as RowCnt
FROM tblStudent
)
SELECT * from CTE where RowCnt > 1
ORDER BY DOB, LName
```

Cet exemple utilise une expression de table commune et une fonction de fenêtre pour afficher toutes les lignes en double (sur un sous-ensemble de colonnes) côte à côte.

Lire Recherche de doublons sur un sous-ensemble de colonne avec détails en ligne: https://riptutorial.com/fr/sql/topic/1585/recherche-de-doublons-sur-un-sous-ensemble-de-colonne-avec-details

Chapitre 51: SAUF

Remarques

`EXCEPT` renvoie toutes les valeurs distinctes de l'ensemble de données situées à gauche de l'opérateur EXCEPT qui ne sont pas également renvoyées par le jeu de données de droite.

Examples

Sélectionnez le jeu de données sauf lorsque les valeurs sont dans cet autre jeu de données

```
--dataset schemas must be identical
SELECT 'Data1' as 'Column' UNION ALL
SELECT 'Data2' as 'Column' UNION ALL
SELECT 'Data3' as 'Column' UNION ALL
SELECT 'Data4' as 'Column' UNION ALL
SELECT 'Data5' as 'Column'
EXCEPT
SELECT 'Data3' as 'Column'
--Returns Data1, Data2, Data4, and Data5
```

Lire SAUF en ligne: https://riptutorial.com/fr/sql/topic/4082/sauf

Chapitre 52: SAUTER LE SAUT (Pagination)

Examples

Ignorer certaines lignes du résultat

ISO / ANSI SQL:

```
SELECT Id, Col1
FROM TableName
ORDER BY Id
OFFSET 20 ROWS
```

MySQL:

```
SELECT * FROM TableName LIMIT 20, 42424242424242;
-- skips 20 for take use very large number that is more than rows in table
```

Oracle:

```
SELECT Id,
   Col1
FROM (SELECT Id,
           Col1,
           row_number() over (order by Id) RowNumber
      FROM TableName)
WHERE RowNumber > 20
```

PostgreSQL:

```
SELECT * FROM TableName OFFSET 20;
```

SQLite:

```
SELECT * FROM TableName LIMIT -1 OFFSET 20;
```

Limiter la quantité de résultats

ISO / ANSI SQL:

```
SELECT * FROM TableName FETCH FIRST 20 ROWS ONLY;
```

MySQL; PostgreSQL; SQLite:

```
SELECT * FROM TableName LIMIT 20;
```

Oracle:

```
SELECT Id,
   Col1
FROM (SELECT Id,
           Col1,
           row_number() over (order by Id) RowNumber
      FROM TableName)
WHERE RowNumber <= 20
```

Serveur SQL:

```
SELECT TOP 20 *
FROM dbo.[Sale]
```

Sauter puis prendre quelques résultats (Pagination)

ISO / ANSI SQL:

```
SELECT Id, Col1
FROM TableName
ORDER BY Id
OFFSET 20 ROWS FETCH NEXT 20 ROWS ONLY;
```

MySQL:

```
SELECT * FROM TableName LIMIT 20, 20; -- offset, limit
```

Oracle; Serveur SQL:

```
SELECT Id,
   Col1
 FROM (SELECT Id,
           Col1,
           row_number() over (order by Id) RowNumber
      FROM TableName)
WHERE RowNumber BETWEEN 21 AND 40
```

PostgreSQL; SQLite:

```
SELECT * FROM TableName LIMIT 20 OFFSET 20;
```

Lire SAUTER LE SAUT (Pagination) en ligne: https://riptutorial.com/fr/sql/topic/2927/sauter-le-saut--pagination-

Chapitre 53: Schéma d'information

Examples

Recherche de schéma d'information de base

L'une des requêtes les plus utiles pour les utilisateurs finaux de gros SGBDR est la recherche d'un schéma d'information.

Une telle requête permet aux utilisateurs de trouver rapidement des tables de base de données contenant des colonnes d'intérêt, par exemple lors de la tentative d'associer indirectement des données provenant de deux tables via une troisième table, sans savoir quelles tables peuvent contenir des clés ou d'autres colonnes utiles. .

À l'aide de T-SQL pour cet exemple, le schéma d'information d'une base de données peut être recherché comme suit:

```
SELECT *
FROM INFORMATION_SCHEMA.COLUMNS
WHERE COLUMN_NAME LIKE '%Institution%'
```

Le résultat contient une liste de colonnes correspondantes, les noms de leurs tables et d'autres informations utiles.

Lire Schéma d'information en ligne: https://riptutorial.com/fr/sql/topic/3151/schema-d-information

Chapitre 54: SÉLECTIONNER

Introduction

L'instruction SELECT est au cœur de la plupart des requêtes SQL. Il définit quel jeu de résultats doit être renvoyé par la requête et est presque toujours utilisé conjointement avec la clause FROM, qui définit les parties de la base de données à interroger.

Syntaxe

- SELECT [DISTINCT] [colonne1] [, [colonne2] ...]
  DE [table]
  [OERE condition]
  [GROUP BY [colonne1] [, [colonne2] ...]

  [HAVING [column1] [, [column2] ...]

  [COMMANDE PAR ASC | DESC]

Remarques

SELECT détermine les données des colonnes à renvoyer et l'ordre dans lequel elles proviennent d'une table donnée (étant donné qu'elles correspondent spécifiquement aux autres exigences de votre requête - où et avec les filtres et les jointures).

```
SELECT Name, SerialNumber
FROM ArmyInfo
```

ne renverra que les résultats des colonnes `Name` et `Serial Number` , mais pas de la colonne intitulée `Rank` , par exemple

```
SELECT *
FROM ArmyInfo
```

indique que **toutes les** colonnes seront renvoyées. Cependant, veuillez noter que `SELECT *` est une mauvaise pratique car vous retournez littéralement toutes les colonnes d'une table.

Examples

Utiliser le caractère générique pour sélectionner toutes les colonnes d'une requête.

Considérons une base de données avec les deux tableaux suivants.

Table d'employés:

Id	FName	LName	DeptId
1	James	Forgeron	3
2	John	Johnson	4

Table des départements:

Id	prénom
1	Ventes
2	Commercialisation
3	La finance
4	IL

Déclaration de sélection simple

`*` est le **caractère générique** utilisé pour sélectionner toutes les colonnes disponibles dans une table.

Utilisé comme substitut pour les noms de colonne explicites, il renvoie toutes les colonnes de toutes les tables qu'une requête sélectionne `FROM` . Cet effet s'applique à **toutes les tables** auxquelles la requête accède via ses clauses `JOIN` .

Considérez la requête suivante:

```
SELECT * FROM Employees
```

Il retournera tous les champs de toutes les lignes de la table `Employees` :

Id	FName	LName	DeptId
1	James	Forgeron	3
2	John	Johnson	4

Notation par points

Pour sélectionner toutes les valeurs d'une table spécifique, le caractère générique peut être appliqué à la table avec une *notation* par *points* .

Considérez la requête suivante:

```
SELECT
    Employees.*,
    Departments.Name
FROM
    Employees
JOIN
    Departments
    ON Departments.Id = Employees.DeptId
```

Cela renverra un ensemble de données avec tous les champs de la table `Employee` , suivi uniquement du champ `Name` dans la table `Departments` :

Id	FName	LName	DeptId	prénom
1	James	Forgeron	3	La finance
2	John	Johnson	4	IL

Avertissements contre l'utilisation

Il est généralement conseillé d'utiliser `*` si possible dans le code de production, car cela peut causer un certain nombre de problèmes potentiels, notamment:

1. Excès d'E / S, charge du réseau, utilisation de la mémoire, etc., car le moteur de base de données lit les données inutiles et les transmet au code frontal. Ceci est particulièrement préoccupant lorsque de grands champs peuvent être utilisés, tels que ceux utilisés pour stocker de longues notes ou des fichiers joints.
2. Le surplus d'E / S se charge si la base de données doit spouler les résultats internes sur le disque dans le cadre du traitement d'une requête plus complexe que `SELECT <columns> FROM <table>` .
3. Traitement supplémentaire (et / ou même plus IO) si certaines des colonnes inutiles sont:
    - colonnes calculées dans les bases de données qui les supportent
    - dans le cas d'une sélection dans une vue, les colonnes d'une table / vue que l'optimiseur de requête pourrait sinon optimiser
4. Potentiel d'erreurs inattendues si des colonnes sont ajoutées aux tables et aux vues ultérieurement, ce qui entraîne des noms de colonnes ambigus. Par exemple `SELECT * FROM orders JOIN people ON people.id = orders.personid ORDER BY displayname` - si une colonne appelée `displayname` est ajoutée à la table des commandes pour permettre aux utilisateurs de donner des noms significatifs à leurs commandes pour référence ultérieure, le nom de la colonne apparaîtra. deux fois dans la sortie de sorte que la clause `ORDER BY` soit ambiguë, ce qui peut provoquer des erreurs ("nom de colonne ambigu" dans les versions récentes de MS SQL Server) et si ce n'est pas le cas, le code de l'application prévu parce que la nouvelle colonne est le premier de ce nom renvoyé, et ainsi de suite.

Quand pouvez-vous utiliser `*` , en tenant compte de l'avertissement ci-dessus?

Bien qu'il soit préférable de l'éviter dans le code de production, l'utilisation de `*` convient

parfaitement pour effectuer des requêtes manuelles sur la base de données pour des travaux d'investigation ou de prototypage.

Parfois, les décisions de conception dans votre application le rendent inévitable (dans de telles circonstances, préférez les `tablealias.*` Plus que `*` si possible).

Lorsque vous utilisez `EXISTS` , tel que `SELECT A.col1, A.Col2 FROM A WHERE EXISTS (SELECT * FROM B where A.ID = B.A_ID)` , nous ne `SELECT A.col1, A.Col2 FROM A WHERE EXISTS (SELECT * FROM B where A.ID = B.A_ID)` aucune donnée de B. Ainsi, une jointure est inutile et le moteur sait qu'aucune valeur de B ne doit être retournée, donc pas de performance pour l'utilisation de `*` . De même, `COUNT(*)` est correct car il ne renvoie pas non plus de colonnes, il suffit donc de lire et de traiter celles qui sont utilisées à des fins de filtrage.

Sélection avec condition

La syntaxe de base de SELECT avec la clause WHERE est la suivante:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition]
```

La *[condition]* peut être toute expression SQL, spécifiée en utilisant des opérateurs de comparaison ou des opérateurs logiques tels que>, <, =, <>,> =, <=, LIKE, NOT, IN, BETWEEN etc.

L'instruction suivante renvoie toutes les colonnes de la table 'Cars' où la colonne status est 'READY':

```
SELECT * FROM Cars WHERE status = 'READY'
```

Voir WHERE et HAVING pour plus d'exemples.

Sélectionner des colonnes individuelles

```
SELECT
    PhoneNumber,
    Email,
    PreferredContact
FROM Customers
```

Cette instruction renvoie les colonnes `PhoneNumber` , `Email` et `PreferredContact` de toutes les lignes de la table `Customers` . Les colonnes seront également renvoyées dans l'ordre dans lequel elles apparaissent dans la clause `SELECT` .

Le résultat sera:

Numéro de téléphone	Email	Contact préféré
3347927472	william.jones@example.com	TÉLÉPHONE

Numéro de téléphone	Email	Contact préféré
2137921892	dmiller@example.net	EMAIL
NUL	richard0123@example.com	EMAIL

Si plusieurs tables sont jointes, vous pouvez sélectionner des colonnes à partir de tables spécifiques en spécifiant le nom de la table avant le nom de la colonne: `[table_name].[column_name]`

```
SELECT
    Customers.PhoneNumber,
    Customers.Email,
    Customers.PreferredContact,
    Orders.Id AS OrderId
FROM
    Customers
LEFT JOIN
    Orders ON Orders.CustomerId = Customers.Id
```

* `AS OrderId` signifie que le champ `Id` de la table `Orders` sera renvoyé sous la forme d'une colonne nommée `OrderId` . Voir la sélection avec l'alias de colonne pour plus d'informations.

Pour éviter d'utiliser des noms de table longs, vous pouvez utiliser des alias de table. Cela atténue la difficulté d'écrire des noms de table longs pour chaque champ que vous sélectionnez dans les jointures. Si vous effectuez une auto-jointure (une jointure entre deux instances de la *même* table), vous devez utiliser des alias de table pour distinguer vos tables. Nous pouvons écrire un alias de table comme `Customers c` ou `Customers AS c` . Ici, `c` fonctionne comme un alias pour les `Customers` et nous pouvons sélectionner, par exemple, `Email` comme ceci: `c.Email` .

```
SELECT
    c.PhoneNumber,
    c.Email,
    c.PreferredContact,
    o.Id AS OrderId
FROM
    Customers c
LEFT JOIN
    Orders o ON o.CustomerId = c.Id
```

SELECT utilisant des alias de colonne

Les alias de colonne sont principalement utilisés pour raccourcir le code et rendre les noms de colonne plus lisibles.

Le code devient plus court que les noms de table longs et l'identification inutile des colonnes *(par exemple, il peut y avoir 2 ID dans la table, mais une seule est utilisée dans l'instruction)* . En plus des alias de table, cela vous permet d'utiliser des noms descriptifs plus longs dans la structure de votre base de données tout en conservant des requêtes concises sur cette structure.

En outre, ils sont parfois *nécessaires* , par exemple dans les vues, pour nommer les sorties calculées.

Toutes les versions de SQL

Les alias peuvent être créés dans toutes les versions de SQL en utilisant des guillemets (`"`).

```
SELECT
    FName AS "First Name",
    MName AS "Middle Name",
    LName AS "Last Name"
FROM Employees
```

Différentes versions de SQL

Vous pouvez utiliser des guillemets simples (`'`), des guillemets doubles (`"`) et des crochets (`[]`) pour créer un alias dans Microsoft SQL Server.

```
SELECT
    FName AS "First Name",
    MName AS 'Middle Name',
    LName AS [Last Name]
FROM Employees
```

Les deux résulteront en:

Prénom	Deuxième nom	Nom de famille
James	John	Forgeron
John	James	Johnson
Michael	Marcus	Williams

Cette instruction `LName` colonnes `FName` et `LName` avec un nom donné (un alias). Ceci est réalisé en utilisant l'opérateur `AS` suivi de l'alias, ou en écrivant simplement un alias directement après le nom de la colonne. Cela signifie que la requête suivante a le même résultat que ci-dessus.

```
SELECT
    FName "First Name",
    MName "Middle Name",
    LName "Last Name"
FROM Employees
```

Prénom	Deuxième nom	Nom de famille
James	John	Forgeron
John	James	Johnson
Michael	Marcus	Williams

Cependant, la version explicite (c.-à-d., En utilisant l'opérateur `AS`) est plus lisible.

Si l'alias a un seul mot qui n'est pas un mot réservé, nous pouvons l'écrire sans guillemets simples, guillemets ou crochets:

```
SELECT
    FName AS FirstName,
    LName AS LastName
FROM Employees
```

Prénom	Nom de famille
James	Forgeron
John	Johnson
Michael	Williams

Une autre variante disponible dans MS SQL Server, entre autres, est `<alias> = <column-or-calculation>` , par exemple:

```
SELECT FullName = FirstName + ' ' + LastName,
       Addr1    = FullStreetAddress,
       Addr2    = TownName
FROM CustomerDetails
```

ce qui équivaut à:

```
SELECT FirstName + ' ' + LastName As FullName
       FullStreetAddress          As Addr1,
       TownName                   As Addr2
FROM CustomerDetails
```

Les deux résulteront en:

Nom complet	Addr1	Addr2
James Smith	123 AnyStreet	TownVille
John Johnson	668 MyRoad	Tout le monde
Michael Williams	999 Dr haut de gamme	Williamsburgh

Certains trouvent que l'utilisation de `=` plutôt que de `As` plus facile à lire, bien que beaucoup recommandent ce format, principalement parce qu'il n'est pas standard, donc pas largement pris en charge par toutes les bases de données. Cela peut entraîner une confusion avec d'autres utilisations du caractère `=` .

Toutes les versions de SQL

De plus, si vous *devez* utiliser des mots réservés, vous pouvez utiliser des parenthèses ou des guillemets pour échapper:

```
SELECT
    FName as "SELECT",
    MName as "FROM",
    LName as "WHERE"
FROM Employees
```

Différentes versions de SQL

De même, vous pouvez échapper des mots-clés dans MSSQL avec toutes les approches différentes:

```
SELECT
    FName AS "SELECT",
    MName AS 'FROM',
    LName AS [WHERE]
FROM Employees
```

SÉLECTIONNER	DE	OÙ
James	John	Forgeron
John	James	Johnson
Michael	Marcus	Williams

En outre, un alias de colonne peut être utilisé pour toutes les clauses finales de la même requête, telles que `ORDER BY` :

```
SELECT
    FName AS FirstName,
    LName AS LastName
FROM
    Employees
ORDER BY
    LastName DESC
```

Cependant, vous *ne* pouvez *pas* utiliser

```
SELECT
    FName AS SELECT,
    LName AS FROM
FROM
    Employees
ORDER BY
    LastName DESC
```

Pour créer un alias à partir de ces mots réservés (`SELECT` et `FROM`).

