









4D - Référence SQL

-  Prise en main
-  Utiliser le SQL dans 4D
-  Commandes SQL
-  Règles de syntaxe
-  Transactions
-  Fonctions
-  Annexes
-  Liste alphabétique des commandes

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

✚ Prise en main

- ✚ Introduction
- ✚ Recevoir le résultat d'une requête SQL dans une variable
- ✚ Utiliser la clause WHERE
- ✚ Recevoir le résultat d'une requête SQL dans un tableau
- ✚ Utiliser la fonction CAST
- ✚ Utiliser la clause ORDER BY
- ✚ Utiliser la clause GROUP BY
- ✚ Utiliser des fonctions statistiques
- ✚ Utiliser la clause HAVING
- ✚ Appeler des méthodes 4D depuis le code SQL
- ✚ Jointures
- ✚ Utiliser des alias
- ✚ Sous-requêtes
- ✚ Tracer et déboguer le code SQL
- ✚ Data Definition Language
- ✚ Connexions externes
- ✚ Connexion au moteur SQL 4D via le driver ODBC

Introduction

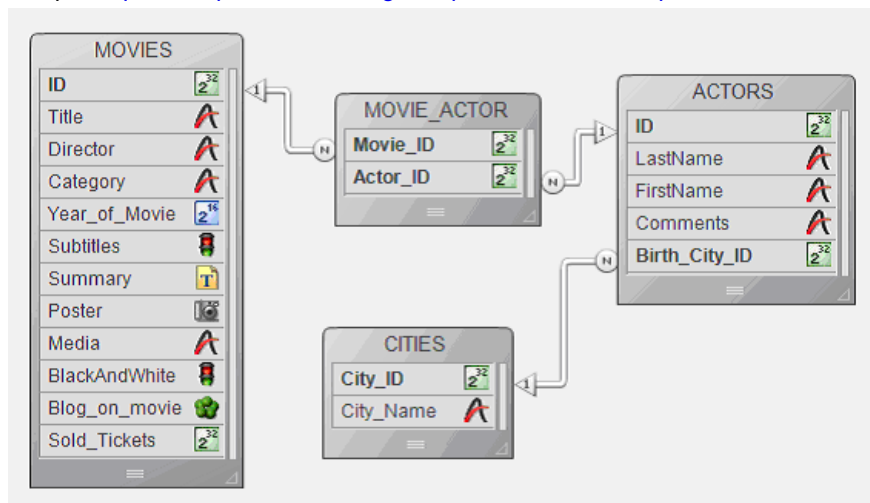
Le SQL (Structured Query Language) est un langage standardisé utilisé pour créer, organiser, gérer et rechercher des informations stockées dans une base de données informatique. Le SQL n'est pas en soi un système de gestion de données, c'est à la fois une composante intégrée de ce système, un langage et une interface de communication avec ce système.

L'objet de ce chapitre n'est pas de vous enseigner le SQL (pour cela, vous pourrez trouver de nombreux liens et sites spécialisés sur Internet), ni le langage intégré de 4D (pour cela, reportez-vous au manuel Langage de 4D).

Ce chapitre a pour but de vous montrer comment faire cohabiter le code SQL et le code 4D, récupérer des données à l'aide des commandes SQL, passer des paramètres à une requête SQL et manipuler les résultats.

Description de la base de données support de cette prise en main

Tous les extraits de code exposés dans ce chapitre proviennent d'une base de données exemple nommée "4D SQL Code Samples" [que vous pouvez télécharger depuis notre serveur ftp](#). La structure de cette base est la suivante :



La table MOVIES regroupe des informations concernant environ 50 films, incluant le titre (Title), le réalisateur (Director), le genre (Category : Action, Animation, Comédie, Science-fiction, Drama, etc.), l'année de sortie (Year_of_Movie), la présence de sous-titres (Subtitles), le résumé (Summary), l'affiche (Poster), le support (Media : DVD, VHS, DivX), s'il est en noir et blanc (BlackAndWhite), un blog stocké dans un BLOB (Blog_on_movie) et le nombre d'entrées réalisées (Sold_Tickets).

La table ACTORS regroupe des informations concernant les acteurs : outre un numéro d'identification (ID), le nom et le prénom (LastName et FirstName), un commentaire (Comments) et un identifiant de la ville de naissance (Birth_City_ID) de l'acteur.

La table CITIES contient l'identifiant et le nom des villes de naissance des acteurs.

La table MOVIE_ACTOR est utilisée pour établir un lien de type N vers N entre les tables MOVIES et ACTORS.

Tous les exemples présentés dans ce chapitre peuvent être exécutés depuis la boîte de dialogue suivante, accessible via la commande de menu **Démo SQL > Montrer les exemples...** :

Ⓜ
✖

Mode de requête

Utiliser le code 4D

Utiliser le code SQL

Utiliser ODBC

Utiliser "Chercher par SQL"

Utiliser SQL Dynamique

Tracer le code

Requêtes

Requêtes SQL vers variables Utilisation de requêtes statistiques

Clause WHERE Clause HAVING

Requêtes SQL vers tableaux Appel de méthode 4D

Utilisation de CAST Jointures

Clause ORDER BY Utilisation d'alias

Clause GROUP BY Sous requêtes

Recherche d'erreurs

Debugger le code SQL

Utiliser "Appeler sur erreur"

Définition des données (DDL)

DDL

Connexions externes

Connexion à ORACLE

Connexion à 4D

Listbox

Années	Titres	Réalisateurs	Média	Billets vendus	Nombre d'acteurs

Tableaux groupés

Années (aMovieYear)	Titre (aTitles)	Réalisateurs (aDirectors)	Média (aMedia)	Billets vendus (aSoldTickets)	Nombre d'acteurs (aNrActors)

Recevoir le résultat d'une requête SQL dans une variable

Commençons par une requête très simple : nous voulons savoir combien de films contient notre vidéothèque. En langage 4D, cette requête peut s'écrire ainsi :

```
C_ENTIER LONG($AllMovies)
$AllMovies:=0
TOUT SELECTIONNER([MOVIES])
$AllMovies:=Enregistrements trouvés([MOVIES])
ALERTE("La vidéothèque contient"+Chaine($AllMovies)+"films")
```

- La première façon d'interagir d'une manière similaire avec le moteur SQL est de placer la requête entre les balises **Debut SQL** et **Fin SQL**. Ainsi, le code précédent devient :

```
C_ENTIER LONG($AllMovies)
$AllMovies:=0
Debut SQL
    SELECT COUNT(*)
    FROM MOVIES
    INTO <<$AllMovies>>
Fin SQL
ALERTE("La vidéothèque contient"+Chaine($AllMovies)+" films")
```

Comme vous pouvez le constater, vous pouvez récupérer le résultat d'une requête dans une variable 4D (dans notre exemple, \$AllMovies) incluse dans les symboles "<<" et ">>".

Une autre manière de référencer tout type d'expression 4D valide (variable, champ, tableau, "expression...") est de la faire précéder du caractère deux-points ":" :

```
C_ENTIER LONG($AllMovies)
$AllMovies:=0
Debut SQL
    SELECT COUNT(*)
    FROM MOVIES
    INTO :$AllMovies
Fin SQL
ALERTE("La vidéothèque contient"+Chaine($AllMovies)+" films")
```

Attention aux variables interprocess : elles doivent en plus être incluses entre des crochets "[" et "]" :

```
C_ENTIER LONG(<>AllMovies)
<>AllMovies:=0
Debut SQL
    SELECT COUNT(*)
    FROM MOVIES
    INTO <<[<>AllMovies]>>
Fin SQL
ALERTE("La vidéothèque contient "+Chaine($AllMovies)+" films")
```

- La deuxième façon d'interagir avec le moteur SQL est d'utiliser les commandes SQL génériques intégrées (compatibles ODBC). Le code devient alors :

```
C_ENTIER LONG($AllMovies)
$AllMovies:=0
\\Initialize a connection with the internal SQL engine
SQL LOGIN(SQL_INTERNAL;"";"")
```

```

\\Execute the query and return the result in the $AllMovies variable
SQL EXECUTER("SELECT COUNT(*) FROM MOVIES";$AllMovies)
\\Retrieve all the records found
SQL CHARGER ENREGISTREMENT (SQL Tous les enregistrements)
\\Close the connection
SQL LOGOUT
ALERTE("The Video Library contains "+Chaine($AllMovies)+" movies")

```

Pour plus d'informations sur les commandes SQL génériques, reportez-vous au chapitre dans le manuel *Langage de 4D*.

- La troisième façon d'interagir avec le moteur SQL est d'utiliser la commande 4D **CHERCHER PAR SQL**. Le code devient :

```

C_ENTIER LONG($AllMovies)
$AllMovies:=0
CHERCHER PAR SQL([MOVIES];"ID <> 0")
$AllMovies:=Enregistrements trouves([MOVIES])
ALERTE("La videothèque contient "+Chaine($AllMovies)+" films")

```

La commande **CHERCHER PAR SQL** exécute une requête de type *SELECT* pouvant s'écrire de la manière suivante :

```

SELECT *
FROM maTable
WHERE <Formule_SQL>

```

maTable est le nom de la table passé en premier paramètre et *Formule_SQL* est le texte de la requête, passé en second paramètre :

```

CHERCHER PAR SQL(maTable;Formule_SQL)

```

Dans notre cas, il n'y a pas de clause **WHERE**, donc nous en ajoutons une : "ID <> 0". L'équivalent de la requête en code SQL serait :

```

SELECT *
FROM MOVIES
WHERE ID <> 0

```

- La quatrième façon d'interagir avec le moteur SQL est d'utiliser la commande SQL dynamique *EXECUTE IMMEDIATE*. Le code devient alors :

```

\C_ENTIER LONG($AllMovies)
$AllMovies:=0
C_TEXTE($tQueryTxt)
$tQueryTxt:="SELECT COUNT(*) FROM MOVIES INTO :$AllMovies"
Debut SQL
EXECUTE IMMEDIATE :$tQueryTxt;
Fin SQL
ALERTE("La videothèque contient "+Chaine($AllMovies)+" films")

```

Pour tester tous ces exemples, lancez la base "4D SQL Code Samples" et affichez la boîte de dialogue principale. Dans la partie gauche de la fenêtre, choisissez le mode d'interrogation du moteur de 4D :

Mode de requête

- Utiliser le code 4D
- Utiliser le code SQL
- Utiliser ODBC
- Utiliser "Chercher par SQL"
- Utiliser SQL Dynamique
- Tracer le code

Puis cliquez sur le bouton **Requêtes SQL vers variables**.

✚ Utiliser la clause WHERE

Nous souhaitons à présent savoir combien de films réalisés à partir de 1960 sont présents dans la base. Voici le code 4D correspondant à cette requête :

```
C_ENTIER LONG($NoMovies)
$NoMovies:=0
REDUIRE SELECTION([MOVIES];0)
CHERCHER([MOVIES];[MOVIES]Year_of_Movie>=1960)
$NoMovies:=Enregistrements trouvés([MOVIES])
ALERTE("La vidéothèque contient "+Chaine($NoMovies)+" films réalisés depuis 1960")
```

- En utilisant du code SQL :

```
C_ENTIER LONG($NoMovies)
$NoMovies:=0
REDUIRE SELECTION([MOVIES];0)
Debut SQL
    SELECT COUNT(*)
    FROM MOVIES
    WHERE Year_of_Movie >= 1960
    INTO :$NoMovies;
Fin SQL
ALERTE("La vidéothèque contient "+Chaine($NoMovies)+" films réalisés depuis 1960")
```

- En utilisant les commandes SQL génériques :

```
C_ENTIER LONG($NoMovies)
$NoMovies:=0
REDUIRE SELECTION([MOVIES];0)
SQL LOGIN(SQL_INTERNAL;"");
SQL EXECUTER("SELECT COUNT(*) FROM MOVIES WHERE Year_of_Movie >= 1960";$NoMovies)
SQL CHARGER ENREGISTREMENT(SQL_Tous_les_enregistrements)
SQL LOGOUT
ALERTE("La vidéothèque contient "+Chaine($NoMovies)+" films réalisés depuis 1960")
```

- En utilisant la commande 4D **CHERCHER PAR SQL** :

```
C_ENTIER LONG($NoMovies)
$NoMovies:=0
REDUIRE SELECTION([MOVIES];0)
CHERCHER PAR SQL([MOVIES];"Year_of_Movie >= 1960")
$NoMovies:=Enregistrements trouvés([MOVIES])
ALERTE("La vidéothèque contient "+Chaine($NoMovies)+" films réalisés depuis 1960")
```

- En utilisant la commande SQL dynamique *EXECUTE IMMEDIATE* :

```
C_ENTIER LONG($NoMovies)
C_TEXTE($tQueryTxt)
$NoMovies:=0
REDUIRE SELECTION([MOVIES];0)
$tQueryTxt:="SELECT COUNT(*) FROM MOVIES WHERE Year_of_Movie >= 1960 INTO :$NoMovies;"
Debut SQL
    EXECUTE IMMEDIATE :$tQueryTxt;
Fin SQL
```

```
ALERTE("La videothèque contient "+Chaine($NoMovies)+" films réalisés depuis 1960")
```

Comme dans la section précédente, pour tester tous ces exemples, lancez la base "4D SQL Code Samples" et affichez la boîte de dialogue principale. Choisissez le mode d'interrogation du moteur de 4D et cliquez sur le bouton **Clause WHERE**.

Recevoir le résultat d'une requête SQL dans un tableau

Nous souhaitons maintenant utiliser dans la requête une variable contenant l'année (et non l'année elle-même) et récupérer la liste de tous les films tournés depuis 1960. De plus, pour chaque film trouvé, nous voulons obtenir plusieurs informations : l'année, le titre, le réalisateur, le support media et le nombre d'entrées. La solution consiste à récupérer ces informations dans des tableaux ou une list box.

- Cette recherche se traduirait ainsi en code 4D :

```
TABLEAU ENTIER LONG (aSoldTickets;0)
TABLEAU ENTIER (aMovieYear;0)
TABLEAU TEXTE (aTitles;0)
TABLEAU TEXTE (aDirectors;0)
TABLEAU TEXTE (aMedias;0)
C_ENTIER LONG ($MovieYear)

REDUIRE SELECTION ([MOVIES];0)
$MovieYear:=1960
CHERCHER ([MOVIES]; [MOVIES]Year_of_Movie>=$MovieYear)
SELECTION VERS
TABLEAU ([MOVIES]Year_of_Movie;aMovieYear;[MOVIES]Title;aTitles;[MOVIES]Director;aDirectors;
[MOVIES]Media;aMedias;[MOVIES]Sold_Tickets;aSoldTickets)
  ` Initialiser le reste des colonnes de la list box afin de visualiser les informations
TABLEAU ENTIER LONG (aNrActors;Taille tableau(aMovieYear))
```

- En utilisant du code SQL :

```
TABLEAU ENTIER LONG (aSoldTickets;0)
TABLEAU ENTIER (aMovieYear;0)
TABLEAU TEXTE (aTitles;0)
TABLEAU TEXTE (aDirectors;0)
TABLEAU TEXTE (aMedias;0)
C_ENTIER LONG ($MovieYear)

$MovieYear:=1960
Debut SQL
  SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets
  FROM MOVIES
  WHERE Year_of_Movie >= :$MovieYear
  INTO :aMovieYear, :aTitles, :aDirectors, :aMedias, :aSoldTickets;
Fin SQL
```

Comme vous pouvez le constater :

- Il est possible de passer une variable (\$MovieYear) à la requête SQL en utilisant la même notation que pour recevoir des paramètres.
- Les résultats de la requête SQL sont stockés dans les tableaux aMovieYear, aTitles, aDirectories, aMedias et aSoldTickets. Ils sont affichés dans la fenêtre principale de deux manières :
 - via des tableaux groupés :

Tableaux groupés

Années (aMovieYear)	Titre (aTitles)	Réalisateurs (aDirectors)	Média (aMedia)	Billets vendus (aSoldTickets)	Nombre d'acteurs (aNrActors)

- via une list box :

Listbox

Années	Titres	Réalisateurs	Média	Billets vendus	Nombre d'acteurs

- En utilisant les commandes SQL génériques :

```

TABLEAU ENTIER LONG (aSoldTickets;0)
TABLEAU ENTIER (aMovieYear;0)
TABLEAU TEXTE (aTitles;0)
TABLEAU TEXTE (aDirectors;0)
TABLEAU TEXTE (aMedias;0)
C_ENTIER LONG ($MovieYear)
C_TEXTE ($tQueryTxt)

REDUIRE SELECTION ([MOVIES];0)
$MovieYear:=1960
SQL LOGIN (SQL_INTERNAL;";")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= :$MovieYear"
SQL EXECUTER ($tQueryTxt;aMovieYear;aTitles;aDirectors;aMedias;aSoldTickets)
SQL CHARGER ENREGISTREMENT (SQL Tous les enregistrements)
SQL LOGOUT

```

- En utilisant la commande **CHERCHER PAR SQL** :

```

TABLEAU ENTIER LONG (aSoldTickets;0)
TABLEAU ENTIER (aMovieYear;0)
TABLEAU TEXTE (aTitles;0)
TABLEAU TEXTE (aDirectors;0)
TABLEAU TEXTE (aMedias;0)
C_ENTIER LONG ($MovieYear)

REDUIRE SELECTION ([MOVIES];0)
$MovieYear:=1960
CHERCHER PAR SQL ([MOVIES];"Year_of_Movie >= :$MovieYear")
SELECTION VERS
TABLEAU ([MOVIES]Year_of_Movie;aMovieYear;[MOVIES]Title;aTitles;[MOVIES]Director;aDirectors;
[MOVIES]Media;aMedias;[MOVIES]Sold_Tickets;aSoldTickets)

```

- En utilisant la commande SQL dynamique **EXECUTE IMMEDIATE** :

```

TABLEAU ENTIER LONG (aSoldTickets;0)
TABLEAU ENTIER (aMovieYear;0)
TABLEAU TEXTE (aTitles;0)
TABLEAU TEXTE (aDirectors;0)
TABLEAU TEXTE (aMedias;0)
C_ENTIER LONG ($MovieYear)
C_TEXTE ($tQueryTxt)

REDUIRE SELECTION ([MOVIES];0)
$MovieYear:=1960
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= :$MovieYear"
$tQueryTxt:=$tQueryTxt+" INTO :aMovieYear, :aTitles, :aDirectors, :aMedias,
:aSoldTickets;"

```

Debut SQL

```
EXECUTE IMMEDIATE :$tQueryTxt;
```

Fin SQL

Pour tester tous ces exemples, lancez la base "4D SQL Code Samples" et affichez la boîte de dialogue principale. Choisissez le mode d'interrogation du moteur de 4D et cliquez sur le bouton **Requêtes SQL vers tableaux**.

Utiliser la fonction CAST

Le langage SQL établit des règles relativement restrictives pour la combinaison de données de différents types dans les expressions. Généralement le SGBD est chargé d'effectuer les conversions de données nécessaires. Cependant, le standard SQL requiert que le SGBD génère une erreur si vous tentez de comparer par exemple des numériques avec des chaînes de caractères. Dans ce contexte, la fonction CAST est très précieuse, en particulier lorsque le SQL est combiné à un langage de programmation manipulant des types de données non pris en charge par le standard SQL.

Cette section adapte l'exemple de la section [Recevoir le résultat d'une requête SQL dans un tableau](#) de manière à utiliser la fonction `CAST`.

- Le code 4D initial est le suivant :

```
TABLEAU ENTIER LONG (aSoldTickets;0)
TABLEAU ENTIER (aMovieYear;0)
TABLEAU TEXTE (aTitles;0)
TABLEAU TEXTE (aDirectors;0)
TABLEAU TEXTE (aMedias;0)

REDUIRE SELECTION ([MOVIES];0)
$MovieYear:=Num("1960")
CHERCHER ([MOVIES]; [MOVIES]Year_of_Movie>=$MovieYear)
SELECTION VERS
TABLEAU ([MOVIES]Year_of_Movie;aMovieYear; [MOVIES]Title;aTitles; [MOVIES]Director;aDirectors;
 [MOVIES]Media;aMedias; [MOVIES]Sold_Tickets;aSoldTickets)
```

- En utilisant du code SQL :

```
TABLEAU ENTIER LONG (aSoldTickets;0)
TABLEAU ENTIER (aMovieYear;0)
TABLEAU TEXTE (aTitles;0)
TABLEAU TEXTE (aDirectors;0)
TABLEAU TEXTE (aMedias;0)

Debut SQL
  SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets
  FROM MOVIES
  WHERE Year_of_Movie >= CAST('1960' AS INT)
  INTO :aMovieYear, :aTitles, :aDirectors, :aMedias, :aSoldTickets;
Fin SQL
```

- En utilisant les commandes SQL génériques :

```
TABLEAU ENTIER LONG (aSoldTickets;0)
TABLEAU ENTIER (aMovieYear;0)
TABLEAU TEXTE (aTitles;0)
TABLEAU TEXTE (aDirectors;0)
TABLEAU TEXTE (aMedias;0)
C_TEXTE ($tQueryTxt)

REDUIRE SELECTION ([MOVIES];0)
SQL LOGIN (SQL_INTERNAL;";")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= CAST('1960' AS INT)"
SQL EXECUTER ($tQueryTxt;aMovieYear;aTitles;aDirectors;aMedias;aSoldTickets)
SQL CHARGER ENREGISTREMENT (SQL Tous les enregistrements)
SQL LOGOUT
```

- En utilisant la commande **CHERCHER PAR SQL** :

```

TABLEAU ENTIER LONG (aSoldTickets;0)
TABLEAU ENTIER (aMovieYear;0)
TABLEAU TEXTE (aTitles;0)
TABLEAU TEXTE (aDirectors;0)
TABLEAU TEXTE (aMedias;0)

REDUIRE SELECTION ([MOVIES];0)
CHERCHER PAR SQL ([MOVIES];"Year_of_Movie >= CAST('1960' AS INT)")
SELECTION VERS
TABLEAU ([MOVIES]Year_of_Movie;aMovieYear;[MOVIES]Title;aTitles;[MOVIES]Director;aDirectors;
[MOVIES]Media;aMedias;[MOVIES]Sold_Tickets;aSoldTickets)

```

- En utilisant la commande SQL dynamique **EXECUTE IMMEDIATE** :

```

TABLEAU ENTIER LONG (aSoldTickets;0)
TABLEAU ENTIER (aMovieYear;0)
TABLEAU TEXTE (aTitles;0)
TABLEAU TEXTE (aDirectors;0)
TABLEAU TEXTE (aMedias;0)
C_TEXTE ($tQueryTxt)

REDUIRE SELECTION ([MOVIES];0)
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= CAST('1960' AS INT)"
$tQueryTxt:=$tQueryTxt+" INTO :aMovieYear, :aTitles, :aDirectors, :aMedias,
:aSoldTickets;"
Debut SQL
EXECUTE IMMEDIATE :$tQueryTxt;
Fin SQL

```

Pour tester tous ces exemples, lancez la base "4D SQL Code Samples" et affichez la boîte de dialogue principale. Choisissez le mode d'interrogation du moteur de 4D et cliquez sur le bouton **Utilisation de CAST**.

✚ Utiliser la clause ORDER BY

Nous souhaitons désormais récupérer tous les films tournés depuis 1960 et, pour chaque film, l'année, le titre, le réalisateur, le support media et le nombre d'entrées. Le résultat devra être trié par année.

Voici le code 4D correspondant à cette requête :

```
TABLEAU ENTIER LONG (aNrActors;0)
TABLEAU ENTIER LONG (aSoldTickets;0)
TABLEAU ENTIER (aMovieYear;0)
TABLEAU TEXTE (aTitles;0)
TABLEAU TEXTE (aDirectors;0)
TABLEAU TEXTE (aMedias;0)
C_ENTIER LONG ($MovieYear)

REDUIRE SELECTION ([MOVIES];0)
$MovieYear:=1960
CHERCHER ([MOVIES]; [MOVIES]Year_of_Movie>=$MovieYear)
SELECTION VERS
TABLEAU ([MOVIES]Year_of_Movie;aMovieYear;[MOVIES]Title;aTitles;[MOVIES]Director;aDirectors;
[MOVIES]Media;aMedias;[MOVIES]Sold_Tickets;aSoldTickets)
TRIER TABLEAU (aMovieYear;aTitles;aDirectors;aMedias;>)
```

- En utilisant du code SQL :

```
TABLEAU ENTIER LONG (aNrActors;0)
TABLEAU ENTIER LONG (aSoldTickets;0)
TABLEAU ENTIER (aMovieYear;0)
TABLEAU TEXTE (aTitles;0)
TABLEAU TEXTE (aDirectors;0)
TABLEAU TEXTE (aMedias;0)
C_ENTIER LONG ($MovieYear)

REDUIRE SELECTION ([MOVIES];0)
$MovieYear:=1960
Debut SQL
    SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets
    FROM MOVIES
    WHERE Year_of_Movie >= :$MovieYear
    ORDER BY 1
    INTO :aMovieYear, :aTitles, :aDirectors, :aMedias, :aSoldTickets;
Fin SQL
```

- En utilisant les commandes SQL génériques :

```
C_TEXTE ($tQueryTxt)
TABLEAU ENTIER LONG (aNrActors;0)
TABLEAU ENTIER LONG (aSoldTickets;0)
TABLEAU ENTIER (aMovieYear;0)
TABLEAU TEXTE (aTitles;0)
TABLEAU TEXTE (aDirectors;0)
TABLEAU TEXTE (aMedias;0)
C_ENTIER LONG ($MovieYear)
REDUIRE SELECTION ([MOVIES];0)
$MovieYear:=1960
SQL LOGIN (SQL_INTERNAL;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
```

```

$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= :$MovieYear"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"
SQL EXECUTER($tQueryTxt;aMovieYear;aTitles;aDirectors;aMedias;aSoldTickets)
SQL CHARGER ENREGISTREMENT (SQL Tous les enregistrements)
SQL LOGOUT

```

- En utilisant la commande **CHERCHER PAR SQL** :

```

TABLEAU ENTIER LONG (aNrActors;0)
TABLEAU ENTIER LONG (aSoldTickets;0)
TABLEAU ENTIER (aMovieYear;0)
TABLEAU TEXTE (aTitles;0)
TABLEAU TEXTE (aDirectors;0)
TABLEAU TEXTE (aMedias;0)
C_ENTIER LONG($MovieYear)

REDUIRE SELECTION ([MOVIES];0)
$MovieYear:=1960
CHERCHER PAR SQL ([MOVIES];"Year_of_Movie >= :$MovieYear")
SELECTION VERS
TABLEAU ([MOVIES]Year_of_Movie;aMovieYear;[MOVIES]Title;aTitles;[MOVIES]Director;aDirectors;
[MOVIES]Media;aMedias;[MOVIES]Sold_Tickets;aSoldTickets)
TRIER TABLEAU (aMovieYear;aTitles;aDirectors;aMedias;>)

```

- En utilisant la commande SQL dynamique **EXECUTE IMMEDIATE** :

```

TABLEAU ENTIER LONG (aNrActors;0)
C_TEXTE ($tQueryTxt)
TABLEAU ENTIER LONG (aSoldTickets;0)
TABLEAU ENTIER (aMovieYear;0)
TABLEAU TEXTE (aTitles;0)
TABLEAU TEXTE (aDirectors;0)
TABLEAU TEXTE (aMedias;0)
C_ENTIER LONG ($MovieYear)

REDUIRE SELECTION ([MOVIES];0)
$MovieYear:=1960
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= :$MovieYear"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"
$tQueryTxt:=$tQueryTxt+" INTO :aMovieYear, :aTitles, :aDirectors, :aMedias,
:aSoldTickets;"
Debut SQL
EXECUTE IMMEDIATE :$tQueryTxt;
Fin SQL

```

Pour tester tous ces exemples, lancez la base "4D SQL Code Samples" et affichez la boîte de dialogue principale. Choisissez le mode d'interrogation du moteur de 4D et cliquez sur le bouton **Clause ORDER BY**.

Utiliser la clause GROUP BY

Nous souhaitons obtenir le nombre annuel total d'entrées depuis 1979. Le résultat devra être trié par année. Voici le code 4D correspondant à cette requête :

```
TABLEAU ENTIER (aMovieYear;0)
TABLEAU ENTIER LONG (aSoldTickets;0)
C_ENTIER LONG ($MovieYear;$vCrtMovieYear;$i)

REDUIRE SELECTION ([MOVIES];0)
$MovieYear:=1979
CHERCHER ([MOVIES]; [MOVIES]Year_of_Movie>=$MovieYear)
TRIER ([MOVIES]; [MOVIES]Year_of_Movie;>)
$vCrtMovieYear:=0
$vInd:=Taille tableau (aMovieYear)
Boucle ($i;1;Enregistrements trouves ([MOVIES]))
  Si ([MOVIES]Year_of_Movie# $vCrtMovieYear)
    $vCrtMovieYear:=[MOVIES]Year_of_Movie
    $vInd:=$vInd+1
    INSERER DANS TABLEAU (aMovieYear;$vInd;1)
    aMovieYear{$vInd}:=$vCrtMovieYear
    INSERER DANS TABLEAU (aSoldTickets;$vInd;1)
  Fin de si
  aSoldTickets{$vInd}:=aSoldTickets{$vInd}+[MOVIES]Sold_Tickets
  ENREGISTREMENT SUIVANT ([MOVIES])
Fin de boucle
` Initialisation du reste des colonnes de list box pour visualiser l'information
TABLEAU TEXTE (aTitles;Taille tableau (aMovieYear))
TABLEAU TEXTE (aDirectors;Taille tableau (aMovieYear))
TABLEAU TEXTE (aMedias;Taille tableau (aMovieYear))
TABLEAU ENTIER LONG (aNrActors;Taille tableau (aMovieYear))
```

- En utilisant du code SQL :

```
TABLEAU ENTIER (aMovieYear;0)
TABLEAU ENTIER LONG (aSoldTickets;0)
C_ENTIER LONG ($MovieYear)

REDUIRE SELECTION ([MOVIES];0)
$MovieYear:=1979
Debut SQL
  SELECT Year_of_Movie, SUM(Sold_Tickets)
  FROM MOVIES
  WHERE Year_of_Movie >= :$MovieYear
  GROUP BY Year_of_Movie
  ORDER BY 1
  INTO :aMovieYear, :aSoldTickets;
Fin SQL
` Initialisation du reste des colonnes de list box pour visualiser l'information
TABLEAU TEXTE (aTitles;Taille tableau (aMovieYear))
TABLEAU TEXTE (aDirectors;Taille tableau (aMovieYear))
TABLEAU TEXTE (aMedias;Taille tableau (aMovieYear))
TABLEAU ENTIER LONG (aNrActors;Taille tableau (aMovieYear))
```

- En utilisant les commandes SQL génériques :

```
C_TEXTE ($tQueryTxt)
TABLEAU ENTIER LONG (aSoldTickets;0)
```



```

TABLEAU ENTIER (aMovieYear;0)
C_ENTIER LONG ($MovieYear)

REDUIRE SELECTION ([MOVIES];0)
$MovieYear:=1979
SQL LOGIN (SQL_INTERNAL;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, SUM(Sold_Tickets)"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= :$MovieYear"
$tQueryTxt:=$tQueryTxt+" GROUP BY Year_of_Movie"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"
SQL EXECUTER ($tQueryTxt;aMovieYear;aSoldTickets)
SQL CHARGER ENREGISTREMENT (SQL Tous les enregistrements)
SQL LOGOUT
  ` Initialisation du reste des colonnes de list box pour visualiser l'information
TABLEAU TEXTE (aTitles;Taille tableau(aMovieYear))
TABLEAU TEXTE (aDirectors;Taille tableau(aMovieYear))
TABLEAU TEXTE (aMedias;Taille tableau(aMovieYear))
TABLEAU ENTIER LONG (aNrActors;Taille tableau(aMovieYear))

```

- En utilisant la commande **CHERCHER PAR SQL** :

```

TABLEAU ENTIER LONG (aSoldTickets;0)
TABLEAU ENTIER (aMovieYear;0)
C_ENTIER LONG ($MovieYear)

REDUIRE SELECTION ([MOVIES];0)
$MovieYear:=1979
CHERCHER PAR SQL ([MOVIES];"Year_of_Movie >= :$MovieYear")
TRIER ([MOVIES];[MOVIES]Year_of_Movie;>)
$vCrtMovieYear:=0
$vInd:=Taille tableau (aMovieYear)
Boucle ($i;1;Enregistrements trouves ([MOVIES]))
  Si ([MOVIES]Year_of_Movie#$vCrtMovieYear)
    $vCrtMovieYear:=[MOVIES]Year_of_Movie
    $vInd:=$vInd+1
    INSERER DANS TABLEAU (aMovieYear;$vInd;1)
    aMovieYear{$vInd}:=$vCrtMovieYear
    INSERER DANS TABLEAU (aSoldTickets;$vInd;1)
  Fin de si
  aSoldTickets{$vInd}:aSoldTickets{$vInd}+[MOVIES]Sold_Tickets
  ENREGISTREMENT SUIVANT ([MOVIES])
Fin de boucle
  ` Initialisation du reste des colonnes de list box pour visualiser l'information
TABLEAU TEXTE (aTitles;Taille tableau(aMovieYear))
TABLEAU TEXTE (aDirectors;Taille tableau(aMovieYear))
TABLEAU TEXTE (aMedias;Taille tableau(aMovieYear))
TABLEAU ENTIER LONG (aNrActors;Taille tableau(aMovieYear))

```

- En utilisant la commande SQL dynamique **EXECUTE IMMEDIATE** :

```

C_TEXTE ($tQueryTxt)
TABLEAU ENTIER LONG (aSoldTickets;0)
TABLEAU ENTIER (aMovieYear;0)
C_ENTIER LONG ($MovieYear)

$MovieYear:=1979
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, SUM(Sold_Tickets)"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= :$MovieYear"
$tQueryTxt:=$tQueryTxt+" GROUP BY Year_of_Movie"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"

```

```
$tQueryTxt:=$tQueryTxt+" INTO :aMovieYear, :aSoldTickets;"  
Debut SQL  
EXECUTE IMMEDIATE :$tQueryTxt;  
Fin SQL  
` Initialisation du reste des colonnes de list box pour visualiser l'information  
TABLEAU TEXTE(aTitles;Taille tableau(aMovieYear))  
TABLEAU TEXTE(aDirectors;Taille tableau(aMovieYear))  
TABLEAU TEXTE(aMedias;Taille tableau(aMovieYear))  
TABLEAU ENTIER LONG(aNrActors;Taille tableau(aMovieYear))
```

Pour tester tous ces exemples, lancez la base "4D SQL Code Samples" et affichez la boîte de dialogue principale. Choisissez le mode d'interrogation du moteur de 4D et cliquez sur le bouton **Clause GROUP BY**.

Utiliser des fonctions statistiques

Les fonctions statistiques permettent d'effectuer des calculs sur des séries de valeurs. Le SQL contient de nombreuses fonctions d'agrégation telles que **MIN**, **MAX**, **AVG**, **SUM**, etc. A l'aide de fonctions d'agrégation, nous souhaitons effectuer des statistiques sur le nombre total d'entrées par an. Le résultat devra être trié par année. Pour cela, nous devons effectuer la somme des entrées pour chaque film puis trier le résultat par année.