Cela provoquera de nombreuses erreurs d'exécution.

Sélection avec résultats triés

```
SELECT * FROM Employees ORDER BY LName
```

Cette instruction retournera toutes les colonnes de la table `Employees` .

Id	FName	LName	Numéro de téléphone
2	John	Johnson	2468101214
1	James	Forgeron	1234567890
3	Michael	Williams	1357911131

```
SELECT * FROM Employees ORDER BY LName DESC
```

Ou

```
 SELECT * FROM Employees ORDER BY LName ASC
```

Cette déclaration modifie le sens du tri.

On peut également spécifier plusieurs colonnes de tri. Par exemple:

```
SELECT * FROM Employees ORDER BY LName ASC, FName ASC
```

Cet exemple va trier les résultats d'abord par `LName` , puis, pour les enregistrements qui ont le même `LName` , trier par `FName` . Cela vous donnera un résultat similaire à ce que vous trouveriez dans un annuaire téléphonique.

Pour enregistrer le nouveau nom de la colonne dans la clause `ORDER BY` , il est possible d'utiliser le numéro de la colonne. Notez que les numéros de colonne commencent à 1.

```
SELECT Id, FName, LName, PhoneNumber FROM Employees ORDER BY 3
```

Vous pouvez également incorporer une instruction `CASE` dans la clause `ORDER BY` .

```
SELECT Id, FName, LName, PhoneNumber FROM Employees ORDER BY CASE WHEN LName='Jones` THEN 0
ELSE 1 END ASC
```

Cela va trier vos résultats pour avoir tous les enregistrements avec le `LName` de "Jones" en haut.

Sélectionner les colonnes nommées d'après les mots-clés réservés

Lorsqu'un nom de colonne correspond à un mot clé réservé, le SQL standard requiert que vous le

placiez entre guillemets doubles:

```
SELECT
    "ORDER",
    ID
FROM ORDERS
```

Notez que le nom de la colonne est sensible à la casse.

Certains SGBD ont des méthodes propriétaires pour citer des noms. Par exemple, SQL Server utilise des crochets à cet effet:

```
SELECT
    [Order],
    ID
FROM ORDERS
```

alors que MySQL (et MariaDB) utilisent par défaut des backticks:

```
SELECT
    `Order`,
    id
FROM orders
```

Sélection du nombre spécifié d'enregistrements

Le standard SQL 2008 définit la clause `FETCH FIRST` pour limiter le nombre d'enregistrements renvoyés.

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
FETCH FIRST 10 ROWS ONLY
```

Cette norme est uniquement prise en charge dans les versions récentes de certains RDMS. La syntaxe non standard spécifique au fournisseur est fournie dans d'autres systèmes. Progress OpenEdge 11.x prend également en charge la syntaxe `FETCH FIRST <n> ROWS ONLY` .

En outre, `OFFSET <m> ROWS` avant `FETCH FIRST <n> ROWS ONLY` permet d'ignorer les lignes avant d'extraire les lignes.

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
OFFSET 5 ROWS
FETCH FIRST 10 ROWS ONLY
```

La requête suivante est prise en charge dans SQL Server et MS Access:

```
SELECT TOP 10 Id, ProductName, UnitPrice, Package
FROM Product
```

```
ORDER BY UnitPrice DESC
```

Pour faire la même chose dans MySQL ou PostgreSQL, le mot-clé `LIMIT` doit être utilisé:

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
LIMIT 10
```

Dans Oracle, on peut faire la même chose avec `ROWNUM` :

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
WHERE ROWNUM <= 10
ORDER BY UnitPrice DESC
```

Résultats : **10** enregistrements.

```
Id    ProductName              UnitPrice            Package
38    Côte de Blaye            263.50               12 - 75 cl bottles
29    Thüringer Rostbratwurst  123.79               50 bags x 30 sausgs.
9    Mishi Kobe Niku           97.00                18 - 500 g pkgs.
20    Sir Rodney's Marmalade   81.00                30 gift boxes
18    Carnarvon Tigers         62.50                16 kg pkg.
59    Raclette Courdavault     55.00                5 kg pkg.
51    Manjimup Dried Apples    53.00                50 - 300 g pkgs.
62    Tarte au sucre           49.30                48 pies
43    Ipoh Coffee              46.00                16 - 500 g tins
28    Rössle Sauerkraut        45.60                25 - 825 g cans
```

Vendor Nuances:

Il est important de noter que le `TOP` de Microsoft SQL fonctionne après la clause `WHERE` et renverra le nombre de résultats spécifié s'il existe dans la table, tandis que `ROWNUM` fonctionne dans le cadre de la clause `WHERE` si d'autres conditions n'existent pas dans la table. nombre spécifié de lignes au début de la table, vous obtiendrez des résultats nuls s'il pouvait y en avoir d'autres à trouver.

Sélection avec alias de table

```
SELECT e.Fname, e.LName
FROM Employees e
```

La table Employees reçoit l'alias 'e' directement après le nom de la table. Cela permet d'éliminer l'ambiguïté dans les scénarios où plusieurs tables ont le même nom de champ et vous devez spécifier la table à partir de laquelle vous souhaitez renvoyer des données.

```
SELECT e.Fname, e.LName, m.Fname AS ManagerFirstName
FROM Employees e
    JOIN Managers m ON e.ManagerId = m.Id
```

Notez qu'une fois que vous avez défini un alias, vous ne pouvez plus utiliser le nom de la table

canonique. c'est à dire,

```
SELECT e.Fname, Employees.LName, m.Fname AS ManagerFirstName
FROM Employees e
JOIN Managers m ON e.ManagerId = m.Id
```

serait jeter une erreur.

Il convient de noter les alias de tables - plus formellement les «variables de plage» - ont été introduits dans le langage SQL pour résoudre le problème des colonnes en double provoquées par `INNER JOIN` . Le standard SQL de 1992 corrigeait ce défaut de conception en introduisant `NATURAL JOIN` (implémenté dans mySQL, PostgreSQL et Oracle mais pas encore dans SQL Server), dont le résultat ne comporte jamais de noms de colonnes en double. L'exemple ci-dessus est intéressant en ce sens que les tables sont jointes sur des colonnes avec des noms différents (`Id` et `ManagerId`) mais ne sont pas supposées être jointes sur les colonnes du même nom (`LName` , `FName`), nécessitant le changement de nom des colonnes *avant* l'adhésion:

```
SELECT Fname, LName, ManagerFirstName
FROM Employees
     NATURAL JOIN
     ( SELECT Id AS ManagerId, Fname AS ManagerFirstName
       FROM Managers ) m;
```

Notez que même si une variable alias / range doit être déclarée pour la table dervied (sinon SQL lancera une erreur), il n’est jamais logique de l’utiliser dans la requête.

Sélectionnez des lignes dans plusieurs tables

```
SELECT *
FROM
    table1,
    table2
```

```
SELECT
    table1.column1,
    table1.column2,
    table2.column1
FROM
    table1,
    table2
```

Ceci est appelé produit croisé dans SQL, il est identique à produit croisé dans les ensembles

Ces instructions renvoient les colonnes sélectionnées de plusieurs tables dans une requête.

Il n'y a pas de relation spécifique entre les colonnes renvoyées par chaque table.

Sélection avec les fonctions d'agrégat

Moyenne

La fonction d'agrégation `AVG()` renvoie la moyenne des valeurs sélectionnées.

```
SELECT AVG(Salary) FROM Employees
```

Les fonctions d'agrégat peuvent également être combinées avec la clause where.

```
SELECT AVG(Salary) FROM Employees where DepartmentId = 1
```

Les fonctions d'agrégat peuvent également être combinées avec une clause group by.