Voici le code 4D correspondant à cette requête :

```
C_ENTIER LONG ($vMin; $vMax; $vSum)
C_REEL ($vAverage)
C_TEXTE ($AlertTxt)

REDUIRE SELECTION ([MOVIES]; 0)
$vMin:=0
$vMax:=0
$vAverage:=0
$vSum:=0
TOUT SELECTIONNER ([MOVIES])
$vMin:=Min ([MOVIES]Sold_Tickets)
$vMax:=Max ([MOVIES]Sold_Tickets)
$vAverage:=Moyenne ([MOVIES]Sold_Tickets)
$vSum:=Somme ([MOVIES]Sold_Tickets)
$AlertTxt:=""
$AlertTxt:=$AlertTxt+"Nombre minimum d'entrées : "+Chaine ($vMin)+Caractere (13)
$AlertTxt:=$AlertTxt+"Nombre maximum d'entrées : "+Chaine ($vMax)+Caractere (13)
$AlertTxt:=$AlertTxt+"Moyenne des entrées : "+Chaine ($vAverage)+Caractere (13)
$AlertTxt:=$AlertTxt+"Total des entrées : "+Chaine ($vSum)+Caractere (13)
ALERTE ($AlertTxt)
```

- En utilisant du code SQL :

```
C_ENTIER LONG ($vMin; $vMax; $vSum)
C_REEL ($vAverage)
C_TEXTE ($AlertTxt)

$vMin:=0
$vMax:=0
$vAverage:=0
$vSum:=0
Debut SQL
SELECT MIN (Sold_Tickets),
MAX (Sold_Tickets),
AVG (Sold_Tickets),
SUM (Sold_Tickets)
FROM MOVIES
INTO :$vMin, :$vMax, :$vAverage, :$vSum;
Fin SQL
$AlertTxt:=""
$AlertTxt:=$AlertTxt+"Nombre minimum d'entrées : "+Chaine ($vMin)+Caractere (13)
$AlertTxt:=$AlertTxt+"Nombre maximum d'entrées : "+Chaine ($vMax)+Caractere (13)
$AlertTxt:=$AlertTxt+"Moyenne des entrées : "+Chaine ($vAverage)+Caractere (13)
$AlertTxt:=$AlertTxt+"Total des entrées : "+Chaine ($vSum)+Caractere (13)
ALERTE ($AlertTxt)
```

- En utilisant les commandes SQL génériques :

```
C_ENTIER LONG ($vMin; $vMax; $vSum)
C_REEL ($vAverage)
C_TEXTE ($tQueryTxt)
```

```

C_TEXTE($AlertTxt)

$vMin:=0
$vMax:=0
$vAverage:=0
$vSum:=0
SQL LOGIN(SQL_INTERNAL;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT MIN(Sold_Tickets), MAX(Sold_Tickets), AVG(Sold_Tickets),
SUM(Sold_Tickets)"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
SQL EXECUTER($tQueryTxt;$vMin;$vMax;$vAverage;$vSum)
SQL CHARGER ENREGISTREMENT(SQL Tous les enregistrements)
SQL LOGOUT
$AlertTxt:=""
$AlertTxt:=$AlertTxt+"Nombre minimum d'entrées : "+Chaine($vMin)+Caractere(13)
$AlertTxt:=$AlertTxt+"Nombre maximum d'entrées : "+Chaine($vMax)+Caractere(13)
$AlertTxt:=$AlertTxt+"Moyenne des entrées : "+Chaine($vAverage)+Caractere(13)
$AlertTxt:=$AlertTxt+"Total des entrées : "+Chaine($vSum)+Caractere(13)
ALERTE($AlertTxt)

```

- En utilisant la commande SQL dynamique *EXECUTE IMMEDIATE* :

```

C_ENTIER LONG($vMin;$vMax;$vSum)
C_REEL($vAverage)
C_TEXTE($tQueryTxt)
C_TEXTE($AlertTxt)

$vMin:=0
$vMax:=0
$vAverage:=0
$vSum:=0
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT MIN(Sold_Tickets), MAX(Sold_Tickets), AVG(Sold_Tickets),
SUM(Sold_Tickets)"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" INTO :$vMin, :$vMax, :$vAverage, :$vSum;"
Debut SQL
EXECUTE IMMEDIATE :$tQueryTxt;
Fin SQL
$AlertTxt:=""
$AlertTxt:=$AlertTxt+"Nombre minimum d'entrées : "+Chaine($vMin)+Caractere(13)
$AlertTxt:=$AlertTxt+"Nombre maximum d'entrées : "+Chaine($vMax)+Caractere(13)
$AlertTxt:=$AlertTxt+"Moyenne des entrées : "+Chaine($vAverage)+Caractere(13)
$AlertTxt:=$AlertTxt+"Total des entrées : "+Chaine($vSum)+Caractere(13)
ALERTE($AlertTxt)

```

Pour tester ces exemples, lancez la base "4D SQL Code Samples" et affichez la boîte de dialogue principale. Choisissez le mode d'interrogation du moteur de 4D et cliquez sur le bouton Utilisation de requêtes statistiques.

Utiliser la clause HAVING

Nous souhaitons obtenir le nombre total d'entrées depuis 1979 classé par année mais en excluant les films totalisant à eux seuls plus de 10 millions d'entrées.

Pour cela, nous devons cumuler le nombre d'entrée de chaque film depuis 1979, enlever les entrées de ceux qui ont fait plus de 10 000 000 entrées puis trier le résultat par année.

Voici le code 4D correspondant à cette requête :

```
TABLEAU ENTIER (aMovieYear;0)
TABLEAU ENTIER LONG (aSoldTickets;0)
C_ENTIER LONG ($MovieYear;$vCrtMovieYear;$i;$MinSoldTickets;$vInd)

REDUIRE SELECTION ([MOVIES];0)
$MovieYear:=1979
$MinSoldTickets:=10000000
CHERCHER ([MOVIES]; [MOVIES]Year_of_Movie>=$MovieYear)
TRIER ([MOVIES]; [MOVIES]Year_of_Movie;>)
$vCrtMovieYear:=0
$vInd:=Taille tableau (aMovieYear)
Boucle ($i;1;Enregistrements trouves ([MOVIES]))
  Si ([MOVIES]Year_of_Movie# $vCrtMovieYear)
    $vCrtMovieYear:=[MOVIES]Year_of_Movie
    Si (aSoldTickets{$vInd}<$MinSoldTickets)
      $vInd:=$vInd+1
      INSERER DANS TABLEAU (aMovieYear;$vInd;1)
      aMovieYear{$vInd}:=$vCrtMovieYear
      INSERER DANS TABLEAU (aSoldTickets;$vInd;1)
    Sinon
      aSoldTickets{$vInd}:=0
    Fin de si
  Fin de si
  aSoldTickets{$vInd}:=aSoldTickets{$vInd}+[MOVIES]Sold_Tickets
  ENREGISTREMENT SUIVANT ([MOVIES])
Fin de boucle
Si (aSoldTickets{$vInd}>=$MinSoldTickets)
  SUPPRIMER DANS TABLEAU (aSoldTickets;$vInd;1)
  SUPPRIMER DANS TABLEAU (aMovieYear;$vInd;1)
Fin de si
  \ Initialisation du reste des colonnes de list box pour visualiser l'information
TABLEAU TEXTE (aTitles;Taille tableau (aMovieYear))
TABLEAU TEXTE (aDirectors;Taille tableau (aMovieYear))
TABLEAU TEXTE (aMedias;Taille tableau (aMovieYear))
TABLEAU ENTIER LONG (aNrActors;Taille tableau (aMovieYear))
```

- En utilisant du code SQL :

```
TABLEAU ENTIER (aMovieYear;0)
TABLEAU ENTIER LONG (aSoldTickets;0)
C_ENTIER LONG ($MovieYear;$MinSoldTickets)

$MovieYear:=1979
$MinSoldTickets:=10000000
Debut SQL
  SELECT Year_of_Movie, SUM(Sold_Tickets)
  FROM MOVIES
  WHERE Year_of_Movie >= :$MovieYear
  GROUP BY Year_of_Movie
  HAVING SUM(Sold_Tickets) < :$MinSoldTickets
  ORDER BY 1
  INTO :aMovieYear, :aSoldTickets;
```

Fin SQL

```

\ Initialisation du reste des colonnes de list box pour visualiser l'information
TABLEAU TEXTE(aTitles;Taille tableau(aMovieYear))
TABLEAU TEXTE(aDirectors;Taille tableau(aMovieYear))
TABLEAU TEXTE(aMedias;Taille tableau(aMovieYear))
TABLEAU ENTIER LONG(aNrActors;Taille tableau(aMovieYear))

```

- En utilisant les commandes SQL génériques :

```

C_TEXTE($tQueryTxt)
TABLEAU ENTIER(aMovieYear;0)
TABLEAU ENTIER LONG(aSoldTickets;0)
C_ENTIER LONG($MovieYear;$MinSoldTickets)

$MovieYear:=1979
$MinSoldTickets:=10000000
SQL LOGIN(SQL_INTERNAL;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, SUM(Sold_Tickets)"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= :$MovieYear"
$tQueryTxt:=$tQueryTxt+" GROUP BY Year_of_Movie"
$tQueryTxt:=$tQueryTxt+" HAVING SUM(Sold_Tickets) < :$MinSoldTickets"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"
SQL EXECUTER($tQueryTxt;aMovieYear;aSoldTickets)
SQL CHARGER ENREGISTREMENT(SQL Tous les enregistrements)
SQL LOGOUT
\ Initialisation du reste des colonnes de list box pour visualiser l'information
TABLEAU TEXTE(aTitles;Taille tableau(aMovieYear))
TABLEAU TEXTE(aDirectors;Taille tableau(aMovieYear))
TABLEAU TEXTE(aMedias;Taille tableau(aMovieYear))
TABLEAU ENTIER LONG(aNrActors;Taille tableau(aMovieYear))

```

- En utilisant la commande **CHERCHER PAR SQL** :

```

C_TEXTE($tQueryTxt)
TABLEAU ENTIER(aMovieYear;0)
TABLEAU ENTIER LONG(aSoldTickets;0)
C_ENTIER LONG($MovieYear;$MinSoldTickets;$vCrtMovieYear;$vInd;$i)

REDUIRE SELECTION([MOVIES];0)
$MovieYear:=1979
$MinSoldTickets:=10000000
CHERCHER PAR SQL([MOVIES];"Year_of_Movie >= :$MovieYear")
TRIER([MOVIES];[MOVIES]Year_of_Movie;>)
$vCrtMovieYear:=0
$vInd:=Taille tableau(aMovieYear)
Boucle($i;1;Enregistrements trouves([MOVIES]))
  Si([MOVIES]Year_of_Movie# $vCrtMovieYear)
    $vCrtMovieYear:=[MOVIES]Year_of_Movie
    Si(aSoldTickets{$vInd}<$MinSoldTickets)
      $vInd:=$vInd+1
      INSERER DANS TABLEAU(aMovieYear;$vInd;1)
      aMovieYear{$vInd}:=$vCrtMovieYear
      INSERER DANS TABLEAU(aSoldTickets;$vInd;1)
    Sinon
      aSoldTickets{$vInd}:=0
    Fin de si
  Fin de si
  aSoldTickets{$vInd}:=aSoldTickets{$vInd}+[MOVIES]Sold_Tickets
  ENREGISTREMENT SUIVANT([MOVIES])
Fin de boucle
Si(aSoldTickets{$vInd}>=$MinSoldTickets)
  SUPPRIMER DANS TABLEAU(aSoldTickets;$vInd;1)

```

```

SUPPRIMER DANS TABLEAU (aMovieYear;$vInd;1)
Fin de si
  \ Initialisation du reste des colonnes de list box pour visualiser l'information
TABLEAU TEXTE (aTitles;Taille tableau(aMovieYear))
TABLEAU TEXTE (aDirectors;Taille tableau(aMovieYear))
TABLEAU TEXTE (aMedias;Taille tableau(aMovieYear))
TABLEAU ENTIER LONG (aNrActors;Taille tableau(aMovieYear))

```

- En utilisant la commande SQL dynamique *EXECUTE IMMEDIATE* :

```

C_TEXTE ($tQueryTxt)
TABLEAU ENTIER (aMovieYear;0)
TABLEAU ENTIER LONG (aSoldTickets;0)
C_ENTIER LONG ($MovieYear;$MinSoldTickets)

$MovieYear:=1979
$MinSoldTickets:=10000000
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, SUM(Sold_Tickets)"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= :$MovieYear"
$tQueryTxt:=$tQueryTxt+" GROUP BY Year_of_Movie"
$tQueryTxt:=$tQueryTxt+" HAVING SUM(Sold_Tickets) < :$MinSoldTickets"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"
$tQueryTxt:=$tQueryTxt+" INTO :aMovieYear, :aSoldTickets;"
Debut SQL
  EXECUTE IMMEDIATE :$tQueryTxt;
Fin SQL
  \ Initialisation du reste des colonnes de list box pour visualiser l'information
TABLEAU TEXTE (aTitles;Taille tableau(aMovieYear))
TABLEAU TEXTE (aDirectors;Taille tableau(aMovieYear))
TABLEAU TEXTE (aMedias;Taille tableau(aMovieYear))
TABLEAU ENTIER LONG (aNrActors;Taille tableau(aMovieYear))

```

Pour tester ces exemples, lancez la base "4D SQL Code Samples" et affichez la boîte de dialogue principale. Choisissez le mode d'interrogation du moteur de 4D et cliquez sur le bouton **Clause HAVING**.

📌 Appeler des méthodes 4D depuis le code SQL

Nous souhaitons effectuer des recherches relatives aux acteurs des films. Plus précisément, nous voulons trouver tous les films auxquels ont participé au moins 7 acteurs. Les résultats devront être triés par année.

Pour cela, nous allons utiliser une fonction 4D (Find_Nr_Of_Actors) qui, à partir d'un numéro d'ID de film reçu en paramètre, retourne le nombre d'acteurs qui ont joué dans ce film :

```
`(F) Find_Nr_Of_Actors
C_ENTIER LONG ($0;$1;$vMovie_ID)
$vMovie_ID:= $1
CHERCHER([MOVIE_ACTOR];[MOVIE_ACTOR]Movie_ID=$vMovie_ID)<tt>
$0:=Enregistrements trouvés([MOVIE_ACTOR</tt>
```

Voici le code 4D correspondant à notre requête :

```
TABLEAU ENTIER LONG(aSoldTickets;0)
TABLEAU ENTIER(aMovieYear;0)
TABLEAU ENTIER LONG(aNrActors;0)
TABLEAU TEXTE(aTitles;0)
TABLEAU TEXTE(aDirectors;0)
TABLEAU TEXTE(aMedias;0)
C_ENTIER LONG($NrOfActors;$i;$vInd)

$vInd:=0
$NrOfActors:=7
TOUT SELECTIONNER([MOVIES])
Boucle($i;1;Enregistrements trouvés([MOVIES]))
  $vCrtActors:=Find_Nr_Of_Actors([MOVIES]ID)
  Si($vCrtActors>=$NrOfActors)
    $vInd:=$vInd+1
    INSERER DANS TABLEAU(aMovieYear;$vInd;1)
    aMovieYear{$vInd}:=[MOVIES]Year_of_Movie
    INSERER DANS TABLEAU(aTitles;$vInd;1)
    aTitles{$vInd}:=[MOVIES]Title
    INSERER DANS TABLEAU(aDirectors;$vInd;1)
    aDirectors{$vInd}:=[MOVIES]Director
    INSERER DANS TABLEAU(aMedias;$vInd;1)
    aMedias{$vInd}:=[MOVIES]Media
    INSERER DANS TABLEAU(aSoldTickets;$vInd;1)
    aSoldTickets{$vInd}:=[MOVIES]Sold_Tickets
    INSERER DANS TABLEAU(aNrActors;$vInd;1)
    aNrActors{$vInd}:=$vCrtActors
  Fin de si
ENREGISTREMENT SUIVANT([MOVIES])
Fin de boucle
TRIER TABLEAU(aMovieYear;aTitles;aDirectors;aMedias;aSoldTickets;aNrActors;>)
```

- En utilisant du code SQL :

```
TABLEAU ENTIER LONG(aSoldTickets;0)
TABLEAU ENTIER(aMovieYear;0)
TABLEAU ENTIER LONG(aNrActors;0)
TABLEAU TEXTE(aTitles;0)
TABLEAU TEXTE(aDirectors;0)
TABLEAU TEXTE(aMedias;0)
C_ENTIER LONG($NrOfActors;$i;$vInd)

$vInd:=0
```



```

$NrOfActors:=7
Debut SQL
    SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets, {fn Find_Nr_Of_Actors(ID)
AS NUMERIC}
    FROM MOVIES
    WHERE {fn Find_Nr_Of_Actors(ID) AS NUMERIC} >= :$NrOfActors
    ORDER BY 1
    INTO :aMovieYear, :aTitles, :aDirectors, :aMedias, :aSoldTickets, :aNrActors;
Fin SQL

```

Vous pouvez constater que nous pouvons appeler une fonction 4D à l'intérieur d'une requête SQL via la syntaxe :
{fn NomFonction4D(paramètre) AS TypeRésultatFonction4D}.

- En utilisant les commandes SQL génériques :

```

TABLEAU ENTIER LONG(aSoldTickets;0)
TABLEAU ENTIER(aMovieYear;0)
TABLEAU ENTIER LONG(aNrActors;0)
TABLEAU TEXTE(aTitles;0)
TABLEAU TEXTE(aDirectors;0)
TABLEAU TEXTE(aMedias;0)
C_ENTIER LONG($NrOfActors;$i;$vInd)
C_TEXTE($tQueryTxt)

$vInd:=0
$NrOfActors:=7
SQL LOGIN(SQL_INTERNAL;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets, {fn
Find_Nr_Of_Actors(ID) AS NUMERIC}"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE {fn Find_Nr_Of_Actors(ID) AS NUMERIC} >= :$NrOfActors"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"
SQL EXECUTER($tQueryTxt;aMovieYear;aTitles;aDirectors;aMedias;aSoldTickets;aNrActors)
SQL CHARGER ENREGISTREMENT(SQL Tous les enregistrements)
SQL LOGOUT

```

- En utilisant la commande **CHERCHER PAR SQL** :

```

TABLEAU ENTIER LONG(aSoldTickets;0)
TABLEAU ENTIER(aMovieYear;0)
TABLEAU ENTIER LONG(aNrActors;0)
TABLEAU TEXTE(aTitles;0)
TABLEAU TEXTE(aDirectors;0)
TABLEAU TEXTE(aMedias;0)
C_ENTIER LONG($NrOfActors;$i;$vInd)

$vInd:=0
$NrOfActors:=7
CHERCHER PAR SQL([MOVIES];"{fn Find_Nr_Of_Actors(ID) AS NUMERIC} >= :$NrOfActors")
Boucle($i;1;Enregistrements trouves([MOVIES]))
    $vInd:=$vInd+1
    INSERER DANS TABLEAU(aMovieYear;$vInd;1)
    aMovieYear{$vInd}:=[MOVIES]Year_of_Movie
    INSERER DANS TABLEAU(aTitles;$vInd;1)
    aTitles{$vInd}:=[MOVIES]Title
    INSERER DANS TABLEAU(aDirectors;$vInd;1)
    aDirectors{$vInd}:=[MOVIES]Director
    INSERER DANS TABLEAU(aMedias;$vInd;1)
    aMedias{$vInd}:=[MOVIES]Media
    INSERER DANS TABLEAU(aSoldTickets;$vInd;1)
    aSoldTickets{$vInd}:=[MOVIES]Sold_Tickets
    INSERER DANS TABLEAU(aNrActors;$vInd;1)
    aNrActors{$vInd}:=Find_Nr_Of_Actors([MOVIES]ID)

```

```
ENREGISTREMENT SUIVANT ([MOVIES])
Fin de boucle
TRIER TABLEAU (aMovieYear;aTitles;aDirectors;aMedias;aSoldTickets;aNrActors;>)
```

- En utilisant la commande SQL dynamique *EXECUTE IMMEDIATE* :

```
TABLEAU ENTIER LONG (aSoldTickets;0)
TABLEAU ENTIER (aMovieYear;0)
TABLEAU ENTIER LONG (aNrActors;0)
TABLEAU TEXTE (aTitles;0)
TABLEAU TEXTE (aDirectors;0)
TABLEAU TEXTE (aMedias;0)
C_ENTIER LONG ($NrOfActors;$i;$vInd)
C_TEXTE ($tQueryTxt)

$vInd:=0
$NrOfActors:=7
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets, {fn
Find_Nr_Of_Actors(ID) AS NUMERIC}"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE {fn Find_Nr_Of_Actors(ID) AS NUMERIC} >= :$NrOfActors"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"
$tQueryTxt:=$tQueryTxt+" INTO :aMovieYear, :aTitles, :aDirectors, :aMedias,
:aSoldTickets, "+" :aNrActors;"
Debut SQL
EXECUTE IMMEDIATE :$tQueryTxt;
Fin SQL
```

Pour tester ces exemples, lancez la base "4D SQL Code Samples" et affichez la boîte de dialogue principale. Choisissez le mode d'interrogation du moteur de 4D et cliquez sur le bouton **Appel de méthode 4D**.

Nous souhaitons récupérer la ville de naissance de chaque acteur. La liste des acteurs est stockée dans la table ACTORS et la liste des villes dans la table CITIES. Pour exécuter cette requête nous devons effectuer une jointure sur deux tables : ACTORS et CITIES.

Voici le code 4D correspondant à cette requête :

```
TABLEAU ENTIER LONG(aSoldTickets;0)
TABLEAU TEXTE(aTitles;0)
TABLEAU TEXTE(aDirectors;0)
C_ENTIER LONG($i;$vInd)

$vInd:=0
TOUT SELECTIONNER([ACTORS])
Boucle($i;1;Enregistrements trouvés([ACTORS]))
    $vInd:=$vInd+1
    INSERER DANS TABLEAU(aTitles;$vInd;1)
    aTitles{$vInd}:=[ACTORS]FirstName+" "+[ACTORS]LastName
    CHARGER SUR LIEN([ACTORS]Birth_City_ID)
    INSERER DANS TABLEAU(aDirectors;$vInd;1)
    aDirectors{$vInd}:=[CITIES]City_Name
    ENREGISTREMENT SUIVANT([ACTORS])
Fin de boucle
    \ Initialisation du reste des colonnes de list box pour visualiser l'information
TABLEAU ENTIER(aMovieYear;Taille tableau(aTitles))
TABLEAU TEXTE(aMedias;Taille tableau(aTitles))
TABLEAU ENTIER LONG(aSoldTickets;Taille tableau(aTitles))
TABLEAU ENTIER LONG(aNrActors;Taille tableau(aTitles))
TABLEAU MULTI TRI(aDirectors;>;aTitles;>;aMovieYear;aMedias;aSoldTickets;aNrActors)
```

- En utilisant du code SQL :

```
TABLEAU TEXTE(aTitles;0)
TABLEAU TEXTE(aDirectors;0)

Debut SQL
    SELECT CONCAT(CONCAT(ACTORS.FirstName,' '),ACTORS.LastName), CITIES.City_Name
    FROM ACTORS, CITIES
    WHERE ACTORS.Birth_City_ID=CITIES.City_ID
    ORDER BY 2,1
    INTO :aTitles, :aDirectors;
Fin SQL
    \ Initialisation du reste des colonnes de list box pour visualiser l'information
TABLEAU ENTIER(aMovieYear;Taille tableau(aTitles))
TABLEAU TEXTE(aMedias;Taille tableau(aTitles))
TABLEAU ENTIER LONG(aSoldTickets;Taille tableau(aTitles))
TABLEAU ENTIER LONG(aNrActors;Taille tableau(aTitles))
```

- En utilisant les commandes SQL génériques :

```
TABLEAU TEXTE(aTitles;0)
TABLEAU TEXTE(aDirectors;0)
C_TEXTE($tQueryTxt)

SQL LOGIN(SQL_INTERNAL;"", "")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT CONCAT(CONCAT(ACTORS.FirstName,' '),ACTORS.LastName),
```

```

CITIES.City_Name"
$tQueryTxt:=$tQueryTxt+" FROM ACTORS, CITIES"
$tQueryTxt:=$tQueryTxt+" WHERE ACTORS.Birth_City_ID=CITIES.City_ID"
$tQueryTxt:=$tQueryTxt+" ORDER BY 2,1"
SQL EXECUTER($tQueryTxt;aTitles;aDirectors)
SQL CHARGER ENREGISTREMENT (SQL Tous les enregistrements)
SQL LOGOUT
  ` Initialisation du reste des colonnes de list box pour visualiser l'information
TABLEAU ENTIER (aMovieYear;Taille tableau(aTitles))
TABLEAU TEXTE (aMedias;Taille tableau(aTitles))
TABLEAU ENTIER LONG (aSoldTickets;Taille tableau(aTitles))
TABLEAU ENTIER LONG (aNrActors;Taille tableau(aTitles))

```

- Il n'est pas possible d'utiliser **CHERCHER PAR SQL** dans ce contexte car cette commande ne peut pas recevoir plus d'une table en paramètre.
- En utilisant la commande SQL dynamique **EXECUTE IMMEDIATE** :

```

TABLEAU TEXTE (aTitles;0)
TABLEAU TEXTE (aDirectors;0)
C_TEXTE ($tQueryTxt)

$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT CONCAT(CONCAT(ACTORS.FirstName,' '),ACTORS.LastName),
CITIES.City_Name"
$tQueryTxt:=$tQueryTxt+" FROM ACTORS, CITIES"
$tQueryTxt:=$tQueryTxt+" WHERE ACTORS.Birth_City_ID=CITIES.City_ID"
$tQueryTxt:=$tQueryTxt+" ORDER BY 2,1"
$tQueryTxt:=$tQueryTxt+" INTO :aTitles, :aDirectors"
Debut SQL
  EXECUTE IMMEDIATE :$tQueryTxt;
Fin SQL
  ` Initialisation du reste des colonnes de list box pour visualiser l'information
TABLEAU ENTIER (aMovieYear;Taille tableau(aTitles))
TABLEAU TEXTE (aMedias;Taille tableau(aTitles))
TABLEAU ENTIER LONG (aSoldTickets;Taille tableau(aTitles))
TABLEAU ENTIER LONG (aNrActors;Taille tableau(aTitles))

```

Pour tester ces exemples, lancez la base "4D SQL Code Samples" et affichez la boîte de dialogue principale. Choisissez le mode d'interrogation du moteur de 4D et cliquez sur le bouton **Jointures**.

✚ Utiliser des alias

Lorsqu'une requête SQL est trop complexe ou contient des libellés trop longs la rendant difficile à lire, il est possible d'utiliser des alias afin d'améliorer sa lisibilité. Dans cette section, nous allons reprendre l'exemple de la section précédente (**Jointures**) en utilisant deux alias : Act pour la table ACTORS et Cit pour la table CITIES.

Voici le code 4D initial :

```
TABLEAU ENTIER LONG(aSoldTickets;0)
TABLEAU TEXTE(aTitles;0)
TABLEAU TEXTE(aDirectors;0)
C_ENTIER LONG($i;$vInd)

$vInd:=0
TOUT SELECTIONNER([ACTORS])
Boucle($i;1;Enregistrements trouvés([ACTORS]))
  $vInd:=$vInd+1
  INSERER DANS TABLEAU(aTitles;$vInd;1)
  aTitles{$vInd}:=[ACTORS]FirstName+" "+[ACTORS]LastName
  CHARGER SUR LIEN([ACTORS]Birth_City_ID)
  INSERER DANS TABLEAU(aDirectors;$vInd;1)
  aDirectors{$vInd}:=[CITIES]City_Name
  ENREGISTREMENT SUIVANT([ACTORS])
Fin de boucle
  \ Initialisation du reste des colonnes de list box pour visualiser l'information
TABLEAU ENTIER(aMovieYear;Taille tableau(aTitles))
TABLEAU TEXTE(aMedias;Taille tableau(aTitles))
TABLEAU ENTIER LONG(aSoldTickets;Taille tableau(aTitles))
TABLEAU ENTIER LONG(aNrActors;Taille tableau(aTitles))
TABLEAU MULTI TRI(aDirectors;>;aTitles;>;aMovieYear;aMedias;aSoldTickets;aNrActors)
```

- En utilisant du code SQL :

```
TABLEAU TEXTE(aTitles;0)
TABLEAU TEXTE(aDirectors;0)

Debut SQL
  SELECT CONCAT(CONCAT(ACTORS.FirstName, ' '),ACTORS.LastName), CITIES.City_Name
  FROM ACTORS AS 'Act', CITIES AS 'Cit'
  WHERE Act.Birth_City_ID=Cit.City_ID
  ORDER BY 2,1
  INTO :aTitles, :aDirectors;
Fin SQL
  \ Initialisation du reste des colonnes de list box pour visualiser l'information
TABLEAU ENTIER(aMovieYear;Taille tableau(aTitles))
TABLEAU TEXTE(aMedias;Taille tableau(aTitles))
TABLEAU ENTIER LONG(aSoldTickets;Taille tableau(aTitles))
TABLEAU ENTIER LONG(aNrActors;Taille tableau(aTitles))
```

- En utilisant les commandes SQL génériques :

```
TABLEAU TEXTE(aTitles;0)
TABLEAU TEXTE(aDirectors;0)
C_TEXTE($tQueryTxt)

SQL LOGIN(SQL_INTERNAL;"","")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT CONCAT(CONCAT(ACTORS.FirstName, ' '),ACTORS.LastName),
```

```

CITIES.City_Name"
$queryTxt:=$queryTxt+" FROM ACTORS AS 'Act', CITIES AS 'Cit'"
$queryTxt:=$queryTxt+" WHERE Act.Birth_City_ID=Cit.City_ID"
$queryTxt:=$queryTxt+" ORDER BY 2,1"
SQL EXECUTER($queryTxt;aTitles;aDirectors)
SQL CHARGER ENREGISTREMENT (SQL Tous les enregistrements)
SQL LOGOUT
  ` Initialisation du reste des colonnes de list box pour visualiser l'information
TABLEAU ENTIER (aMovieYear;Taille tableau(aTitles))
TABLEAU TEXTE (aMedias;Taille tableau(aTitles))
TABLEAU ENTIER LONG (aSoldTickets;Taille tableau(aTitles))
TABLEAU ENTIER LONG (aNrActors;Taille tableau(aTitles))

```

- Il n'est pas possible d'utiliser **CHERCHER PAR SQL** dans ce contexte car cette commande ne peut pas recevoir plus d'une table en paramètre.
- En utilisant la commande SQL dynamique **EXECUTE IMMEDIATE** :

```

TABLEAU TEXTE (aTitles;0)
TABLEAU TEXTE (aDirectors;0)
C_TEXTE ($queryTxt)

$queryTxt:=""
$queryTxt:=$queryTxt+"SELECT CONCAT(CONCAT(ACTORS.FirstName,' '),ACTORS.LastName),
CITIES.City_Name"
$queryTxt:=$queryTxt+" FROM ACTORS AS 'Act', CITIES AS 'Cit'"
$queryTxt:=$queryTxt+" WHERE Act.Birth_City_ID=Cit.City_ID"
$queryTxt:=$queryTxt+" ORDER BY 2,1"
$queryTxt:=$queryTxt+" INTO :aTitles, :aDirectors"
Debut SQL
  EXECUTE IMMEDIATE :$queryTxt;
Fin SQL
TABLEAU ENTIER (aMovieYear;Taille tableau(aTitles))
TABLEAU TEXTE (aMedias;Taille tableau(aTitles))
TABLEAU ENTIER LONG (aSoldTickets;Taille tableau(aTitles))
TABLEAU ENTIER LONG (aNrActors;Taille tableau(aTitles))

```

Pour tester ces exemples, lancez la base "4D SQL Code Samples" et affichez la boîte de dialogue principale. Choisissez le mode d'interrogation du moteur de 4D et cliquez sur le bouton **Utilisation d'alias**.

📌 Sous-requêtes

Nous souhaitons à présent obtenir une statistique par rapport aux nombres d'entrées : quels sont les films ayant réalisé plus d'entrées que la moyenne de tous les films. Pour exécuter cette requête en SQL, nous allons utiliser une requête incluse dans une requête, autrement dit une sous-requête.

Voici le code 4D correspondant à cette requête :

```
TABLEAU ENTIER LONG(aSoldTickets;0)
TABLEAU TEXTE(aTitles;0)
C_ENTIER LONG($i;$vInd;$vAvgSoldTickets)

$vInd:=0
TOUT SELECTIONNER([MOVIES])
$vAvgSoldTickets:=Moyenne([MOVIES]Sold_Tickets)
Boucle($i;1;Enregistrements trouvés([MOVIES]))
  Si([MOVIES]Sold_Tickets>$vAvgSoldTickets)
    $vInd:=$vInd+1
    INSERER DANS TABLEAU(aTitles;$vInd;1)
    aTitles{$vInd}:=[MOVIES]Title
    INSERER DANS TABLEAU(aSoldTickets;$vInd;1)
    aSoldTickets{$vInd}:=[MOVIES]Sold_Tickets
  Fin de si
  ENREGISTREMENT SUIVANT([MOVIES])
Fin de boucle
` Initialisation du reste des colonnes de list box pour visualiser l'information
TABLEAU ENTIER(aMovieYear;Taille tableau(aTitles))
TABLEAU TEXTE(aDirectors;Taille tableau(aTitles))
TABLEAU TEXTE(aMedias;Taille tableau(aTitles))
TABLEAU ENTIER LONG(aNrActors;Taille tableau(aTitles))
TRIER TABLEAU(aTitles;aDirectors;aMovieYear;aMedias;aSoldTickets;aNrActors;>)
```

- En utilisant du code SQL :

```
TABLEAU ENTIER LONG(aSoldTickets;0)
TABLEAU TEXTE(aTitles;0)
Debut SQL
  SELECT Title, Sold_Tickets
  FROM MOVIES
  WHERE Sold_Tickets > (SELECT AVG(Sold_Tickets) FROM MOVIES)
  ORDER BY 1
  INTO :aTitles, :aSoldTickets;
Fin SQL
` Initialisation du reste des colonnes de list box pour visualiser l'information
TABLEAU ENTIER(aMovieYear;Taille tableau(aTitles))
TABLEAU TEXTE(aDirectors;Taille tableau(aTitles))
TABLEAU TEXTE(aMedias;Taille tableau(aTitles))
TABLEAU ENTIER LONG(aNrActors;Taille tableau(aTitles))
TRIER TABLEAU(aTitles;aDirectors;aMovieYear;aMedias;aSoldTickets;aNrActors;>)
```

- En utilisant les commandes SQL génériques :

```
TABLEAU ENTIER LONG(aSoldTickets;0)
TABLEAU TEXTE(aTitles;0)
C_TEXTE($tQueryTxt)

SQL LOGIN(SQL_INTERNAL;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Title, Sold_Tickets"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
```

```

$tQueryTxt:=$tQueryTxt+" WHERE Sold_Tickets > (SELECT AVG(Sold_Tickets) FROM MOVIES)"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"
SQL EXECUTER($tQueryTxt;aTitles;aSoldTickets)
SQL CHARGER ENREGISTREMENT (SQL Tous les enregistrements)
SQL LOGOUT
  ` Initialisation du reste des colonnes de list box pour visualiser l'information
TABLEAU ENTIER(aMovieYear;Taille tableau(aTitles))
TABLEAU TEXTE(aDirectors;Taille tableau(aTitles))
TABLEAU TEXTE(aMedias;Taille tableau(aTitles))
TABLEAU ENTIER LONG(aNrActors;Taille tableau(aTitles))
TRIER TABLEAU(aTitles;aDirectors;aMovieYear;aMedias;aSoldTickets;aNrActors;>)

```

- En utilisant la commande **CHERCHER PAR SQL** :

```

TABLEAU ENTIER LONG(aSoldTickets;0)
TABLEAU TEXTE(aTitles;0)

CHERCHER PAR SQL([MOVIES];"Sold_Tickets > (SELECT AVG(Sold_Tickets) FROM MOVIES)")
TRIER([MOVIES];[MOVIES]Title;>)
SELECTION VERS TABLEAU([MOVIES]Title;aTitles;[MOVIES]Sold_Tickets;aSoldTickets)
  ` Initialisation du reste des colonnes de list box pour visualiser l'information
TABLEAU ENTIER(aMovieYear;Taille tableau(aTitles))
TABLEAU TEXTE(aDirectors;Taille tableau(aTitles))
TABLEAU TEXTE(aMedias;Taille tableau(aTitles))
TABLEAU ENTIER LONG(aNrActors;Taille tableau(aTitles))
TRIER TABLEAU(aTitles;aDirectors;aMovieYear;aMedias;aSoldTickets;aNrActors;>)

```

- En utilisant la commande SQL dynamique **EXECUTE IMMEDIATE** :

```

TABLEAU ENTIER LONG(aSoldTickets;0)
TABLEAU TEXTE(aTitles;0)
C_TEXTE($tQueryTxt)

$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Title, Sold_Tickets"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Sold_Tickets > (SELECT AVG(Sold_Tickets) FROM MOVIES)"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"
$tQueryTxt:=$tQueryTxt+" INTO :aTitles, :aSoldTickets"
Debut SQL
  EXECUTE IMMEDIATE :$tQueryTxt;
Fin SQL
  ` Initialisation du reste des colonnes de list box pour visualiser l'information
TABLEAU ENTIER(aMovieYear;Taille tableau(aTitles))
TABLEAU TEXTE(aDirectors;Taille tableau(aTitles))
TABLEAU TEXTE(aMedias;Taille tableau(aTitles))
TABLEAU ENTIER LONG(aNrActors;Taille tableau(aTitles))

```

Pour tester ces exemples, lancez la base "4D SQL Code Samples" et affichez la boîte de dialogue principale. Choisissez le mode d'interrogation du moteur de 4D et cliquez sur le bouton **Sous-requêtes**.

Tracer et déboguer le code SQL

Dans 4D, il existe deux principales manières de tracer et de déboguer votre code : soit utiliser le **Débogueur** pour tracer et corriger les erreurs, soit utiliser la commande **APPELER SUR ERREUR** pour intercepter et traiter les erreurs de façon appropriée.

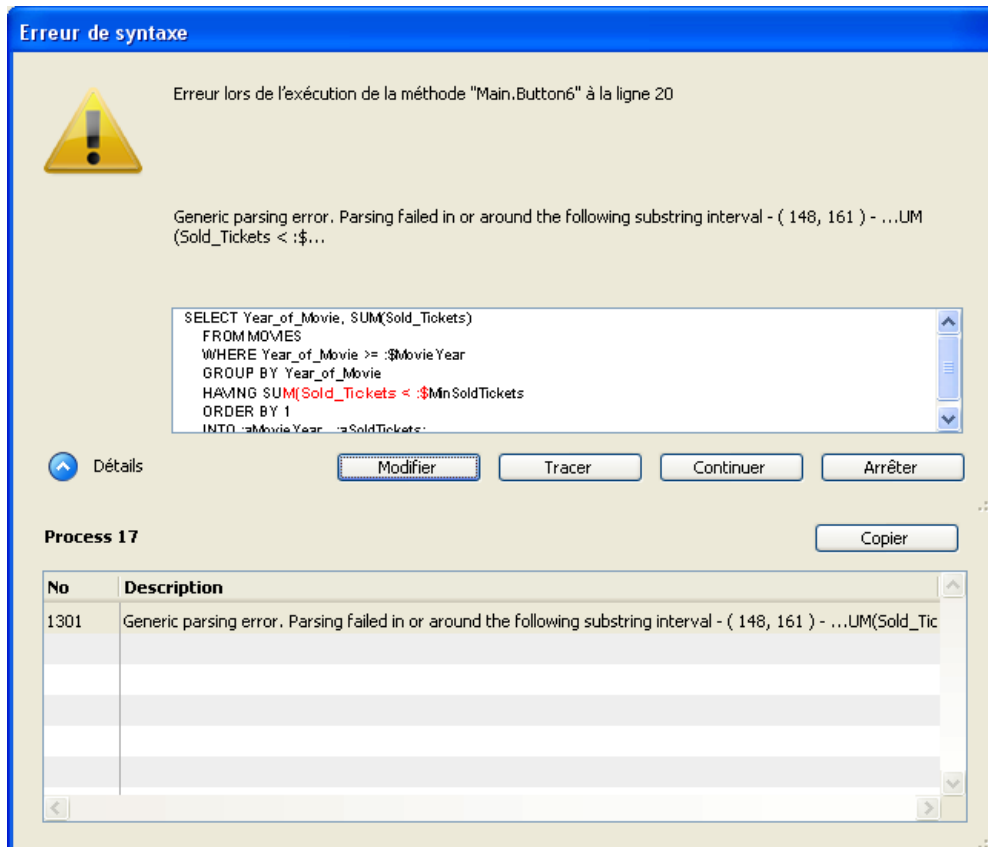
Ces deux techniques peuvent être mises en oeuvre avec le code SQL.

Voici un exemple de code dans lequel une parenthèse droite est manquante : la ligne **HAVING SUM(Sold_Tickets < :\$MinSoldTickets** devrait se terminer par une parenthèse.

```
TABLEAU ENTIER LONG (aSoldTickets;0)
TABLEAU ENTIER (aMovieYear;0)
C_ENTIER LONG ($MovieYear;$MinSoldTickets;SQL_Error)
$MovieYear:=1979
$MinSoldTickets:=10000000
SQL_Error:=0

` Installation de la méthode SQL_Error_Handler pour intercepter les erreurs
APPELER SUR ERREUR ("SQL_Error_Handler")
Debut SQL
  SELECT Year_of_Movie, SUM(Sold_Tickets)
  FROM MOVIES
  WHERE Year_of_Movie >= :$MovieYear
  GROUP BY Year_of_Movie
  HAVING SUM(Sold_Tickets < :$MinSoldTickets
  ORDER BY 1
  INTO :aMovieYear, :aSoldTickets;
Fin SQL
```

Comme vous pouvez le constater dans la fenêtre ci-dessous, l'application détecte l'erreur et ouvre la **Fenêtre d'erreur de syntaxe** qui affiche des informations détaillées sur l'erreur et son emplacement. Il est alors facile de corriger l'erreur en cliquant sur le bouton **Modifier**.



Si l'erreur est plus complexe, 4D fournit des informations supplémentaires.

Pour tester l'exemple ci-dessus, dans la boîte de dialogue principale de la base "4D SQL Code Samples", cliquez sur le bouton

Debugguer le code SQL.

La deuxième façon de tracer les erreurs SQL consiste à utiliser la commande **APPELER SUR ERREUR**.

Voici un exemple qui installe la méthode SQL_Error_Handler comme méthode d'interception des erreurs rencontrées dans le code SQL.

```
TABLEAU ENTIER LONG (aSoldTickets;0)
TABLEAU ENTIER (aMovieYear;0)
C_ENTIER LONG ($MovieYear;$MinSoldTickets;SQL_Error)
$MovieYear:=1979
$MinSoldTickets:=10000000
SQL_Error:=0

` Installation de la méthode SQL_Error_Handler pour intercepter les erreurs
APPELER SUR ERREUR ("SQL_Error_Handler")
Debut SQL
  SELECT Year_of_Movie, SUM(Sold_Tickets)
  FROM MOVIES
  WHERE Year_of_Movie >= :$MovieYear
  GROUP BY Year_of_Movie
  HAVING SUM(Sold_Tickets < :$MinSoldTickets)
  ORDER BY 1
  INTO :aMovieYear, :aSoldTickets;
Fin SQL
` Désinstallation de la méthode SQL_Error_Handler
APPELER SUR ERREUR ("")

Si (SQL_Error#0)
  ALERTE ("SQL Error number: "+Chaine (MySQL_Error))
Fin de si
```

La méthode SQL_Error_Handler contient le code suivant :

```
` (P) SQL_Error_Handler
MySQL_Error:=Error
```

Pour tester l'exemple ci-dessus, dans la boîte de dialogue principale de la base "4D SQL Code Samples", cliquez sur le bouton **Utiliser "Appeler sur erreur"**.

Data Definition Language

Les commandes DDL (Data Definition Language) du SQL vous permettent de définir et de gérer la structure de la base. Avec les commandes DDL, vous pouvez créer ou modifier des tables et des champs, ainsi qu'ajouter et/ou supprimer des données.

Cet exemple simple crée une table, ajoute quelques champs puis remplit ces champs avec des données.

Debut SQL

```
DROP TABLE IF EXISTS ACTOR_FANS;

CREATE TABLE ACTOR_FANS

(ID INT32,
Name VARCHAR);

INSERT INTO ACTOR_FANS
(ID, Name)
VALUES (1, 'Francis');

ALTER TABLE ACTOR_FANS
ADD Phone_Number VARCHAR;

INSERT INTO ACTOR_FANS
(ID, Name, Phone_Number)
VALUES (2, 'Florence', '01446677888');
```

Fin SQL

Pour tester cet exemple, dans la boîte de dialogue principale de la base "4D SQL Code Samples", cliquez sur le bouton **DDL**.

Note : Cet exemple ne fonctionnera qu'une fois car si vous cliquez une seconde fois sur le bouton "DDL", un message d'erreur vous signalera que la table existe déjà.

🌿 Connexions externes

4D vous permet d'utiliser des bases de données externes, c'est-à-dire d'exécuter des requêtes SQL sur d'autres bases que la base locale. Pour cela, vous pouvez vous connecter à toute source de données externe via ODBC ou directement à d'autres bases 4D.

Voici les commandes permettant la connexion à des bases de données externes :

- **LISTE SOURCES DONNEES** retourne la liste des sources de données ODBC installées sur la machine.
- **Lire source donnees courante** indique la source de données ODBC utilisée par l'application.
- **SQL LOGIN** vous permet de vous connecter à une base externe directement ou via une source de données ODBC installée sur la machine.
- **SQL LOGOUT** peut être utilisée pour refermer toute connexion externe et se reconnecter à la base 4D locale.
- **USE DATABASE** (commande SQL) permet d'ouvrir une base de données 4D externe en utilisant le moteur SQL de 4D.

L'exemple suivant illustre la connexion à une source de données externe (ORACLE), la récupération de données depuis cette base ORACLE ainsi que la déconnexion de cette base et le retour à la base locale. Cet exemple suppose qu'une source de données externe valide nommée "Test_ORACLE_10g" est installée dans le système.

```
TABLEAU TEXTE(aDSN;0)
TABLEAU TEXTE(aDS_Driver;0)
C_TEXTE($CrI_DSN;$My_ORACLE_DSN)
TABLEAU TEXTE(aTitles;0)
TABLEAU ENTIER LONG(aNrActors;0)
TABLEAU ENTIER LONG(aSoldTickets;0)
TABLEAU ENTIER(aMovieYear;0)
TABLEAU TEXTE(aTitles;0)
TABLEAU TEXTE(aDirectors;0)
TABLEAU TEXTE(aMedias;0)
C_ENTIER LONG($MovieYear)
C_TEXTE($tQueryTxt)
REDUIRE SELECTION(!MOVIES;0)
$MovieYear:=1960

` Par défaut, la source de données (DSN) courante est interne : "DB4D_SQL_LOCAL" (valeur de la constante SQL_INTERNAL)
$CrI_DSN:=Lire source donnees courante
ALERTE("La DSN courante est "+$CrI_DSN)

` Faire quelque chose avec la base interne
☐ Debut SQL
    SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets
    FROM MOVIES
    WHERE Year_of_Movie >= :$MovieYear
    INTO :aMovieYear, :aTitles, :aDirectors, :aMedias, :aSoldTickets;
Fin SQL
` Récupérer les sources de données utilisateur définies dans le gestionnaire ODBC
LISTE SOURCES DONNEES(Source de données utilisateur;aDSN;aDS_Driver)
$My_ORACLE_DSN:="Test_Oracle_10g"
☐ Si (Chercher dans tableau(aDSN;$My_ORACLE_DSN)>0)
    ` Connexion entre 4D et la source $My_ORACLE_DSN="Test_Oracle_10g"

    SQL LOGIN($My_ORACLE_DSN,"scott","tiger",*)

    ` La DSN est la base ORACLE
    $CrI_DSN:=Lire source donnees courante
    ALERTE("The current DSN is "+$CrI_DSN)
    TABLEAU TEXTE(aTitles;0)
    TABLEAU ENTIER LONG(aNrActors;0)
    TABLEAU ENTIER LONG(aSoldTickets;0)
    TABLEAU ENTIER(aMovieYear;0)
    TABLEAU TEXTE(aTitles;0)
    TABLEAU TEXTE(aDirectors;0)
    TABLEAU TEXTE(aMedias;0)

    ` Faire quelque chose avec la base externe (ORACLE)
    ☐ Debut SQL
        SELECT ENAME FROM EMP INTO :aTitles
    Fin SQL

    ` Refermer la connexion ouverte avec la commande SQL LOGIN
    SQL LOGOUT
    ` La DSN courante redevient la base interne
    $CrI_DSN:=Lire source donnees courante
    ALERTE("La DSN courante est "+$CrI_DSN)
☐ Simon
    ALERTE("DSN ORACLE non installée")
Fin de si
```

Pour tester cet exemple, dans la boîte de dialogue principale de la base "4D SQL Code Samples", cliquez sur le bouton **Connexion à ORACLE**.

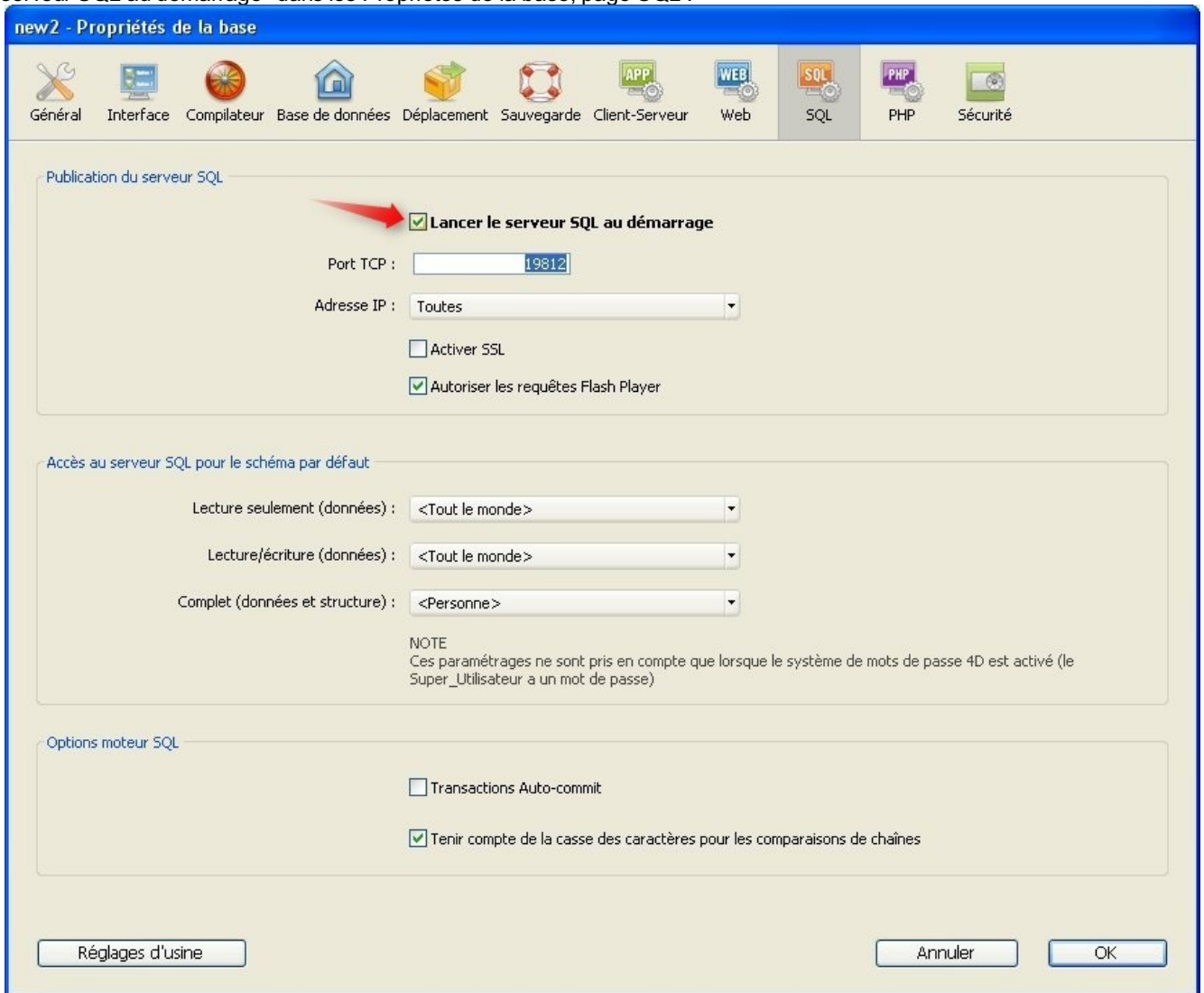
Connexion au moteur SQL 4D via le driver ODBC

Vous pouvez vous connecter au moteur SQL de 4D depuis toute base de données externe via le driver (pilote) ODBC pour 4D.

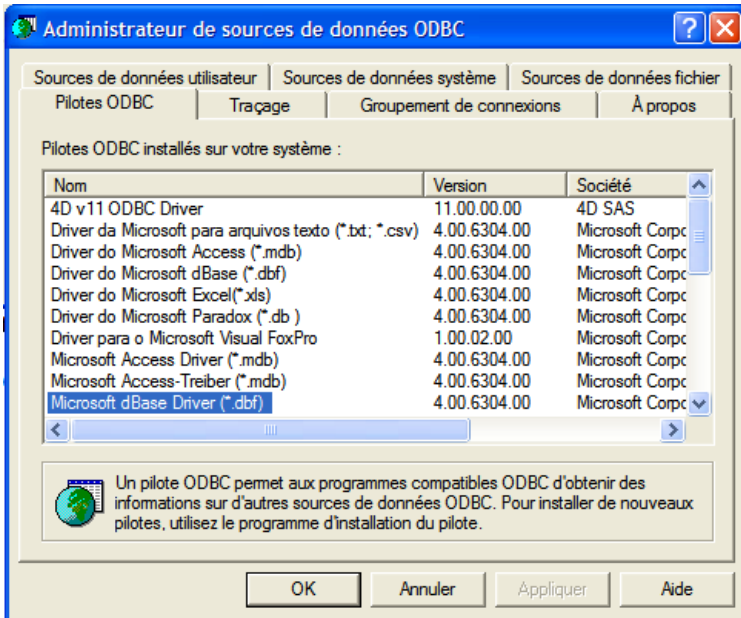
L'exemple proposé dans cette section illustre la procédure de connexion d'une base 4D à une autre base 4D via le driver ODBC :

Note : Cette configuration est utilisée à titre d'exemple. Il est possible de connecter directement des applications 4D entre elles via le SQL. Pour plus d'informations, reportez-vous à la description de la commande [SQL LOGIN](#).

1. Dupliquez la base exemple "4D SQL Code Samples".
2. Renommez les deux dossiers contenant les bases, par exemple "Client" et "Server".
3. Lancez la base du dossier "Server" et activez le lancement du serveur SQL au démarrage en cochant l'option "Lancer le serveur SQL au démarrage" dans les Propriétés de la base, page SQL :



4. Quittez puis relancez la base du dossier Server afin d'activer le serveur SQL.
5. Installez le pilote "4D ODBC Driver for 4D" puis vérifiez qu'il apparaît bien dans la boîte de dialogue d'administration de sources de données ODBC :



6. Créez une nouvelle source de données nommée "Test_ODBC_Driver_v11"



et testez-la en cliquant sur le bouton **Connection test** :

4D v11 SQL - Configure data source

Data Source Name : 4D test

Description : 4D v11 SQL datasource.

Server

Server : localhost

Port : 1919

SSL

Default user

User : Administrateur

Password :

Timeouts

Connection : 3

Login : 3

Query : 30

Connection test Cancel Ok

Data source name [DSN]

A unique name for this Data Source
Default: ""
Optional: No

7. Lancez la base exemple du dossier Client, affichez la fenêtre principale et cliquez sur le bouton "Connexion à 4D". Le code de ce bouton est le suivant :

```
C_TEXTE($CrI_DSN;$My_4D_DSN)
TABLEAU TEXTE(aDSN;0)
TABLEAU TEXTE(aDS_Driver;0)
TABLEAU TEXTE(aTitles;0)
TABLEAU TEXTE(aDirectors;0)
TABLEAU TEXTE(aMedias;0)
```

```
REDUIRE SELECTION((MOVIES);0)
```

```
` Par défaut, la source de données (DSN) courante est interne
```

```
$CrI_DSN:=Lire source donnees courante
ALERTE("La DSN courante est"+"$CrI_DSN)
```

```
` Faire quelque chose avec la base interne
```

```
▣ Debut SQL
```

```
SELECT Title, Director, Media
FROM MOVIES
ORDER BY 1
INTO :aTitles, :aDirectors, :aMedias;
```

```
Fin SQL
```



```

^ Récupérer les sources de données utilisateur définies dans le gestionnaire ODBC
LISTE SOURCES DONNEES(Source de données utilisateur ;aDSN;aDS_Driver)
$My_4D_DSN:="Test_ODBC_Driver_v11"
☐ Si (Chercher dans tableau(aDSN;$My_4D_DSN)>0)
  ^ Connexion entre 4D et une autre base 4D via le Driver ODBC 4D v11
  SQL LOGIN($My_4D_DSN;"Administrator";";")
  ☐ Si (Ok=1)
    ^ La DSN est la base 4D externe
    $CrI_DSN:=Lire source donnees courante
    ALERTE("La DSN courante est "+$CrI_DSN)

    TABLEAU TEXTE(aTitles;0)
    TABLEAU TEXTE(aDirectors;0)
    TABLEAU TEXTE(aMedias;0)

    ^ Faire quelque chose avec la base externe (4D)
    ☐ Debut SQL
      SELECT Title, Director, Media
      FROM MOVIES
      ORDER BY 1
      INTO :aTitles, :aDirectors, :aMedias;
    Fin SQL







    ^ Refermer la connexion ouverte avec la commande SQL LOGIN
    SQL LOGOUT
    ^ La DSN courante redevient la base interne
    $CrI_DSN:=Lire source donnees courante
    ALERTE("La DSN courante est "+$CrI_DSN)

  ☐ Sinon
    ALERTE("Impossible de se connecter à la source de données externe")
  Fin de si
☐ Sinon
  ALERTE("Source de données ODBC Driver introuvable")
Fin de si

```

Comme vous pouvez le constater, dans la première partie de la méthode nous effectuons une requête sur la base interne. Puis, dans la seconde partie, nous nous connectons à l'autre base 4D via le driver ODBC et effectuons la même requête. Le résultat obtenu dans les deux cas est identique, bien entendu.

Utiliser le SQL dans 4D

-  Utiliser le SQL dans 4D
-  Accéder au moteur SQL de 4D
-  Configuration du serveur SQL de 4D
-  Implémentations du moteur SQL de 4D
-  Réplication via le SQL
-  Prise en charge des jointures

Utiliser le SQL dans 4D

Ce chapitre propose une vue d'ensemble des possibilités d'utilisation du SQL dans 4D.

Il décrit les différents modes d'accès au moteur SQL intégré ainsi que les différentes façons d'exécuter des requêtes et de récupérer des données. Il explique également la configuration du serveur SQL de 4D. Enfin, il présente les principes de l'implémentation du SQL au niveau du moteur de 4D.

Envoi de requêtes au moteur SQL de 4D

Le moteur SQL intégré de 4D peut être interrogé de trois manières :

- via la commande **CHERCHER PAR SQL**. Le principe consiste à passer la clause **WHERE** d'une commande SQL **SELECT** en tant que paramètre à la commande. Exemple :

```
CHERCHER PAR SQL ([OFFICES]; "SALES > 100")
```

- via les commandes SQL intégrées de 4D, placées dans le thème "SQL" (**SQL FIXER PARAMETRE**, **SQL EXECUTER**, etc.). Ces commandes peuvent fonctionner avec une source de données ODBC ou le moteur SQL 4D de la base de données courante.
- via l'éditeur de méthodes standard de 4D. Les instructions SQL peuvent être saisies directement dans l'éditeur de méthodes de 4D. Il suffit d'insérer les requêtes SQL entre les mots-clés **Debut SQL** et **Fin SQL**. Le code compris entre ces balises ne sera pas analysé par l'interpréteur de 4D et sera exécuté par le moteur SQL (ou par un autre moteur, si défini par la commande **SQL LOGIN**).

Echanger des données entre 4D et le moteur SQL

Référencer les expressions 4D

Il est possible de faire référence à tout type d'expression 4D valide (variable, champ, tableau, expression...) au sein des clauses **WHERE** et **INTO** des expressions SQL. Pour désigner une référence 4D, vous pouvez utiliser indifféremment l'une des deux notations suivantes :

- placer la référence entre chevrons "<<" et ">>"
- faire précéder la référence de deux-points ":"

Exemples :

```
C_ALPHA (80; vNom)
vNom:=Demander ("Nom :")
SQL EXECUTER ("SELECT age FROM PEOPLE WHERE nom=<<vNom>>")
```

ou bien :

```
C_ALPHA (80; vNom)
vNom:=Demander ("Nom :")
Debut SQL
    SELECT age FROM PEOPLE WHERE name= :vNom
Fin SQL
```

Notes :

- Lorsque vous manipulez des variables interprocess (nommées <>mavar), vous devez encadrer le nom des variables avec des crochets [] (par exemple <<[<>mavar]>> ou :[<>mavar]).
- En mode compilé, vous ne pouvez pas utiliser de variables locales (débutant par le symbole \$).

Récupérer les données des requêtes SQL dans 4D

Les données issues d'une commande **SELECT** doivent être manipulées soit à l'intérieur d'un bloc **Debut SQL/Fin SQL** via la clause **INTO** de la commande **SELECT**, soit via les commandes 4D du thème "SQL" :

- Dans le cas d'une structure **Debut SQL/Fin SQL**, vous pouvez utiliser la clause **INTO** de la requête SQL pour désigner tout objet 4D valide (champ, variable ou tableau) comme destinataire des valeurs.

```
Debut SQL
    SELECT ename FROM emp INTO <<[Employés]Nom>>
Fin SQL
```

- Avec la commande **SQL EXECUTER**, vous pouvez également utiliser les paramètres supplémentaires :

```
SQL EXECUTER("SELECT ename FROM emp";[Employés]Nom)
```

La principale différence entre les deux façons de récupérer les données issues d'une requête SQL (balises **Debut SQL/Fin SQL** et commandes SQL) est que dans le premier cas toutes les informations sont retournées à 4D en une seule fois, tandis que dans le second cas chaque enregistrement doit être chargé explicitement à l'aide de la commande **SQL CHARGER ENREGISTREMENT**.

Par exemple, en supposant que la table PERSONS comporte 100 enregistrements :

- Récupération dans un tableau en utilisant les commandes SQL génériques de 4D :

```
TABLEAU ENTIER(tAnneeNaiss;0)
C_ALPHA(40;vNom)
vNom:="Smith"
$$SQLStm:="SELECT Birth_Year FROM PERSONS WHERE ename= <<vNom>>"
SQL EXECUTER($$SQLStm;tAnneeNaiss)
Tant que(Non(SQL Fin de selection))
    SQL CHARGER ENREGISTREMENT(10)
Fin tant que
```

Ici, il est donc nécessaire d'effectuer 10 boucles afin de récupérer les 100 enregistrements. Pour charger tous les enregistrements en une seule fois, il suffit d'écrire :

```
SQL CHARGER ENREGISTREMENT(SQL Tous les enregistrements)
```

- Récupération dans un tableau en utilisant les balises **Debut SQL/Fin SQL** :

```
TABLEAU ENTIER(tAnneeNaiss;0)
C_ALPHA(40;vNom)
vNom:="Smith"
Debut SQL
    SELECT Birth_Year FROM PERSONS WHERE ename= <<vNom>> INTO <<tAnneeNaiss>>
Fin SQL
```

Dans ce cas, après l'exécution de l'instruction **SELECT**, le tableau tAnneeNaiss contient 100 éléments, chaque élément stockant une année de naissance provenant des 100 enregistrements.

Si, au lieu d'un tableau nous souhaitons récupérer les données dans un champ 4D, 4D créera automatiquement autant d'enregistrements que nécessaire pour contenir toutes les valeurs. Reprenons l'exemple d'une table PERSONS contenant 100 enregistrements :

- Récupération dans un champ en utilisant les commandes SQL génériques de 4D :

```
C_ALPHA(40;vNom)
vNom:="Smith"
$$SQLStm:="SELECT Birth_Year FROM PERSONS WHERE ename= <<vNom>>"
SQL EXECUTER($$SQLStm;[Matable]Annee_Naiss)
Tant que(Non(SQL Fin de selection))
    SQL CHARGER ENREGISTREMENT(10)
Fin tant que
```

Ici, il est nécessaire d'effectuer 10 boucles afin de récupérer les 100 enregistrements. Chaque passage dans la boucle entraînera la création de 10 enregistrements dans la table [Matable] et chaque valeur Birth_Year récupérée de la table PERSONS sera stockée dans le champ Annee_Naiss.

- Récupération dans un champ en utilisant les balises **Debut SQL/FinSQL** :

```
C_ALPHA(40;vNom)
vNom:="Smith"
Debut SQL
    SELECT Birth_Year FROM PERSONS WHERE ename= <<vNom>> INTO <<[Matable]Annee_Naiss>>
Fin SQL
```

Dans ce cas, pendant l'exécution de l'instruction **SELECT**, 100 enregistrements seront créés dans la table [Matable], les valeurs de la colonne Birth_Year étant stockées dans le champ 4D Annee_Naiss.

Utilisation d'une listbox

4D inclut un automatisme spécifique permettant de placer les données issues d'une requête *SELECT* dans une listbox. Pour plus d'informations, reportez-vous au manuel Mode Développement.

Optimisation des requêtes

Pour des raisons d'optimisation, il est préférable d'utiliser des expressions 4D plutôt que des fonctions SQL dans les requêtes. En effet, les expressions 4D seront calculées une fois avant l'exécution de la requête tandis que les fonctions SQL sont évaluées pour chaque enregistrement trouvé.

Par exemple, avec l'instruction suivante :

```
SQL EXECUTER ("SELECT nomcomplet FROM PEOPLE WHERE nomcomplet=<<vNom+vPrenom>>")
```

... l'expression *vNom+vPrenom* est calculée une fois, avant l'exécution de la requête. Avec l'instruction suivante :

```
SQL EXECUTER ("SELECT nomcomplet FROM PEOPLE WHERE nomcomplet=CONCAT (<<vNom>>,<<vPrenom>> )")
```

... la fonction **CONCAT(<<vNom>>,<<vPrenom>>)** est appelée pour chaque enregistrement de la table, autrement dit l'expression est évaluée pour chaque enregistrement.

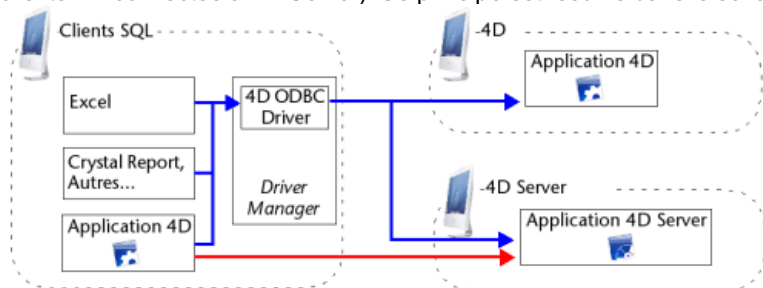
🌱 Configuration du serveur SQL de 4D

Le serveur SQL de 4D autorise les accès externes aux données stockées dans la base 4D. Pour les applications tierces et les applications 4D, ces accès s'effectuent via un pilote ODBC 4D. Il est également possible d'effectuer des connexions directes entre une application cliente 4D et 4D Server. Toutes les connexions sont effectuées via le protocole TCP/IP.

Le serveur SQL d'une application 4D peut être stoppé ou démarré à tout moment. En outre, pour des raisons de performances et de sécurité, vous pouvez définir le port TCP et l'adresse IP d'écoute et restreindre les possibilités d'accès à la base de données 4D.

Accès externes au serveur SQL

L'accès externe au serveur SQL de 4D peut être effectué soit via ODBC (toutes configurations), soit directement (application cliente 4D connectée à 4D Server). Ce principe est résumé dans le schéma suivant :



En bleu : connexions via ODBC

En rouge : connexions directes

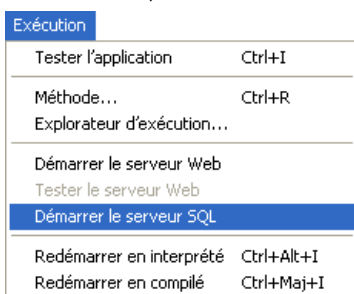
Au niveau des requêtes, l'ouverture d'une connexion externe directe ou via ODBC s'effectue à l'aide de la commande **SQL LOGIN**. Pour plus d'informations, reportez-vous à la description de cette commande.

- **Connexions via ODBC** : 4D fournit un pilote ODBC (ou driver ODBC) permettant à toute application tierce (tableur de type Excel®, autre SGBD...) ou à une autre application 4D de se connecter au serveur SQL de 4D. Le pilote ODBC 4D doit être installé sur le poste de la partie SQL cliente. L'installation et la configuration du pilote ODBC 4D sont détaillées dans un manuel séparé.
- **Connexions directes** : Seule une application 4D Server peut répondre aux requêtes SQL directes provenant d'autres applications 4D. De même, seules les applications 4D de la gamme "Professional" peuvent ouvrir une connexion directe vers une autre application 4D. Durant une connexion directe, l'échange de données s'effectue automatiquement en mode synchrone, ce qui élimine les questions liées à la synchronisation et à l'intégrité des données. Une seule connexion est autorisée par process. Si vous souhaitez établir plusieurs connexions simultanément, vous devez créer autant de process que nécessaire. Les connexions directes peuvent être sécurisées via la sélection de l'option **Activer SSL** du côté cible de la connexion (4D Server) dans la page "SQL" des Propriétés de la base. Les connexions directes ne sont autorisées par 4D Server que si le serveur SQL est démarré. Les connexions directes ont pour principal avantage d'accélérer les échanges.