Si l'employé est classé avec plusieurs départements et que nous voulons trouver un salaire moyen pour chaque service, nous pouvons utiliser la requête suivante.

```
SELECT AVG(Salary) FROM Employees GROUP BY DepartmentId
```

Le minimum

La fonction d'agrégation `MIN()` renvoie le minimum de valeurs sélectionnées.

```
SELECT MIN(Salary) FROM Employees
```

Maximum

La fonction d'agrégation `MAX()` renvoie le maximum de valeurs sélectionnées.

```
SELECT MAX(Salary) FROM Employees
```

Compter

La fonction d'agrégation `COUNT()` renvoie le nombre de valeurs sélectionnées.

```
SELECT Count(*) FROM Employees
```

Il peut également être combiné avec les conditions pour obtenir le nombre de lignes satisfaisant des conditions spécifiques.

```
SELECT Count(*) FROM Employees where ManagerId IS NOT NULL
```

Des colonnes spécifiques peuvent également être spécifiées pour obtenir le nombre de valeurs dans la colonne. Notez que les valeurs `NULL` ne sont pas comptabilisées.

```
Select Count(ManagerId) from Employees
```

Count peut également être combiné avec le mot-clé distinct pour un compte distinct.

```
Select Count(DISTINCT DepartmentId) from Employees
```

Somme

La fonction d'agrégation `SUM()` renvoie la somme des valeurs sélectionnées pour toutes les lignes.

```
SELECT SUM(Salary) FROM Employees
```

Sélection avec null

```
SELECT Name FROM Customers WHERE PhoneNumber IS NULL
```

La sélection avec des valeurs NULL a une syntaxe différente. N'utilisez pas `=` , utilisez `IS NULL` ou `IS NOT NULL` place.

Sélection avec CASE

Lorsque les résultats doivent avoir une logique appliquée «à la volée», on peut utiliser l'instruction CASE pour l'implémenter.

```
SELECT CASE WHEN Col1 < 50 THEN 'under' ELSE 'over' END threshold
FROM TableName
```

peut également être enchaîné

```
SELECT
    CASE WHEN Col1 < 50 THEN 'under'
         WHEN Col1 > 50 AND Col1 <100 THEN 'between'
         ELSE 'over'
    END threshold
FROM TableName
```

on peut aussi avoir `CASE` dans une autre déclaration `CASE`

```
SELECT
    CASE WHEN Col1 < 50 THEN 'under'
         ELSE
            CASE WHEN Col1 > 50 AND Col1 <100 THEN Col1
            ELSE 'over' END
    END threshold
FROM TableName
```

Sélectionner sans verrouiller la table

Parfois, lorsque les tables sont utilisées principalement (ou uniquement) pour les lectures, l'indexation n'aide plus et chaque bit compte, on peut utiliser selects sans LOCK pour améliorer les performances.

---

serveur SQL

```
SELECT * FROM TableName WITH (nolock)
```

MySQL

```
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM TableName;
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Oracle

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM TableName;
```

DB2

```
SELECT * FROM TableName WITH UR;
```

où `UR` signifie "lecture non validée".

S'il est utilisé sur une table dont les modifications d'enregistrement sont en cours, les résultats risquent d'être imprévisibles.

Sélectionnez distinct (valeurs uniques uniquement)

```
SELECT DISTINCT ContinentCode
FROM Countries;
```

Cette requête renvoie toutes les valeurs `DISTINCT` (uniques, différentes) de `ContinentCode` colonne `ContinentCode` du tableau `Countries`

ContinentCode
OC
UE
COMME
N / A

ContinentCode
UN F

Démo SQLFiddle

Sélectionnez avec la condition de plusieurs valeurs de la colonne

```
SELECT * FROM Cars  WHERE status IN ( 'Waiting', 'Working' )
```

Ceci est sémantiquement équivalent à

```
SELECT * FROM Cars  WHERE ( status = 'Waiting' OR status = 'Working' )
```

la `value IN ( <value list> )` est un raccourci pour la disjonction (`OR` logique).

Obtenir un résultat agrégé pour les groupes de lignes

Comptage des lignes en fonction d'une valeur de colonne spécifique:

```
SELECT category, COUNT(*) AS item_count
FROM item
GROUP BY category;
```

Obtenir un revenu moyen par département:

```
SELECT department, AVG(income)
FROM employees
GROUP BY department;
```

L'important est de ne sélectionner que les colonnes spécifiées dans la clause `GROUP BY` ou utilisées avec les fonctions d'agrégat .

La clause `WHERE` peut également être utilisée avec `GROUP BY` , mais `WHERE` filtre les enregistrements *avant* tout regroupement:

```
SELECT department, AVG(income)
FROM employees
WHERE department <> 'ACCOUNTING'
GROUP BY department;
```

Si vous devez filtrer les résultats après le regroupement, par exemple pour ne voir que les départements dont le revenu moyen est supérieur à 1000, vous devez utiliser la clause `HAVING` :

```
SELECT department, AVG(income)
FROM employees
WHERE department <> 'ACCOUNTING'
GROUP BY department
HAVING avg(income) > 1000;
```

Sélection avec plus d'une condition

Le mot clé `AND` est utilisé pour ajouter plus de conditions à la requête.

prénom	Âge	Le genre
Sam	18	M
John	21	M
Bob	22	M
Marie	23	F

```
SELECT name FROM persons WHERE gender = 'M' AND age > 20;
```

Cela va retourner:

prénom
John
Bob

en utilisant le mot clé `OR`

```
SELECT name FROM persons WHERE gender = 'M' OR age < 20;
```

Cela va retourner:

prénom
Sam
John
Bob

Ces mots-clés peuvent être combinés pour permettre des combinaisons de critères plus complexes:

```
SELECT name
FROM persons
WHERE (gender = 'M' AND age < 20)
   OR (gender = 'F' AND age > 20);
```

Cela va retourner:

prénom
Sam
Marie

Lire SÉLECTIONNER en ligne: https://riptutorial.com/fr/sql/topic/222/selectionner

Chapitre 55: Séquence

Examples

Créer une séquence

```
CREATE SEQUENCE orders_seq
START WITH     1000
INCREMENT BY   1;
```

Crée une séquence avec une valeur de départ de 1000 qui est incrémentée de 1.

Utiliser des séquences

une référence à *nom_seq* .NEXTVAL est utilisée pour obtenir la valeur suivante dans une séquence. Une seule instruction ne peut générer qu'une seule valeur de séquence. S'il existe plusieurs références à NEXTVAL dans une instruction, elles utiliseront le même numéro généré.

NEXTVAL peut être utilisé pour INSERTS

```
INSERT INTO Orders (Order_UID, Customer)
        VALUES (orders_seq.NEXTVAL, 1032);
```

Il peut être utilisé pour les mises à jour

```
UPDATE Orders
SET Order_UID = orders_seq.NEXTVAL
WHERE Customer = 581;
```

Il peut également être utilisé pour SELECTS

```
SELECT Order_seq.NEXTVAL FROM dual;
```

Lire Séquence en ligne: https://riptutorial.com/fr/sql/topic/1586/sequence

Chapitre 56: Sous-requêtes

Remarques

Les sous-requêtes peuvent apparaître dans différentes clauses d'une requête externe ou dans l'opération définie.

Ils doivent être entre parenthèses `()` . Si le résultat de la sous-requête est comparé à autre chose, le nombre de colonnes doit correspondre. Les alias de table sont requis pour les sous-requêtes dans la clause FROM pour nommer la table temporaire.

Examples

Sous-requête dans la clause WHERE

Utilisez une sous-requête pour filtrer le jeu de résultats. Par exemple, tous les employés recevront un salaire égal au salaire le plus élevé.

```
SELECT *
FROM Employees
WHERE Salary = (SELECT MAX(Salary) FROM Employees)
```

Sous-requête dans la clause FROM

Une sous-requête dans une clause `FROM` agit de manière similaire à une table temporaire générée lors de l'exécution d'une requête et perdue par la suite.

```
SELECT Managers.Id, Employees.Salary
FROM (
  SELECT Id
  FROM Employees
  WHERE ManagerId IS NULL
) AS Managers
JOIN Employees ON Managers.Id = Employees.Id
```

Sous-requête dans la clause SELECT

```
SELECT
  Id,
  FName,
  LName,
  (SELECT COUNT(*) FROM Cars WHERE Cars.CustomerId = Customers.Id) AS NumberOfCars
FROM Customers
```

Sous-requêtes dans la clause FROM

Vous pouvez utiliser des sous-requêtes pour définir une table temporaire et l'utiliser dans la clause

FROM d'une requête "externe".

```
SELECT * FROM (SELECT city, temp_hi - temp_lo AS temp_var FROM weather) AS w
WHERE temp_var > 20;
```

Le ci-dessus trouve des villes de la table météo dont la variation de température quotidienne est supérieure à 20. Le résultat est:

ville	temp_var
SAINT LOUIS	21
LOS ANGELES	31
LOS ANGELES	23
LOS ANGELES	31
LOS ANGELES	27
LOS ANGELES	28
LOS ANGELES	28
LOS ANGELES	32

.