Démarrer et stopper le serveur SQL

Le serveur SQL de 4D peut être démarré et stoppé de trois manières :

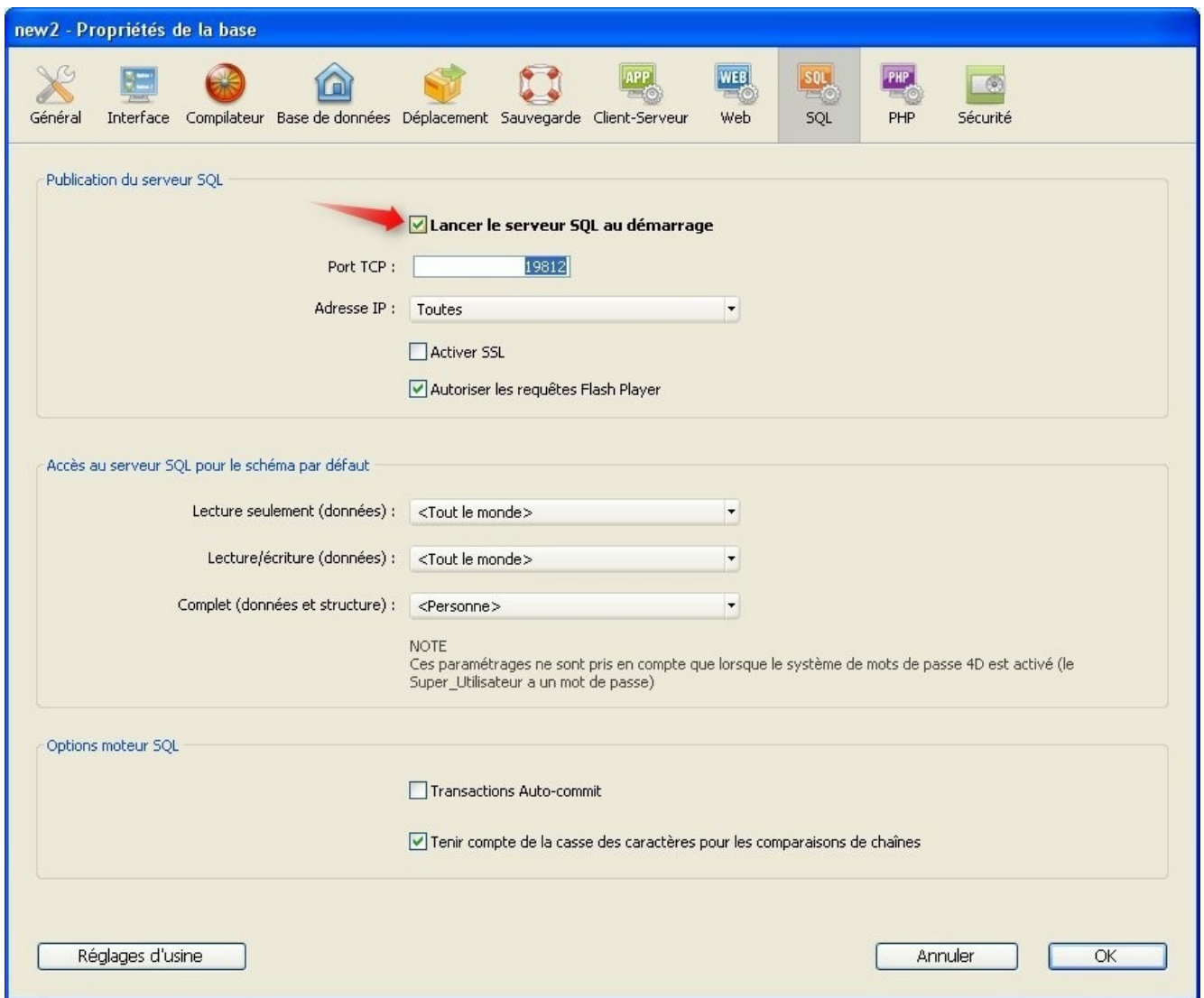
- Manuellement, via la commande **Démarrer le serveur SQL** dans le menu **Exécution** de 4D :



Note : Avec 4D Server, cette commande est accessible sous forme de bouton dans la **Page Serveur SQL**.

Lorsque le serveur est lancé, le libellé de cette commande devient **Arrêter le serveur SQL**.

- Automatiquement au démarrage de l'application, via les Propriétés de la base. Pour cela, affichez la page **SQL** et cochez l'option **Lancer le serveur SQL au démarrage** :



- Par programmation, via les commandes **LANCER SERVEUR SQL** et **ARRETER SERVEUR SQL** (thème "SQL"). Lorsque le serveur SQL est arrêté (ou tant qu'il n'a pas été lancé), 4D ne répond à aucune requête SQL externe.

Note : L'arrêt du serveur SQL n'a pas d'influence sur le fonctionnement du moteur SQL de 4D. Le moteur SQL est accessible en permanence pour les requêtes internes.

Préférences de publication du serveur SQL

Il est possible de configurer les paramètres de publication par défaut du serveur SQL intégré de 4D. Ces paramètres sont accessibles dans la page **SQL** des Propriétés de la base.

- L'option **Lancer le serveur SQL au démarrage** permet de démarrer le serveur SQL au démarrage de l'application
- **Port TCP :** Par défaut, le serveur SQL de 4D répond aux requêtes sur le port TCP 19812. Si ce port est déjà utilisé par un autre service ou si vos paramètres de connexion requièrent une autre configuration, vous pouvez changer le port TCP utilisé par le serveur SQL de 4D.
Note : Si vous passez 0, 4D utilisera le numéro de port TCP par défaut, c'est-à-dire 19812.
- **Adresse IP :** Vous pouvez définir l'adresse IP de la machine sur laquelle le serveur SQL doit traiter les requêtes SQL. Par défaut, le serveur répond sur toutes les adresses IP (option **Toutes**). La liste déroulante "Adresse IP" liste automatiquement toutes les adresses IP présentes sur la machine. Lorsque vous sélectionnez une adresse particulière, le serveur ne répond qu'aux requêtes dirigées sur cette adresse. Cette fonctionnalité est destinée aux applications 4D hébergées sur des machines ayant plusieurs adresses TCP/IP.
Note : Côté client, l'adresse IP et le port TCP du serveur SQL auquel l'application se connecte doivent être correctement configurées dans la définition de la source de données ODBC.
- **Activer SSL :** Cette option indique si le serveur SQL doit activer le protocole SSL pour traiter les connexions SQL. A noter que lorsque ce protocole est activé, le mot-clé "ssl" devra être ajouté à l'adresse IP du serveur SQL lorsque vous ouvrirez une connexion via la commande **SQL LOGIN**.
 Par défaut, le serveur SQL utilise des fichiers internes pour la clé et certificat SSL. Vous pouvez toutefois utiliser des éléments personnalisés : pour cela, il vous suffit de recopier vos propres fichiers *key.pem* et *cert.pem* à l'emplacement suivant : MaBase/Preferences/SQL (où "MaBase" représente le dossier/package de la base).

- **Autoriser les requêtes Flash Player** : Cette option permet d'activer le mécanisme de prise en charge des requêtes Flash Player par le serveur SQL de 4D. Ce mécanisme est basé sur la présence d'un fichier, nommé "socketpolicy.xml", dans le dossier de préférences de la base (Préférences/SQL/Flash/). Ce fichier est requis par Flash Player afin d'autoriser les connexions cross-domains ou les connexions par sockets des applications Flex (Web 2.0). Dans la version précédente de 4D, ce fichier devait être ajouté manuellement. Désormais, lorsque vous cochez cette option, les requêtes Flash Player sont acceptées et un fichier "socketpolicy.xml" générique est créé si nécessaire pour la base. Lorsque vous désélectionnez cette option, le fichier "socketpolicy.xml" est désactivé (renommé). Les requêtes Flash Player ultérieures reçues par le serveur SQL sont alors rejetées. A l'ouverture de la base, l'option est cochée ou non en fonction de la présence d'un fichier "socketpolicy.xml" actif dans le dossier de préférences.
Note : Il est possible de définir l'encodage utilisé par le serveur SQL pour le traitement des requêtes externes à l'aide de la commande 4D [SQL FIXER OPTION](#).

Contrôle des accès SQL pour le schéma par défaut

Pour des raisons de sécurité, il est possible de contrôler les actions que les requêtes externes adressées au serveur SQL peuvent effectuer dans la base de données 4D.

Ce contrôle est effectué à deux niveaux :

- au niveau du type d'action autorisé,
- au niveau de l'utilisateur ayant effectué la requête.
Ces paramétrages peuvent être effectués via la page **SQL** des Propriétés de la base.

Les paramétrages définis dans cette boîte de dialogue s'appliquent au schéma par défaut. En effet, le contrôle des accès externes à la base est basé sur le concept des schémas SQL (cf. section [Implémentations du moteur SQL de 4D](#)). Si vous ne créez pas de schémas personnalisés, le schéma par défaut inclut toutes les tables de la base. Si vous créez d'autres schémas dotés de droits spécifiques et leur associez des tables, le schéma par défaut inclut uniquement les tables non incluses dans les schémas personnalisés.

Vous pouvez configurer séparément trois types d'accès au schéma par défaut via le serveur SQL :

- **Lecture seulement (données)** : libre accès en lecture à toutes les données des tables de la base mais ne permet pas l'ajout, la modification ou la suppression d'enregistrements ni la modification de la structure de la base.
- **Lecture/écriture (données)** : accès en lecture et en écriture (ajout, modification et suppression) à toutes les données des tables de la base mais ne permet pas la modification de la structure de la base.
- **Complet (données et structure)** : accès en lecture et en écriture (ajout, modification et suppression) à toutes les données des tables de la base et à la structure de la base (tables, champs, liens, etc).

Vous pouvez désigner un ensemble d'utilisateurs pour chaque type d'accès. Pour cela, vous disposez de trois types d'options :

- **<Personne>** : si vous sélectionnez cette option, le type d'accès sera refusé pour toutes les requêtes, quelle que soit leur provenance. Ce paramètre peut être utilisé même si le système de gestion des accès via des mots de passe de 4D n'est pas activé.
- **<Tout le monde>** : si vous sélectionnez cette option, le type d'accès sera accepté pour toutes les requêtes (aucun contrôle n'est effectué).
- **groupe d'utilisateurs** : cette option permet de désigner un groupe d'utilisateurs comme seul autorisé à effectuer le type d'accès associé. Cette option requiert que la gestion des mots de passe de 4D ait été activée. L'utilisateur à l'origine des requêtes fournit son nom et son mot de passe lors de la connexion au serveur SQL.

ATTENTION : Ce mécanisme s'appuie sur les mots de passe de 4D. Pour que le contrôle d'accès au serveur SQL soit effectif, le système de mots de passe de 4D doit être actif (un mot de passe doit avoir été attribué au Super_Utilisateur).

Note : Une option de sécurité supplémentaire peut être définie au niveau de chaque méthode projet 4D. Pour plus d'informations sur ce point, reportez-vous au paragraphe "Option "Disponible via SQL"" dans la section [Implémentations du moteur SQL de 4D](#).



Fondamentalement, le moteur SQL de 4D est conforme à la norme SQL92. Cela signifie que, pour une description détaillée des commandes, fonctions, opérateurs et syntaxes à utiliser, vous pouvez vous référer à n'importe quelle documentation sur le SQL92. De multiples ressources sur ce thème sont disponibles, par exemple, sur Internet.

Cependant, le moteur SQL de 4D ne prend pas en charge 100 % des fonctions du SQL92 et propose en outre des fonctions supplémentaires, non présentes dans cette norme.

Ce paragraphe décrit les principales implémentations et limitations du moteur SQL de 4D.

Limitations générales

Le moteur SQL de 4D étant intégré au cœur de la base de données de 4D, les limitations relatives au nombre maximum de tables, de colonnes (champs) et d'enregistrements par base ainsi que les règles de nommage des tables et des colonnes sont identiques à celles du moteur standard de 4D. Elles sont listées ci-dessous :

- Nombre maximum de tables par base : deux milliards en théorie, mais limitation à 32767 pour des raisons de compatibilité avec 4D.
- Nombre maximum de colonnes (champs) par table : deux milliards en théorie, mais limitation à 32767 pour des raisons de compatibilité avec 4D.
- Nombre maximum de lignes (enregistrements) par table : un milliard.
- Nombre maximum de clés d'index : un milliard x 64.
- Une clé primaire ne peut pas être une valeur NULL et doit être unique. Il n'est pas obligatoire d'indexer les colonnes (champs) clés primaires.
- Nombre maximum de caractères pour les noms de tables et de champs : 31 caractères (limitation de 4D).
- Il n'est pas possible de créer plusieurs tables avec le même nom. Les mécanismes de contrôle standard de 4D sont appliqués.

Types de données

Le tableau suivant indique les types de données pris en charge dans le SQL de 4D ainsi que leur type correspondant dans 4D :

Type 4D SQL	Description	Type 4D
Varchar	Texte aphanumérique	Texte ou Alpha
Real	Nombre à virgule flottante compris dans l'intervalle +/-1.7E308	Réel
Numeric	Nombre compris dans l'intervalle +/-2E64	Entier 64 bits
Float	Nombre à virgule flottante (virtuellement infini)	Float
Smallint	Nombre compris entre -32 768 et 32 767	Entier
Int	Nombre compris entre -2 147 483 648 et 2 147 483 647	Entier long, Entier
Int64	Nombre compris dans l'intervalle +/-2E64	Entier 64 bits
UUID	Nombre de 16 octets (128 bits) contenant 32 caractères hexadécimaux	Alpha format UUID
Bit	Champ qui n'accepte que la valeur TRUE/FALSE ou 1/0	Booléen
Boolean	Champ qui n'accepte que la valeur TRUE/FALSE ou 1/0	Booléen
Blob	Jusqu'à 2 Go ; tout objet binaire tel qu'une image, un document, une application...	BLOB
Bit varying	Jusqu'à 2 Go ; tout objet binaire tel qu'une image, un document, une application...	BLOB
Clob	Jusqu'à 2 Go de texte. Ce type de colonne (champ) ne peut pas être indexé. Il n'est pas stocké dans l'enregistrement lui-même.	Texte
Text	Jusqu'à 2 Go de texte. Ce type de colonne (champ) ne peut pas être indexé. Il n'est pas stocké dans l'enregistrement lui-même.	Texte
Timestamp	Date sous la forme 'YYYY/MM/DD' et Heure sous la forme 'HH:MM:SS:ZZ'	Parties Date et Heure gérées séparément (conversion auto)
Duration	Durée sous la forme 'HH:MM:SS:ZZ'	Heure
Interval	Durée sous la forme 'HH:MM:SS:ZZ'	Heure
Picture	Image PICT jusqu'à 2 Go	Image

La conversion entre les types de données numériques est automatique. Les chaînes qui représentent un nombre ne sont pas converties en valeur numérique. L'opérateur de transtypage **CAST** permet de convertir des valeurs d'un type en un autre. Les types de données SQL suivants ne sont pas implémentés :

- NCHAR
- NCHAR VARYING

Valeurs NULL dans 4D

La valeur NULL est implémentée dans le langage SQL de 4D ainsi que dans le moteur de base de données de 4D. En revanche, cette valeur n'existe pas dans le langage de 4D. Il est toutefois possible de lire et d'écrire la valeur NULL dans un champ 4D via les commandes **Valeur champ Null** et **FIXER VALEUR CHAMP NULL**.

Compatibilité des traitements et option Traduire les NULL en valeurs vides

Pour des raisons de compatibilité dans 4D, les valeurs NULL stockées dans les tables des bases de données 4D sont automatiquement converties en valeurs par défaut lors des manipulations effectuées via le langage de 4D. Par exemple, dans le cas de l'instruction suivante :

```
mavarAlpha:=[matable]MonChpAlpha
```

... si le champ MonchpAlpha contient la valeur NULL, la variable mavarAlpha contiendra "" (chaîne vide).

Les valeurs par défaut dépendent du type de données :

- Pour les types Alpha et Texte : ""
- Pour les types Réel, Entier et Entier long : 0
- Pour le type Date : "00/00/00"
- Pour le type Heure : "00:00:00"
- Pour le type Booléen : Faux
- Pour le type Image : Image vide
- Pour le type BLOB : BLOB vide

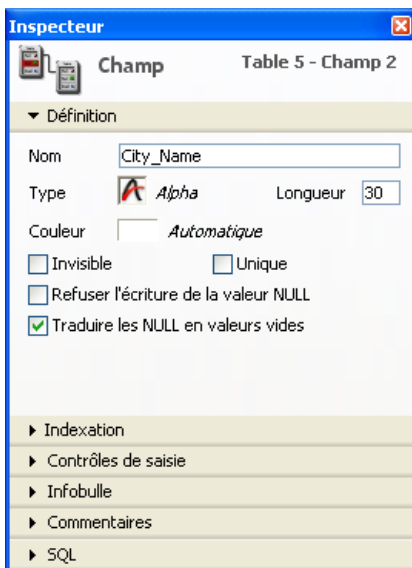
En revanche, ce mécanisme ne s'applique pas en principe aux traitements effectués au niveau du moteur de la base de données 4D, tels que les recherches. En effet, la recherche d'une valeur "vide" (par exemple mavaleur=0) ne trouvera pas les enregistrements stockant la valeur NULL, et inversement. Lorsque les deux types de valeurs (valeurs par défaut et NULL) cohabitent dans les enregistrements pour un même champ, certains traitements peuvent être faussés ou nécessiter du code supplémentaire.

Pour éviter ces désagréments, une option permet d'uniformiser tous les traitements dans le langage de 4D : **Traduire les NULL en valeurs vides**. Cette option, accessible dans l'Inspecteur des champs de l'éditeur de structure, permet d'étendre le principe d'usage des valeurs par défaut à tous les traitements. Les champs contenant la valeur NULL seront systématiquement considérés comme contenant la valeur par défaut. Cette option est cochée par défaut.

La propriété Traduire les NULL en valeurs vides est prise en compte à un niveau très bas du moteur de la base de données. Elle agit notamment sur la routine **Valeur champ Null**.

Refuser l'écriture de la valeur NULL

La propriété de champ **Refuser l'écriture de la valeur NULL** permet d'interdire le stockage de la valeur NULL :



Lorsque cet attribut est coché pour un champ, il ne sera pas possible de stocker la valeur NULL dans ce champ. Cette propriété de bas niveau correspond précisément à l'attribut NOT NULL du SQL.

De manière générale, si vous souhaitez pouvoir utiliser des valeurs NULL dans votre base de données 4D, il est conseillé d'utiliser exclusivement le langage SQL de 4D.

Note: Dans 4D, les champs peuvent également avoir l'attribut "Obligatoire". Les deux notions sont proches mais leur portée est différente : l'attribut "Obligatoire" est un contrôle de saisie, tandis que l'attribut "Refuser l'écriture de la valeur NULL" agit au niveau du moteur de la base de données.

Si un champ disposant de cet attribut reçoit la valeur NULL, une erreur est générée.

Option “Disponible via SQL”

La propriété "Disponible via SQL", disponible pour les méthodes projet, permet de contrôler l'exécution des méthodes projet de 4D via le SQL.



Lorsqu'elle est cochée, cette option autorise l'exécution de la méthode projet par le moteur SQL de 4D. Elle est désélectionnée par défaut, ce qui signifie que, sauf autorisation explicite, les méthodes projet de 4D sont protégées et peuvent pas être appelées par le moteur SQL de 4D.

Cette propriété s'applique à toutes les requêtes SQL internes et externes — exécutées via le driver ODBC, le code SQL inséré dans les balises **Debut SQL/Fin SQL** ou la commande **CHERCHER PAR SQL**.

Notes :

- Même si une méthode dispose de l'attribut "Disponible via SQL", les accès définis au niveau des Propriétés de la base et des propriétés de la méthode sont pris en compte pour l'exécution de la méthode.
- La fonction ODBC SQLProcedure retourne uniquement les méthodes projet disposant de l'attribut "Disponible via SQL".

Options moteur SQL

- **Transactions Auto-commit** : cette option permet d'activer le mécanisme d'auto-commit dans le moteur SQL. Le mode auto-commit a pour but de préserver l'intégrité référentielle des données. Lorsque cette option est cochée, toute requête **SELECT**, **INSERT**, **UPDATE** et **DELETE** (SIUD) effectuée en-dehors d'une transaction est automatiquement incluse dans une transaction ad hoc. Ce principe garantit que les requêtes seront entièrement exécutées ou, en cas d'erreur, intégralement annulées.

Les requêtes incluses dans une transaction (gestion personnalisée de l'intégrité référentielle) ne sont pas affectées par cette option.

Lorsque cette option n'est pas cochée, aucune transaction automatique n'est générée (à l'exception des requêtes **SELECT... FOR UPDATE**, reportez-vous à la description de la commande **SELECT**). Par défaut, cette option n'est pas cochée.

Vous pouvez également gérer cette option par programmation à l'aide de la commande **FIXER PARAMETRE BASE**.

Note : Seules les bases locales interrogées par le moteur SQL de 4D sont affectées par ce paramètre. Dans le cas de connexions externes à d'autres bases SQL, le mécanisme d'auto-commit est pris en charge par les moteurs SQL distants.

- **Tenir compte de la casse des caractères pour les comparaisons de chaînes** : cette option permet de modifier le mode de prise en compte de la casse des caractères dans les requêtes SQL. Elle est cochée par défaut, ce qui signifie que le moteur SQL établit une différence entre les majuscules et les minuscules lors des comparaisons de chaînes (tris et recherches). Par exemple "ABC"="ABC" mais "ABC" # "Abc".

Dans certains cas, par exemple pour aligner le fonctionnement du moteur SQL sur celui du moteur 4D, vous pourrez souhaiter que les comparaisons de chaînes ne tiennent pas compte de la casse ("ABC"="Abc"). Pour cela, il suffit de désélectionner l'option.

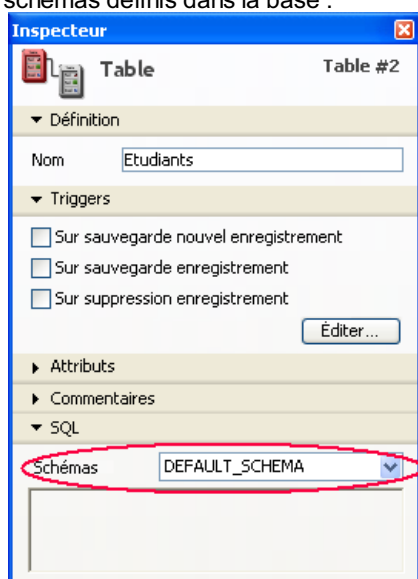
Vous pouvez également gérer cette option par programmation à l'aide de la commande **FIXER PARAMETRE BASE**.

Schémas

4D implémente le concept de **schémas**. Un schéma est un objet virtuel contenant des tables de la base. Dans le SQL, le concept

de schémas a pour but de permettre l'attribution de droits d'accès spécifiques à des ensembles d'objets de la base de données. Les schémas découpent la base en entités indépendantes dont l'assemblage représente la base entière. Autrement dit, une table appartient toujours à un et un seul schéma.

- Pour créer un schéma, vous devez utiliser la commande *CREATE SCHEMA*. Vous utilisez ensuite les commandes *GRANT* et *REVOKE* pour configurer les types d'accès des schémas.
- Pour associer une table à un schéma, vous pouvez appeler les commandes *CREATE TABLE* ou *ALTER TABLE*. Vous pouvez également utiliser le pop up menu "Schémas" de l'Inspecteur de l'éditeur de Structure de 4D, qui liste tous les schémas définis dans la base :



- La commande *DROP SCHEMA* permet de supprimer un schéma.

Note : Le contrôle des accès via les schémas s'applique uniquement aux connexions depuis l'extérieur. Le code SQL exécuté à l'intérieur de 4D via les balises *Debut SQL/Fin SQL*, *SQL EXECUTER*, *CHERCHER PAR SQL*... dispose toujours d'un accès complet.

Tables système

Le catalogue SQL de 4D comporte sept tables système, accessible à tout utilisateur SQL disposant des droits d'accès en lecture : *_USER_TABLES*, *_USER_COLUMNS*, *_USER_INDEXES*, *_USER_CONSTRAINTS*, *_USER_IND_COLUMNS*, *_USER_CONS_COLUMNS* et *_USER_SCHEMAS*.

Conformément aux usages dans le monde SQL, les tables système décrivent la structure de la base de données. Voici le descriptif de ces tables et de leurs champs :

_USER_TABLES		Décrit les tables utilisateurs de la base
TABLE_NAME	VARCHAR	Nom de table
TEMPORARY	BOOLEAN	Vrai si la table est temporaire, faux sinon
TABLE_ID	INT64	Numéro de table
SCHEMA_ID	INT32	Numéro de schéma

_USER_COLUMNS		Décrit les colonnes des tables utilisateurs de la base
TABLE_NAME	VARCHAR	Nom de table
COLUMN_NAME	VARCHAR	Nom de colonne
DATA_TYPE	INT32	Type de colonne
DATA_LENGTH	INT32	Longueur de colonne
NULLABLE	BOOLEAN	Vrai si la colonne accepte des valeurs NULL, faux sinon
TABLE_ID	INT64	Numéro de table
COLUMN_ID	INT64	Numéro de colonne

_USER_INDEXES		Décrit les index utilisateurs de la base
INDEX_ID	VARCHAR	Numéro d'index
INDEX_NAME	VARCHAR	Nom d'index
INDEX_TYPE	INT32	Type d'index (1=BTree / Composite, 3=Cluster / Mots-clés, 7=Auto)
TABLE_NAME	VARCHAR	Nom de table avec index
UNIQUENESS	BOOLEAN	Vrai si l'index impose une contrainte d'unicité, faux sinon
TABLE_ID	INT64	Numéro de table avec index

_USER_IND_COLUMNS		Décrit les colonnes des index utilisateurs de la base
INDEX_ID	VARCHAR	Numéro d'index
INDEX_NAME	VARCHAR	Nom d'index
TABLE_NAME	VARCHAR	Nom de table avec index
COLUMN_NAME	VARCHAR	Nom de colonne avec index
COLUMN_POSITION	INT32	Position de colonne dans l'index
TABLE_ID	INT64	Numéro de table avec index
COLUMN_ID	INT64	Numéro de colonne

_USER_CONSTRAINTS		Décrit les contraintes d'intégrité de la base
CONSTRAINT_ID	VARCHAR	Numéro de contrainte
CONSTRAINT_NAME	VARCHAR	Nom de la définition de contrainte
CONSTRAINT_TYPE	VARCHAR	Type de contrainte (P=clé primaire, R=intégrité référentielle - clé étrangère, 4DR=relation 4D)
TABLE_NAME	VARCHAR	Nom de table avec contrainte
TABLE_ID	INT64	Numéro de table avec contrainte
DELETE_RULE	VARCHAR	Règle de suppression de la contrainte – CASCADE ou RESTRICT
RELATED_TABLE_NAME	VARCHAR	Nom de table liée
RELATED_TABLE_ID	INT64	Numéro de table liée

_USER_CONS_COLUMNS		Décrit les colonnes des contraintes utilisateurs de la base
CONSTRAINT_ID	VARCHAR	Numéro de contrainte
CONSTRAINT_NAME	VARCHAR	Nom de contrainte
TABLE_NAME	VARCHAR	Nom de table avec contrainte
TABLE_ID	INT64	Numéro de table avec contrainte
COLUMN_NAME	VARCHAR	Nom de colonne avec contrainte
COLUMN_ID	INT64	Numéro de colonne avec contrainte
COLUMN_POSITION	INT32	Position de colonne dans une contrainte
RELATED_COLUMN_NAME	VARCHAR	Nom de colonne liée dans une contrainte
RELATED_COLUMN_ID	INT32	Numéro de colonne liée dans une contrainte

_USER_SCHEMAS		Décrit les schémas de la base
SCHEMA_ID	INT32	Numéro de schéma
SCHEMA_NAME	VARCHAR	Nom de schéma
READ_GROUP_ID	INT32	Numéro du groupe ayant accès en lecture
READ_GROUP_NAME	VARCHAR	Nom du groupe ayant accès en lecture
READ_WRITE_GROUP_ID	INT32	Numéro du groupe ayant accès en lecture-écriture
READ_WRITE_GROUP_NAME	VARCHAR	Nom du groupe ayant accès en lecture-écriture
ALL_GROUP_ID	INT32	Numéro du groupe ayant un accès complet
ALL_GROUP_NAME	VARCHAR	Nom du groupe ayant un accès complet

Note : Les tables système sont affectées à un schéma particulier nommé **SYSTEM_SCHEMA**. Ce schéma ne peut être ni modifié ni supprimé. Il n'apparaît pas dans la liste des schémas affichée dans l'Inspecteur de tables. Il est accessible en mode Lecture seulement à tout utilisateur.

Connexions aux sources SQL

L'architecture multi-bases est implémentée au niveau du serveur SQL de 4D. Depuis 4D, il est possible :

- de se connecter à une base existante à l'aide de la commande **SQL LOGIN**.
- de passer d'une base à l'autre en utilisant les commandes 4D **SQL LOGIN** et **SQL LOGOUT**.
- d'ouvrir et d'utiliser une autre base 4D que la base courante à l'aide de la commande **USE DATABASE**.

Clé primaire

Dans le langage SQL, la clé primaire permet d'identifier dans une table la ou les colonnes (champs) chargée(s) de désigner de façon unique les enregistrements (lignes). La définition d'une clé primaire est notamment indispensable à la fonction de réplication des enregistrements d'une table de 4D (cf. section).

4D vous permet de gérer la clé primaire d'une table de deux manières :

- via le langage SQL
- depuis l'éditeur de structure de 4D.

Définir la clé primaire via le langage SQL

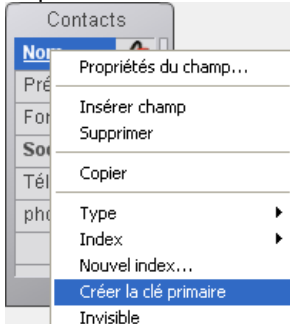
Vous pouvez définir une clé primaire (PRIMARY KEY) lors de la création d'une table (via la commande *CREATE TABLE*) ou de l'ajout ou de la modification d'une colonne (via la commande *ALTER TABLE*). La clé primaire est définie à l'aide de la clause PRIMARY KEY suivie du nom de la colonne ou d'une liste de colonnes. Pour plus d'informations, reportez-vous à la section .

Définir la clé primaire via l'éditeur de structure

4D vous permet de créer et de supprimer directement une clé primaire via le menu contextuel de l'éditeur de structure.

Pour créer une clé primaire :

1. Sélectionnez le ou les champ(s) devant constituer la clé primaire de la table.
2. Cliquez avec le bouton droit de la souris et choisissez la commande **Créer la clé primaire** dans le menu contextuel :



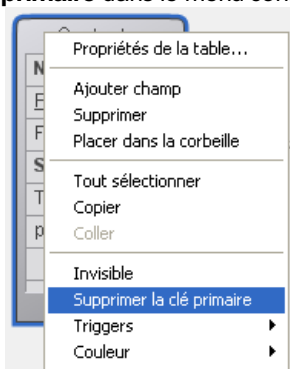
Les champs inclus dans la clé primaire apparaissent alors soulignés dans l'éditeur et leur description SQL fait apparaître le mot-clé PRIMARY KEY.

Le ou les champ(s) appartenant à la clé primaire ne doivent pas comporter de valeurs dupliquées. Si des valeurs dupliquées existaient déjà dans les enregistrements de la table, une boîte de dialogue d'alerte vous le signale.

Note : La ou les colonne(s) appartenant à la clé primaire n'acceptent pas la valeur NULL.

Pour supprimer la clé primaire d'une table :

1. Cliquez avec le bouton droit de la souris sur la table contenant la clé primaire et choisissez la commande **Supprimer la clé primaire** dans le menu contextuel :



Une boîte de dialogue de confirmation apparaît. Si vous cliquez sur **OK**, la clé primaire est immédiatement supprimée.

➤ Réplication via le SQL

4D propose un mécanisme permettant de répliquer ou de synchroniser les données de deux ou plusieurs bases 4D via le SQL. Cette fonctionnalité spécifique autorise la mise en place d'une ou plusieurs base(s) miroir, garantissant la disponibilité permanente des données.

Le principe est le suivant : une base de données cible réplique en local les données d'une base de données source distante. La mise à jour s'effectue périodiquement par la base locale qui va chercher les données sur la base distante. La réplication est effectuée au niveau de la table : vous répliquez les données d'une table de la base distante dans une table de la base locale. Ce principe s'appuie sur des "marqueurs" (*stamps*) et des commandes SQL spécifiques.

Dans l'éditeur de structure, une propriété de table permet d'activer le mécanisme de réplication dans la base distante et la base locale. Côté base locale, la commande SQL **REPLICATE** permet de rapatrier les données d'une table de la base distante puis de les intégrer dans une table de la base locale. La commande SQL **SYNCHRONIZE** permet quant à elle d'effectuer la synchronisation des deux tables.

Champs virtuels

Chaque table de la base 4D se voit attribuer trois champs "virtuels" : `__ROW_ID`, `__ROW_STAMP` et `__ROW_ACTION`. Ces champs sont appelés virtuels pour les différencier des champs "classiques", car ils disposent de propriétés spécifiques : ils sont renseignés automatiquement, peuvent être lus mais non modifiés par les utilisateurs et n'apparaissent pas dans les tables système de la base. Le tableau suivant décrit ces champs ainsi que leur mode d'utilisation :

Champ virtuel	Type	Contenu	Utilisation
<code>__ROW_ID</code>	Int32	ID d'enregistrement	Dans toute instruction SQL sauf REPLICATE ou SYNCHRONIZE
<code>__ROW_STAMP</code>	Int64	Informations de réplication de l'enregistrement	Dans toute instruction SQL
<code>__ROW_ACTION</code>	Int16	Type d'action effectuée sur l'enregistrement : 1 = Ajout ou modification, 2 = Suppression	Uniquement avec la commande REPLICATE ou SYNCHRONIZE

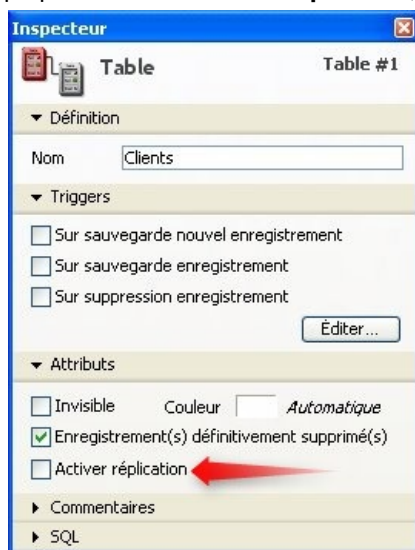
Lorsque les mécanismes de réplication sont activés, dès qu'un enregistrement est créé, modifié ou supprimé, les informations correspondantes sont automatiquement mises à jour dans les champs virtuels de cet enregistrement.

Activation de la réplication

Par défaut les mécanismes permettant la réplication ne sont pas activés. Vous devez les activer explicitement côté base distante et côté base locale pour chaque table utilisée dans la réplication ou synchronisation.

A noter que l'activation ne déclenche pas la réplication ; pour que les données soient effectivement répliquées dans la base locale ou synchronisées, vous devez utiliser les commandes **REPLICATE** ou **SYNCHRONIZE**.

Pour activer le mécanisme interne de réplication, vous devez utiliser dans chaque table (côté base distante et côté base locale) la propriété de table **Activer réplication**, accessible dans l'Inspecteur des tables :



Note : Pour que le mécanisme de réplication puisse fonctionner, une clé primaire doit avoir été définie dans les tables impliquées côté base distante et côté base locale. Vous pouvez effectuer cette création via l'éditeur de structure ou les commandes SQL. Si aucune clé primaire n'a été définie, l'option est grisée.

Lorsque cette option est cochée, 4D génère les informations nécessaires au mécanisme de réplication des enregistrements de la

table (basées notamment sur la clé primaire de la table). Ces informations sont stockées dans les champs virtuels `__ROW_STAMP` et `__ROW_ACTION`.

Note : Il est possible d'activer et d'inactiver la génération des informations de réplication via les commandes SQL [CREATE TABLE](#) et [ALTER TABLE](#), en utilisant les mots-clés `ENABLE REPLICATE` et `DISABLE REPLICATE`. Pour plus d'informations, reportez-vous à la description de ces commandes.

ATTENTION : L'action de cocher cette option entraîne la publication d'informations nécessaires aux mécanismes de réplication. Pour des raisons de sécurité, l'accès à ces informations doit être protégé – comme doivent l'être les accès à vos données lorsqu'elles sont publiées. Par conséquent, lorsque vous implémentez un système de réplication à l'aide de cette option, vous devez veiller à ce que :

- si le serveur SQL est lancé, les accès soient protégés à l'aide des mots de passe 4D et/ou les schémas SQL (cf. section [Configuration du serveur SQL de 4D](#)),
- si le serveur HTTP est lancé, les accès soient protégés à l'aide des mots de passe 4D et/ou des schémas SQL (cf. section [Configuration du serveur SQL de 4D](#)) et/ou de la [Méthode base Sur authentification Web](#) et/ou de la définition d'une structure virtuelle via les commandes [FIXER TITRES TABLES](#) et [FIXER TITRES CHAMPS](#). Pour plus d'informations, reportez-vous au paragraphe "URL 4DSYNC/" dans la section [URLs et actions de formulaires](#).

Mise à jour côté base locale

Une fois que le mécanisme de réplication est activé dans chaque table de chaque base, vous pouvez l'exploiter depuis la base locale à l'aide de la commande SQL [REPLICATE](#). Pour plus d'informations, reportez-vous à la description de cette commande.

🌱 Prise en charge des jointures

Le moteur SQL de 4D permet une prise en charge étendue des jointures.

Les jointures peuvent être internes ou externes, implicites ou explicites. Les jointures internes (*INNER joins*) implicites sont prises en charge via la commande *SELECT*. Vous pouvez également générer des jointures internes et externes explicites à l'aide du mot-clé SQL *JOIN*.

Note : L'implémentation actuelle des jointures dans le moteur SQL de 4D n'inclut pas :

- les jointures naturelles.
- le constructeur *USING* dans les jointures internes.

Présentation

Les jointures permettent de mettre en relation les enregistrements de deux ou plusieurs tables et de combiner le résultat dans une nouvelle table, appelée jointure.

Vous générez des jointures via des instructions *SELECT* définissant des conditions de jointure. Avec les jointures explicites, les conditions peuvent être complexes mais elles doivent toujours être basées sur un test d'égalité entre des colonnes incluses dans la jointure. Par exemple, il n'est pas possible d'utiliser l'opérateur *>=* dans une condition de jointure explicite. Les jointures implicites autorisent tout type de comparaison.

En interne, les comparaisons d'égalité sont effectuées directement par le moteur de 4D, ce qui leur assure une grande rapidité d'exécution.

Note : Généralement, dans le moteur de base de données, l'ordre des tables est déterminé par l'ordre défini dans la recherche.

Lorsque vous utilisez des jointures, l'ordre des tables est déterminé par la liste des tables. Dans l'exemple suivant :

```
SELECT * FROM T1 RIGHT OUTER JOIN T2 ON T2.depID = T1.depID;
```

... l'ordre des tables est T1 puis T2 (comme elles apparaissent dans la liste des tables) et non T2 puis T1 (comme elles apparaissent dans la condition de jointure).

Jointures internes explicites

Une jointure interne (*inner join*) est une jointure basée sur une comparaison d'égalité entre deux colonnes. Considérons par exemple les trois tables suivantes :

- Employees

name	depID	cityID
Alain	10	30
Anne	11	39
Bernard	10	33
Fabrice	12	35
Martine	15	30
Philippe	NULL	33
Thomas	10	NULL

- Departments

depID	depName
10	Program
11	Engineering
NULL	Marketing
12	Development
13	Quality

- Cities

cityID	cityName
30	Paris
33	New York
NULL	Berlin

Note : Cette structure d'exemple sera utilisée tout au long de ce chapitre.

Voici un exemple de jointure interne implicite :

```
SELECT *
      FROM employees, departments
      WHERE employees.DepID = departments.DepID;
```

Dans 4D, vous pouvez également utiliser le mot-clé JOIN afin de définir une jointure interne explicite :

```
SELECT *
      FROM employees
      INNER JOIN departments
            ON employees.DepID = departments.DepID;
```

Cette requête peut être insérée dans le code 4D de la manière suivante :

```
TABLEAU TEXTE (aName;0)
TABLEAU TEXTE (aDepName;0)
TABLEAU ENTIER (aEmpDepID;0)
TABLEAU ENTIER (aDepID;0)
Debut SQL
      SELECT *
      FROM employees
            INNER JOIN departments
                  ON employees.depID = departments.depID
      INTO :aName, :aEmpDepID, :aDepID, :aDepName;
Fin SQL
```

Résultat de cette jointure :

aName	aEmpDepID	aDepID	aDepName
Alain	10	10	Program
Anne	11	11	Engineering
Bernard	10	10	Program
Fabrice	12	12	Development
Thomas	10	10	Program

A noter que ni les employés Philippe et Martine ni les départements Marketing et Quality n'apparaissent dans la jointure résultante car :

- Philippe n'a pas de département associé (NULL),
- l'ID de département de Martine n'existe pas dans la table Departments,
- il n'y a pas d'employé associé au département d'ID 13 (Quality),
- le département Marketing n'a pas d'ID associé (NULL).

Jointures croisées (CROSS)

Une jointure interne dans laquelle la clause WHERE et la clause ON sont omises est appelée jointure croisée ou cartésienne. Effectuer une jointure croisée consiste à associer chaque ligne d'une table à chaque ligne d'une autre table.

Le résultat d'une jointure croisée est le produit cartésien des tables, contenant $m \times n$ lignes, où m est le nombre de lignes de la première table et n est le nombre de lignes de la seconde table. Ce produit représente l'ensemble des combinaisons possibles formées par la concaténation des lignes des tables.

Les syntaxes suivantes sont équivalentes :

```
SELECT * FROM T1 INNER JOIN T2
SELECT * FROM T1, T2
SELECT * FROM T1 CROSS JOIN T2;
```

Voici un exemple de code 4D intégrant une jointure croisée :

```
TABLEAU TEXTE (aName;0)
TABLEAU TEXTE (aDepName;0)
TABLEAU ENTIER (aEmpDepID;0)
TABLEAU ENTIER (aDepID;0)
Debut SQL
      SELECT *
      FROM employees CROSS JOIN departments
      INTO :aName, :aEmpDepID, :aDepID, :aDepName;
Fin SQL
```

Résultat de cette jointure avec notre base d'exemple :

aName	aEmpDepID	aDepID	aDepName
Alain	10	10	Program
Anne	11	10	Program
Bernard	10	10	Program
Fabrice	12	10	Program
Martine	15	10	Program
Philippe	NULL	10	Program
Thomas	10	10	Program
Alain	10	11	Engineering
Anne	11	11	Engineering
Bernard	10	11	Engineering
Fabrice	12	11	Engineering
Martine	15	11	Engineering
Philippe	NULL	11	Engineering
Thomas	10	11	Engineering
Alain	10	NULL	Marketing
Anne	11	NULL	Marketing
Bernard	10	NULL	Marketing
Fabrice	12	NULL	Marketing
Martine	15	NULL	Marketing
Philippe	NULL	NULL	Marketing
Thomas	10	NULL	Marketing
Alain	10	12	Development
Anne	11	12	Development
Bernard	10	12	Development
Fabrice	12	12	Development
Martine	15	12	Development
Philippe	NULL	12	Development
Thomas	10	12	Development
Alain	10	13	Quality
Anne	11	13	Quality
Bernard	10	13	Quality
Fabrice	12	13	Quality
Martine	15	13	Quality
Philippe	NULL	13	Quality
Thomas	10	13	Quality

Note : Pour des raisons de performances, les jointures croisées sont à utiliser avec précaution.

Jointures externes

4D vous permet de générer des jointures externes (*OUTER JOINS*). Dans une jointure externe, il n'est pas nécessaire qu'il existe une correspondance entre les lignes des tables jointes. La table résultante contient toutes les lignes des tables (ou d'au moins une des tables de la jointure) même s'il n'y a pas de ligne correspondante. Ce principe permet de s'assurer que toutes les informations d'une table sont exploitées, même si des lignes ne sont pas renseignées entre les différentes tables jointes.

Il existe trois types de jointures externes, définies par les mots-clés LEFT, RIGHT et FULL. LEFT et RIGHT permettent de désigner la table (située à gauche ou à droite du mot-clé) dont la totalité des données devra être traitée. FULL indique une jointure externe bilatérale.

Note : Seules les jointures externes explicites sont prises en charge par 4D.

Jointures externes gauches (LEFT OUTER JOIN)

Le résultat d'une jointure externe gauche (ou jointure gauche) contient toujours tous les enregistrements de la table située à gauche du mot-clé même si la condition de jointure ne trouve pas d'enregistrement correspondant dans la table de droite. Cela signifie que si pour une ligne de la table gauche la requête trouve zéro ligne correspondant dans la table droite, la jointure contiendra la ligne avec la valeur NULL pour chaque colonne de la table de droite. Autrement dit, une jointure externe gauche retourne toutes les lignes de la table gauche *plus* celles de la table droite qui correspondent à la condition de jointure (ou NULL si aucune ne correspond). A noter que si la table la droite contient plus d'une ligne correspondant au prédicat de la jointure pour une ligne de la table gauche, les valeurs de la table gauche seront répétées pour chaque ligne distincte de la table droite.

Voici un exemple de code 4D effectuant une jointure externe gauche :

```
TABLEAU TEXTE (aName; 0)
TABLEAU TEXTE (aDepName; 0)
```

```
TABLEAU ENTIER (aEmpDepID;0
```

```
TABLEAU ENTIER (aDepID;0)
```

```
Debut SQL
```

```
SELECT *
    FROM employees
    LEFT OUTER JOIN departments
        ON employees.DepID = departments.DepID;
    INTO :aName, :aEmpDepID, :aDepID, :aDepName;
```

```
Fin SQL
```

Résultat de cette jointure avec notre base d'exemple (les lignes additionnelles sont en rouge) :

aName	aEmpDepID	aDepID	aDepName
Alain	10	10	Program
Anne	11	11	Engineering
Bernard	10	10	Program
Fabrice	12	12	Development
Thomas	10	10	Program
Martine	15	NULL	NULL
Philippe	NULL	NULL	NULL

Jointures externes droites (RIGHT OUTER JOIN)

A l'exact opposé de la jointure externe gauche, le résultat d'une jointure externe droite contient toujours tous les enregistrements de la table située à droite du mot-clé même si la condition de jointure ne trouve pas d'enregistrement correspondant dans la table gauche.

Voici un exemple de code 4D effectuant une jointure externe droite :

```
TABLEAU TEXTE (aName;0)
```

```
TABLEAU TEXTE (aDepName;0)
```

```
TABLEAU ENTIER (aEmpDepID;0
```

```
TABLEAU ENTIER (aDepID;0)
```

```
Debut SQL
```

```
SELECT *
    FROM employees
    RIGHT OUTER JOIN departments
        ON employees.DepID = departments.DepID;
    INTO :aName, :aEmpDepID, :aDepID, :aDepName;
```

```
Fin SQL
```

Résultat de cette jointure avec notre base d'exemple (les lignes additionnelles sont en rouge) :

aName	aEmpDepID	aDepID	aDepName
Alain	10	10	Program
Anne	11	11	Engineering
Bernard	10	10	Program
Fabrice	12	12	Development
Thomas	10	10	Program
NULL	NULL	NULL	Marketing
NULL	NULL	13	Quality

Jointures externes bilatérales (FULL OUTER JOIN)

La jointure externe bilatérale combine simplement les résultats d'une jointure externe gauche et d'une jointure externe droite. La table jointure résultante contient tous les enregistrements des tables gauche et droite et remplit les champs manquants de chaque côté avec des valeurs NULL.

Voici un exemple de code 4D effectuant une jointure externe bilatérale :

```
TABLEAU TEXTE (aName;0)
```

```
TABLEAU TEXTE (aDepName;0)
```

```
TABLEAU ENTIER (aEmpDepID;0
```

```
TABLEAU ENTIER (aDepID;0)
```

```
Debut SQL
```

```
SELECT *
    FROM employees
    FULL OUTER JOIN departments
        ON employees.DepID = departments.DepID;
    INTO :aName, :aEmpDepID, :aDepID, :aDepName;
```

```
Fin SQL
```

Résultat de cette jointure avec notre base d'exemple (les lignes additionnelles sont en rouge) :

aName	aEmpDepID	aDepID	aDepName
Alain	10	10	Program
Anne	11	11	Engineering
Bernard	10	10	Program
Fabrice	12	12	Development
Thomas	10	10	Program
Martine	15	NULL	NULL
Philippe	NULL	NULL	NULL
NULL	NULL	NULL	Marketing
NULL	NULL	13	Quality

Jointures multiples dans une seule requête

Il est possible de combiner plusieurs jointures dans une même instruction *SELECT*. Il est également possible de mixer des jointures internes implicites ou explicites et des jointures externes explicites.

Voici un exemple de code 4D effectuant des jointures multiples :

```
TABLEAU TEXTE (aName;0)
TABLEAU TEXTE (aDepName;0)
TABLEAU TEXTE (aCityName;0)
TABLEAU ENTIER (aEmpDepID;0)
TABLEAU ENTIER (aEmpCityID;0)
TABLEAU ENTIER (aDepID;0)
TABLEAU ENTIER (aCityID;0)
Debut SQL
    SELECT *
        FROM (employees RIGHT OUTER JOIN departments
            ON employees.depID = departments.depID)
            LEFT OUTER JOIN cities
            ON employees.cityID = cities.cityID
        INTO :aName, :aEmpDepID, :aEmpCityID, :aDepID, :aDepName, :aCityID, :aCityName;
Fin SQL
```

Résultat de cette jointure avec notre base d'exemple :

aName	aEmpDepID	aEmpCityID	aDepID	aDepName	aCityID	aCityName
Alain	10	30	10	Program	30	Paris
Anne	11	39	11	Engineering	0	
Bernard	10	33	10	Program	33	New York
Fabrice	12	35	12	Development	0	
Thomas	10	NULL	10	Program	0	

Commandes SQL

Commandes SQL

 SELECT

 INSERT

 UPDATE

 DELETE

 CREATE DATABASE

 USE DATABASE

 ALTER DATABASE


 CREATE TABLE

 ALTER TABLE

 DROP TABLE

 CREATE INDEX

 DROP INDEX

 LOCK TABLE

 UNLOCK TABLE

 EXECUTE IMMEDIATE

 CREATE SCHEMA

 ALTER SCHEMA

 DROP SCHEMA

 GRANT

 REVOKE

 REPLICATE

 SYNCHRONIZE

Commandes SQL

Les commandes SQL sont généralement classées en deux catégories :

- Les commandes de manipulation de données, utilisées pour récupérer, ajouter, supprimer et/ou modifier des informations dans la base de données. En particulier, cette catégorie inclut les commandes *SELECT*, **INSERT**, *UPDATE* et *DELETE*.
- Les commandes de définition de données, utilisées pour créer ou supprimer des objets de structure de la base de données ou des bases de données elles-mêmes. Cette catégorie inclut notamment les commandes **CREATE DATABASE**, *CREATE TABLE*, *ALTER TABLE*, *DROP INDEX*, *DROP TABLE* ou *CREATE SCHEMA*.

Dans la syntaxe de chaque commande, les noms et les mots-clés apparaissent en **caractères gras** et doivent être passés tels quels. Les autres éléments des instructions apparaissent en caractères italiques et sont détaillés dans le chapitre . Les mots-clés et/ou les clauses inclus entre crochets [] sont optionnels. Le caractère barre verticale | sépare différentes alternatives. Lorsque des éléments sont inclus entre accolades {} et séparés par des barres verticales, un seul élément de l'ensemble doit être passé.

SELECT

```
SELECT [ALL | DISTINCT]
{* | select_élément, ..., select_élément}
FROM ref_table, ..., ref_table
[WHERE critère_recherche]
[ORDER BY liste_tri]
[GROUP BY liste_tri]
[HAVING critère_recherche]
[LIMIT {ref_langage_4d | nombre_entier | ALL}]
[OFFSET ref_langage_4d | nombre_entier ]
[INTO {ref_langage_4d, ..., ref_langage_4d}]
[FOR UPDATE]
```

Description

La commande **SELECT** permet de récupérer des données stockées dans une ou plusieurs table(s).

Si vous passez le paramètre *****, toutes les colonnes seront retournées ; ou bien, vous pouvez passer un ou plusieurs paramètre(s) de type *select_élément* pour spécifier individuellement chaque colonne à sélectionner (les colonnes doivent être séparées par des virgules). Si vous ajoutez le mot-clé facultatif **DISTINCT** à l'instruction **SELECT**, les valeurs dupliquées ne seront pas retournées. Il n'est pas possible d'exécuter des requêtes contenant à la fois le "*" et des champs explicites. Par exemple, l'instruction suivante :

```
SELECT *, SALES, TARGET FROM OFFICES
```

... n'est pas autorisée, tandis que :

```
SELECT * FROM OFFICES
```

... est valide.

La clause **FROM** doit être suivie d'un ou plusieurs argument(s) de type *ref_table* permettant de désigner la ou les table(s) contenant les données à sélectionner. Vous pouvez passer comme argument soit un nom SQL standard soit une chaîne. Il n'est pas possible de passer d'expression de type requête à la place d'un nom de table.

Facultativement, vous pouvez passer le mot-clé **AS** afin d'affecter un alias à la colonne. Si ce mot-clé est passé, il doit être suivi du nom de l'alias qui peut également être soit un nom SQL standard soit une chaîne.

Toutes les clauses suivantes sont facultatives.

La clause **WHERE** définit des conditions que les valeurs doivent satisfaire pour être sélectionnées. Pour cela, passez un *critère_recherche* qui sera appliqué aux données récupérées par la clause **FROM**. L'expression *critère_recherche* retourne toujours une valeur de type booléen.

La clause **ORDER BY** peut être utilisée afin d'indiquer que les données sélectionnées doivent être triées en fonction des valeurs de la *liste_tri*. Vous pouvez ajouter le mot-clé **ASC** ou **DESC** afin de définir si le tri doit être ascendant ou descendant. Par défaut, le tri est ascendant.

La clause **GROUP BY** peut être utilisée afin de demander le regroupement des données identiques en fonction des valeurs de la *liste_tri*, passée en argument. Vous pouvez passer des colonnes de groupes multiples. Cette clause permet généralement d'éviter les redondances ou de traiter une fonction d'agrégation (**SUM**, **COUNT**, **MIN** ou **MAX**) qui sera appliquée à ces groupes. Vous pouvez également ajouter le mot-clé **ASC** ou **DESC** comme avec la clause **ORDER BY**.

La clause **HAVING** peut être utilisée afin d'appliquer un *critère_recherche* à l'un des groupes. La clause **HAVING** peut être utilisée sans la clause **GROUP BY**.

La clause **LIMIT** permet de restreindre le nombre de données sélectionnées à la quantité définie par la variable *ref_langage_4d* ou le *nombre_entier*.

La clause **OFFSET** permet de définir un nombre (variable *ref_langage_4d* ou *nombre_entier*) de valeurs à ignorer avant de

débuter le décompte pour l'application de la clause **LIMIT**.

La clause **INTO** permet de désigner des variables *ref langage 4d* auxquelles les données sélectionnées seront affectées. Vous pouvez également passer le mot-clé **LISTBOX** afin de placer les données dans la list box désignée par le paramètre *ref langage 4d*.

Une instruction **SELECT** comportant la clause **FOR UPDATE** tentera de verrouiller en écriture tous les enregistrements sélectionnés. Si un enregistrement au moins ne peut pas être verrouillé, l'ensemble de la commande échouera et une erreur sera retournée. Dans le cas contraire, les enregistrements resteront verrouillés jusqu'à ce que la transaction courante soit validée ou annulée.

Exemple 1

Imaginons une base de données de films contenant une table qui, pour chaque film, stocke le titre, l'année de sortie et le nombre d'entrées vendues.

Nous voulons sélectionner les années depuis 1979 ainsi que le nombre d'entrées pour les films ayant fait moins de 10 millions d'entrées. Nous souhaitons ignorer les 5 premières années et afficher uniquement 10 années, classées par ordre croissant.

```
C_ENTIER LONG ($AnneeFilm;$EntreesMin;$Depart;$Limite)
TABLEAU ENTIER (tAnneeFilm;0)
TABLEAU ENTIER LONG (tEntreesMin;0)
$AnneeFilm:=1979
$EntreesMin:=10000000
$Depart:=5
$Limite:=10
```

Debut SQL

```
SELECT Annee_Film, SUM(Entrees)
FROM FILMS
WHERE Annee_Film >= :$AnneeFilm
GROUP BY Annee_Film
HAVING SUM(Entrees) < :$EntreesMin
ORDER BY 1
LIMIT :$Limite
OFFSET :$Depart
INTO :tAnneeFilm, :tEntreesMin;
```

Fin SQL

Exemple 2

Cet exemple utilise une combinaison de critères de recherches :

```
SELECT id_fournisseurs
FROM fournisseurs
WHERE (nom= 'CANON')
      OR (nom= 'Hewlett Packard' AND city = 'New York')
      OR (nom= 'Firewall' AND status = 'Closed' AND city = 'Chicago');
```

Exemple 3

Soit la table **COMMERCIAUX** où **QUOTA** est le montant des ventes prévu pour le commercial et **VENTES** le montant des ventes effectivement réalisé.

```
TABLEAU REEL (tValeurs_Min;0)
TABLEAU REEL (tValeurs_Max;0)
TABLEAU REEL (tValeurs_Sommes;0)
Debut SQL
SELECT MIN ( ( VENTES * 100 ) / QUOTA ),
MAX ( ( VENTES * 100 ) / QUOTA ),
SUM ( QUOTA ) - SUM ( VENTES )
FROM COMMERCIAUX
INTO :tValeurs_Min, :tValeurs_Max, :tValeurs_Sommes;
Fin SQL
```

Exemple 4

Cet exemple sélectionne tous les acteurs nés dans une ville donnée :

```
TABLEAU TEXTE (tNomsActeurs;0)
```

```
TABLEAU TEXTE (tVilles;0)
```

```
Debut SQL
```

```
SELECT ACTEURS.Prenom, VILLES.Nom  
FROM ACTEURS AS 'Act', VILLES AS 'Vil'  
WHERE Act.ID_Ville_Naissance=Vil.ID_Ville  
ORDER BY 2 ASC  
INTO : tNomsActeurs, : tVilles;
```

```
Fin SQL
```

```
INSERT INTO {nom_sql | chaîne_sql}
[(ref_colonne, ..., ref_colonne)]
[VALUES({[INFILE]expression_arithmétique | NULL}, ..., {[INFILE]expression_arithmétique | NULL};) | sous_requête]
```

Description

La commande **INSERT** permet d'ajouter des données dans une table existante. La table dans laquelle les données sont insérées est désignée via un argument de type *nom_sql* ou *chaîne_sql*. Les arguments facultatifs *ref_colonne* permettent de définir les colonnes dans lesquelles insérer les valeurs. Si aucune *ref_colonne* n'est passée, les valeurs seront insérées dans l'ordre des colonnes de la base (la première valeur passée sera insérée dans la première colonne, la deuxième dans la deuxième colonne, et ainsi de suite).

Le mot-clé **VALUES** permet de passer la ou les valeur(s) à insérer dans la ou les colonne(s) spécifiée(s). Vous pouvez passer soit une *expression_arithmétique*, soit **NULL**. Alternativement, une *sous_requête* peut être passée au mot-clé **VALUES** afin d'insérer une sélection de données en tant que valeurs.

Le nombre de valeurs passées via le mot-clé **VALUES** doit correspondre au nombre de colonnes défini par le ou les argument(s) *ref_colonne*. En outre, le type des données insérées doit correspondre à celui des colonnes de la table, ou au moins pouvoir être converti dans ce type.

Le mot-clé **INFILE** permet d'utiliser le contenu d'un fichier externe pour définir les valeurs d'un nouvel enregistrement. Ce mot-clé doit être utilisé uniquement avec des expressions de type VARCHAR. Lorsque le mot-clé **INFILE** est passé, la valeur *expression_arithmétique* est évaluée en tant que chemin d'accès de fichier ; si le fichier est trouvé, le contenu du fichier est inséré dans la colonne correspondante. Seuls des champs de type texte ou BLOB peuvent recevoir des valeurs issues d'un **INFILE**. Le contenu du fichier est transféré sous forme de données brutes, sans interprétation.

Le fichier recherché doit se trouver sur l'ordinateur hébergeant le moteur SQL, même si la requête provient d'un client distant. De même, le chemin d'accès doit être exprimé en respectant la syntaxe du système d'exploitation du moteur SQL. Il peut être absolu ou relatif.

La commande **INSERT** est utilisable dans les requêtes mono et multi-lignes. Toutefois, une requête **INSERT** multi-lignes ne permet pas d'effectuer d'opérations **UNION** et **JOIN**.

Le moteur de 4D admet les insertions multi-lignes de valeurs, ce qui permet d'alléger et d'optimiser le code, notamment lors de l'insertion de grandes quantités de données. La syntaxe des insertions multi-lignes est du type :

```
INSERT INTO {nom_sql | chaîne_sql}
[(ref_colonne, ..., ref_colonne)]
VALUES(expression_arithmétique, ..., expression_arithmétique), ..., (expression_arithmétique,
..., expression_arithmétique);
```

Cette syntaxe est illustrée dans les exemples 3 et 4.

Exemple 1

Cet exemple simple permet d'insérer une sélection de la table2 dans la table1 :

```
INSERT INTO table1 (SELECT * FROM table2)
```

Exemple 2

Cet exemple crée une table et insère des valeurs :

```
CREATE TABLE ACTEUR_FANS
(ID INT32, Nom VARCHAR);
INSERT INTO ACTEUR_FANS
(ID, Nom)
VALUES (1, 'Francis');
```

Exemple 3

La syntaxe multi-lignes permet d'éviter la répétition fastidieuse de lignes :

```
INSERT INTO MaTable
(Chp1,Chp2,ChpBol,ChpDate,ChpHeure, ChpInfo)
VALUES
(1,1,1,'11/01/01','11:01:01','Première ligne'),
(2,2,0,'12/01/02','12:02:02','Deuxième ligne'),
(3,3,1,'13/01/03','13:03:03','Troisième ligne'),
.....
(7,7,1,'17/01/07','17:07:07','Septième ligne');
```

Exemple 4

Vous pouvez également utiliser des variables ou des tableaux 4D avec la syntaxe multi-lignes :

```
INSERT INTO MaTable
(Chp1,Chp2,ChpBol,ChpDate,ChpHeure, ChpInfo)
VALUES
( :vTabId, :vTabIdx, :vTabbol, :vTabdate, :vTabL, :vTabText);
```

Note : Vous ne pouvez pas combiner des variables simples et des tableaux dans la même instruction **INSERT**.

```
UPDATE {nom_sql | chaîne_sql}  
SET nom_sql = {expression_arithmétique | NULL}, ..., nom_sql = {expression_arithmétique | NULL}  
[WHERE critère_recherche]
```

Description

La commande *UPDATE* permet de modifier les données contenues dans la table désignée par l'argument *nom_sql* ou *chaîne_sql*.

La clause **SET** permet de définir les nouvelles valeurs à assigner (via une *expression_arithmétique* ou la valeur **NULL**).

La clause **WHERE** (facultative) peut être utilisée pour sélectionner, via l'évaluation de l'expression *critère_recherche*, les données à mettre à jour. Si cette clause n'est pas passée, toutes les données de la table seront mises à jour à l'aide des nouvelles valeurs définies dans la clause **SET**.

La commande *UPDATE* est prise en charge dans les requêtes et les sous-requêtes. Toutefois, l'instruction *UPDATE* positionnée n'est pas permise. Les mises à jour en cascade sont implémentées dans 4D, mais les règles de suppression **SET NULL** et **SET DEFAULT** ne sont pas prises en charge.

Exemple

L'exemple suivant modifie la table FILMS pour porter à 3 500 000 le nombre d'entrées pour le film "Air Force One" :

```
UPDATE FILMS  
SET nombre_entrees = 3500000  
WHERE Titre = 'Air Force One';
```

DELETE

```
DELETE FROM {nom_sql | chaîne_sql}  
[WHERE critère_recherche]
```

Description

La commande *DELETE* permet de supprimer une partie ou la totalité des données de la table désignée par l'argument *nom_sql* ou *chaîne_sql* après le mot-clé **FROM**.

La clause **WHERE** (facultative) peut être utilisée pour sélectionner, via l'évaluation de l'expression *critère_recherche*, les données à supprimer. Si cette clause n'est pas passée, toutes les données de la table seront supprimées.

L'instruction *DELETE* positionnée n'est pas prise en charge. Les suppressions en cascade sont implémentées dans 4D, mais les règles de suppression **SET DEFAULT** et **SET NULL** ne sont pas prises en charge.

Exemple

Dans cet exemple nous supprimons de la table FILMS tous les films dont l'année de sortie est égale ou antérieure à 2000 :

```
DELETE FROM FILMS  
WHERE Annee_sortie <= 2000;
```

CREATE DATABASE [IF NOT EXISTS] DATAFILE <Chemin d'accès complet>

Description

La commande **CREATE DATABASE** vous permet de créer une nouvelle base de données externe (fichiers standard .4db et .4dd) à un emplacement spécifique.

Si la contrainte **IF NOT EXISTS** est passée, la base de données n'est pas créée et aucune erreur n'est générée si une base du même nom existe déjà à l'emplacement défini.

Si la contrainte **IF NOT EXISTS** n'est pas passée, la base de données n'est pas créée et le message d'erreur "Cette base de données existe déjà. Echec de la commande CREATE DATABASE" est affiché si une base du même nom existe déjà à l'emplacement défini.

La clause **DATAFILE** vous permet de définir le nom complet (chemin d'accès complet + nom) de la nouvelle base de données externe. Vous devez passer le nom du fichier de structure. Le programme ajoute automatiquement l'extension ".4db" au fichier si elle n'est pas déjà définie et crée le fichier de données. Le chemin d'accès peut être exprimé soit en syntaxe POSIX, soit en syntaxe système. Il peut être absolu ou relatif au fichier de structure de la base 4D principale.

- syntaxe POSIX (type URL) : les noms de dossiers sont séparés par une barre oblique ("/"), quelle que soit la plate-forme que vous utilisez, par exemple : ../basesexternes/mabase.4db
- syntaxe système : chemin d'accès respectant la syntaxe de la plate-forme courante, par exemple :
 - (Mac OS) Disque:Applications:monserv:basesexternes:mabase.4db
 - (Windows) C:\Applications\monserv\basesexternes\mabase.4db

Après l'exécution réussie de la commande **CREATE DATABASE**, la nouvelle base de données créée ne devient pas automatiquement la base courante. Pour cela, vous devez explicitement la déclarer en tant que base courante à l'aide de la commande **USE DATABASE**.

A propos des bases externes

Une base externe est une base 4D indépendante de la base 4D principale, mais que vous pouvez utiliser et manipuler depuis la base 4D principale, en utilisant le moteur SQL de 4D. Utiliser une base externe signifie désigner temporairement cette base comme base courante, c'est-à-dire comme base cible des requêtes SQL exécutées par 4D. Par défaut, la base courante est la base principale.

Vous pouvez créer autant de bases externes que vous voulez depuis la base 4D principale avec la commande **CREATE DATABASE**. Une fois créée, une base externe peut être désignée comme base courante à l'aide de la commande **USE DATABASE**. Elle peut alors être modifiée via les commandes SQL standard (*CREATE TABLE*, *ALTER TABLE*, etc.) et vous pouvez y stocker des données. La fonction **DATABASE_PATH** permet de connaître à tout moment la base courante.

L'intérêt majeur des bases externes réside dans le fait qu'elles peuvent être créées et manipulées via des composants 4D. Ce principe permet de développer des composants pouvant créer des tables et des champs en fonction de leurs besoins.

Note : Une base externe est une base 4D standard. Elle peut être ouverte et manipulée en tant que base principale par une application 4D ou 4D Server. Inversement, toute base 4D standard peut être utilisée comme base externe. Toutefois, il est impératif de ne pas activer le système de gestion des accès (via l'affectation d'un mot de passe au Super_Utilisateur) dans une base externe sinon elle ne sera plus accessible via la commande **USE DATABASE**.

Exemple 1

Création des fichiers de base externe ExternalDB.4DB et ExternalDB.4DD à l'emplacement C:/MaBase/ :

```
Debut SQL
CREATE DATABASE IF NOT EXISTS DATAFILE 'C:/MaBase/ExternalDB';
Fin SQL
```

Exemple 2

Création des fichiers de base externe TestDB.4DB et TestDB.4DD à côté du fichier de structure de la base principale :

```
Debut SQL
```



```
CREATE DATABASE IF NOT EXISTS DATAFILE 'TestDB';  
Fin SQL
```

Exemple 3

Création des fichiers de base externe External.4DB et External.4DD à l'emplacement défini par l'utilisateur :

```
C_TEXTE($chemin)  
$chemin:=Selectionner dossier("Dossier de destination de la base externe :")  
$chemin:=$chemin+"External"  
Debut SQL  
CREATE DATABASE DATAFILE <<$chemin>>;  
Fin SQL
```

USE [LOCAL | REMOTE] DATABASE

{**DATAFILE** <Chemin d'accès complet> | **SQL_INTERNAL** | **DEFAULT**}
[**AUTO_CLOSE**]

Description

La commande **USE DATABASE** vous permet de désigner une base externe 4D comme base de données courante, c'est-à-dire vers laquelle seront dirigées les prochaines requêtes SQL dans le process courant. Tous les types de requêtes SQL sont concernés : requêtes incluses dans une structure **Debut SQL/Fin SQL**, commandes **SQL EXECUTER** ou **SQL EXECUTER SCRIPT**, etc.