Sous-requêtes dans la clause WHERE

L'exemple suivant trouve des villes (exemple des villes) dont la population est inférieure à la température moyenne (obtenue via une sous-requête):

```
SELECT name, pop2000 FROM cities
WHERE pop2000 < (SELECT avg(pop2000)  FROM cities);
```

Ici: la sous-requête (SELECT avg (pop2000) FROM cities) est utilisée pour spécifier des conditions dans la clause WHERE. Le résultat est:

prénom	pop2000
San Francisco	776733
SAINT LOUIS	348189
Kansas City	146866

Sous-requêtes dans la clause SELECT

Les sous-requêtes peuvent également être utilisées dans la partie `SELECT` de la requête externe. La requête suivante affiche toutes les colonnes de la table météo avec les états correspondants de la table cities .

```
SELECT w.*,  (SELECT c.state FROM cities AS c WHERE c.name = w.city ) AS state
FROM weather AS w;
```

Filtrer les résultats de la requête à l'aide d'une requête sur une table différente

Cette requête sélectionne tous les employés qui ne figurent pas dans la table Supervisors.

```
SELECT *
FROM Employees
WHERE EmployeeID not in (SELECT EmployeeID
                         FROM Supervisors)
```

Les mêmes résultats peuvent être obtenus en utilisant un LEFT JOIN.

```
SELECT *
FROM Employees AS e
LEFT JOIN Supervisors AS s ON s.EmployeeID=e.EmployeeID
WHERE s.EmployeeID is NULL
```

Sous-requêtes corrélées

Les sous-requêtes corrélées (également appelées synchronisées ou coordonnées) sont des requêtes imbriquées qui font référence à la ligne actuelle de leur requête externe:

```
SELECT EmployeeId
    FROM Employee AS eOuter
    WHERE Salary > (
       SELECT AVG(Salary)
       FROM Employee eInner
       WHERE eInner.DepartmentId = eOuter.DepartmentId
    )
```

La sous-requête `SELECT AVG(Salary) ...` est *corrélée* car elle fait référence à la ligne de l' `Employee eOuter` de sa requête externe.

Lire Sous-requêtes en ligne: https://riptutorial.com/fr/sql/topic/1606/sous-requetes

Chapitre 57: SQL Group By vs Distinct

Examples

Différence entre GROUP BY et DISTINCT

`GROUP BY` est utilisé en combinaison avec les fonctions d'agrégation. Considérons le tableau suivant:

numéro de commande	identifiant d'utilisateur	nom du magasin	valeur de la commande	date de commande
1	43	Magasin a	25	20-03-2016
2	57	Magasin B	50	22-03-2016
3	43	Magasin a	30	25-03-2016
4	82	Magasin C	dix	26-03-2016
5	21	Magasin a	45	29-03-2016

La requête ci-dessous utilise `GROUP BY` pour effectuer des calculs agrégés.

```
SELECT
    storeName,
    COUNT(*) AS total_nr_orders,
    COUNT(DISTINCT userId) AS nr_unique_customers,
    AVG(orderValue) AS average_order_value,
    MIN(orderDate) AS first_order,
    MAX(orderDate) AS lastOrder
FROM
    orders
GROUP BY
    storeName;
```

et renverra les informations suivantes

nom du magasin	total_nr_orders	nr_unique_customers	average_order_value	Premier ordre	dernière commande
Magasin a	3	2	33,3	20-03-2016	29-03-2016
Magasin B	1	1	50	22-03-2016	22-03-2016
Magasin	1	1	dix	26-03-	26-03-2016

nom du magasin	total_nr_orders	nr_unique_customers	average_order_value	Premier ordre	dernière commande
C				2016	

Alors que `DISTINCT` est utilisé pour répertorier une combinaison unique de valeurs distinctes pour les colonnes spécifiées.

```
SELECT DISTINCT
    storeName,
    userId
FROM
    orders;
```

nom du magasin	identifiant d'utilisateur
Magasin a	43
Magasin B	57
Magasin C	82
Magasin a	21

Lire SQL Group By vs Distinct en ligne: https://riptutorial.com/fr/sql/topic/2499/sql-group-by-vs-distinct

Chapitre 58: Supprimer en cascade

Examples

ON DELETE CASCADE

Supposons que vous ayez une application qui administre des salles.
Supposons en outre que votre application fonctionne par client (locataire).
Vous avez plusieurs clients.
Votre base de données contiendra donc une table pour les clients et une pour les salles.

Maintenant, chaque client a N chambres.

Cela devrait signifier que vous avez une clé étrangère sur votre table de salle, référençant la table client.

```
ALTER TABLE dbo.T_Room  WITH CHECK ADD  CONSTRAINT FK_T_Room_T_Client FOREIGN KEY(RM_CLI_ID)
REFERENCES dbo.T_Client (CLI_ID)
GO
```

En supposant qu'un client passe à un autre logiciel, vous devrez supprimer ses données dans votre logiciel. Mais si tu le fais

```
DELETE FROM T_Client WHERE CLI_ID = x
```

Ensuite, vous obtenez une violation de clé étrangère, car vous ne pouvez pas supprimer le client lorsqu'il dispose encore de salles.

Désormais, votre application contient du code d'écriture qui supprime les pièces du client avant de supprimer le client. Supposons encore qu'à l'avenir, de nombreuses dépendances de clés étrangères seront ajoutées à votre base de données, car les fonctionnalités de votre application se développent. Horrible. Pour toute modification de votre base de données, vous devrez adapter le code de votre application à N endroits. Vous devrez peut-être également adapter le code dans d'autres applications (par exemple, interfaces avec d'autres systèmes).

Il y a une meilleure solution que de le faire dans votre code.
Vous pouvez simplement ajouter `ON DELETE CASCADE` à votre clé étrangère.

```
ALTER TABLE dbo.T_Room  -- WITH CHECK -- SQL-Server can specify WITH CHECK/WITH NOCHECK
ADD  CONSTRAINT FK_T_Room_T_Client FOREIGN KEY(RM_CLI_ID)
REFERENCES dbo.T_Client (CLI_ID)
ON DELETE CASCADE
```

Maintenant vous pouvez dire

```
DELETE FROM T_Client WHERE CLI_ID = x
```

et les pièces sont automatiquement supprimées lorsque le client est supprimé.
Problème résolu - sans modification du code d'application.

Un mot d'avertissement: dans Microsoft SQL-Server, cela ne fonctionnera pas si vous avez une table qui fait référence à elle-même. Donc, si vous essayez de définir une cascade de suppression sur une arborescence récursive, comme ceci:

```
IF NOT EXISTS (SELECT * FROM sys.foreign_keys WHERE object_id =
OBJECT_ID(N'[dbo].[FK_T_FMS_Navigation_T_FMS_Navigation]') AND parent_object_id =
OBJECT_ID(N'[dbo].[T_FMS_Navigation]'))
ALTER TABLE [dbo].[T_FMS_Navigation]  WITH CHECK ADD  CONSTRAINT
[FK_T_FMS_Navigation_T_FMS_Navigation] FOREIGN KEY([NA_NA_UID])
REFERENCES [dbo].[T_FMS_Navigation] ([NA_UID])
ON DELETE CASCADE
GO

IF  EXISTS (SELECT * FROM sys.foreign_keys WHERE object_id =
OBJECT_ID(N'[dbo].[FK_T_FMS_Navigation_T_FMS_Navigation]') AND parent_object_id =
OBJECT_ID(N'[dbo].[T_FMS_Navigation]'))
ALTER TABLE [dbo].[T_FMS_Navigation] CHECK CONSTRAINT [FK_T_FMS_Navigation_T_FMS_Navigation]
GO
```

cela ne fonctionnera pas, car Microsoft-SQL-server ne vous permet pas de définir une clé étrangère avec `ON DELETE CASCADE` sur une arborescence récursive. Une des raisons à cela est que l'arbre est probablement cyclique et que cela pourrait conduire à une impasse.

PostgreSQL peut en revanche faire cela;
la condition est que l'arbre soit non cyclique.
Si l'arbre est cyclique, vous obtenez une erreur d'exécution.
Dans ce cas, vous devrez simplement implémenter la fonction de suppression vous-même.

Un mot d'avertissement:
Cela signifie que vous ne pouvez plus simplement supprimer et réinsérer la table client, car si vous faites cela, elle supprimera toutes les entrées dans "T_Room" ... (plus de mises à jour non-delta)

Lire Supprimer en cascade en ligne: https://riptutorial.com/fr/sql/topic/3518/supprimer-en-cascade

Chapitre 59: Transactions

Remarques

Une transaction est une unité de travail logique contenant une ou plusieurs étapes, chacune devant aboutir pour que la transaction soit validée dans la base de données. S'il y a des erreurs, toutes les modifications de données sont effacées et la base de données est restaurée à son état initial au début de la transaction.