Note : Pour plus d'informations sur les bases externes, reportez-vous à la description de la commande **CREATE DATABASE**.

- Si vous travaillez en configuration monoposte, la base externe doit être située sur votre machine 4D.
- Si vous travaillez en mode distant, la base externe peut être située sur le poste local ou sur la machine 4D Server.

Si vous utilisez 4D en mode distant, le mot-clé **REMOTE** vous permet de désigner une base externe située sur 4D Server. Pour des raisons de sécurité, ce mécanisme fonctionne uniquement avec les connexions distantes natives, c'est-à-dire dans le contexte d'une base 4D distante connectée à 4D Server. Les connexions via ODBC ou pass-through ne sont pas autorisées. Si aucun mot-clé n'est spécifié, l'option **LOCAL** est utilisée par défaut. Si vous utilisez 4D en mode local, les mot-clés **REMOTE** et **LOCAL** sont ignorés : les connexions sont toujours locales.

Pour désigner la base externe à utiliser, passez son chemin complet (chemin d'accès + nom) dans la clause **DATAFILE**. Le chemin d'accès peut être exprimé soit en syntaxe POSIX, soit en syntaxe système. Il peut être absolu ou relatif au fichier de structure de la base 4D principale.

En mode distant, si le mot-clé **REMOTE** est passé, ce paramètre désigne le chemin d'accès de la base à partir du poste serveur. S'il est omis ou si le mot-clé **LOCAL** est passé, ce paramètre désigne le chemin d'accès de la base sur le poste 4D local.

Important : Vous devez désigner une base 4D externe valide et dans laquelle le système de contrôle des accès n'a pas été activé (via l'attribution d'un mot de passe au Super_Utilisateur). Dans le cas contraire, une erreur est générée.

Pour rétablir la base principale en tant que base courante, exécutez la commande en passant le mot-clé **SQL_INTERNAL** ou **DEFAULT**.

Passez **AUTO_CLOSE** si vous souhaitez fermer physiquement la base externe à l'issue de son utilisation, c'est-à-dire lorsque vous changerez de base courante. En effet, l'ouverture d'une base externe étant une opération qui nécessite du temps, pour des raisons d'optimisation 4D maintient en mémoire des informations relatives aux bases externes ouvertes durant la session utilisateur. Ces informations sont maintenues en mémoire tant que l'application 4D est lancée. Les réouvertures suivantes d'une même base externe sont alors accélérées. Toutefois, ce principe empêche le partage des bases externes entre plusieurs applications 4D car la base externe reste ouverte en lecture/écriture pour la première application qui l'a utilisée. Si plusieurs applications 4D doivent pouvoir utiliser simultanément une même base externe, passez le mot-clé **AUTO_CLOSE** afin de libérer physiquement la base externe après son utilisation.

Cette restriction ne s'applique pas aux process d'une même application : différents process d'une application peuvent toujours accéder à une même base externe en lecture/écriture sans qu'il soit nécessaire de forcer sa fermeture.

A noter que lorsque plusieurs process utilisent la même base externe, elle n'est libérée physiquement que lorsque le dernier process qui l'utilise est refermé, même lorsque l'option **AUTO_CLOSE** a été passée. Vous devez tenir compte de ce fonctionnement pour les opérations de partage inter-applications et de suppression des bases externes.

Exemple

Utilisation d'une base externe pour une requête puis retour à la base principale :

```
Debut SQL
USE DATABASE DATAFILE 'C:/MaBase/Noms'
SELECT Name FROM emp INTO :tNoms1
USE DATABASE SQL_INTERNAL
Fin SQL
```

ALTER DATABASE {ENABLE | DISABLE} {INDEXES | CONSTRAINTS}

Description

La commande **ALTER DATABASE** active ou inactive des options SQL de la base courante pour la session courante.

Cette commande vous permet de désactiver temporairement des options SQL afin d'accélérer certaines opérations consommatrices en ressources. Par exemple, désactiver les index et les contraintes avant de débiter un import d'une grande quantité de données peut réduire de façon significative la durée de l'import. Les contraintes incluent les clés primaires, les clés étrangères, les attributs d'unicité et de nullité.

ALTER DATABASE s'applique à la totalité de la base. Autrement dit, si un utilisateur inactive une option, elle sera inactivée pour tous les utilisateurs de la base.

Exemple

Exemple d'import avec désactivation temporaire de toutes les options SQL :

```
Debut SQL
ALTER DATABASE DISABLE INDEXES;
ALTER DATABASE DISABLE CONSTRAINTS;
Fin SQL
SQL EXECUTER SCRIPT("C:\Exported_data\Export.sql";SQL Continuer si erreur)
Debut SQL
ALTER DATABASE ENABLE INDEXES;
ALTER DATABASE ENABLE CONSTRAINTS;
Fin SQL
```

CREATE TABLE

```
CREATE TABLE [IF NOT EXISTS] {nom_sql.}nom_sql({définition_colonne |contrainte_table}[PRIMARY KEY], ... ,
{définition_colonne |contrainte_table} [PRIMARY KEY]) [{ENABLE | DISABLE} REPLICATE]
```

Description

La commande *CREATE TABLE* permet de créer une table nommée *nom_sql* comportant un ou plusieurs champ(s) défini(s) par les paramètres *définition_colonne* et/ou *contrainte_table*.

Si la contrainte **IF NOT EXISTS** est passée, la table n'est pas créée et aucune erreur n'est générée si une table du même nom existe déjà dans la base.

Le premier paramètre *nom_sql* (facultatif) permet de désigner le schéma SQL auquel vous souhaitez affecter la table. Si vous ne passez pas ce paramètre ou si vous passez un nom de schéma inexistant, la table est automatiquement affectée au schéma par défaut, nommé "DEFAULT_SCHEMA". Pour plus d'informations sur les schémas SQL, reportez-vous à la section [Implémentations du moteur SQL de 4D](#).

Note : Il est également possible d'affecter une table à un schéma SQL via le pop up menu "Schémas" situé dans l'Inspecteur de tables de 4D. Ce menu contient la liste des schémas définis dans la base.

Une *définition_colonne* contient le nom (*nom_sql*) et le type de données (*type_données_sql*) d'une colonne et une *contrainte_table* restreint les valeurs qu'une table peut stocker.

Le mot-clé **PRIMARY KEY** permet de définir la clé primaire au moment de la création de la table. Pour plus d'informations sur les clés primaires, reportez-vous à la section [Implémentations du moteur SQL de 4D](#).

Les mots-clés **ENABLE REPLICATE** et **DISABLE REPLICATE** permettent d'activer et d'inactiver le mécanisme autorisant la réplication de la table (cf. section).

Exemple 1

Voici un exemple simple de création de table avec 2 colonnes :

```
CREATE TABLE ACTEUR_FANS
(ID INT32, Nom VARCHAR);
```

Exemple 2

Cet exemple crée la même table que précédemment mais uniquement s'il n'y a pas de table existante du même nom :

```
CREATE TABLE IF NOT EXISTS ACTEUR_FANS
(ID INT32, Nom VARCHAR);
```

Exemple 3

Cet exemple crée la table "Preferences" et l'affecte au schéma "Control" :

```
CREATE TABLE Control.Preferences
(ID INT32, Value VARCHAR);
```

ALTER TABLE

```
ALTER TABLE nom_sql
{ADD définition_colonne [PRIMARY KEY] |
DROP nom_sql |
ADD définition_clé_primaire |
DROP PRIMARY KEY |
ADD définition_clé_étrangère |
DROP CONSTRAINT nom_sql |
[{ENABLE | DISABLE} REPLICATE] |
SET SCHEMA nom_sql}
```

Description

La commande **ALTER TABLE** permet de modifier la table existante désignée par l'argument *nom_sql*. Vous pouvez effectuer l'une des actions suivantes :

Passer **ADD** *définition_colonne* ajoute une colonne à la table. Le mot-clé **PRIMARY KEY** permet de définir la clé primaire au moment de l'ajout de la colonne.

Passer **DROP** *nom_sql* supprime la colonne *nom_sql* de la table.

Passer **ADD** *définition_clé_primaire* ajoute une clé primaire (**PRIMARY KEY**) à la table.

Passer **DROP PRIMARY KEY** supprime la clé primaire (**PRIMARY KEY**) de la table.

Passer **ADD** *définition_clé_étrangère* ajoute une clé étrangère (**FOREIGN KEY**) à la table.

Passer **DROP CONSTRAINT** *nom_sql* supprime la contrainte spécifiée par *nom_sql* de la table.

Passer **ENABLE REPLICATE** ou **DISABLE REPLICATE** permet d'activer et d'inactiver le mécanisme autorisant la réplification de la table (cf. section).

Passer **SET SCHEMA** *nom_sql* transfère la table vers le schéma *nom_sql*.

Exemple

Cet exemple crée une table, y insère un ensemble de valeurs, puis ajoute une colonne Num_Tel, insère un autre ensemble de valeurs puis supprime la colonne ID :

```
CREATE TABLE ACTEUR_FANS
(ID INT32, Nom VARCHAR);

INSERT INTO ACTOR_FANS
(ID, Nom)
VALUES (1, 'Francis');

ALTER TABLE ACTEUR_FANS
ADD Num_Tel VARCHAR;

INSERT INTO ACTEUR_FANS
(ID, Nom, Num_Tel)
VALUES (2, 'Florence', '01446677888');

ALTER TABLE ACTEUR_FANS
DROP ID;
```

DROP TABLE

DROP TABLE [**IF EXISTS**] *nom_sql*

Description

La commande *DROP TABLE* permet de supprimer la table nommée *nom_sql* de la base. Lorsque la contrainte **IF EXISTS** est passée, si la table à supprimer n'existe pas dans la base, la commande ne fait rien et aucune erreur n'est générée.

Cette commande ne supprime pas uniquement la table de la structure, elle efface également ses données ainsi que tous les éventuel index, triggers et contraintes qui lui sont associés. Il n'est pas possible d'utiliser cette commande avec une table référencée par une contrainte **FOREIGN KEY**.

Note : Vous devez veiller à ce qu'au moment de l'exécution de la commande *DROP TABLE* aucun enregistrement de la table *nom_sql* ne soit chargé en mémoire en lecture écriture. Dans le cas contraire, l'erreur 1272 est générée.

Exemple 1

Cet exemple supprime la table ACTEUR_FANS :

```
DROP TABLE ACTEUR_FANS
```

Exemple 2

Cet exemple supprime la même table que dans l'exercice précédent, sauf que dans ce cas, si la table ACTEUR_FANS n'existe pas, aucune erreur n'est générée :

```
DROP TABLE IF EXISTS ACTEUR_FANS
```

CREATE INDEX

```
CREATE [UNIQUE] INDEX nom_sql ON nom_sql (ref_colonne, ... , ref_colonne)
```

Description

La commande *CREATE INDEX* permet de créer un index (*nom_sql*) sur une ou plusieurs colonne(s), désignée(s) par le ou les paramètre(s) *ref_colonne*, d'une table existante (*nom_sql*). Les index sont transparents pour les utilisateurs et permettent d'accélérer l'exécution des requêtes.

Vous pouvez également passer le mot-clé facultatif **UNIQUE** afin de créer un index ne permettant pas la saisie de valeurs dupliquées.

Exemple

Voici un exemple simple de création d'index :

```
CREATE INDEX ID_INDEX ON ACTEUR_FANS (ID)
```

DROP INDEX

```
DROP INDEX nom_sql
```

Description

La commande *DROP INDEX* permet de supprimer de la base l'index existant désigné par *nom_sql*. Elle ne peut pas être utilisée avec les index créés pour les contraintes **PRIMARY KEY** ou **UNIQUE**.

Exemple

Voici un exemple simple de suppression d'index :

```
DROP INDEX ID_INDEX
```


LOCK TABLE

```
LOCK TABLE nom_sql IN {EXCLUSIVE | SHARE} MODE
```

Description

La commande *LOCK TABLE* permet de verrouiller la table nommée *nom_sql* dans le mode **EXCLUSIVE** ou le mode **SHARE**.

Dans le mode **EXCLUSIVE**, les données de la table ne peuvent pas être lues ou modifiées par une autre transaction.

Dans le mode **SHARE**, les données de la table peuvent être lues par d'autres transactions mais les modifications restent interdites.

Exemple

Cet exemple verrouille la table FILMS de manière à ce qu'elle puisse être lue mais non modifiée par d'autres transactions :

```
LOCK TABLE FILMS IN SHARE MODE
```

UNLOCK TABLE

```
UNLOCK TABLE nom_sql
```

Description

La commande *UNLOCK TABLE* permet de déverrouiller la table *nom_sql* ayant été préalablement verrouillée à l'aide de la commande *LOCK TABLE*. Cette commande ne fonctionnera pas si elle est appelée à l'intérieur d'une transaction ou si elle est utilisée sur une table verrouillée par un autre process.

Exemple

Cette commande supprime le verrou de la table FILMS :

```
UNLOCK TABLE FILMS
```

EXECUTE IMMEDIATE

```
EXECUTE IMMEDIATE <<nom_sql>> | <<$nom_sql>> | :nom_sql | :$nom_sql
```

Description

La commande *EXECUTE IMMEDIATE* permet d'exécuter une expression SQL dynamique. L'argument *nom_sql* représente une variable contenant un ensemble d'instructions SQL qui seront exécutées en bloc.

Notes

- La commande *EXECUTE IMMEDIATE* ne peut pas être utilisée dans le cadre d'une connexion SQL externe (pass-through) initiée via la commande 4D **UTILISER BASE EXTERNE**.
- En mode compilé, il n'est pas possible d'utiliser des variables 4D locales (débutant par le caractère \$) dans la chaîne de requête passée à la commande *EXECUTE IMMEDIATE*.

Exemple

L'exemple suivant récupère le nombre de films réalisés depuis 1960 présents dans la base :

```
C_ENTIER LONG (NbFilms)
C_TEXTE ($tQueryTxt)
NbFilms:=0

$tQueryTxt:="SELECT COUNT(*) FROM FILMS WHERE Annee_Film >= 1960 INTO :NbFilms;"
Debut SQL
    EXECUTE IMMEDIATE :$tQueryTxt;
Fin SQL

ALERTE("La vidéothèque contient"+Chaine(NbFilms)+" films réalisés depuis 1960")
```

CREATE SCHEMA

```
CREATE SCHEMA nom_sql
```

Description

La commande *CREATE SCHEMA* permet de créer un nouveau schéma SQL nommé *nom_sql* dans la base de données.

Note : Pour plus d'informations sur les schémas, reportez-vous à la section **Implémentations du moteur SQL de 4D**.

Lorsque vous créez un nouveau schéma, par défaut les droits associés sont les suivants :

- Lecture seulement (Données) : <Tout le monde>
- Lecture/Ecriture (Données) : <Tout le monde>
- Complet (Données & Structure) : <Personne>

Chaque schéma peut se voir attribuer un type de droits d'accès externes à l'aide de la commande *GRANT*.

Seuls le Super_Utilisateur et l'Administrateur de la base peuvent créer, modifier ou supprimer des schémas.

Si le système de gestion des accès de 4D n'est pas activé (c'est-à-dire, si aucun mot de passe n'est assigné au Super_Utilisateur), tous les utilisateurs peuvent créer et modifier des schémas sans restriction.

Lorsqu'une base de données est créée ou convertie avec 4D v11 SQL (à compter de la release 3), un schéma par défaut est créé afin de regrouper toutes les tables de la base. Ce schéma est nommé "DEFAULT_SCHEMA". Il ne peut pas être supprimé ni renommé.

Exemple

Création d'un schéma nommé "Droits_Compta" :

```
CREATE SCHEMA Droits_Compta
```

ALTER SCHEMA

```
ALTER SCHEMA nom_sql RENAME TO nom_sql
```

Description

La commande *ALTER SCHEMA* permet de renommer le schéma SQL *nom_sql* (premier paramètre) en *nom_sql* (second paramètre).

Seuls le Super_Utilisateur et l'Administrateur de la base peuvent modifier des schémas.

Exemple

Renommage du schéma MyFirstSchema en MyLastSchema :

```
ALTER SCHEMA MyFirstSchema RENAME TO MyLastSchema
```

DROP SCHEMA

DROP SCHEMA *nom_sql*

Description

La commande *DROP SCHEMA* permet de supprimer le schéma désigné par *nom_sql*.

Il est possible de supprimer tout schéma, à l'exception du schéma par défaut (DEFAULT_SCHEMA). Lorsque vous supprimez un schéma, toutes les tables qui lui étaient affectées sont transférées au schéma par défaut. Les tables transférées héritent des droits d'accès du schéma par défaut.

Si vous tentez de supprimer un schéma inexistant ou ayant déjà été supprimé, une erreur est générée.

Seuls le Super_Utilisateur et l'Administrateur de la base peuvent supprimer des schémas.

Exemple

Vous souhaitez supprimer le schéma MyFirstSchema (auquel sont affectées les tables Table1 et Table2) :

```
DROP SCHEMA MyFirstSchema
```

Après cette opération, les deux tables Table1 et Table2 sont réaffectées au schéma par défaut.

```
GRANT [READ | READ_WRITE | ALL] ON nom_sql TO nom_sql
```

Description

La commande *GRANT* permet de définir les droits d'accès associés au schéma *nom_sql*. Ces droits seront attribués au groupe d'utilisateurs 4D désigné par le second paramètre *nom_sql*.

Les mots-clés **READ**, **READ_WRITE** et **ALL** permettent de définir les types d'accès autorisés pour la table :

- **READ** instaure le mode d'accès Lecture seulement (données)
- **READ_WRITE** instaure le mode d'accès Lecture/Ecriture (données)
- **ALL** instaure le mode d'accès complet (données et structure).

Le contrôle des accès s'applique uniquement aux connexions depuis l'extérieur. Le code SQL exécuté à l'intérieur de 4D via les balises **Debut SQL/Fin SQL** ou les commandes telles que **SQL EXECUTER** bénéficie toujours d'un accès sans restriction.

Note de compatibilité : Lors de la conversion d'une ancienne base en version 11.3 ou suivante, les droits d'accès globaux (tels que définis dans la page SQL des Préférences de l'application) sont transférés au schéma par défaut.

Le premier paramètre *nom_sql* doit contenir le nom du groupe d'utilisateurs 4D auquel vous souhaitez affecter les droits d'accès au schéma. Ce groupe doit exister dans la base 4D.

Note : 4D permet de définir des noms de groupes comportant des espaces ou des caractères accentués, qui ne sont pas acceptés par le standard SQL. Dans ce cas, vous devez encadrer le nom avec les caractères [et]. Par exemple : **GRANT READ ON [le schéma] TO [les admins]**

Seuls le Super_Utilisateur et l'Administrateur de la base peuvent modifier des schémas.

Note à propos de l'intégrité référentielle

4D assure le principe d'intégrité référentielle indépendamment des droits d'accès. Imaginons par exemple que vous disposiez de deux tables Table1 et Table2 reliées par un lien de type N vers 1 (Table2 -> Table1). Table1 appartient au schéma S1 et Table2 au schéma S2. Un utilisateur disposant des droits d'accès du schéma S1 mais pas de ceux du schéma S2 peut supprimer des enregistrements dans la Table1. Dans ce cas, afin de respecter les principes d'intégrité référentielle, tous les enregistrements de la Table2 liés aux enregistrements supprimés de la Table1 seront également supprimés.

Exemple

Vous souhaitez autoriser l'accès en lecture écriture des données du schéma MonSchema1 au groupe "Power_Users" :

```
GRANT READ_WRITE ON MonSchema1 TO POWER_USERS
```

REVOKE

```
REVOKE [READ | READ_WRITE | ALL] ON nom_sql
```

Description

La commande *REVOKE* permet de supprimer les droits d'accès spécifiques associés au schéma défini par le paramètre *nom_sql*.

En fait, lorsque vous exécutez cette commande, vous affectez le pseudo-groupe d'utilisateurs <Personne> au droit d'accès défini.

Exemple

Vous souhaitez supprimer tout droit en lecture écriture au schéma MonSchema1 :

```
REVOKE READ_WRITE ON MonSchema1
```



```
REPLICATE liste_répliquée
FROM ref_table
[WHERE critère_recherche]
[LIMIT {nombre_entier | ref_langage_4d}]
[OFFSET {nombre_entier | ref_langage_4d}]
FOR REMOTE [STAMP] {nombre_entier | ref_langage_4d}
[, LOCAL [STAMP] {nombre_entier | ref_langage_4d}]
[{REMOTE OVER LOCAL | LOCAL OVER REMOTE}]
[LATEST REMOTE [STAMP] ref_langage_4d]
[, LATEST LOCAL [STAMP] ref_langage_4d]
INTO {liste_cible | ref_table(nom_sql_1,...,nom_sql_N)};
```

Description

La commande **REPLICATE** vous permet de répliquer les données d'une table d'une base A dans celles d'une table d'une base B. Par convention, la base sur laquelle est exécutée la commande est nommée "base locale" et la base de laquelle les données sont répliquées est nommée "base distante".

Cette commande peut être utilisée uniquement dans le cadre d'un système de réplication de base. Pour que le système fonctionne, la réplication doit avoir été activée côté base distante et côté base locale et chaque table impliquée doit comporter une clé primaire. Pour plus d'informations, reportez-vous à la section [Réplication via le SQL](#).

Note : Si vous souhaitez mettre en place un système de synchronisation complète, reportez-vous à la description de la commande **SYNCHRONIZE**.

Passez dans *liste_répliquée* une liste de champs (virtuels ou classiques) séparés par une virgule. Les champs doivent appartenir à la table *ref_table* de la base distante.

La clause **FROM** doit être suivie d'un argument de type *ref_table* permettant de désigner la table de la base distante depuis laquelle répliquer les données des champs *liste_répliquée*.

Note : Les champs virtuels de la table distante ne pourront être stockés que dans des tableaux de la base locale.

Côté base distante

La clause optionnelle **WHERE** permet d'appliquer un filtre préalable aux enregistrements de la table dans la base distante ; seuls les enregistrements qui satisfont au *critère_recherche* sont pris en compte par la commande.

4D récupère ensuite les valeurs des champs *liste_répliquée* pour tous les enregistrements désignés par la clause **FOR REMOTE STAMP**. La valeur passée dans cette clause peut être soit :

- une valeur de type **entier long > 0** : dans ce cas, les enregistrements dont la valeur du marqueur `__ROW_STAMP` est supérieure ou égale à cette valeur sont récupérés.
- **0** : dans ce cas, tous les enregistrements dont la valeur du marqueur `__ROW_STAMP` est différente de 0 sont récupérés. A noter que les enregistrements existant éventuellement avant l'activation de la réplication ne seront donc pas pris en compte (leur valeur de `__ROW_STAMP` = 0).
- **-1** : dans ce cas, tous les enregistrements de la table distante sont récupérés, autrement dit tous les enregistrements dont la valeur du marqueur `__ROW_STAMP` ≥ 0 . A la différence du cas précédent, tous les enregistrements de la table, y compris les enregistrements existant éventuellement avant l'activation, seront pris en compte.
- **-2** : dans ce cas, tous les enregistrements supprimés de la table distante (après activation de la réplication) sont récupérés, autrement dit tous les enregistrements dont la valeur du marqueur `__ROW_ACTION` = 2.

Enfin, vous pouvez appliquer à la sélection obtenue les clauses optionnelles **OFFSET** et **LIMIT** :

- Lorsqu'elle est passée, la clause **OFFSET** permet d'exclure les N premiers enregistrements de la sélection (N étant la valeur passée à la clause).
- Lorsqu'elle est passée, la clause **LIMIT** permet de restreindre la sélection aux M premiers enregistrements (M étant la valeur passée à la clause). Si la clause **OFFSET** a également été passée, la clause **LIMIT** est appliquée à la sélection obtenue après exécution de **OFFSET**.

Une fois l'ensemble des clauses appliquées, la sélection résultante est envoyée à la base locale.

Côté base locale

Les valeurs récupérées sont directement écrites dans la *liste_cible* de la base locale ou dans les champs classiques définis par *nom_sql* de la table *ref_table* de la base locale. L'argument *liste_cible* peut contenir soit une liste de champs standard, soit une

liste de tableaux du même type que les champs distants (mais pas une combinaison des deux). Si la destination de la commande est une liste de champs, les enregistrements cibles seront automatiquement créés, modifiés ou supprimés en fonction de l'action stockée dans le champ virtuel `__ROW_ACTION`.

Vous résolvez les conflits pour les enregistrements répliqués qui existent déjà dans la base cible (clés primaires identiques) à l'aide des clauses de priorité **REMOTE OVER LOCAL** et **LOCAL OVER REMOTE** :

- Si vous passez l'option **REMOTE OVER LOCAL** ou omettez la clause de priorité, tous les enregistrements source (base distante) désignés par la clause **FOR REMOTE STAMP** remplacent les enregistrements cible (base locale) s'ils existent déjà – qu'ils aient été modifiés ou non d'un côté comme de l'autre. Dans ce cas, il est inutile de passer la clause **LOCAL STAMP** car elle est ignorée.
- Si vous passez l'option **LOCAL OVER REMOTE**, la commande tient compte du marqueur local (**LOCAL STAMP**). Dans ce cas, les enregistrements cible (base locale) dont la valeur de marqueur est inférieure ou égale à celle qui est passée dans **LOCAL STAMP** ne sont pas remplacés par les enregistrements source (base distante). Par exemple, si vous passez 100 dans **LOCAL STAMP**, tous les enregistrements de la base locale dont le marqueur est ≤ 100 ne seront pas remplacés par les enregistrements équivalents de la base distante. Ce principe permet de préserver les données modifiées localement et de réduire la sélection d'enregistrements à répliquer dans la table locale.
- Si vous passez les clauses **LATEST REMOTE STAMP** et/ou **LATEST LOCAL STAMP**, 4D retourne dans les variables *ref_langage_4d* correspondantes les valeurs des derniers marqueurs des tables distante et locale. Ces informations sont utiles si vous souhaitez automatiser la gestion de la procédure de synchronisation. Ces valeurs correspondent à la valeur des marqueurs juste après la fin de l'opération de réplication : si vous les utilisez dans une instruction **REPLICATE** ou **SYNCHRONIZE** ultérieure, vous n'avez pas besoin de les incrémenter, ils le sont automatiquement avant d'être retournés par la commande **REPLICATE**.

Si l'opération de réplication se déroule correctement, la variable système OK prend la valeur 1. Vous pouvez contrôler cette valeur depuis une méthode 4D.

Si des erreurs se produisent durant l'opération de réplication, l'opération est stoppée à la première erreur rencontrée. La dernière variable source (si elle a été définie) est valorisée avec le marqueur de l'enregistrement dans lequel l'erreur s'est produite. La variable système OK prend la valeur 0. L'erreur générée peut être interceptée par une méthode de gestion d'erreurs installée par la commande **APPELER SUR ERREUR**.

Note : Les opérations effectuées par la commande **REPLICATE** ne tiennent pas compte des contraintes d'intégrité des données. Cela signifie par exemple que les règles de clés étrangères, d'unicité, etc. ne sont pas vérifiées. Si les données reçues peuvent remettre en cause l'intégrité des données, vous devez les vérifier à l'issue de l'opération de réplication. Le moyen le plus simple est de verrouiller via le langage 4D ou le langage SQL les enregistrements devant être modifiés.

SYNCHRONIZE

```
[LOCAL] TABLE ref_table (ref_colonne_1,...,ref_colonne_N)  
WITH  
[REMOTE] TABLE ref_table (ref_colonne_1,...,ref_colonne_N)  
FOR REMOTE [STAMP] {nombre_entier | ref_langage_4d},  
LOCAL [STAMP] {nombre_entier | ref_langage_4d}  
{REMOTE OVER LOCAL | LOCAL OVER REMOTE}  
LATEST REMOTE [STAMP] ref_langage_4d,  
LATEST LOCAL [STAMP] ref_langage_4d;
```

Description

La commande **SYNCHRONIZE** permet de synchroniser deux tables situées sur deux serveurs 4D SQL différents. Toute modification effectuée sur une table est reportée dans l'autre. Le serveur 4D SQL qui exécute la commande est appelé serveur local (LOCAL), l'autre serveur est appelé serveur distant (REMOTE).

La commande **SYNCHRONIZE** équivaut à la combinaison de deux appels internes à la commande **REPLICATE**. Le premier appel réplique les données depuis le serveur distant vers le serveur local, le second effectue l'opération inverse : réplique des données du serveur local vers le serveur distant. Les tables à synchroniser doivent donc être configurées pour la réplique :

- elles doivent comporter une clé primaire,
- l'option "Activer réplique" doit être cochée dans l'Inspecteur de chaque table.

Pour plus d'informations, reportez-vous à la description de la commande **REPLICATE**.

La commande **SYNCHRONIZE** accepte quatre marqueurs (stamps) en "paramètres" : deux marqueurs en entrée et deux marqueurs en sortie (dernière modification). Les marqueurs d'entrée sont utilisés pour indiquer le moment de la dernière synchronisation sur chaque serveur. Les marqueurs de sortie retournent la valeur des marqueurs de modification sur chaque serveur juste après la dernière modification. Grâce à ce principe, lorsque la commande **SYNCHRONIZE** est appelée régulièrement, il est possible d'utiliser les marqueurs de sortie de la dernière synchronisation en tant que marqueurs d'entrée pour la suivante.

Note : Les marqueurs d'entrée et de sortie sont exprimés sous forme de valeurs numériques (stamps), et non de marqueurs de temps (timestamps). Pour plus d'informations sur ces marqueurs, reportez-vous à la description de la commande **REPLICATE**.

En cas d'erreur, le marqueur de sortie du serveur concerné contient le marqueur de l'enregistrement à l'origine de l'erreur. Si l'erreur provient d'une cause externe à la synchronisation (problèmes réseau par exemple), le marqueur contient 0.

Il y a deux codes d'erreurs différents, l'un pour indiquer une erreur de synchronisation sur le site local et l'autre sur le site distant. En cas d'erreur, l'état des données dépend de celui de la transaction sur le serveur local. Sur le serveur distant, la synchronisation est toujours effectuée dans le contexte d'une transaction, les données ne peuvent donc pas être altérées par l'opération. Sur le serveur local en revanche, le processus de synchronisation est placé sous le contrôle du développeur. Il est effectué en-dehors de toute transaction si la préférence **Transactions Auto-commit** est désélectionnée (dans le cas contraire, un contexte de transaction est automatiquement créé). Le développeur peut décider de démarrer une transaction, il lui revient de la valider ou de l'annuler après la synchronisation des données.

Vous pouvez "forcer" le sens de la synchronisation à l'aide des clauses **REMOTE OVER LOCAL** et **LOCAL OVER REMOTE** en fonction des caractéristiques de votre application. Pour plus d'informations sur les mécanismes mis en oeuvre, reportez-vous à la description de la commande **REPLICATE**.

Note : Les opérations effectuées par la commande **SYNCHRONIZE** ne tiennent pas compte des contraintes d'intégrité des données. Cela signifie par exemple que les règles de clés étrangères, d'unicité, etc. ne sont pas vérifiées. Si les données reçues peuvent remettre en cause l'intégrité des données, vous devez les vérifier à l'issue de l'opération de synchronisation. Le moyen le plus simple est de verrouiller via le langage 4D ou le langage SQL les enregistrements devant être modifiés.

4D retourne dans les variables *ref_langage_4d* des clauses **LATEST REMOTE STAMP** et **LATEST LOCAL STAMP** les valeurs des derniers marqueurs des tables distante et locale. Ces informations permettent d'automatiser la gestion de la procédure de synchronisation. Elles correspondent à la valeur des marqueurs juste après la fin de l'opération de réplique : si vous les utilisez dans une instruction **REPLICATE** ou **SYNCHRONIZE** ultérieure, vous n'avez pas besoin de les incrémenter, ils le sont automatiquement avant d'être retournés par la commande **REPLICATE**.

Exemple

Pour bien comprendre les mécanismes mis en jeu lors d'une opération de synchronisation, nous allons examiner les différentes éventualités relatives à la mise à jour d'un enregistrement existant dans les deux bases synchronisées.

La méthode de synchronisation est de la forme suivante :

```

C_LONGINT(vRemoteStamp)
C_LONGINT(vLocalStamp)
C_LONGINT(vLatestRemoteStamp)
C_LONGINT(vLatestLocalStamp)

vRemoteStamp:=X... // voir valeurs dans le tableau ci-dessous
vLocalStamp:=X... // voir valeurs dans le tableau ci-dessous
vLatestRemoteStamp:=X... // valeur retournée dans un précédent LATEST REMOTE STAMP
vLatestLocalStamp:=X... // valeur retournée dans un précédent LATEST LOCAL STAMP

```

Debut SQL

```

SYNCHRONIZE
  LOCAL MYTABLE (MyField)
  WITH
  REMOTE MYTABLE (MyField)
  FOR REMOTE STAMP :vRemoteStamp,
  LOCAL STAMP :vLocalStamp
  LOCAL OVER REMOTE // ou REMOTE OVER LOCAL, voir dans le tableau ci-dessous
  LATEST REMOTE STAMP :vLatestRemoteStamp,
  LATEST LOCAL STAMP :vLatestLocalStamp;

```

Fin SQL

Les données de départ sont les suivantes :

- Le marqueur de l'enregistrement dans la base LOCAL a pour valeur 30 et celui dans la base REMOTE a pour valeur 400
- Les valeurs du champ MyField sont les suivantes :

LOCAL		REMOTE	
Ancienne valeur	Nouvelle valeur	Ancienne valeur	Nouvelle valeur
AAA	BBB	AAA	CCC





























- Nous utilisons les valeurs retournées par des précédents **LATEST LOCAL STAMP** et **LATEST REMOTE STAMP** de manière à synchroniser uniquement ce qui a été modifié depuis la dernière synchronisation.

Voici les synchronisations effectuées par la commande **SYNCHRONIZE** en fonction des valeurs passées dans les paramètres **LOCAL STAMP** et **REMOTE STAMP** ainsi que de l'option de priorité utilisée : ROL (pour **REMOTE OVER LOCAL**) ou LOR (pour **LOCAL OVER REMOTE**) :

LOCAL STAMP	REMOTE STAMP	Priorité	LOCAL après sync	REMOTE Après sync	Synchronisation LOCAL - REMOTE
20	3000	ROL	CCC	CCC	<---->
20	3000	LOR	BBB	BBB	<---->
31	3000	ROL	CCC	CCC	<--
31	3000	LOR	CCC	CCC	<--
20	4001	ROL	BBB	BBB	-->
20	4001	LOR	BBB	BBB	-->
31	4001	ROL	BBB	CCC	Pas de synchronisation
31	4001	LOR	BBB	CCC	Pas de synchronisation
40	3000	ROL	CCC	CCC	<--
40	3000	LOR	CCC	CCC	<--
20	5000	ROL	BBB	BBB	-->
20	5000	LOR	BBB	BBB	-->
40	5000	ROL	BBB	CCC	Pas de synchronisation
40	5000	LOR	BBB	CCC	Pas de synchronisation

Règles de syntaxe

Règles de syntaxe

-  appel_fonction_4d
-  ref_langage_4d
-  prédicat_all_or_any
-  expression_arithmétique
-  prédicat_between
-  expression_conditionnelle
-  définition_colonne
-  ref_colonne
-  paramètre_commande
-  prédicat_comparaison
-  prédicat_exists
-  définition_clé_étrangère
-  appel_fonction
-  prédicat_in
-  prédicat_is_null
-  prédicat_like
-  val_littérale
-  prédicat
-  définition_clé_primaire
-  critère_recherche
-  select_élément
-  liste_tri
-  type_données_sql
-  nom_sql
-  chaîne_sql
-  sous_requête
-  contrainte_table
-  ref_table

Règles de syntaxe

Cette section décrit le contenu et l'usage des divers éléments constituant les prédicats utilisés dans les instructions SQL. Nous les avons regroupés par entités élémentaires de manière à pouvoir les décrire de la façon la plus simple et fournir des indications générales sur leur mode d'utilisation dans l'environnement 4D. Les mots-clés (en gras) sont invariables et doivent toujours être passés tels quels.

appel_fonction_4d

```
{FN nom_sql ([expression_arithmétique, ..., expression_arithmétique]) AS type_données_sql}
```

Description

Un *appel_fonction_4d* permet d'exécuter une fonction 4D retournant une valeur.