Examples

Transaction simple

```
BEGIN TRANSACTION
    INSERT INTO DeletedEmployees(EmployeeID, DateDeleted, User)
        (SELECT 123, GetDate(), CURRENT_USER);
    DELETE FROM Employees WHERE EmployeeID = 123;
COMMIT TRANSACTION
```

Transaction d'annulation

Lorsque quelque chose échoue dans votre code de transaction et que vous souhaitez l'annuler, vous pouvez annuler votre transaction:

```
BEGIN TRY
    BEGIN TRANSACTION
        INSERT INTO Users(ID, Name, Age)
        VALUES(1, 'Bob', 24)

        DELETE FROM Users WHERE Name = 'Todd'
   COMMIT TRANSACTION
END TRY
BEGIN CATCH
   ROLLBACK TRANSACTION
END CATCH
```

Lire Transactions en ligne: https://riptutorial.com/fr/sql/topic/2424/transactions

Chapitre 60: TRONQUER

Introduction

L'instruction TRUNCATE supprime toutes les données d'une table. Ceci est similaire à DELETE sans filtre, mais, en fonction du logiciel de base de données, présente certaines restrictions et optimisations.

Syntaxe

- TRUNCATE TABLE nom_table;

Remarques

TRUNCATE est une commande DDL (Data Definition Language), et il existe donc des différences significatives entre cette commande et DELETE (un langage de manipulation de données, DML, commande). Bien que TRUNCATE puisse être un moyen de supprimer rapidement de gros volumes d'enregistrements d'une base de données, ces différences doivent être comprises afin de décider si l'utilisation d'une commande TRUNCATE convient à votre situation particulière.

- TRUNCATE est une opération de page de données. Par conséquent, les déclencheurs DML (ON DELETE) associés à la table ne se déclenchent pas lorsque vous effectuez une opération TRUNCATE. Bien que cela économise beaucoup de temps pour des opérations de suppression massives, vous devrez peut-être alors supprimer manuellement les données associées.
- TRUNCATE libère l'espace disque utilisé par les lignes supprimées, DELETE libère de l'espace
- Si la table à tronquer utilise des colonnes d'identité (MS SQL Server), la graine est réinitialisée par la commande TRUNCATE. Cela peut entraîner des problèmes d’intégrité référentielle
- Selon les rôles de sécurité en place et la variante de SQL utilisée, il se peut que vous ne disposiez pas des autorisations nécessaires pour exécuter une commande TRUNCATE.

Examples

Supprimer toutes les lignes de la table Employee

```
TRUNCATE TABLE Employee;
```

L'utilisation de la table tronquée est souvent préférable à l'utilisation de DELETE TABLE car elle ignore tous les index et tous les déclencheurs et supprime tout.

Supprimer la table est une opération basée sur des lignes, cela signifie que chaque ligne est supprimée. Truncate table est une opération de page de données sur laquelle toute la page de

données est réallouée. Si vous avez une table avec un million de lignes, il sera beaucoup plus rapide de tronquer la table que d'utiliser une instruction delete table.

Bien que nous puissions supprimer des lignes spécifiques avec DELETE, nous ne pouvons pas TRUNCATE de lignes spécifiques, nous ne pouvons que TRUNCATE tous les enregistrements à la fois. La suppression de toutes les lignes et l'insertion d'un nouvel enregistrement continueront d'ajouter la valeur de la clé primaire incrémentée automatiquement à partir de la valeur précédemment insérée.

Notez que lorsque vous tronquez une table, **aucune clé étrangère ne doit être présente** , sinon vous obtiendrez une erreur.

Lire TRONQUER en ligne: https://riptutorial.com/fr/sql/topic/1466/tronquer

Chapitre 61: Types de données

Examples

DECIMAL et NUMERIC

Correction des nombres décimaux de précision et d'échelle. `DECIMAL` et `NUMERIC` sont fonctionnellement équivalents.

Syntaxe:

```
DECIMAL ( precision [ , scale] )
NUMERIC ( precision [ , scale] )
```

Exemples:

```
SELECT CAST(123 AS DECIMAL(5,2)) --returns 123.00
SELECT CAST(12345.12 AS NUMERIC(10,5)) --returns 12345.12000
```

FLOAT et REAL

Types de données à nombre approximatif à utiliser avec des données numériques à virgule flottante.

```
SELECT CAST( PI() AS FLOAT) --returns 3.14159265358979
SELECT CAST( PI() AS REAL) --returns 3.141593
```

Entiers

Types de données à nombre exact qui utilisent des données entières.

Type de données	Gamme	Espace de rangement
bigint	-2 ^ 63 (-9 223 372 036 854 775 808) à 2 ^ 63-1 (9 223 372 036 854 775 807)	8 octets
int	-2 ^ 31 (-2 147 443 648) à 2 ^ 31-1 (2 147 483 647)	4 octets
smallint	-2 ^ 15 (-32 768) à 2 ^ 15-1 (32 767)	2 octets
tinyint	0 à 255	1 octet

L'argent et le petit argent

Types de données représentant des valeurs monétaires ou monétaires.

Type de données	Gamme	Espace de rangement
argent	-922 337 203 685 477,5808 à 922 337 203 685 477,5807	8 octets
petit argent	-214.748.3648 à 214.748.3647	4 octets

BINARY et VARBINARY

Types de données binaires de longueur fixe ou variable.

Syntaxe:

```
BINARY [ ( n_bytes ) ]
VARBINARY [ ( n_bytes | max ) ]
```

`n_bytes` peut être un nombre compris entre 1 et 8 000 octets. `max` indique que l'espace de stockage maximal est 2 ^ 31-1.

Exemples:

```
SELECT CAST(12345 AS BINARY(10)) -- 0x0000000000000003039
SELECT CAST(12345 AS VARBINARY(10)) -- 0x00003039
```

CHAR et VARCHAR

Types de données de chaîne de longueur fixe ou variable.

Syntaxe:

```
CHAR [ ( n_chars ) ]
VARCHAR [ ( n_chars ) ]
```

Exemples:

```
SELECT CAST('ABC' AS CHAR(10)) -- 'ABC       ' (padded with spaces on the right)
SELECT CAST('ABC' AS VARCHAR(10)) -- 'ABC' (no padding due to variable character)
SELECT CAST('ABCDEFGHIJKLMNOPQRSTUVWXYZ' AS CHAR(10))  -- 'ABCDEFGHIJ' (truncated to 10
characters)
```

NCHAR et NVARCHAR

Types de données de chaîne UNICODE de longueur fixe ou variable.

Syntaxe:

```
NCHAR [ ( n_chars ) ]
NVARCHAR [ ( n_chars | MAX ) ]
```

Utilisez `MAX` pour de très longues chaînes pouvant dépasser 8 000 caractères.

IDENTIFIANT UNIQUE

Un GUID / UUID de 16 octets.

```
DECLARE @GUID UNIQUEIDENTIFIER = NEWID();
SELECT @GUID -- 'E28B3BD9-9174-41A9-8508-899A78A33540'
DECLARE @bad_GUID_string VARCHAR(100) = 'E28B3BD9-9174-41A9-8508-899A78A33540_foobarbaz'
SELECT
    @bad_GUID_string,   -- 'E28B3BD9-9174-41A9-8508-899A78A33540_foobarbaz'
    CONVERT(UNIQUEIDENTIFIER, @bad_GUID_string) -- 'E28B3BD9-9174-41A9-8508-899A78A33540'
```

Lire Types de données en ligne: https://riptutorial.com/fr/sql/topic/1166/types-de-donnees

Chapitre 62: UNION / UNION ALL

Introduction

Le mot-clé ***UNION*** dans SQL est utilisé pour combiner les résultats de l'instruction ***SELECT*** avec aucun duplicata. Pour utiliser UNION et combiner les résultats, les deux instructions SELECT doivent avoir le même nombre de colonnes avec le même type de données dans le même ordre, mais la longueur de la colonne peut être différente.

Syntaxe

- SELECT column_1 [, column_2] FROM table_1 [, table_2] [WHERE condition]
  UNION | UNION ALL
  SELECT column_1 [, column_2] FROM table_1 [, table_2] [WHERE condition]

Remarques

`UNION` clauses `UNION` et `UNION ALL` combinent l'ensemble de résultats de deux instructions SELECT structurées de manière identique en un seul résultat.

Le nombre de colonnes et les types de colonne pour chaque requête doivent correspondre pour qu'un `UNION` / `UNION ALL` fonctionne.

La différence entre une requête `UNION` et une requête `UNION ALL` est que la clause `UNION` supprime toutes les lignes en double dans le résultat, contrairement à la clause `UNION ALL` .

Cette suppression distincte des enregistrements peut considérablement ralentir les requêtes, même si aucune ligne distincte ne doit être supprimée. Par conséquent, si vous savez qu'il n'y aura pas de doublons (ou ne vous inquiétez pas), `UNION ALL` toujours optimisée.

Examples

Requête de base UNION ALL

```
CREATE TABLE HR_EMPLOYEES
(
    PersonID int,
    LastName VARCHAR(30),
    FirstName VARCHAR(30),
    Position VARCHAR(30)
);

CREATE TABLE FINANCE_EMPLOYEES
(
    PersonID INT,
    LastName VARCHAR(30),
    FirstName VARCHAR(30),
```

```
    Position VARCHAR(30)
);
```

Disons que nous voulons extraire les noms de tous les `managers` de nos départements.

En utilisant un `UNION` nous pouvons obtenir tous les employés des départements des ressources humaines et des finances, qui occupent le `position` de `manager`

```
SELECT
    FirstName, LastName
FROM
    HR_EMPLOYEES
WHERE
    Position = 'manager'
UNION ALL
SELECT
    FirstName, LastName
FROM
    FINANCE_EMPLOYEES
WHERE
    Position = 'manager'
```

L'instruction `UNION` supprime les lignes dupliquées des résultats de la requête. Comme il est possible d'avoir des personnes ayant le même nom et la même position dans les deux départements, nous utilisons `UNION ALL` afin de ne pas supprimer les doublons.

Si vous souhaitez utiliser un alias pour chaque colonne de sortie, vous pouvez simplement les placer dans la première instruction select, comme suit:

```
SELECT
    FirstName as 'First Name', LastName as 'Last Name'
FROM
    HR_EMPLOYEES
WHERE
    Position = 'manager'
UNION ALL
SELECT
    FirstName, LastName
FROM
    FINANCE_EMPLOYEES
WHERE
    Position = 'manager'
```

Explication simple et exemple

En termes simples:

- `UNION` joint 2 jeux de résultats tout en supprimant les doublons du jeu de résultats
- `UNION ALL` joint 2 jeux de résultats sans tenter de supprimer les doublons

  Une erreur commise par de nombreuses personnes consiste à utiliser un `UNION` sans avoir à supprimer les doublons. Le coût de performance supplémentaire par rapport aux ensembles de résultats importants peut être très important.

Quand vous avez besoin d' `UNION`

Supposons que vous deviez filtrer une table avec 2 attributs différents et que vous ayez créé des index non clusterisés distincts pour chaque colonne. Un `UNION` vous permet de tirer parti des deux index tout en empêchant les doublons.

```
SELECT C1, C2, C3 FROM Table1 WHERE C1 = @Param1
UNION
SELECT C1, C2, C3 FROM Table1 WHERE C2 = @Param2
```

Cela simplifie le réglage des performances car seuls des index simples sont nécessaires pour effectuer ces requêtes de manière optimale. Vous pourriez même être en mesure de vous débrouiller avec beaucoup moins d'index non clusterisés améliorant également les performances d'écriture globales par rapport à la table source.

Quand vous pourriez avoir besoin de `UNION ALL`

Supposons que vous ayez toujours besoin de filtrer une table avec 2 attributs, mais que vous n'avez pas besoin de filtrer les enregistrements en double (soit parce que vos données ne produiraient aucun doublon au cours de l'union).

```
SELECT C1 FROM Table1
UNION ALL
SELECT C1 FROM Table2
```

Cela est particulièrement utile lors de la création de vues qui associent des données conçues pour être partitionnées physiquement sur plusieurs tables (peut-être pour des raisons de performances, mais qui veulent toujours regrouper des enregistrements). Comme les données sont déjà divisées, le fait de supprimer les doublons du moteur de base de données n'ajoute aucune valeur et ajoute simplement du temps de traitement supplémentaire aux requêtes.

Lire UNION / UNION ALL en ligne: https://riptutorial.com/fr/sql/topic/349/union---union-all

Chapitre 63: Vues matérialisées

Introduction

Une vue matérialisée est une vue dont les résultats sont stockés physiquement et doivent être régulièrement actualisés afin de rester à jour. Ils sont donc utiles pour stocker les résultats de requêtes complexes de longue durée lorsque des résultats en temps réel ne sont pas requis. Des vues matérialisées peuvent être créées dans Oracle et PostgreSQL. D'autres systèmes de base de données offrent des fonctionnalités similaires, telles que les vues indexées de SQL Server ou les tables de requêtes matérialisées de DB2.

Examples

Exemple PostgreSQL

```
CREATE TABLE mytable (number INT);
INSERT INTO mytable VALUES (1);

CREATE MATERIALIZED VIEW myview AS SELECT * FROM mytable;

SELECT * FROM myview;
 number
--------
      1
(1 row)

INSERT INTO mytable VALUES(2);

SELECT * FROM myview;
 number
--------
      1
(1 row)

REFRESH MATERIALIZED VIEW myview;

SELECT * FROM myview;
 number
--------
      1
      2
(2 rows)
```

Lire Vues matérialisées en ligne: https://riptutorial.com/fr/sql/topic/8367/vues-materialisees

Chapitre 64: XML

Examples

Requête à partir du type de données XML

```
DECLARE @xmlIN XML = '<TableData>
<aaa Main="First">
  <row name="a" value="1" />
  <row name="b" value="2" />
  <row name="c" value="3" />
</aaa>
<aaa Main="Second">
  <row name="a" value="3" />
  <row name="b" value="4" />
  <row name="c" value="5" />
</aaa>
<aaa Main="Third">
  <row name="a" value="10" />
  <row name="b" value="20" />
  <row name="c" value="30" />
</aaa>
</TableData>'

SELECT t.col.value('../@Main', 'varchar(10)') [Header],
t.col.value('@name', 'VARCHAR(25)') [name],
t.col.value('@value',  'VARCHAR(25)') [Value]
FROM    @xmlIn.nodes('//TableData/aaa/row') AS t (col)
```

Résultats

```
Header    name    Value
First      a        1
First      b        2
First      c        3
Second     a        3
Second     b        4
Second     c        5
Third      a        10
Third      b        20
Third      c        30
```

Lire XML en ligne: https://riptutorial.com/fr/sql/topic/4421/xml

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec SQL	Arjan Einbu, brichins, Burkhard, cale_b, CL., Community, Devmati Wadikar, Epodax, geeksal, H. Pauwelyn, Hari, Joey, JohnLBevan, Jon Ericson, Lankymart, Laurel, Mureinik, Nathan, omini data, PeterRing, Phrancis, Prateek, RamenChef, Ray, Simone Carletti, SZenC, t1gor, ypercube
2	Algèbre relationnelle	CL., Darren Bartrup-Cook, Martin Smith
3	ALTER TABLE	Aidan, blackbishop, bluefeet, CL., Florin Ghita, Francis Lord, guiguiblitz, Joe W, KIRAN KUMAR MATAM, Lexi, mithra chintha , Ozair Kafray, Simon Foster, Siva Rama Krishna
4	application croisée, application extérieure	Karthikeyan, RamenChef
5	Base de données DROP ou DELETE	Abhilash R Vankayala, John Odom
6	Blocs d'exécution	Phrancis
7	CAS	ɐlǝx, Christos, CL., Dariusz, Fenton, Infinity, Jaydles, Matt, MotKohn, Mureinik, Peter Lang, Stanislovas Kalašnikovas
8	CLAUSE EXISTE	Blag, Özgür Öztürk
9	Clause IN	CL., juergen d, walid, Zaga
10	Clés étrangères	CL., Harjot, Yehuda Shapira
11	Clés primaires	Andrea Montanari, CL., FlyingPiMonster, KjetilNordin
12	COMMANDÉ PAR	Andi Mohr, CL., Cristian Abelleira, Jaydles, mithra chintha, nazark, Özgür Öztürk, Parado, Phrancis, Wolfgang
13	commentaires	CL., Phrancis
14	CREATE Base de données	Emil Rowland
15	CREER LA TABLE	Aidan, alex9311, Almir Vuk, Ares, CL., drunken_monkey, Dylan Vander Berg, Franck Dernoncourt, H. Pauwelyn, Jojodmo, KIRAN KUMAR MATAM, Matas Vaitkevicius, Prateek