L'argument *nom_sql* de l'appel doit être précédé du mot-clé **FN** et suivi d'un ou plusieurs argument(s) de type *expression_arithmétique*. La valeur retournée par la fonction sera du type défini par *type_données_sql*.

Exemple

Cet exemple utilise des fonctions pour extraire de la table FILMS le nombre d'acteurs pour chaque film comprenant au moins 7 acteurs :

```
C_ENTIER LONG ($NbActeurs)
TABLEAU TEXTE (tTitresFilms;0)
TABLEAU ENTIER LONG (tNbActeurs;0)

$NbActeurs:=7
Debut SQL
  SELECT Titre_Film, {FN Quel_Nb_Acteurs(ID) AS NUMERIC}
  FROM FILMS
  WHERE {FN Quel_Nb_Acteurs(ID) AS NUMERIC} >= :$NbActeurs
  ORDER BY 1
  INTO :tTitresFilms; :tNbActeurs
Fin SQL
```

```
<<nom_sql>> | <<$nom_sql>> | <<[nom_sql]nom_sql>> | :nom_sql |:$nom_sql |:nom_sql.nom_sql
```

Description

Un argument *ref_langage_4d* désigne un nom de variable ou de champ 4D (*nom_sql*) destiné(e) à recevoir des données. Ce nom peut être passé d'une des manières suivantes :

`<<nom_sql>>`

`<<$nom_sql>>` (*)

`<<[nom_sql]nom_sql>>` (correspond à la syntaxe 4D standard : [NomTable]NomChamp)

`:nom_sql`

`:$nom_sql` (*)

`:nom_sql.nom_sql` (correspond à la syntaxe SQL standard : NomTable.NomChamp)

(*) En mode compilé, vous ne pouvez pas utiliser de références à des variables locales (débutant par le symbole \$).

prédicat_all_or_any

expression_arithmétique{< | <= | = | >= | > | <>} {**ANY** | **ALL** | **SOME**} (*sous_requête*)

Description

Un *prédicat_all_or_any* est utilisé pour comparer une *expression_arithmétique* avec une *sous_requête*. Vous pouvez utiliser des opérateurs de comparaison tels que <, <=, =, >=, > et <> ainsi que les mots-clés **ANY**, **ALL** et **SOME** pour effectuer la comparaison avec la *sous_requête*.

Exemple

L'exemple effectue une sous-requête sélectionnant les meilleures ventes de logiciels. La requête principale sélectionne les enregistrements des tables VENTES et CLIENTS dont la colonne Valeur_Totale est supérieure aux enregistrements sélectionnés par la sous-requête :

```
SELECT Valeur_Totale, CLIENTS.Client
FROM VENTES, CLIENTS
WHERE VENTES.Client_ID = CLIENTS.Client_ID
AND Valeur_Totale > ALL (SELECT MAX (Valeur_Totale)
FROM VENTES
WHERE Type_produit = 'Logiciel');
```

```
val_littérale /  
ref_colonne /  
appel_fonction /  
paramètre_commande /  
expression_conditionnelle /  
(expression_arithmétique) /  
+ expression_arithmétique /  
- expression_arithmétique /  
expression_arithmétique + expression_arithmétique /  
expression_arithmétique - expression_arithmétique /  
expression_arithmétique * expression_arithmétique /  
expression_arithmétique / expression_arithmétique
```

Description

Une *expression_arithmétique* peut contenir une valeur littérale (*val_littérale*), une référence de colonne (*ref_colonne*), un appel de fonction (*appel_fonction*), un paramètre de commande (*paramètre_commande*) ou une expression du type "au cas où" (*expression_conditionnelle*). Vous pouvez également passer des combinaisons d'expressions arithmétiques à l'aide des opérateurs +, -, * et /.

prédicat_between

expression_arithmétique [**NOT**] **BETWEEN** *expression_arithmétique* **AND** *expression_arithmétique*

Description

Le *prédicat_between* permet de rechercher des données dont les valeurs sont comprises dans l'intervalle défini par deux valeurs de type *expression_arithmétique* (passées dans l'ordre croissant). Vous pouvez également passer le mot-clé facultatif **NOT** afin de rechercher les valeurs n'appartenant pas à l'intervalle défini.

Exemple

Cet exemple sélectionne tous les clients dont le prénom débute par une lettre comprise entre A et E :

```
SELECT CLIENT_PRENOM, CLIENT_NOM
FROM T_CLIENT
WHERE CLIENT_PRENOM BETWEEN 'A' AND 'E'
```

expression_conditionnelle

Description

Une *expression_conditionnelle* permet d'appliquer une ou plusieurs conditions du type "Au cas où" afin de sélectionner une expression.

Une *expression_conditionnelle* peut être utilisée par exemple de la manière suivante :

```
CASE  
WHEN critère_recherche THEN expression_arithmétique  
...  
WHEN critère_recherche THEN expression_arithmétique  
[ELSE expression_arithmétique]  
END
```

Ou bien :

```
CASE expression_arithmétique  
WHEN expression_arithmétique THEN expression_arithmétique  
...  
WHEN expression_arithmétique THEN expression_arithmétique  
[ELSE expression_arithmétique]  
END
```

Exemple

Cet exemple sélectionne les enregistrements de la colonne des numéros de chambre en fonction de la valeur de la colonne ETAGE_CHBR :

```
SELECT NUM_CHBR  
CASE ETAGE_CHBR  
WHEN 'RDC' THEN 0  
WHEN '1er' THEN 1  
WHEN '2e' THEN 2  
END AS ETAGES, NB_COUCHAGE  
FROM T_CHAMBRES  
ORDER BY ETAGES, NB_COUCHAGE
```

```
nom_sql type_données_sql [(nombre_entier)][NOT NULL [UNIQUE]] [AUTO_INCREMENT] [AUTO_GENERATE]
```

Description

Une définition de colonne (*définition_colonne*) contient le nom (*nom_sql*) et le type de données (*type_données_sql*) d'une colonne. Facultativement, vous pouvez également passer un *nombre_entier* ainsi que les mots-clés **NOT NULL**, **UNIQUE**, **AUTO_INCREMENT** et/ou **AUTO_GENERATE**.

- Passer **NOT NULL** dans la *définition_colonne* signifie que la colonne n'acceptera pas de valeurs NULL.
- Passer **UNIQUE** signifie que la même valeur ne pourra pas être insérée deux fois dans la colonne. A noter que seules les colonnes **NOT NULL** peuvent comporter l'attribut **UNIQUE**. Le mot-clé **UNIQUE** doit toujours être précédé de **NOT NULL**.
- Passer **AUTO_INCREMENT** signifie que la colonne générera un numéro unique pour chaque nouvelle ligne. Cet attribut est utilisable avec des colonnes numériques uniquement.
- Passer **AUTO_GENERATE** signifie qu'un UUID sera généré automatiquement dans la colonne à chaque nouvelle ligne. Cet attribut est utilisable avec des colonnes UUID uniquement.

Chaque colonne doit avoir un type de données et être définie comme "null" ou "not null" ; si cet attribut n'est pas défini, la base considère par défaut que la colonne est "null". Le type de données de la colonne ne limite pas les données qui peuvent être placées dans la colonne.

Exemple

Cet exemple crée une table avec deux colonnes, ID et Nom :

```
CREATE TABLE ACTEUR_FANS  
(ID INT32, Nom VARCHAR NOT NULL UNIQUE);
```

nom_sql | nom_sql.nom_sql | chaîne_sql.chaîne_sql

Description

Une référence de colonne (*ref_colonne*) est constituée d'un *nom_sql* ou d'une *chaîne_sql* passé(e) d'une des manières suivantes :

nom_sql

nom_sql.nom_sql

chaîne_sql.chaîne_sql

paramètre_commande

? | <<nom_sql>> | <<\$nom_sql>> | <<[nom_sql]nom_sql>> | :nom_sql | :\$nom_sql | :nom_sql.nom_sql

Description

Un argument *paramètre_commande* peut être un point d'interrogation (?) ou un *nom_sql* passé dans l'une des formes suivantes :

?

<<nom_sql>>

<<\$nom_sql>>

<<[nom_sql]nom_sql>>

:nom_sql

:\$nom_sql

:nom_sql.nom_sql

```
expression_arithmétique {< |<= | = | >= | > | <> } expression_arithmétique |  
expression_arithmétique {< |<= | = | >= | > | <> } (sous_requête) |  
(sous_requête) {< |<= | = | >= | > | <> } expression_arithmétique
```

Description

Un argument *prédicat_comparaison* utilise les opérateurs <, <=, =, >=, > ou <> pour comparer deux arguments de type *expression_arithmétique* ou pour comparer une *expression_arithmétique* à une *sous_requête* en tant que partie d'un *critère_recherche* appliqué aux données.

EXISTS (*sous_requête*)

Description

Le prédicat **EXISTS** permet d'exécuter une *sous_requête* et de vérifier si elle retourne une valeur. Ce résultat est obtenu en passant le mot-clé **EXISTS** suivi de la *sous_requête*.

Exemple

Cet exemple retourne le total des ventes lorsqu'il existe un magasin dans la région spécifiée :

```
SELECT SUM (Ventes)
FROM Information_Magasin
WHERE EXISTS
(SELECT * FROM Geographie
WHERE nom_region = 'Ouest')
```

définition_clé_étrangère

```
CONSTRAINT nom_sql  
FOREIGN KEY (ref_colonne, ... , ref_colonne)  
REFERENCES nom_sql [(ref_colonne, ... , ref_colonne)]  
[ON DELETE {RESTRICT | CASCADE}]  
[ON UPDATE {RESTRICT | CASCADE}]
```

Description

Une définition de clé étrangère (*définition_clé_étrangère*) permet de faire correspondre les champs clés primaires (*ref_colonne*) définis dans une autre table afin de préserver l'intégrité des données. La contrainte **FOREIGN KEY** permet de passer la ou les référence(s) des colonne(s) (*ref_colonne*) à définir comme clés étrangères (correspondant aux clés primaires d'une autre table).

La clause **CONSTRAINT** *nom_sql* permet de nommer la contrainte **FOREIGN KEY**.

La clause **REFERENCES** permet de spécifier les champs clés primaires source correspondants dans l'autre table (désignée par *nom_sql*). Vous pouvez omettre la liste des champs (arguments *ref_colonne*) si la table *nom_sql* définie dans la clause **REFERENCES** a une clé primaire qui doit être utilisée comme correspondance pour la contrainte clé étrangère.

La clause facultative **ON DELETE CASCADE** spécifie que lorsqu'un enregistrement (row) est supprimé de la table parente (contenant les champs clés primaires), les enregistrements associés dans la table enfant (contenant les champs clés étrangers) doivent également être supprimés. Passer la clause facultative **ON DELETE RESTRICT** empêche la suppression des données d'une table si d'autres tables les référencent.

La clause facultative **ON UPDATE CASCADE** spécifie qu'à chaque fois qu'un enregistrement (row) est mis à jour dans la table parente (contenant les champs clés primaires), il est également mis à jour dans tous les enregistrements associés dans la table enfant (contenant les champs clés étrangers). Passer la clause facultative **ON UPDATE RESTRICT** empêche la mise à jour des données d'une table si d'autres tables les référencent.

A noter que si les deux clauses **ON DELETE** et **ON UPDATE** sont passées, elles doivent être du même type (c'est-à-dire **ON DELETE CASCADE** et **ON UPDATE CASCADE**, ou **ON DELETE RESTRICT** et **ON UPDATE RESTRICT**).

Si ni la clause **ON DELETE** ni **ON UPDATE** n'est passée, **CASCADE** est utilisée comme règle par défaut.

Exemple

Cet exemple crée la table COMMANDES puis définit la colonne SID_Clients comme clé étrangère, associée à la colonne SID de la table CLIENTS.

```
CREATE TABLE COMMANDES  
  (ID_Commande INT32,  
   SID_Clients INT32,  
   LeTotal NUMERIC,  
   PRIMARY KEY (ID_Commande),  
   CONSTRAINT fk_1 FOREIGN KEY (SID_Clients) REFERENCES CLIENTS(SID));
```

*appel_fonction_sql |
appel_fonction_4d*

Description

Un *appel_fonction* peut être soit un appel à une des **Fonctions SQL** soit un appel à une fonction 4D (*appel_fonction_4d*). Les deux types de fonctions peuvent manipuler les données, retourner des résultats et peuvent traiter un ou plusieurs arguments.

Exemple

Cet exemple utilise la fonction SQL *COUNT* :

```
C_ENTIER LONG (vNumPersonne)
Debut SQL
    SELECT COUNT (*)
    FROM COMMERCIAUX
    INTO :vNumPersonne;
Fin SQL
```

```
expression_arithmétique [NOT] IN (sous_requête) |  
expression_arithmétique [NOT] IN (expression_arithmétique, ..., expression_arithmétique)
```

Description

Le prédicat **IN** permet d'évaluer si une *expression_arithmétique* est incluse (ou non incluse si le mot-clé **NOT** est également passé) dans un ensemble de valeurs. L'ensemble de valeurs utilisé pour la comparaison peut être soit une séquence d'expressions arithmétiques, soit le résultat d'une sous-requête.

Exemple

Cet exemple sélectionne les enregistrements de la table COMMANDES dont la valeur de la colonne id_commande est égale à 10000, 10001, 10003 ou 10005 :

```
SELECT *  
FROM COMMANDES  
WHERE id_commande IN (10000, 10001, 10003, 10005);
```

expression_arithmétique **IS [NOT] NUL**

Description

Le prédicat **IS NULL** permet de trouver une *expression_arithmétique* ayant la valeur **NULL**. Il est également possible de passer le mot-clé **NOT** afin de trouver les expressions sans la valeur **NULL**.

Exemple

Cet exemple sélectionner les produits dont le poids est inférieur à 15 ou dont la colonne Couleur contient la valeur NULL :

```
SELECT Nom, Poids, Couleur
FROM PRODUITS
WHERE Poids < 15.00 OR Couleur IS NULL
```

prédicat_like

expression_arithmétique [**NOT**] **LIKE** *expression_arithmétique* [**ESCAPE** *chaîne_sql*]

Description

Le prédicat **LIKE** permet de sélectionner les données correspondant à l'*expression_arithmétique* passée après le mot-clé **LIKE**. Vous pouvez également passer le mot-clé **NOT** afin de sélectionner les données ne correspondant pas à cette expression. Vous pouvez utiliser le mot-clé **ESCAPE** afin que le caractère passé dans *chaîne_sql* ne soit pas interprété comme joker. Il est généralement utilisé lorsque vous voulez rechercher les caractères '%' ou '_'.

Exemple 1

Cet exemple sélectionne les fournisseurs dont le nom contient "bob" :

```
SELECT * FROM fournisseurs
WHERE nom LIKE '%bob%';
```

Exemple 2

Sélection des fournisseurs dont le nom ne commence pas par la lettre T :

```
SELECT * FROM fournisseurs
WHERE nom NOT LIKE 'T%';
```

Exemple 3

Sélection des fournisseurs dont le nom commence par "Sm" et se termine par "th" :

```
SELECT * FROM fournisseurs
WHERE nom LIKE 'Sm_th'
```

nombre_entier | nombre_fractionnaire | chaîne_sql | nombre_hexadécimal

Description

Une valeur *val_littérale* est une donnée de type *nombre_entier* (Entier), un *nombre_fractionnaire* (fraction), une *chaîne_sql* ou un *nombre_hexadécimal*.

La notation hexadécimale (apparue dans 4D 12.1) permet d'exprimer tout type de données sous forme de représentation d'octets. Un octet est toujours défini par deux valeurs hexadécimales. Pour indiquer l'emploi de cette notation dans une commande SQL, vous devez simplement utiliser la syntaxe hexadécimale SQL standard :

X'<valeur hexadécimale>'

Par exemple, pour la valeur décimale 15, vous pouvez écrire **X'0f'**. Il est possible de définir une valeur vide (zéro octet) en écrivant **X''**.

Note : Les commandes **SQL EXPORTER BASE** et **SQL EXPORTER SELECTION** exportent les données binaires au format hexadécimal lorsqu'elles sont incluses dans le fichier principal.

prédicat

Description

Un *prédicat* suit la clause **WHERE** et permet d'ajouter des conditions de recherche pour la sélection des données. Il peut être d'un des types suivants :

prédicat_comparaison

prédicat_between

prédicat_like

prédicat_is_null

prédicat_in

prédicat_all_or_any

prédicat_exists

définition_clé_primaire

```
[CONSTRAINT nom_sql] PRIMARY KEY (nom_sql, ... , nom_sql)
```

Description

Un argument de type *définition_clé_primaire* permet de passer la colonne ou la combinaison de colonnes (*nom_sql*) qui sera utilisée comme clé primaire (**PRIMARY KEY**, numéro unique) de la table. La ou les colonne(s) passée(s) ne doi(ven)t pas contenir de doublons ni de valeurs **NULL**.

Une contrainte facultative (**CONSTRAINT**) peut également précéder la clé **PRIMARY KEY** afin de limiter les valeurs pouvant être insérées dans la colonne.

Exemple

Cet exemple crée une table et définit la colonne SID comme clé primaire :

```
CREATE TABLE Clients
(SID int32,
Nom varchar(30),
Prenom varchar(30),
PRIMARY KEY (SID));
```

```
prédicat |  
NOT critère_recherche |  
(critère_recherche) |  
critère_recherche OR critère_recherche |  
critère_recherche AND critère_recherche
```

Description

Un *critère_recherche* spécifie une condition à appliquer aux données sélectionnées. Il est également possible de passer une combinaison de conditions à l'aide des mots-clés **AND** et **OR**. Vous pouvez également faire précéder un *critère_recherche* du mot-clé **NOT** afin de sélectionner uniquement les données ne correspondant pas à la condition définie.

Il est enfin possible de passer un *prédicat* comme *critère_recherche*.

Exemple

Cet exemple utilise une combinaison de critères de recherche dans la clause **WHERE** :

```
SELECT fournisseur_id  
FROM fournisseurs  
WHERE (nom= 'CANON')  
OR (nom= 'Hewlett Packard' AND ville = 'New York')  
OR (nom= 'Firewall' AND statut = 'Fermé' AND ville = 'Chicago');
```

select_élément

expression_arithmétique [[**AS**] {*chaîne_sql* | *nom_sql*}]

Description

Un *select_élément* désigne un ou plusieurs éléments à inclure dans les résultats. Une colonne est générée pour chaque *select_élément* passé. Un *select_élément* consiste en une *expression_arithmétique*. Vous pouvez également utiliser le mot-clé facultatif **AS** afin de spécifier une *chaîne_sql* ou un *nom_sql* à passer à la colonne (passer une *chaîne_sql* ou un *nom_sql* sans le mot-clé **AS** produit le même effet).

Exemple

Cet exemple crée une colonne nommée `Annee_Film` contenant les films réalisés à partir de l'année 2000 :

```
TABLEAU ENTIER(tAnneeFilm;0)
Debut SQL
  SELECT Annee_de_sortie AS Annee_Film
  FROM FILMS
  WHERE Annee_Film >= 2000
  ORDER BY 1
  INTO :tAnneeFilm;
Fin SQL
```

```
{ref_colonne | nombre_entier} [ASC | DESC], ... , {ref_colonne | nombre_entier} [ASC | DESC]
```

Description

Un argument de type *liste_tri* contient soit une référence de colonne (*ref_colonne*) soit un numéro (*nombre_entier*) désignant la colonne dont les valeurs doivent être triées. Vous pouvez également passer les mots-clés **ASC** ou **DESC** afin de spécifier si le tri doit être ascendant ou descendant. Par défaut, les tris sont ascendants.

ALPHA_NUMERIC | VARCHAR | TEXT | TIMESTAMP | INTERVAL | DURATION | BOOLEAN | BIT | BYTE | INT16 | SMALLINT | INT32 | INT | INT64 | NUMERIC | REAL | FLOAT | DOUBLE PRECISION | BLOB | BIT VARYING | CLOB | PICTURE

Description

Un argument *type_données_sql* suit le mot-clé **AS** dans un *appel_fonction_4d* et peut prendre l'une des valeurs suivantes :

ALPHA_NUMERIC
VARCHAR
TEXT
TIMESTAMP
INTERVAL
DURATION
BOOLEAN
BIT
BYTE
INT16
SMALLINT
INT32
INT
INT64
NUMERIC
REAL
FLOAT
DOUBLE PRECISION
BLOB
BIT VARYING
CLOB
PICTURE

nom_sql

Description

Un *nom_sql* est soit un nom SQL standard débutant par un caractère de l'alphabet Latin et contenant uniquement des caractères latins, des chiffres et des traits de soulignement, soit une chaîne encadrée par des crochets. Passez deux crochets droits comme caractères d'échappement pour le crochet droit.

Exemples :

Chaîne à passer	nom_sql
MonNomSQL_2	MonNomSQL_2
Mon nom !&^#%!&#% non standard	[Mon nom !&^#%!&#% non standard]
[nom déjà entre crochets]	[[nom déjà entre crochets]]
nom contenant [] des crochets	[nom contenant [] des crochets]

chaîne_sql

Description

Une *chaîne_sql* contient une chaîne de caractères entre apostrophes. Les apostrophes incluses dans la chaîne doivent être doublées ; pour les chaînes qui sont déjà entre apostrophes, elles doivent être triplées.

Exemples :

Chaîne à passer	chaîne_sql
ma chaîne	'ma chaîne'
chaîne avec ' au milieu	'chaîne avec '' au milieu'
'chaîne déjà entre apostrophes'	''chaîne déjà entre apostrophes''

```
SELECT [ALL | DISTINCT]  
{* | select_élément, ..., select_élément}  
FROM ref_table, ..., ref_table  
[WHERE critère_recherche]  
[GROUP BY liste_tri]  
[HAVING critère_recherche]  
[LIMIT {nombre_entier | ALL.}]  
[OFFSET nombre_entier]
```

Description

Une *sous_requête* s'apparente à une instruction *SELECT* indépendante incluse entre parenthèses et passée dans le prédicat d'une autre instruction SQL (*SELECT*, **INSERT**, *UPDATE* ou *DELETE*). Il s'agit d'une requête dans une requête et elle est souvent passée comme partie d'une clause **WHERE** ou **HAVING**.

{*définition_clé_primaire* | *définition_clé_étrangère*}

Description

Une *contrainte_table* restreint les valeurs qu'une table peut stocker. Vous pouvez passer une *définition_clé_primaire* ou une *définition_clé_étrangère*. La *définition_clé_primaire* désigne la clé primaire pour la table et la *définition_clé_étrangère* désigne la clé étrangère (qui correspond à la clé primaire d'une autre table).

Exemple

Cet exemple définit Nom comme champ clé primaire :


```
CREATE TABLE ACTEUR_FANS  
(ID INT32, Nom VARCHAR NOT NULL UNIQUE);
```

```
{nom_sql | chaîne_sql} [[AS] {nom_sql|chaîne_sql}]
```

Description

Une référence de table peut être soit un nom SQL standard soit une chaîne. Vous pouvez aussi passer le mot-clé facultatif **AS** pour assigner un alias (sous la forme d'un *nom_sql* ou d'une *chaîne_sql*) à la colonne (passer une *chaîne_sql* ou un *nom_sql* sans le mot-clé **AS** produit le même effet).

Transactions

 Transactions

 START

 COMMIT

 ROLLBACK

Description

Une transaction est un ensemble d'instructions SQL exécutées en bloc. Soit toutes les instructions sont exécutées, soit aucune d'entre elles. Les transactions placent des verrous logiques pendant leur exécution afin de préserver l'intégrité des données. Si la transaction se termine correctement, vous pouvez appeler la commande **COMMIT** afin de valider et d'enregistrer définitivement les modifications effectuées. Dans le cas contraire, appelez la commande **ROLLBACK** afin d'annuler toutes les modifications et de restituer la base de données dans son état précédent.

Au niveau de la base de données, il n'y a pas de différence entre une transaction 4D et une transaction SQL. Les deux types de transactions partagent les mêmes données et le même process. Les instructions SQL incluses dans une structure **Debut SQL/Fin SQL**, la commande **CHERCHER PAR SQL** et les commandes SQL génériques intégrées appliquées à la base locale sont toujours exécutées dans le même contexte que les commandes 4D standard.

Note : 4D propose l'option "Auto-commit" permettant de démarrer et de valider automatiquement des transactions lors de l'utilisation des commande SIUD (**SELECT**, **INSERT**, **UPDATE** et **DELETE**) afin de préserver l'intégrité des données. Pour plus d'informations, reportez-vous à la section **Implémentations du moteur SQL de 4D**.

Les exemples suivants illustrent les différentes combinaisons de transactions.

Ni "John" ni "Smith" ne seront ajoutés dans la table emp :

```
SQL LOGIN (SQL_INTERNAL;"";"" ) `Initialiser le moteur SQL de 4D SQL
DEBUT TRANSACTION `Démarrer une transaction dans le process courant
Debut SQL
    INSERT INTO emp
    (NAME)
    VALUES ('John');
Fin SQL
SQL EXECUTER ("START") `Autre transaction dans le process courant
SQL ANNULER CHARGEMENT
SQL EXECUTER ("INSERT INTO emp (NAME) VALUES ('Smith')") `Cette instruction est exécutée dans le
même process
SQL ANNULER CHARGEMENT
SQL EXECUTER ("ROLLBACK") `Annuler la transaction interne du process
ANNULER TRANSACTION `Annuler la transaction externe du process
SQL LOGOUT
```

Seul "John" sera ajouté dans la table emp :

```
SQL LOGIN (SQL_INTERNAL;"";"" )
DEBUT TRANSACTION
Debut SQL
    INSERT INTO emp
    (NAME)
    VALUES ('John');
Fin SQL
SQL EXECUTER ("START")
SQL ANNULER CHARGEMENT
SQL EXECUTER ("INSERT INTO emp (NAME) VALUES ('Smith')")
SQL ANNULER CHARGEMENT
SQL EXECUTER ("ROLLBACK") `Annuler la transaction interne du process
VALIDER TRANSACTION `Valider la transaction externe du process
SQL LOGOUT
```

Ni "John" ni "Smith" ne seront ajoutés dans la table emp. La transaction externe annule la transaction interne :

```
SQL LOGIN (SQL_INTERNAL;"";"" )
DEBUT TRANSACTION
Debut SQL
    INSERT INTO emp
    (NAME)
    VALUES ('John');
Fin SQL
SQL EXECUTER ("START")
SQL ANNULER CHARGEMENT
```

```
SQL EXECUTER("INSERT INTO emp (NAME) VALUES ('Smith')")
SQL ANNULER CHARGEMENT
SQL EXECUTER("COMMIT") `Valider la transaction interne du process
ANNULER TRANSACTION `Annuler la transaction externe du process
SQL LOGOUT
```

“John” et “Smith” seront ajoutés dans la table emp :

```
SQL LOGIN(SQL_INTERNAL;"";"")
DEBUT TRANSACTION
Debut SQL
    INSERT INTO emp
    (NAME)
    VALUES ('John');
Fin SQL
SQL EXECUTER("START")
SQL ANNULER CHARGEMENT
SQL EXECUTER("INSERT INTO emp (NAME) VALUES ('Smith')")
SQL ANNULER CHARGEMENT
SQL EXECUTER("COMMIT") `Valider la transaction interne du process
VALIDER TRANSACTION `Valider la transaction externe du process
SQL LOGOUT
```

START

START [TRANSACTION]

Description

La commande *START* démarre une transaction. Si cette commande est exécutée alors qu'une transaction est déjà en cours, elle démarre une sous-transaction.

Le mot-clé **TRANSACTION** est facultatif.

Exemple

Cet exemple effectue une sélection à l'intérieur d'une transaction :

```
START TRANSACTION
SELECT * FROM suppliers
WHERE supplier_name LIKE '%bob%';
COMMIT TRANSACTION;
```

COMMIT [TRANSACTION]

Description

La commande *COMMIT* termine une transaction exécutée avec succès. Elle a pour effet de valider et d'enregistrer dans les données de la base toutes les modifications effectuées durant la transaction. Elle libère également toutes les ressources utilisées par la transaction.

Rappelez-vous que vous ne pouvez pas utiliser la commande *ROLLBACK* après un *COMMIT* puisque dans ce cas les modifications ont déjà été entérinées.

Le mot-clé **TRANSACTION** est facultatif.

Exemple

Reportez-vous à l'exemple de la commande *START*.

ROLLBACK [TRANSACTION]

Description

La commande *ROLLBACK* annule la transaction en cours et restitue les données dans l'état où elles se trouvaient avant le début de la transaction. Elle libère également toutes les ressources utilisées par la transaction. Le mot-clé **TRANSACTION** est facultatif.






















































Exemple


























Cet exemple illustre l'emploi de la commande *ROLLBACK* :

```
START TRANSACTION
SELECT * FROM suppliers
WHERE supplier_name like '%bob%';
ROLLBACK TRANSACTION;
```


Fonctions

Fonctions SQL

-  ABS
-  ACOS
-  ASCII
-  ASIN
-  ATAN
-  ATAN2
-  AVG
-  BIT_LENGTH
-  CAST
-  CEILING
-  CHAR
-  CHAR_LENGTH
-  COALESCE
-  CONCAT
-  CONCATENATE
-  COS
-  COT
-  COUNT
-  CURDATE
-  CURRENT_DATE
-  CURRENT_TIME
-  CURRENT_TIMESTAMP
-  CURTIME
-  DATABASE_PATH
-  DATE_TO_CHAR
-  DAY
-  DAYNAME
-  DAYOFMONTH
-  DAYOFWEEK
-  DAYOFYEAR
-  DEGREES
-  EXP
-  EXTRACT
-  FLOOR
-  HOUR
-  INSERT
-  LEFT
-  LENGTH
-  LOCATE
-  LOG
-  LOG10
-  LOWER
-  LTRIM
-  MAX
-  MILLISECOND
-  MIN
-  MINUTE
-  MOD
-  MONTH
-  MONTHNAME
-  NULLIF
-  OCTET_LENGTH
-  PI

 POSITION
 POWER
 QUARTER
 RADIANS
 RAND
 REPEAT
 REPLACE
 RIGHT
 ROUND
 RTRIM
 SECOND
 SIGN
 SIN
 SPACE
 SQRT
 SUBSTRING
 SUM
 TAN
 TRANSLATE
 TRIM
 TRUNC
 TRUNCATE
 UPPER
 WEEK
 YEAR

Fonctions SQL

Les fonctions SQL s'appliquent aux colonnes de données afin de retourner un résultat spécifique dans 4D. Dans ce manuel, les noms des fonctions apparaissent en gras et sont passés tels quels, généralement suivis d'un ou plusieurs arguments(s) de type *expression_arithmétique*.

ABS (*expression_arithmétique*)

Description

La fonction **ABS** retourne la valeur absolue de l'*expression_arithmétique*.

Exemple

Cet exemple retourne la valeur absolue des prix et la multiplie par une certaine quantité :

```
ABS(Price) * quantity
```

ACOS (*expression_arithmétique*)

Description

La fonction *ACOS* retourne l'arc cosinus de l'*expression_arithmétique* passée en paramètre. Il s'agit de la fonction réciproque de la fonction cosinus (**COS**). L'*expression_arithmétique* représente l'angle exprimé en radians.

Exemple

Cet exemple sélectionne l'arc cosinus de l'angle de -0.73 radians :

```
SELECT ACOS(-0.73)
FROM TABLES_OF_ANGLES;
```

ASCII (*expression_arithmétique*)

Description

La fonction *ASCII* retourne sous forme d'entier le code du caractère le plus à gauche de l'*expression_arithmétique* passée en paramètre. Si l'*expression_arithmétique* est vide, la fonction retourne la valeur **NULL**.

Exemple

Cet exemple retourne le code de la première lettre de chaque nom :

```
SELECT ASCII (SUBSTRING (Nom, 1, 1) )  
FROM PERSONNES;
```

ASIN (*arithmetic_expression*)

Description

La fonction *ASIN* retourne l'arcsinus de l'*expression_arithmétique* passée en paramètre. Il s'agit de la réciproque de la fonction sinus (**SIN**). L'*expression_arithmétique* désigne un angle exprimé en radians.

Exemple

Cet exemple sélectionne l'arcsinus de l'angle de -0.73 radians :

```
SELECT ASIN(-0.73)
FROM TABLES_OF_ANGLES;
```

ATAN (*expression_arithmétique*)

Description

La fonction *ATAN* retourne l'arctangente de l'*expression_arithmétique* passée en paramètre. Il s'agit de la réciproque de la fonction tangente (**TAN**). L'*expression_arithmétique* désigne un angle exprimé en radians.

Exemple

Cet exemple sélectionne l'arctangente de l'angle de -0.73 radians :

```
SELECT ATAN(-0.73)
FROM TABLES_OF_ANGLES;
```


ATAN2 (*expression_arithmétique, expression_arithmétique*)

Description

La fonction *ATAN2* retourne l'arctangente des coordonnées "x" et "y", "x" étant la première *expression_arithmétique* passée et "y" étant la seconde.

Exemple

Cet exemple retourne l'arctangente des coordonnées 0,52 et 0,60 :

```
SELECT ATAN2( 0.52, 0.60 );
```

AVG ([**ALL** | **DISTINCT**] *expression_arithmétique*)

Description

La fonction *AVG* retourne la moyenne des valeurs de l'*expression_arithmétique* passée en paramètre. Les mots-clés facultatifs **ALL** et **DISTINCT** permettent respectivement d'inclure ou d'éliminer les valeurs dupliquées (doublons) du calcul.

Exemple

Cet exemple retourne plusieurs valeurs statistiques relatives aux entrées des films stockés dans la table FILMS : le plus petit et le plus grand nombre, le nombre moyen et le nombre total :

```
SELECT MIN(Nb_Entrees),  
MAX(Nb_Entrees),  
AVG(Nb_Entrees),  
SUM (Nb_Entrees)  
FROM FILMS
```

BIT_LENGTH

BIT_LENGTH (*expression_arithmétique*)

Description

La fonction *BIT_LENGTH* retourne la longueur en bits de l'*expression_arithmétique* passée en paramètre.