16	CREER UNE FONCTION	John Odom, Ricardo Pontual
17	CURSEUR SQL	Stefan Steiger
18	Déclencheurs	Daryl, IncrediApp
19	Des synonymes	Daryl
20	Des vues	Amir978, CL., Florin Ghita
21	Design de table	Darren Bartrup-Cook
22	DROP Table	CL., Joel, KIRAN KUMAR MATAM, Stu
23	EFFACER	Batsu, Chip, CL., Dylan Vander Berg, fredden, Joel, KIRAN KUMAR MATAM, Phrancis, Umesh, xenodevil, Zoyd
24	ESSAYEZ / CATCH	Uberzen1
25	Exemples de bases de données et de tables	Abhilash R Vankayala, Arulkumar, Athafoud, bignose, Bostjan, Brad Larson, Christian, CL., Dariusz, Dr. J. Testington, enrico.bacis, Florin Ghita, FlyingPiMonster, forsvarir, Franck Dernoncourt, hairboat, JavaHopper, Jaydles, Jon Ericson, Magisch, Matt, Mureinik, Mzzzzzz, Prateek, rdans, Shiva, tinlyx, Tot Zam, WesleyJohnson
26	EXPLIQUEZ et DÉCRIVEZ	Simulant
27	Expressions de table communes	CL., Daniel, dd4711, fuzzy_logic, Gidil, Luis Lema, ninesided, Peter K, Phrancis, Sibeesh Venu
28	Filtrer les résultats en utilisant WHERE et HAVING	Arulkumar, Bostjan, CL., Community, Franck Dernoncourt, H. Pauwelyn, Jon Chan, Jon Ericson, juergen d, Matas Vaitkevicius, Mureinik, Phrancis, Tot Zam
29	Fonctions (agrégées)	ashja99, CL., Florin Ghita, Ian Kenney, Imran Ali Khan, Jon Chan, juergen d, KIRAN KUMAR MATAM, Mark Stewart, Maverick, Nathan, omini data, Peter K, Reboot, Tot Zam, William Ledbetter, winseybash, Алексей Неудачин
30	Fonctions (analytique)	CL., omini data
31	Fonctions (Scalar / Single Row)	CL., Kewin Björk Nielsen, Mark Stewart
32	Fonctions de chaîne	ɐlǝx, Allan S. Hansen, Arthur D, Arulkumar, Batsu, Chris, CL., Damon Smithies, Franck Dernoncourt, Golden Gate, hatchet,

		Imran Ali Khan, IncrediApp, Jaydip Jadhav, Jones Joseph, Kewin Björk Nielsen, Leigh Riffel, Matas Vaitkevicius, Mateusz Piotrowski, Neria Nachum, Phrancis, RamenChef, Robert Columbia, vmaroli, ypercube
33	Fonctions de fenêtre	Arkh, beercohol, bhs, Gidil, Jerry Jeremiah, Mureinik, mustaccio
34	FUSIONNER	Abhilash R Vankayala, CL., Kyle Hale, SQLFox, Zoyd
35	GRANT et REVOKE	RamenChef, user2314737
36	Identifiant	Andreas, CL.
37	Index	a1ex07, Almir Vuk, carlosb, CL., David Manheim, FlyingPiMonster, forsvarir, Franck Dernoncourt, Horaciux, Jenism, KIRAN KUMAR MATAM, mauris, Parado, Paulo Freitas , Ryan
38	Injection SQL	120196, CL., Clomp, Community, Epodax, Knickerless-Noggins, Stefan Steiger
39	INSÉRER	Ameya Deshpande, CL., Daniel Langemann, Dipesh Poudel, inquisitive_mind, KIRAN KUMAR MATAM, rajarshig, Tot Zam, zplizzi
40	JOINDRE	A_Arnold, Akshay Anand, Andy G, bignose, Branko Dimitrijevic, Casper Spruit, CL., Daniel Langemann, Darren Bartrup-Cook, Dipesh Poudel, enrico.bacis, Florin Ghita, forsvarir, Franck Dernoncourt, hairboat, Hari K M, HK1, HLGEM, inquisitive_mind , John C, John Odom, John Slegers, Mark Iannucci, Marvin, Mureinik, Phrancis, raholling, Raidri, Saroj Sasmal, Stefan Steiger, sunkuet02, Tot Zam, xenodevil, ypercube, Рахул Маквана
41	METTRE À JOUR	Akshay Anand, CL., Daniel Vérité, Dariusz, Dipesh Poudel, FlyingPiMonster, Gidil, H. Pauwelyn, Jon Chan, KIRAN KUMAR MATAM, Matas Vaitkevicius, Matt, Phrancis, Sanjay Bharwani, sunkuet02, Tot Zam, TriskalJM, vmaroli, WesleyJohnson
42	Nettoyer le code en SQL	CL., Stivan
43	NUL	Bart Schuijt, CL., dd4711, Devmati Wadikar, Phrancis, Saroj Sasmal, StanislavL, walid, ypercube
44	Numéro de ligne	CL., Phrancis, user1221533
45	Opérateur LIKE	Abhilash R Vankayala, Aidan, ashja99, Bart Schuijt, CL., Cristian Abelleira, guiguiblitz, Harish Gyanani, hellyale, Jenism,

		Lohitha Palagiri, Mark Perera, Mr. Developer, Ojen, Phrancis, RamenChef, Redithion, Stefan Steiger, Tot Zam, Vikrant, vmaroli
46	Opérateurs ET & OU	guiguiblitz
47	Ordre d'exécution	a1ex07, Gallus, Ryan Rockey, ypercube
48	PAR GROUPE	3N1GM4, Abe Miessler, Bostjan, Devmati Wadikar, Filipe Manuel, Frank, Gidil, Jaydles, juergen d, Nathaniel Ford, Peter Gordon, Simone - Ali One, WesleyJohnson, Zahiro Mor, Zoyd
49	Procédures stockées	brichins, John Odom, Lamak, Ryan
50	Recherche de doublons sur un sous-ensemble de colonne avec détails	Darrel Lee, mnoronha
51	SAUF	LCIII
52	SAUTER LE SAUT (Pagination)	CL., Karl Blacquiere, Matas Vaitkevicius, RamenChef
53	Schéma d'information	Hack-R
54	SÉLECTIONNER	Abhilash R Vankayala, aholmes, Alok Singh, Amnon, Andrii Abramov, apomene, Arpit Solanki, Arulkumar, AstraSerg, Brent Oliver, Charlie West, Chris, Christian Sagmüller, Christos, CL., controller, dariru, Daryl, David Pine, David Spillett, day_dreamer , Dean Parker, DeepSpace, Dipesh Poudel, Dror, Durgpal Singh , Epodax, Eric VB, FH-Inway, Florin Ghita, FlyingPiMonster, Franck Dernoncourt, geeksal, George Bailey, Hari K M, HoangHieu, iliketocode, Imran Ali Khan, Inca, Jared Hooper, Jaydles, John Odom, John Slegers, Jojodmo, JonH, Kapep, KartikKannapur, Lankymart, Mark Iannucci, Mark Perera, Mark Wojciechowicz, Matas Vaitkevicius, Matt, Matt S, Mattew Whitt, Matthew Moisen, MegaTom, Mihai-Daniel Virna, Mureinik, mustaccio, mxmissile, Oded, Ojen, onedaywhen, Paul Bambury, penderi, Peter Gordon, Prateek, Praveen Tiwari, Přemysl Šťastný, Preuk, Racil Hilan, Robert Columbia, Ronnie Wang, Ryan, Saroj Sasmal, Shiva, SommerEngineering, sqluser, stark, sunkuet02, ThisIsImpossible, Timothy, user1336087, user1605665, waqasahmed, wintersolider, WMios, xQbert, Yury Fedorov, Zahiro Mor, zedfoxus
55	Séquence	John Smith

56	Sous-requêtes	CL., dasblinkenlight, KIRAN KUMAR MATAM, Nunie123, Phrancis, RamenChef, tinlyx
57	SQL Group By vs Distinct	carlosb
58	Supprimer en cascade	Stefan Steiger
59	Transactions	Amir Pourmand, CL., Daryl, John Odom
60	TRONQUER	Abhilash R Vankayala, CL., Cristian Abelleira, DaImTo, Hynek Bernard, inquisitive_mind, KIRAN KUMAR MATAM, Paul Bambury, ss005
61	Types de données	bluefeet, Jared Hooper, John Odom, Jon Chan, JonMark Perry, Phrancis
62	UNION / UNION ALL	Andrea, Athafoud, Daniel Langemann, Jason W, Jim, Joe Taras , KIRAN KUMAR MATAM, Lankymart, Mihai-Daniel Virna, sunkuet02
63	Vues matérialisées	dmfay
64	XML	Steven