Exemple

Cet exemple retourne 8 :

```
SELECT BIT_LENGTH( '01101011' );
```

CAST

CAST (*expression_arithmétique* **AS** *type_données_sql*)

Description

La fonction *CAST* convertit l'*expression_arithmétique* passée en paramètre dans le *type_données_sql* passé après le mot-clé **AS**.

Exemple

Dans cet exemple, l'année de réalisation du film est convertie en Entier :

```
SELECT Annee_Realisation, Titre, Realisateur, Media, Nb_Entrees
FROM FILMS
WHERE Annee_Realisation >= CAST('1960' AS INT)
```

CEILING (*expression_arithmétique*)

Description

La fonction *CEILING* retourne le plus petit entier supérieur ou égal à l'*expression_arithmétique* passée en paramètre.

Exemple

Cet exemple retourne le plus petit entier supérieur ou égal à -20,9 :

```
CEILING (-20,9) `retourne -20
```

CHAR (*expression_arithmétique*)

Description

La fonction *CHAR* retourne une chaîne de caractères de longueur fixe dépendante du type de l'*expression_arithmétique* passée en paramètre.

Exemple

Cet exemple retourne une chaîne de caractères basée sur le code de la première lettre de chaque nom :

```
SELECT CHAR (ASCII (SUBSTRING (Nom, 1, 1) ))  
FROM PERSONNES;
```

CHAR_LENGTH

CHAR_LENGTH (*expression_arithmétique*)

Description

La fonction *CHAR_LENGTH* retourne le nombre de caractères présents dans l'*expression_arithmétique* passée en paramètre.

Exemple

Cet exemple retourne le nombre de caractères dans le nom des produits dont le poids ne dépasse pas 15 kg :

```
SELECT CHAR_LENGTH (Nom)
FROM PRODUITS
WHERE poids < 15,00
```

COALESCE

COALESCE (*expression_arithmétique, ..., expression_arithmétique*)

Description

La fonction *COALESCE* retourne la valeur de la première expression non-NULL trouvée parmi le ou les argument(s) de type *expression_arithmétique* passé(s) en paramètre(s). La fonction retourne **NULL** si toutes les expressions passées sont NULL.

Exemple

Cet exemple retourne tous les numéros des factures depuis l'année 2007 pour lesquelles la TVA est supérieure à 0 :

```
SELECT NO_FACTURES
FROM FACTURES
WHERE EXTRACT(YEAR (DATE_FACTURE)) = 2007
HAVING (COALESCE (TVA_FACTURE;0) > 0)
```


CONCAT

CONCAT (*expression_arithmétique, expression_arithmétique*)

Description

La fonction *CONCAT* retourne les deux expressions passées en paramètres sous la forme d'une seule chaîne concaténée.

Exemple

Cet exemple retourne le prénom et le nom sous forme de chaîne concaténée :

```
SELECT CONCAT (CONCAT (PERSONNES.Prenom, ' '), PERSONNES.Nom) FROM PERSONNES;
```

CONCATENATE

CONCATENATE (*expression_arithmétique, expression_arithmétique*)

Description

La fonction *CONCATENATE* retourne les deux expressions passées en paramètres sous la forme d'une seule chaîne concaténée.

Exemple

Reportez-vous à l'exemple de la fonction [CONCAT](#).

COS (*expression_arithmétique*)

Description

La fonction **COS** retourne le cosinus de l'*expression_arithmétique* passée en paramètre. L'*expression_arithmétique* définit un angle exprimé en radians.

Exemple

Cet exemple retourne le cosinus de l'angle défini par (degrees * 180 / 3,1416) :

```
SELECT COS(degrees * 180 / 3,1416)
FROM TABLES_OF_ANGLES;
```

COT (*expression_arithmétique*)

Description

La fonction *COT* retourne la cotangente de l'*expression_arithmétique*. L'*expression_arithmétique* définit un angle exprimé en radians.

Exemple

Cet exemple retourne la cotangente de l'angle défini par la valeur 3,1416 :

```
SELECT COT(3,1416)
FROM TABLES_OF_ANGLES;
```

COUNT

```
COUNT ( { [ ALL | DISTINCT ] expression_arithmétique | * } )
```

Description

La fonction *COUNT* retourne le nombre de valeurs non-NULL présentes dans l'*expression_arithmétique*. Les mots-clés facultatifs **ALL** et **DISTINCT** permettent respectivement d'inclure ou d'exclure les valeurs dupliquées (doublons) dans le calcul.

Si vous passez le caractère * , la fonction retourne le nombre total d'enregistrements dans l'*expression_arithmétique*, y compris les valeurs NULL et les doublons.

Exemple

Cet exemple retourne le nombre total de films stockés dans la table FILMS :

```
SELECT COUNT (*)  
FROM FILMS
```

CURDATE ()

Description

La fonction *CURDATE* retourne la date courante.

Exemple

Cet exemple crée une table de factures et insère la date courante dans la colonne *DATE_FACT* :

```
TABLEAU ALPHA (30; tDate; 0)
```

```
Debut SQL
```

```
CREATE TABLE FACTURES  
(DATE_FACT VARCHAR(40));
```

```
INSERT INTO FACTURES  
(DATE_FACT)  
VALUES (CURDATE());
```

```
SELECT *  
FROM FACTURES  
INTO :tDate;
```

```
Fin SQL
```

```
`Le tableau tDate retournera la date et l'heure d'exécution de la commande INSERT
```

CURRENT_DATE ()

Description

La fonction *CURRENT_DATE* retourne la date locale courante.

Exemple

Cet exemple crée une table de factures et insère la date courante dans la colonne DATE_FACT :

```
TABLEAU ALPHA (30; tDate; 0)
```

```
Debut SQL
```

```
CREATE TABLE FACTURES  
(DATE_FACT VARCHAR(40));
```

```
INSERT INTO FACTURES  
(DATE_FACT)  
VALUES (CURRENT_DATE());
```

```
SELECT *  
FROM FACTURES  
INTO :tDate;
```

```
Fin SQL
```

```
`Le tableau tDate retournera la date et l'heure d'exécution de la commande INSERT
```

CURRENT_TIME ()

Description

La fonction *CURRENT_TIME* retourne l'heure locale courante.

Exemple

Cet exemple crée une table de factures et insère l'heure courante dans la colonne HEURE_FACT :

```
TABLEAU ALPHA (30; tDate; 0)
```

```
Debut SQL
```

```
CREATE TABLE FACTURES  
(HEURE_FACT VARCHAR(40));
```

```
INSERT INTO FACTURES  
(HEURE_FACT)  
VALUES (CURRENT_TIME());
```

```
SELECT *  
FROM FACTURES  
INTO :tDate;
```

```
Fin SQL
```

```
`Le tableau tDate retournera l'heure d'exécution de la commande INSERT
```


CURRENT_TIMESTAMP

CURRENT_TIMESTAMP ()

Description

La fonction *CURRENT_TIMESTAMP* retourne la date et l'heure locales courantes.

Exemple

Cet exemple crée une table de factures et insère la date et l'heure courantes dans la colonne DATEHEURE_FACT :

```
TABLEAU ALPHA (30; tDate; 0)
```

```
Debut SQL
```

```
CREATE TABLE FACTURES  
(DATEHEURE_FACT VARCHAR(40));
```

```
INSERT INTO FACTURES  
(DATEHEURE_FACT )  
VALUES (CURRENT_TIMESTAMP());
```

```
SELECT *  
FROM FACTURES  
INTO :tDate;
```

```
Fin SQL
```

```
`Le tableau tDate retournera la date et l'heure d'exécution de la commande INSERT
```

CURTIME ()

Description

La fonction *CURTIME* retourne l'heure courante avec une précision d'une seconde.

Exemple

Cet exemple crée une table de factures et insère l'heure courante dans la colonne HEURE_FACT :

```
TABLEAU ALPHA (30; tDate; 0)
```

```
Debut SQL
```

```
CREATE TABLE FACTURES  
(HEURE_FACT VARCHAR(40));
```

```
INSERT INTO FACTURES  
(HEURE_FACT)  
VALUES (CURTIME());
```

```
SELECT *  
FROM FACTURES  
INTO :tDate;
```

```
Fin SQL
```

```
`Le tableau tDate retournera l'heure d'exécution de la commande INSERT
```

DATABASE_PATH

DATABASE_PATH()

Description

La fonction **DATABASE_PATH** retourne le chemin d'accès complet de la base courante. La base courante peut être modifiée à l'aide de la commande SQL **USE DATABASE**. Par défaut, la base courante est la base 4D principale.

Exemple

Soit une base externe courante nommée TestBase.4DB et située dans le dossier "C:\MesBases". Après l'exécution du code suivant :

```
C_TEXTE ($vCrtDatabasePath)
Debut SQL
    SELECT DATABASE_PATH()
    FROM _USER_SCHEMA
    LIMIT 1
    INTO :$vCrtDatabasePath;
Fin SQL
```

... la variable *vCrtDatabasePath* contiendra "C:\MesBases\TestBase.4DB".

DATE_TO_CHAR (*expression_arithmétique*, *expression_arithmétique*)

Description

La fonction `DATE_TO_CHAR` retourne une représentation sous forme de texte de la date passée dans la première *expression_arithmétique* en fonction du format défini dans la deuxième *expression_arithmétique*. La première *expression_arithmétique* doit être de type Timestamp ou Duration et la deuxième doit être de type texte.

Les codes de formatage pouvant être utilisés sont fournis ci-dessous. En général, si un code de formatage débute par un caractère majuscule, le nombre débutera par un ou plusieurs zéros si nécessaire. Sinon, il n'y aura pas de zéros en tête de la valeur. Par exemple, si `dd` retourne 7, `Dd` retournera 07.

L'usage de caractères majuscules et minuscules dans les codes de formatage pour les jours et les mois sera reproduit dans le résultat retourné. Par exemple, passer "day" retournera "lundi", passer "Day" retournera "Lundi" et passer "DAY" retournera "LUNDI".

am - am ou pm en fonction de la valeur de l'heure

pm - am ou pm en fonction de la valeur de l'heure

a.m. - a.m. ou p.m. en fonction de la valeur de l'heure

p.m. - a.m. ou p.m. en fonction de la valeur de l'heure

d - numéro du jour de la semaine (1-7)

dd - numéro du jour dans le mois (1-31)

ddd - numéro du jour dans l'année

day - nom du jour de la semaine

dy - nom du jour de la semaine abrégé sur 3 lettres

hh - heure en chiffres, basée sur 12 heures (0-11)

hh12 - heure en chiffres, basée sur 12 heures (0-11)

hh24 - heure en chiffres, basée sur 24 heures (0-23)

J - jour Julien

mi - minutes (0-59)

mm - mois en chiffres (0-12)

q - trimestre

ss - secondes (0-59)

sss - millisecondes (0-999)

w - numéro de la semaine dans le mois (1-5)

ww - numéro de la semaine dans l'année (1-53)

yy - année

yyyy - année

[Tout texte] - le texte placé entre des crochets ([]) n'est pas interprété et est inséré tel quel

-,.; 'espace' 'tabulation' - ces caractères sont conservés tels quels, sans modification

Exemple

Cet exemple retourne la date de naissance sous la forme du numéro de jour de la semaine (1-7):

```
SELECT DATE_TO_CHAR (Date_naissance, 'd')
FROM EMPLOYES;
```

DAY (*expression_arithmétique*)

Description

La fonction *DAY* retourne le jour du mois de la date passée dans l'*expression_arithmétique*.

Exemple

Cet exemple retourne le jour du mois de la date "05-07-2007":

```
SELECT DAY('05-07-2007');  
`retourne 5
```

DAYNAME

DAYNAME (*expression_arithmétique*)

Description

La fonction *DAYNAME* retourne le nom du jour de la semaine pour la date passée dans l'*expression_arithmétique*.

Exemple

Cet exemple retourne le nom du jour de la date de naissance :

```
SELECT DAYNAME(Date_naissance);
```

DAYOFMONTH (*expression_arithmétique*)

Description

La fonction *DAYOFMONTH* retourne un nombre représentant le jour du mois (de 1 à 31) de la date passée dans l'*expression_arithmétique*.

Exemple

Soit une table *PERSONNES* comportant un champ *Date_Naissance*. Le code suivant permet d'obtenir le numéro du jour de la date de naissance de chaque personne stockée dans la table :

```
SELECT DAYOFMONTH(Date_Naissance)
FROM PERSONNES;
```

DAYOFWEEK (*expression_arithmétique*)

Description

La fonction *DAYOFWEEK* retourne un nombre représentant le jour de la semaine (allant de 1 à 7, où 1 représente le dimanche et 7 le samedi) de la date passée dans l'*expression_arithmétique*.

Exemple

Soit une table PERSONNES comportant un champ Date_Naissance. Le code suivant permet d'obtenir le numéro de jour de la semaine de la date de naissance de chaque personne stockée dans la table :

```
SELECT DAYOFWEEK(Date_Naissance)
FROM PERSONNES;
```


DAYOFYEAR (*expression_arithmétique*)

Description

La fonction *DAYOFYEAR* retourne un nombre représentant le jour de l'année (de 1 à 366, où 1 est le 1er janvier) de la date passée dans l'*expression_arithmétique*.

Exemple

Soit une table PERSONNES comportant un champ Date_Naissance. Le code suivant permet d'obtenir le jour de l'année de la date de naissance de chaque personne stockée dans la table :

```
SELECT DAYOFYEAR(Date_Naissance)
FROM PERSONNES;
```

DEGREES (*expression_arithmétique*)

Description

La fonction *DEGREES* retourne le nombre de degrés de l'*expression_arithmétique*. La valeur passée dans l'*expression_arithmétique* représente un angle exprimé en radians.

Exemple

Cet exemple crée une table et insère des valeurs basées sur le nombre de degrés de la valeur Pi :

```
CREATE TABLE Degrees_table (PI_value float);
INSERT INTO Degrees_table VALUES
(DEGREES(PI()));
SELECT * FROM Degrees_table
```

EXP (*expression_arithmétique*)

Description

La fonction **EXP** retourne l'exponentielle de l'*expression_arithmétique*, c'est-à-dire e élevé à la puissance x, "x" étant la valeur passée dans l'*expression_arithmétique*.

Exemple

Cette exemple retourne e puissance 15 :

```
SELECT EXP( 15 ); `retourne 3269017,3724721107
```

EXTRACT

```
EXTRACT ({YEAR | MONTH | DAY | HOUR | MINUTE | SECOND | MILLISECOND} FROM expression_arithmétique)
```

Description

La fonction *EXTRACT* retourne de l'*expression_arithmétique* la partie spécifiée par le mot-clé associé à la fonction. L'*expression_arithmétique* doit être de type Timestamp.

Exemple

Cet exemple retourne tous les numéros de facture du mois de janvier :

```
SELECT NUM_FACTURE
FROM FACTURES
WHERE EXTRACT (MONTH (DATE_FACTURE)) = 1;
```

FLOOR (*expression_arithmétique*)

Description

La fonction *FLOOR* retourne le plus grand nombre entier inférieur ou égal à l'*expression_arithmétique* passée en paramètre.

Exemple

Cet exemple retourne le plus grand entier inférieur ou égal à -20.9:

```
FLOOR (-20.9); `retourne -21
```

hour (*expression_arithmétique*)

Description

La fonction *hour* retourne la partie 'heure' d'une heure passée dans l'*expression_arithmétique*. La valeur retournée est dans l'intervalle 0 à 23.

Exemple

Soit une table **FACTURES** avec un champ **Heure_livraison**. Pour afficher l'heure de la livraison :

```
SELECT hour(Heure_livraison)
FROM FACTURES;
```

INSERT

INSERT (*expression_arithmétique*, *expression_arithmétique*, *expression_arithmétique*, *expression_arithmétique*)

Description

La fonction **INSERT** insère une chaîne dans une autre à un emplacement donné. La première *expression_arithmétique* passée est la chaîne de destination. La seconde *expression_arithmétique* est l'indice auquel la chaîne passée dans la quatrième *expression_arithmétique* sera insérée ; la troisième *expression_arithmétique* indique le nombre de caractères à supprimer à partir du point d'insertion.

Exemple

Cet exemple insère "Cher " devant les prénoms dans la table PERSONNES :

```
SELECT INSERT (PERSONNES.Prenom, 0, 0, 'Cher ') FROM PERSONNES;
```

LEFT

LEFT (*expression_arithmétique*, *expression_arithmétique*)

Description

La fonction *LEFT* retourne la partie la plus à gauche de l'*expression_arithmétique* passée. La seconde *expression_arithmétique* indique le nombre de caractères à retourner à partir de la gauche de la première.

Exemple

Cet exemple retourne les prénoms et les deux premiers caractères des noms de la table PERSONNES :

```
SELECT Prenom, LEFT(Nom, 2)
FROM PERSONNES;
```


LENGTH

LENGTH (*expression_arithmétique*)

Description

La fonction *LENGTH* retourne le nombre de caractères de l'*expression_arithmétique* passée en paramètre.

Exemple

Cet exemple retourne le nombre de caractères des noms des produits de moins de 15 kg :

```
SELECT LENGTH (Nom)
FROM PRODUITS
WHERE Poids < 15.00
```

LOCATE

LOCATE (*expression_arithmétique*, *expression_arithmétique*₁, *expression_arithmétique*)

Description

La fonction *LOCATE* retourne la position de départ de la première occurrence d'une *expression_arithmétique* trouvée dans une seconde *expression_arithmétique*. Vous pouvez également passer une troisième *expression_arithmétique* pour spécifier la position du caractère auquel doit débuter la recherche.

Exemple

Cet exemple retourne la position de la première lettre X trouvée parmi les noms de la table PERSONNES :

```
SELECT Prenom, LOCATE('X',Nom)
FROM PERSONNES;
```

LOG (*expression_arithmétique*)

Description

La fonction **LOG** retourne le logarithme népérien (ou logarithme naturel) de l'*expression_arithmétique*.

Exemple

Cet exemple retourne le logarithme népérien de 50 :

```
SELECT LOG ( 50 );
```

LOG10 (*expression_arithmétique*)

Description

La fonction *LOG10* retourne le logarithme décimal (logarithme base 10) de l'*expression_arithmétique*.

Exemple

Cet exemple retourne le logarithme décimal de la valeur 50:

```
SELECT LOG10( 50 );
```

LOWER (*expression_arithmétique*)

Description

La fonction *LOWER* retourne la chaîne *expression_arithmétique* avec tous les caractères convertis en minuscules.

Exemple

Cet exemple retourne les noms des produits en minuscules :

```
SELECT LOWER (Nom)
FROM PRODUITS;
```

LTRIM (*expression_arithmétique*₁, *expression_arithmétique*₂)

Description

La fonction *LTRIM* supprime les éventuels espaces situés au début de l'*expression_arithmétique*. La deuxième *expression_arithmétique* (facultative) permet de désigner des caractères spécifiques à supprimer au début de la première *expression_arithmétique*.

Exemple

Cet exemple supprime tout espace situé au début du nom des produits :

```
SELECT LTRIM(Nom)
FROM PRODUITS;
```

MAX (*expression_arithmétique*)

Description

La fonction **MAX** retourne la valeur maximale de l'*expression_arithmétique*.

Exemple

Reportez-vous aux exemples des fonctions *SUM* et *AVG*.

MILLISECOND (*expression_arithmétique*)

Description

La fonction *MILLISECOND* retourne la partie millisecondes de l'heure passée dans *expression_arithmétique*.

Exemple

Soit une table *FACTURES* avec un champ *Heure_Livraison*. Pour afficher les millisecondes de l'heure de livraison :

```
SELECT MILLISECOND(Heure_Livraison)
FROM FACTURES;
```


MIN (*expression_arithmétique*)

Description

La fonction **MIN** retourne la valeur minimale de l'*expression_arithmétique*.

Exemple

Reportez-vous aux exemples des fonctions *SUM* et *AVG*.

MINUTE (*expression_arithmétique*)

Description

La fonction *MINUTE* retourne la partie 'minutes' de l'heure passée dans *expression_arithmétique*. La valeur retournée est située dans l'intervalle 0 à 59.

Exemple

Soit une table **FACTURES** avec un champ **Heure_Livraison**. Pour afficher les minutes de l'heure de livraison :

```
SELECT MINUTE(Heure_Livraison)
FROM FACTURES;
```

MOD (*expression_arithmétique*, *expression_arithmétique*)

Description

La fonction *MOD* retourne le reste de la division de la première *expression_arithmétique* par la seconde.

Exemple

Cet exemple retourne le reste de la division de 10 par 3 :

```
MOD(10,3) `retourne 1
```

MONTH (*expression_arithmétique*)

Description

La fonction *MONTH* retourne le numéro du mois (de 1 à 12) de la date passée dans l'*expression_arithmétique*.

Exemple

Soit la table *PERSONNES* avec un champ *Date_Naissance*. Pour obtenir le mois de la date de naissance de chaque personne :

```
SELECT MONTH(Date_Naissance)
FROM PERSONNES;
```

MONTHNAME

MONTHNAME (*expression_arithmétique*)

Description

La fonction *MONTHNAME* retourne le nom du mois de la date passée dans l'*expression_arithmétique*.

Exemple

Cet exemple retourne le nom du mois de chaque date de naissance :

```
SELECT MONTHNAME(Date_Naissance) ;
```

NULLIF (*expression_arithmétique*, *expression_arithmétique*)

Description

La fonction *NULLIF* retourne **NULL** si la première *expression_arithmétique* est égale à la seconde. Sinon, la valeur de première expression est retournée. Les deux expressions doivent être comparables.

Exemple

Cet exemple retourne NULL si le total de la facture est 0 :

```
NULLIF (FACTURES_TOTAL, 0) ;
```

OCTET_LENGTH

OCTET_LENGTH (*expression_arithmétique*)

Description

La fonction *OCTET_LENGTH* retourne le nombre d'octets de l'*expression_arithmétique*, incluant les éventuels espaces.

Exemple

Cet exemple retourne le nombre d'octets d'une colonne comportant des données binaires :

```
SELECT OCTET_LENGTH (MaCol_Binaire )
FROM MaTable
WHERE MaCol_Binaire = '93FB';
` retourne 2
```

PI ()

Description

La fonction *PI* retourne la valeur de la constante Pi.

Exemple

Reportez-vous à l'exemple de la fonction *DEGREES*.

POSITION

POSITION (*expression_arithmétique* **IN** *expression_arithmétique*)

Description

La fonction **POSITION** retourne une valeur indiquant la position de départ de la première *expression_arithmétique* au sein de la seconde. Si la chaîne n'est pas trouvée, la fonction retourne zéro.

Exemple

Cet exemple retourne l'emplacement du mot "York" dans tous les noms stockés dans la table PERSONNES :

```
SELECT Prenom, POSITION('York' IN Nom)
FROM PERSONNES;
```

POWER (*expression_arithmétique*, *expression_arithmétique*)

Description

La fonction *POWER* élève la première *expression_arithmétique* passée à la puissance "x", où "x" est la seconde *expression_arithmétique* passée.

Exemple

Cet exemple élève chaque valeur à la puissance 3:

```
SELECT Sources, POWER(Sources, 3)
FROM Valeurs
ORDER BY Sources
`retourne 8 pour Sources = 2 par exemple
```

QUARTER (*expression_arithmétique*)

Description

La fonction *QUARTER* retourne le trimestre de l'année (valeur comprise entre 1 et 4) auquel appartient la date passée dans l'*expression_arithmétique*.

Exemple

Soit la table *PERSONNES* avec un champ *Date_Naissance*. Pour obtenir le trimestre de la date de naissance de chaque personne :

```
SELECT QUARTER(Date_Naissance)
FROM PERSONNES;
```

RADIANS (*expression_arithmétique*)

Description

La fonction *RADIANS* retourne le nombre de radians de l'*expression_arithmétique*. La valeur de cette expression doit représenter un angle exprimé en degrés.

Exemple

Cet exemple retourne le nombre de radians d'un angle de 30 degrés :

```
RADIANS (30 ); `retourne la valeur 0,5236
```

RAND ([*expression_arithmétique*])

Description

La fonction *RAND* retourne une valeur de type Float aléatoire située entre 0 et 1. L'*expression_arithmétique* facultative peut être utilisée pour passer une valeur seed (valeur clé).

Exemple

Cet exemple insère des valeurs d'ID générées grâce à la fonction *RAND* :

```
CREATE TABLE PERSONNES
(ID INT32,
Nom VARCHAR);

INSERT INTO PERSONNES
(ID, Nom)
VALUES (RAND, 'Francis');
```

REPEAT

REPEAT(*expression_arithmétique*, *expression_arithmétique*)

Description

La fonction *REPEAT* retourne la première *expression_arithmétique* répétée le nombre de fois défini dans le second argument *expression_arithmétique*.

Exemple

Cet exemple illustre le principe de la fonction *REPEAT* :

```
SELECT REPEAT( 'repete', 3 ) ` retourne 'repeterepeterepete'
```

REPLACE

REPLACE (*expression_arithmétique*, *expression_arithmétique*, *expression_arithmétique*)

Description

La fonction *REPLACE* recherche dans la première *expression_arithmétique* toutes les occurrences de la seconde *expression_arithmétique* et remplace chaque occurrence trouvée par la troisième *expression_arithmétique* passée. Si aucune occurrence n'est trouvée, la première *expression_arithmétique* est retournée inchangée.

Exemple

Cet exemple remplacera la chaîne "Francs" par "Euros" dans la colonne Devise :

```
SELECT Nom, REPLACE(Devise, 'Francs', 'Euros')
FROM PRODUITS;
```

RIGHT (*expression_arithmétique*, *expression_arithmétique*)

Description

La fonction *RIGHT* retourne la partie la plus à droite de l'*expression_arithmétique* passée. La seconde *expression_arithmétique* indique le nombre de caractères à retourner à partir de la droite de la première.

Exemple

Cet exemple retourne les prénoms et les deux premiers caractères des noms de la table **PERSONNES** :

```
SELECT Prenom, RIGHT(Nom, 2)
FROM PERSONNES;
```


ROUND (*expression_arithmétique* [, *expression_arithmétique*])

Description

La fonction *ROUND* arrondit la valeur passée dans la première *expression_arithmétique* à "n" décimales, "n" étant facultativement passé dans la seconde *expression_arithmétique*. Si la seconde *expression_arithmétique* n'est pas passée, la valeur d'origine est arrondie au nombre entier le plus proche.

Exemple

Cet exemple arrondit le nombre d'origine à deux décimales :

```
ROUND (1234.1966, 2) `retourne 1234.2000
```

RTRIM (*expression_arithmétique* [, *expression_arithmétique*])

Description

La fonction *RTRIM* supprime les éventuels espaces situés à la fin de l'*expression_arithmétique*. La deuxième *expression_arithmétique* (facultative) permet de désigner des caractères spécifiques à supprimer à la fin de la première *expression_arithmétique*.

Exemple

Cet exemple supprime tout espace superflu situé à la fin du nom des produits :

```
SELECT RTRIM(Nom)
FROM PRODUITS;
```

SECOND

SECOND (*expression_arithmétique*)

Description

La fonction *SECOND* retourne la partie "secondes" (valeur comprise entre 0 et 59) de l'heure passée dans l'*expression_arithmétique*.

Exemple

Soit une table *FACTURES* avec un champ *Heure_Livraison*. Pour afficher les secondes de l'heure de livraison :

```
SELECT SECOND(Heure_Livraison)
FROM FACTURES;
```

SIGN (*expression_arithmétique*)

Description

La fonction *SIGN* retourne le signe de l'*expression_arithmétique*. La fonction retourne 1 pour un nombre positif, -1 pour un nombre négatif ou 0.

Exemple

Cet exemple retourne tous les montants négatifs trouvés dans la table *FACTURES* :

```
SELECT MONTANT
FROM FACTURES
WHERE SIGN(MONTANT) = -1;
```

SIN (*expression_arithmétique*)

Description

La fonction **SIN** retourne le sinus de l'*expression_arithmétique*. La valeur de cette expression doit représenter un angle exprimé en radians.

Exemple

Cet exemple retourne le sinus de l'angle en radians :

```
SELECT SIN(radians)
FROM TABLES_OF_ANGLES;
```

SPACE

SPACE (*expression_arithmétique*)

Description

La fonction *SPACE* retourne une chaîne de caractères constituée du nombre d'espacements défini dans l'argument *expression_arithmétique*. Si la valeur de l'*expression_arithmétique* est négative, une valeur **NULL** est retournée.

Exemple

Cet exemple ajoute trois espaces devant les noms de la table **PERSONNES** :

```
SELECT CONCAT (SPACE (3) , PERSONNES.Nom) FROM PERSONNES;
```

SQRT (*expression_arithmétique*)

Description

La fonction *SQRT* retourne la racine carrée de l'*expression_arithmétique*.

Exemple

Cet exemple retourne la racine carrée du fret :

```
SELECT Fret, SQRT(Fret) AS "Racine carrée du fret"  
FROM Commandes
```

SUBSTRING

SUBSTRING (*expression_arithmétique*, *expression_arithmétique*, [*expression_arithmétique*])

Description

La fonction *SUBSTRING* retourne une sous-chaîne de la première *expression_arithmétique* passée. La seconde *expression_arithmétique* fournit la position de départ de la sous-chaîne et la troisième *expression_arithmétique* (facultative) indique le nombre de paramètres à retourner à partir de la position de départ. Si la troisième *expression_arithmétique* est omise, la fonction retournera tous les caractères à partir de la position de départ.

Exemple

Cet exemple retourne 4 caractères du nom de la boutique à partir du 2e caractère :

```
SELECT SUBSTRING(Nom_boutique,2,4)
FROM Geographie
WHERE Nom_boutique= 'Paris';
```


SUM ([**ALL** | **DISTINCT**] *expression_arithmétique*)

Description

La fonction *SUM* retourne la somme de l'*expression_arithmétique*. Les mots-clés facultatifs **ALL** et **DISTINCT** permettent respectivement d'inclure ou d'éliminer les éventuelles valeurs dupliquées (doublons) du calcul.

Exemple

Cet exemple retourne la somme des ventes prévues moins la somme des ventes réalisées, ainsi que le montant minimum et maximum des ventes réalisées multiplié par 100 et divisé par les ventes prévues dans la table **COMMERCIAUX** :

```
SELECT MIN( ( VENTES* 100 ) / QUOTA ),  
MAX( ( VENTES* 100 ) / QUOTA ),  
SUM( QUOTA ) - SUM( VENTES)  
FROM COMMERCIAUX
```

TAN (*expression_arithmétique*)

Description

La fonction **TAN** retourne la tangente de l'*expression_arithmétique*. La valeur de cette expression doit représenter un angle exprimé en radians.

Exemple

Cet exemple retourne la tangente de l'angle exprimé en radians :

```
SELECT TAN(radians)
FROM TABLES_OF_ANGLES;
```

TRANSLATE (*expression_arithmétique*, *expression_arithmétique*, *expression_arithmétique*)

Description

La fonction *TRANSLATE* retourne la première *expression_arithmétique* avec toutes les occurrences de chacun des caractères passés dans la seconde *expression_arithmétique* remplacés par leurs caractères correspondants passés dans la troisième *expression_arithmétique*.

Le remplacement est effectué caractère par caractère : le premier caractère de la seconde *expression_arithmétique* est remplacé à chaque fois qu'il apparaît dans la première *expression_arithmétique* par le premier caractère de la troisième *expression_arithmétique*, et ainsi de suite.

S'il y a moins de caractères dans la troisième *expression_arithmétique* que dans la seconde, tous les caractères de la seconde *expression_arithmétique* qui n'ont pas de correspondance dans la troisième seront supprimés de la première. Par exemple, si la seconde *expression_arithmétique* comprend 5 caractères et que la troisième n'en contient que 4, à chaque fois que le 5e caractère de la seconde *expression_arithmétique* est trouvé dans la première *expression_arithmétique*, il est supprimé de la valeur retournée.

Exemple

Cet exemple remplace toutes les occurrences de "a" par "1" et les occurrences de "b" par "2" :

```
TRANSLATE ('abcd', 'ab', '12') ` retourne '12cd'
```

TRIM ([[**LEADING** | **TRAILING** | **BOTH**] [*expression_arithmétique*] **FROM**] *expression_arithmétique*)

Description

La fonction *TRIM* supprime des extrémités de l'*expression_arithmétique* tout espace superflu (ou tout autre caractère tel que défini dans la première *expression_arithmétique*).

Vous pouvez passer le mot-clé **LEADING** afin d'indiquer que les espaces/caractères doivent être supprimés du début de l'*expression_arithmétique* uniquement, **TRAILING** afin d'indiquer qu'ils doivent être supprimés à la fin de l'expression uniquement, ou **BOTH** pour combiner les deux opérations. N'utiliser aucun de ces mots-clés équivaut à passer **BOTH** (les espaces/caractères sont supprimés du début et de la fin de l'*expression_arithmétique*).

Le premier argument *expression_arithmétique* (facultatif) permet de désigner le ou les caractère(s) à supprimer individuellement de la seconde *expression_arithmétique*. S'il est omis, seuls les espaces seront supprimés.

Exemple

Cet exemple supprime les espaces des noms de produits :

```
SELECT TRIM(Nom)
FROM PRODUITS;
```

TRUNC (*expression_arithmétique* [, *expression_arithmétique*])

Description

La fonction *TRUNC* retourne la première *expression_arithmétique* tronquée à N chiffres à droite du séparateur décimal, N étant la valeur passée dans la seconde *expression_arithmétique* (facultative). Si la seconde *expression_arithmétique* est omise, la première expression est retournée tronquée sans décimale.

Exemple

Cette fonction tronque la valeur passée à un chiffre après le séparateur :

```
TRUNC(2.42 , 1) `retourne 2.40
```

TRUNCATE

TRUNCATE (*expression_arithmétique*[, *expression_arithmétique*])

Description

La fonction *TRUNCATE* retourne la première *expression_arithmétique* tronquée à N chiffres à droite du séparateur décimal, N étant la valeur passée dans la seconde *expression_arithmétique* (facultative). Si la seconde *expression_arithmétique* est omise, la première expression est retournée tronquée sans décimale.

Exemple

Reportez-vous à l'exemple de la fonction *TRUNC*.

UPPER (*expression_arithmétique*)

Description

La fonction *UPPER* retourne la chaîne *expression_arithmétique* avec tous les caractères convertis en majuscules.

Exemple

Cet exemple retourne les noms des produits en majuscules :

```
SELECT UPPER (Nom)
FROM PRODUITS;
```

WEEK (*expression_arithmétique*)

Description

La fonction *WEEK* retourne le numéro de semaine de l'année (valeur comprise entre 1 et 54) de la date passée dans l'*expression_arithmétique*. La semaine débute le dimanche et le 1er janvier appartient toujours à la première semaine.

Exemple

Cet exemple retourne un nombre indiquant la semaine de l'année pour chaque date de naissance passée :

```
SELECT WEEK(Date_naissance);
```


YEAR (*expression_arithmétique*)

Description

La fonction *YEAR* retourne la partie "année" de la date passée dans l'*expression_arithmétique*.

Exemple

Soit la table *PERSONNES* avec un champ *Date_Naissance*. Pour obtenir l'année de la date de naissance de chaque personne :

```
SELECT YEAR(Date_Naissance)
FROM PERSONNES;
```

